Universidad Nacional de La Plata
Facultad de Informática

Análisis del rendimiento de algoritmos paralelos de propósito general en GPGPU. Aplicación a un problema de mallado de elementos finitos.

Adriana A. Gaudiani

Director:
Dr. Marcelo Naiouf
Codirector:
Ing. Armando De Giusti

Trabajo Final presentado para obtener el grado de Especialista en

Cómputo de Altas Prestaciones y Tecnología Grid

Junio de 2012

Agradecimientos:

A la memoria de mis padres, quienes me enseñaron a luchar por mis objetivos.

Al Dr. Gabriel Acosta, quién fue el inspirador del tema de aplicación, a los investigadores del III-LIDI, que me ayudaron a superar varias dificultades, y a Santiago Montiel.

Índice de contenidos

Objetivos

El objetivo de este trabajo es analizar el desarrollo, optimización y evaluación de algoritmos paralelos de propósito general cuya ejecución se implementa sobre la arquitectura de cómputo de alto rendimiento que ofrecen los multiprocesadores gráficos de las placas de video. Este tipo de arquitectura de cómputo, como las GPU de la placa de video NVIDIA, permite obtener mejoras significativas en el tiempo de procesamiento de aplicaciones no gráficas. Este objetivo es alcanzable con un uso eficiente de su modelo de memoria y de programación.

En este trabajo se estudiará el rendimiento de un algoritmo de mallado de elementos finitos, cuya paralelización es adecuada para un modelo de programación paralela de memoria compartida. Este objetivo incluye determinar las técnicas de optimización del modelo de programación CUDA que hagan más eficiente el uso de los niveles de memoria de la GPU para esta aplicación y que permitan su escalado a unidades con mayor cantidad de multiprocesadores.

El aporte de este trabajo es mostrar la ganancia obtenida con el uso de multiprocesadores gráficos al ejecutar aplicaciones paralelas de propósito general, que se caracterizan por proveer un alto paralelismo de datos. Como también exponer el potencial presente en las GPU, de eficiencia y escalabilidad, cuando se dan estas condiciones en la aplicación sobre la que se trabaja.

Temas de Investigación

Los temas centrales de investigación necesarios para la elaboración de este trabajo son:

- **Programación sobre modelos de memoria compartida**: El entorno de cómputo de la GPU está compuesto por un conjunto de multiprocesadores, sobre los que se mapean los bloques de threads en ejecución, dependiendo de los recursos disponibles. La cantidad de bloques y de threads a utilizar al momento de hacer la ejecución, deben ser investigados para lograr un uso eficiente de la arquitectura.

- **Optimización del cómputo sobre arquitectura de multiprocesadores**: La programación del algoritmo paralelo requiere utilizar CUDA para escribir *kernels*, o funciones que corran en la GPU. Es necesario escribir un código que utilice eficientemente los recursos como registros y niveles de memoria de manera explícita en el código. Esto permitirá optimizarlo.

- **Estudio del rendimiento de algoritmos paralelos de propósito general sobre GPU**: El análisis de los resultados obtenidos permitirá determinar el rendimiento logrado y el nivel de escalabilidad que se alcanza en función de recursos como la cantidad de memoria.

Capítulo 1

Cómputo de Alto Rendimiento

1.1 Introducción

Este capítulo hace un repaso acerca de la evolución histórica de la tecnología de microprocesadores y posteriormente de las arquitecturas de cómputo de alto rendimiento (HPC), las cuales surgen tratando de brindar una respuesta a la incesante demanda de recursos de cómputo desde la comunidad de desarrolladores. En un comienzo, la necesidad provenía de las aplicaciones científicas e ingenieriles, y aun siguen siendo problemas de gran desafío y en búsqueda constante de mejores rendimientos. Actualmente, se agregan otras áreas de desarrollo a la demanda, como sucede con problemas críticos de defensa, estrategia e inteligencia cuyas operaciones son primordialmente, en aritmética entera.

El alcance de PetaFlops en el cómputo, o 10^{15} operaciones por segundo, fue posible gracias a arquitecturas altamente concurrentes, como *RoadRunner* de Los Alamos National Laboratory, o como *BlueGene/P* de IBM, o *BlueWaters* de la University of Illinois.

Los microprocesadores convencionales evolucionaron de mono-procesador a multi-procesador y multi-threading. Las arquitecturas de cómputo evolucionaron para reunir conjuntos de computadoras o clusters, y posteriormente pasar a arquitecturas híbridas que combinan multiprocesadores con un área de memoria común, y múltiples hojas de multiprocesadores que se comunican a través de redes de interconexión, con pasaje de mensajes.

En 2007, el estado del arte en Cómputo de Alto Rendimiento era el *Cómputo a Peta-Escala*. El actual es el *Cómputo a Exa-Escala*, cuyo alcance es cree posible gracias a estar inmersos en la *Era del Cómputo Heterogéneo*.

Los multiprocesadores gráficos, GPUs, se transformaron en una atractiva opción tanto para aplicaciones científicas como para otro tipo de cómputo no gráfico. Esta arquitectura incluye cientos de arreglos de procesadores que son capaces de realizar una operación de punto flotante a la vez, por ciclo de reloj. Las arquitecturas actuales ya integran al hardware los multiprocesadores gráficos presentes en las placas de video, los cuales funcionan como coprocesadores paralelos de extremo poder de cómputo. De esta forma, los desarrolladores están obligados a programar las aplicaciones en paralelo si desean software escalable y eficiente, que esté a la altura de estas nuevas arquitecturas y obtenga el mayor provecho de sus prestaciones.

Las nuevas plataformas de desarrollo integran todas estas tecnologías de hardware y nuevos modelos de programación paralela, en búsqueda de alcanzar la escala de ExaFlops como potencia de cómputo.

La exigencia de la comunidad científica de más potencia de cómputo y más recursos computacionales quedó plasmada en varios reportes y estudios, como [1] y [2]. En el

reporte de DARPA se expresa el estado del arte del cómputo avanzado y las recomendaciones del *Exascale Working Group* para lograr este objetivo para el 2015 [3].

1.2 La crisis del hardware

Desde la llegada de "*la crisis del hardware*" en el 2003, los investigadores se ocuparon de encontrar nuevas técnicas para reducir su impacto en el cómputo de alto rendimiento. La necesidad de enfrentar la latencia de memoria llevó a complejizar la tecnología de los microprocesadores agregando concurrencia interna, cada vez en mayor medida.

Como la velocidad de los procesadores era mucho mayor que la velocidad de la memoria, los diseñadores agregaron varios niveles de cache a la memoria principal. La necesidad de aumentar la potencia de cómputo llevó a poner más y más procesadores en un mismo chip y correr múltiples hilos de ejecución en cada núcleo, pero pronto surgió la complicación de no poder alimentar a todos ellos con los datos que demandaban durante el cómputo.

Las aplicaciones fueron necesitando grandes conjuntos de datos a procesar, llegando pronto al orden del Terabyte. Estos datos podían estar físicamente distribuidos lejos del procesador – en otros chips, necesitando pasar uno o más ruteadores, a través de fibra óptica –, resultando en una gran tasa de fallos de cache y más tiempo ocioso de los procesadores esperando por los datos.

La mayoría de las aplicaciones no fueron programadas explotando la concurrencia, por lo tanto, no aprovechaban los avances de la arquitectura de los microprocesadores:

- muchos núcleos en un chip (*multicore*).
- múltiples hilos de ejecución (*multithreading*).

Estos avances optimizaron el rendimiento de los microprocesadores, pero no necesariamente extendieron esta propiedad a las aplicaciones.

Los investigadores anunciaron entonces, que la solución sería escribir software escalable en paralelo, tratando de explotar la concurrencia propia de la aplicación, anunciando esta opción como la base de la programación de alto rendimiento [4] [5] [6].

A continuación se presenta una síntesis de la evolución de los microprocesadores de un núcleo y los problemas que surgieron a medida que los ingenieros del hardware implementaban mejoras para lograr más y más rendimiento en el cómputo hasta llegar al momento conocido como "la crisis del hardware" y el problema originado por la latencia de memoria.

1.3 De procesadores de un núcleo a multiprocesadores con múltiples hilos

La evolución de los microprocesadores basados en una única unidad central de proceso (CPU), tales como la familia Intel Pentium ® o la familia AMD ® Opteron™, llevaron a un rápido incremento en el rendimiento y fuerte descenso en el costo de las aplicaciones de computadora por más de veinte años. Estos procesadores alcanzaron el orden de miles de millones (GIGA) de instrucciones de punto flotante por segundo (GFLOPS) en las

computadoras de escritorio o personales y cientos de GFLOPS en los servidores basados en clusters. Para los años ’80, la idea generalizada era que el rendimiento de las computadoras aumentaba a medida que aumentaba la velocidad de los procesadores [7]. Durante este tiempo, los desarrolladores de software vieron como sus aplicaciones funcionaban más rápido con cada nueva generación de procesadores, sin modificar ninguna línea de código. Todo esto, gracias a la escalabilidad obtenida al aumentar las prestaciones ofrecidas por los procesadores de un núcleo. Pero, la complejidad computacional de los problemas a resolver también crecía constantemente, lo cual exigió soportar volúmenes mayores de datos a procesar, demandando más y más rendimiento de los microprocesadores.

¿Cómo se enfrentó la demanda de más rendimiento de cómputo?

Uno de los objetivos de los investigadores en micro-arquitectura fue diseñar microprocesadores que realizaran las tareas en el menor tiempo posible, usando mínima potencia y al menor costo posible. Los diseñadores de CPUs lograron incrementar la potencia de los procesadores durante casi 30 años minimizando la latencia y maximizando el trabajo del procesador por ciclo de reloj. Su trabajo estuvo enfocado en:

- Aumentar la velocidad del reloj para obtener más ciclos en igual período de tiempo.

- Optimizar la ejecución con el uso de técnicas como pipeline, predicción de saltos y reordenamiento del flujo de instrucciones, para obtener más instrucciones por ciclo.

- Aumentar la memoria cache para poner los datos lo más cerca posible del procesador debido a que el tiempo de acceso a la memoria principal era cada vez más lento que la CPU.

Esto fue beneficioso para incrementar el rendimiento y escalabilidad de las aplicaciones secuenciales – como lo eran buena parte de las aplicaciones que se beneficiaban de esta evolución [8]. La industria de los microprocesadores y de los circuitos integrados ha evolucionado en este tiempo siguiendo los pasos de la “Ley de Moore” [9]. Esta ley predecía un crecimiento exponencial de la complejidad de los circuitos integrados, y caracterizó los avances tecnológicos de esta industria. En [10], los autores realizan una interesante descripción de los problemas que surgían en 2001 en micro-arquitectura y los desafíos que imponían al futuro del desarrollo de los microprocesadores.

Desde 2003, la carrera por lograr mayores velocidades en el reloj de las CPU de un solo núcleo fue desacelerando debido a muchas razones, como la física de los semiconductores y el diseño de los chips. Los microprocesadores de un núcleo habían alcanzado los límites impuestos en sus tamaños debido a la naturaleza de los semiconductores. La estructura atómica del silicio y los efectos cuánticos son factores que detuvieron la carrera de la “miniaturización” de los componentes. A este efecto se suma haber llegado a los límites en la explotación del paralelismo a nivel de instrucción [11].

Uno de los obstáculos de más impacto en el futuro de los microprocesadores fue la latencia provocada por el acceso a memoria, especialmente cuando los datos debían traerse de un segundo nivel de cache y, peor aún, si provenían de la memoria principal. Esta situación representó un importante problema para el rendimiento del cómputo, a

pesar que la CPU podía ejecutar instrucciones fuera de orden para optimizar su trabajo, no era suficiente para atender la latencia.

Para estos años, los ingenieros del hardware ya se enfrentaban a los límites en la búsqueda del alto rendimiento en procesadores secuenciales y multicores. En resumen, estos límites estuvieron relacionados con los siguientes factores:

- Disipación de calor.
- Velocidad de la luz.
- Latencia de memoria.
- Paralelismo a nivel de instrucción.

Fueron también los ingenieros del hardware quienes buscaron soluciones urgentes a cada uno de estos factores limitantes del rendimiento. Los avances orientados a minimizar sus efectos estuvieron orientados en las siguientes direcciones:

- *Hyperthreading*. Intel presenta, en el 2004, la técnica de Hyper-Threading (tecnología Intel® HT), presente en la familia Intel® Xeon™, con el objetivo de manejar la latencia utilizando concurrencia. Esta técnica, presente en los procesadores actuales, mediante duplicación de los registros hace que el sistema operativo vea dos procesadores lógicos cuando en realidad hay un solo procesador físico, y envíe así un hilo de ejecución a cada uno, minimizando el tiempo de espera del procesador. [12]

- *Multicore*. La búsqueda incesante de procesadores con mayor rendimiento y los inconvenientes ya nombrados, llevaron a los fabricantes a aumentar la cantidad de procesadores en un chip impulsando el mercado de los procesadores de muchos núcleos o multiprocesadores. En el año 2005, el chip SPARC64 VI tenía 2 procesadores por UCP[1] y desde mucho antes, PowerPC de RISC también ofrecía una tecnología SMP de múltiples procesadores con su propia cache[2]. Pero, estos avances tecnológicos sólo podrían ser aprovechados en todo su potencial, corriendo software programado y compilado en paralelo [11].

- *Cache*. En esta dirección, se mantuvo la idea deaumentar el tamaño de la cache, que fueran de tecnologías de acceso más rápido y estén ubicadas más cerca del procesador. Debido a que la rapidez se conseguía con caches pequeñas, los fabricantes debieron diseñar una jerarquía de caches rápidas y pequeñas. Aparece un nuevo inconveniente que exige una pronta respuesta pues, al crecer el número de procesadores en un chip DRAM, no crecen al mismo ritmo las conexiones entre el chip y el resto de la computadora. Por esto, no hay relación entre la tarea que un procesador está realizando y en qué lugar pueden residir el próximo conjunto de datos que necesitará, provocando fallos en la localidad de los datos [13].

[1] http://www.fujitsu.com/global/services/computing/server/sparcenterprise/technology/performance/processor.html

[2] http://archive.rootvg.net/column_risc.htm

¿Cómo se impactó en el desarrollo de aplicaciones?

Murphy presentó un artículo en el 2007 [14], exponiendo los resultados de sus experiencias en Sandia National Laboratories[3]. Murphy explica que a las necesidades de arquitecturas de cómputo de alto rendimiento tradicionales para correr aplicaciones orientadas a cálculo en punto flotante, se está agregando otra área orientada a la aritmética entera. Esta s aplicaciones críticas eran emergentes del área de la Matemática Discreta y la Programación Entera. La evolución de las supercomputadoras desde los '70 acompañó las necesidades de cálculo en punto flotante de las aplicaciones científicas e ingenieriles, las cuales fueron demandantes de gran cantidad de potencia de cómputo en ese entonces, y lo siguen siendo. Pero, esta evolución no ofrecía las prestaciones de alto rendimiento necesarias para este nuevo tipo de aplicaciones, como son las aplicaciones a teoría de grafos. El autor las presenta como dominadas por la latencia más que por el ancho de banda y propensas a mayor tasa de fallos de cache y de saltos, imponiendo así un desafío para las arquitecturas de los nuevos procesadores superescalares de ese momento.

Herb Sutter, en su artículo para *Dr. Dobbs Journal* del 2005 [8], decía: "*Your free lunch will soon be over*", anunciando el fin de la era del código secuencial y exponiendo el impacto en el desarrollo de software. La transición de CPUs mono-procesador a multiprocesadores con múltiples hilos afectaría permanentemente el modo de escribir el software. Si se desea aprovechar en todo su potencial la última tecnología de los procesadores, será necesario escribir aplicaciones eficientemente concurrentes, a pesar de la dificultad que esto impone al momento de programar. Es sabido que encontrar el paralelismo inherente a un problema no fue ni será tarea fácil. Sutter presentó esta condición como esencial, para explotar los avances que las nuevas tecnologías de procesadores ofrecían al rendimiento de las prestaciones.

Para 2008, los investigadores aseguraban que el rendimiento logrado con multiprocesadores era muy pobre para aplicaciones con uso intensivo de datos. La causa principal aun seguía siendo la falta de ancho de banda de memoria y los conflictos entre procesadores y las conexiones con la memoria; estos últimos son indispensables para relacionar direcciones de memoria y los datos que circulan desde y hacia la memoria principal (RAM) [5] [6].

1.4 Hacia el cómputo a petaescala… y a exaescala

Por décadas, al hablar de cómputo de alto rendimiento, *rendimiento* fue sinónimo de *velocidad*. Y esta velocidad se refirió por mucho tiempo a *velocidad del procesador*.

En la década del 80, la idea era lograr más eficiencia en el cómputo con procesadores más rápidos. Este pensamiento cambió cuando el procesamiento en paralelo demostró que también era posible lograrlo comunicando dos o más computadoras, para que juntas colaboren en el procesamiento.

En los años 90, la tendencia fue ir dejando las supercomputadoras vectoriales de procesadores masivamente paralelos, todas ellas propietarias y excesivamente caras,

[3] http://www.cs.sandia.gov/ Laboratorio que investiga en áreas tecnológicas basadas en la ciencia, para la seguridad nacional de EEUU.

migrando a redes de computadoras de costo mucho menor debido a que podían ser construidos con componentes básicos de fácil acceso, tanto de hardware como de redes de interconexión. De esta manera, los clusters de PCs, estaciones de trabajo o SMPs se convirtieron rápidamente en plataformas estándar para el cómputo de alto rendimiento.

En 1992, según el *Mannheim Supercomputer Statistics* [15], llevaba la cuenta de las supercomputadoras a nivel global. En ese año fueron relevadas 530 supercomputadoras existentes en todo el mundo. A partir de 1993, es el proyecto *Top500* el encargado hasta la actualidad del relevamiento de las 500 más potentes supercomputadoras instaladas en el mundo. La clasificación se establece midiendo los mejores tiempos de ejecución al correr programas de prueba de Linpack[4].

Esta evolución fue impulsada por una frenética carrera hacia un ilimitado poder de cómputo, que en la actualidad rompe la barrera de los Pentaflop/s, y estuvo marcada por momentos de logros significativos desde la década del 70. En 1971, la predecesora de las computadoras vectoriales, CDC 7600 de Control Data Corporation, rompe la barrera del Megaflop/s. En 1986, Cray 2 supera la barrera del Gigaflop/s y en 1997, ASCI Red de Intel rompe la barrera del Teraflop/s y se posiciona en el puesto Nro.1 del TOP500 de 1999. Para entonces, los microprocesadores de alto rendimiento, las redes de alta velocidad y las herramientas disponibles para el desarrollo con cómputo distribuido impulsaron la construcción de *clusters* de computadoras. Estos nuevos sistemas de cómputo abrieron camino a una nueva área del cómputo paralelo gracias a la fácil disponibilidad de sus componentes básicos, al costo de una fracción de los valores de las supercomputadoras paralelas de ese momento.

El impacto de este desarrollo se hace evidente en el mercado de los fabricantes, a lo largo de los 15 años que van de 1993 a 2008. En ese tiempo, Cray pasó de ser el diseñador líder del mercado del cómputo de alto rendimiento, a quedar relegado a un sector de dicho mercado formado por laboratorios de investigación gubernamentales y algunos clientes en ámbitos académicos. Por otro lado, IBM comienza a dominar el mercado desde 1999, creciendo año a año, acompañado por Oracle al principio y por Hewlett-Packard desde 2001. Cray reaparece en el mercado y ocupa un tercer lugar a

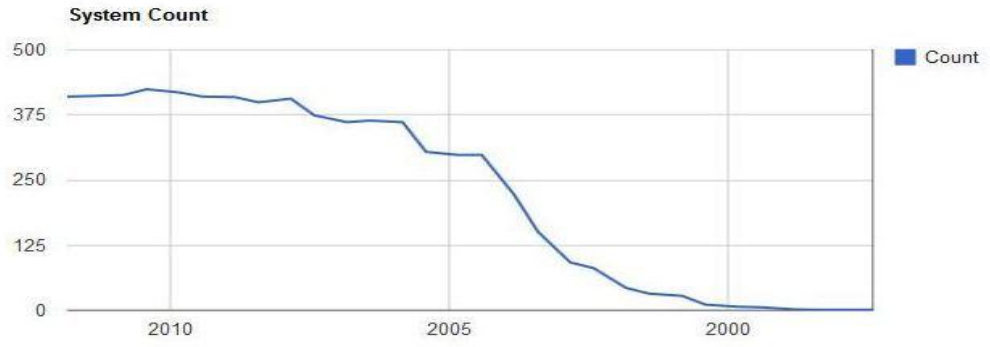

Figura 1: Cantidad de computadoras basadas en **clusters** registradas por "TOP500" del 2000 al 2011

[4] http://top500.org/project/linpack

partir de 2010 con cuatro supercomputadoras dentro de las *TOP10*, dejando atrás las computadoras puramente vectoriales por *supercomputadoras híbridas* [3].

En el reporte *TOP500* de Noviembre de 2004, entre las diez supercomputadoras más rápidas figuraban cinco basadas en *clusters*. La más rápida de las construidas con esta arquitectura era MareNostrum del Centro de Supercómputo de Barcelona, una IBM Blade Center JS20 con una conexión Myrinet que alcanzaba los 20.53 Tflop/s[5]. De la misma fuente, se muestra en la Figura 1, el vertiginoso aumento de supercomputadoras basadas en *clusters* desde el comienzo de su desarrollo hasta la actualidad.

Más potencia de cómputo a menor costo: *clusters*

Un *cluster* es un tipo de sistema de cómputo paralelo distribuido, el cual consiste de un conjunto de computadoras interconectadas y procesando juntas como si fuesen una sola, ofreciendo un recurso de cómputo integrado. Cada computadora o nodo que integra el *cluster* puede ser un sistema simple o multiprocesador. Estos sistemas dominan el cómputo de alto rendimiento para el año 2000, gracias al uso de tecnología de bajo costo y como resultado de unir recursos de hardware de fácil acceso. A este desarrollo se suman los avances logrados en modelos de programación por pasaje de mensajes para cómputo en sistemas débilmente acoplados, como lo eran PVM (Parallel Virtual Machine) y MPI (Message Passing Interface). Décadas de investigación permitieron contar con este tipo de bibliotecas de comunicación a bajo nivel que garantizan portabilidad, arquitectura y transparencia a nivel de red. Actualmente se utilizan las distribuciones de MPI y su interface de desarrollo para C/C++ y Fortran, como también para programación orientada a objetos. MPICH2 es la última distribución para cómputo de alto rendimiento[6].

Según Silva y Buyya [16], "*Un cluster es una clase de sistema de procesamiento paralelo o distribuido, el cual consiste en una colección de computadoras independientes – stand-alone – interconectadas de manera que trabajen todas en conjunto, como si fuesen un solo recurso de cómputo integrado*."

Clusters Beowulf

En 1993 la NASA emprende un proyecto llamado Beowulf Project[7], para conseguir una estación de trabajo que brinde un Gflops de potencia de cómputo a U$S 50.000. Para lograrlo se propusieron utilizar hardware de bajo costo que se encontraba en el mercado comercial. El bajo costo se logró utilizando *commodities* o bienes informáticos de consumo masivo, dando lugar a una nueva arquitectura para cómputo de alto rendimiento.

Los clusters Beowulf, pasaron a ser *commodities clusters* del estilo "ármelo Ud. mismo", ya que utilizando recursos de hardware de propósito general y el modelo de programación para cómputo distribuido de pasaje de mensajes, era posible tener un sistema de altas prestaciones para el cómputo en paralelo y una flexibilidad para su configuración superior a los sistemas MPP. Por lo general,

[5] http://top500.org/lists/2004/11 – Accedido en febrero de 2012

[6] http://www.mcs.anl.gov/research/projects/mpich2/ - Accedido en marzo de 2012

[7] El nombre proviene de una conocida fábula llamada "La sopa de piedras", que trata sobre la cooperación frente a la escasez.

Linux pasó a ser el SO generalmente utilizado en los nodos; debido a esto se los llamó *clusters Linux*. Thomas Sterling escribió el libro "How to Build a Beowulf" en 2001, el cual hacía posible crear su propio cluster Linux [17].

Los avances tecnológicos de los microprocesadores y las redes de área local permitieron lograr en pocos años decenas, y cientos, de Gflops en el rendimiento de los clusters. Por otro lado, el hardware presente en instituciones de investigación, oficinas y aulas de informática, podía utilizarse para crear clusters de bajo costo, logrando una potencia de cómputo similar a las supercomputadoras. Los siguientes artículos brindan un amplio panorama sobre el cómputo en clusters y su evolución [18] [19] [20].

Para el año 2000, estaban en funcionamiento sistemas de cómputo tipo Beowulf con varios miles de nodos y se contaba con varios miles de estos sistemas.

La tendencia en ese momento, era migrar de las plataformas de supercómputo especializado, como Cray/SGI T3E, a los sistemas de propósito general y de mucho menor costo que ofrecían los clusters de PCs o estaciones de trabajo, de simple o multi-procesadores. Baker y Buyya hacen una detallada exposición sobre el cómputo sobre clusters y su evolución en [21]. Los sistemas basados en *clusters* pasaron rápidamente a dominar el mercado del cómputo de alto rendimiento durante la década pasada.

A pesar de todos estos beneficios, como anunciaron Bell y Gray, en 2002, surgió un "*lado oscuro*" del cómputo sobre *clusters*. El inconveniente que se presentó con el desarrollo de aplicaciones paralelas para *clusters*, homogéneos o heterogéneos, fue el pobre rendimiento logrado cuando las aplicaciones requerían gran cantidad de memoria compartida [22].

El crecimiento exponencial del hardware de los sistemas de cómputo de alto rendimiento por un lado, y las demandas impuestas por el desarrollo de aplicaciones científicas e ingenieriles por otro, llevan hoy en día a plantear exigencias y desafíos con el fin de alcanzar el cómputo a *exa-escala*[8], como se muestra a continuación.

1.5 La era del cómputo heterogéneo

Thomas Sterling anunciaba en 2001 [19], "*Sin lugar a dudas, el futuro de muchos de los campos de vanguardia que determinan la evolución de la civilización en este nuevo siglo, será determinado por el grado en que sus contribuciones exploten las posibilidades del cómputo en escala de trans-Teraflops (quizás Petaflops).*"

¿Por qué ingenieros y científicos de las Ciencias de la Computación invierten todo su esfuerzo en llevar los límites del cómputo a escala peta, exa y continuar por más?

El siglo pasado la comunidad científica aceptó, de manera explícita o implícita, que la computación se convirtió en el tercer pilar de los medios que llevan a los nuevos descubrimientos científicos, junto a la teoría y la experimentación [1]. El cómputo a *peta-escala* y *exa-escala* refleja el estado del arte en cómputo de alto rendimiento, que aprovecha los recursos de punta para resolver problemas de gran envergadura, por lo general, problemas de alta prioridad global. Alcanzar estos límites y lograr un uso

[8] 1 *exa* = 1000 *peta* = 1000000 *tera*

eficiente de estos recursos de cómputo, abrirá caminos para resolver problemas globales como prevención de enfermedades, predicción de desastres naturales, formación del universo, evolución de la humanidad y muchos otros problemas en constante estudio, pero que requieren cómputo a muy alta escala [23] .

En junio de 2008, el mercado del cómputo de alto rendimiento pasó a la era de la *peta-escala* con el sistema que encabezaba la lista TOP500, Roadrunner, un cluster BladeCenter QS22/LS21 construido por IBM para U.S. Department of Energy's Los Alamos National Laboratory, alcanzando un rendimiento de 1,02 petaflop/s.

Luego de unos años de haber entrado a la era del cómputo a *peta-escala*, el objetivo es entrar en la era de la *exa-escala* antes de finalizar esta década. Ahora, no bastará con una evolución de las tecnologías existentes y de los enfoques utilizados actualmente en el desarrollo del software, para alcanzar esta meta. Existen importantes retos a lograr con computadoras a exa-escala:

- La gestión de grandes volúmenes de datos. Las aplicaciones que corran en estas supercomputadoras generarán ingentes cantidades de datos.
- La reducción del consumo energético. Se trataría de lograr eficiencia energética mediante el uso de procesadores que actualmente están en uso en dispositivos móviles [24].

Desde Noviembre de 2011, en el Centro de Supercómputo de Barcelona, sus investigadores trabajan en el proyecto UE Mont-Blanc[9] para el desarrollo de una supercomputadora híbrida de CPUs y GPUs, la cual soportará la arquitectura de programación paralela CUDA, de NVIDIA.

Esta nueva era del cómputo de alto rendimiento establece un nuevo paradigma llamado la era del **Cómputo Heterogéneo**.

En los últimos años el "cómputo heterogéneo" ganó mercado y se convirtió en la tecnología de punta para el cómputo de alto rendimiento. Según estudios de IDC para NVIDIA, en 2008 casi el 10% de los sitios de HPC estaban utilizando arquitecturas aceleradoras del cómputo, porcentaje que llega al 28% en 2010, y casi todo este hardware acelerador eran unidades de multiprocesadores gráficos conocidos como GPUs. Pareciera que las GPUs traen nuevamente el modelo del cómputo vectorial al alcance de los programadores, pero les impone el desafío de conocer a fondo un escenario de desarrollo de altas prestaciones totalmente nuevo [25].

Desde que existen los desarrollos informáticos fue necesario unir tres ingredientes para lograr resultados de calidad: computadoras, software y la pericia correcta para utilizarlos. Con procesadores que proveen tanta capacidad de cómputo, más que nunca, se impone un uso inteligente de los tres componentes.

Estas tecnologías podrían resumirse en dos [26]:

- Tecnología *multicore*. Especialmente adecuado para cargas de cómputo intenso y cuando el cómputo puede dividirse en paralelo, en tareas individuales con una funcionalidad específica. La granularidad del procesamiento puede ser tan pequeña como el nivel de carga de los procesadores individuales de la arquitectura utilizada.

[9] http://www.montblanc-project.eu/

- Tecnología *many-core*. Es la provista actualmente por el cómputo en GPUs y es adecuada para aplicaciones que manejan grandes volúmenes de datos que pueden ser divididos en conjuntos y cada dato puede ser procesado en paralelo. Este paralelismo de datos es especialmente adecuado para procesarlos en GPUs con cientos de procesadores en un chip.

Se espera que con el cómputo heterogéneo se pueda alcanzar la Exa-Escala en esta década y, seguramente, el cómputo en GPUs jugará un rol importante.

El aumento en el consumo de energía puede ser un factor que frene este objetivo. Es por eso que hay grandes esperanzas en el uso de GPUs para optimizar la eficiencia de energía necesaria para tan grandes potencias de cómputo. Por ejemplo, haciendo que el código de las aplicaciones envíen a procesar aquellas partes con fuerte paralelismo de datos a las GPUs.

Capítulo 2

Cómputo en GPUs

En los últimos años, el cómputo en GPUs - Graphics Processing Units - ha sido capaz de proveer una arquitectura de cómputo paralelo cuyo rendimiento fue en constante aumento, especialmente con aquellas aplicaciones escritas para tomar ventaja del procesamiento sobre una arquitectura multiprocesador. La idea de usar GPUs para correr aplicaciones de propósito general comenzó en 2003, aunque esto fue reservado sólo para desarrolladores especialistas en visualización y representación gráfica. Recién fue posible lograrlo a partir de 2006, cuando NVIDIA ofrece su modelo de programación paralela CUDA™ – Compute Unified Device Architecture –, en conjunto con la arquitectura TESLA™. Desde entonces, las GPUs incrementaron continuamente su capacidad de cómputo y facilidad de programación, logrando gran popularidad entre la comunidad de investigadores [27].

Owens et al., en su artículo [28] explican por qué las GPUs se desarrollaron tan rápido, superando a las CPUs. Una de las razones es contar con una tecnología de fácil alcance y barata, que puede ser encontrada en la mayoría de las PCs de escritorio o computadoras portátiles. Por otro lado, CUDA está diseñada para extender el código C/C++, y así manejar las funcionalidades para cómputo paralelo propias de las GPUs. De esta forma, CUDA brinda una plataforma unificada de cómputo que aprovecha al máximo la potencia de sus multiprocesadores y acelera aplicaciones de propósito general.

2. 1 Cómputo de propósito general con GPUs

La tecnología CUDA permite utilizar la arquitectura de la placa gráfica para cómputo de propósito general. Esto es posible porque ofrece una abstracción del hardware, compuesto por un conjunto de multiprocesadores MIMD (Multiple Instruction Multiple Data) y cada uno compuesto por un conjunto de procesadores SIMD (Single Instruction Multiple Data). El modelo CUDA sigue el modelo de programación SPMD (Single Program Multiple Data) con funciones de procesamiento vectoriales [29]. Los programadores escriben programas seriales, desde los cuáles invocan funciones paralelas llamadas kernels. Estas funciones se ejecutan en paralelo mediante un conjunto de threads, los cuales son mapeados sobre los procesadores de la GPU. Su filosofía de cómputo se resume en utilizar el mismo programa en paralelo sobre muchos elementos de un conjunto de datos de entrada, siendo independiente el procesamiento de cada elemento y sin comunicación entre ellos durante el cómputo.

En la programación de GPUs para cómputo de propósito general, se reconocen las siguientes etapas, comunes a toda implementación que utiliza CUDA [28]:

1. El programador define el dominio del cómputo, organizando los threads dentro de una grilla estructurada.
2. Un programa de cómputo general, o kernel, dirige la ejecución de cada thread.

3. Cada thread realiza un conjunto de operaciones matemáticas, pudiendo leer y escribir en una memoria compartida por todos los threads en ejecución. El programa puede hacer la lectura y escritura utilizando el mismo buffer, pudiendo así ahorrar memoria.
4. Los resultados que un kernel deja en la memoria global compartida pueden ser utilizados como datos de entradas para la ejecución futura de otro kernel.

No todos los problemas son adecuados para ser resueltos con GPUs. Los que se ajustan a esta arquitectura son los problemas paralelizables mediante el paradigma de paralelismo de datos, o sea la misma operación puede ejecutarse en paralelo sobre un conjunto de datos de la entrada, ajustándose a una arquitectura SIMD y NUMA. Las GPUs que surgen desde la serie G80 en adelante ofrecen la suficiente abstracción del hardware gráfico, permitiendo así correr aplicaciones que no tienen nada en común con las aplicaciones gráficas, siendo programadas utilizando las APIs gráficas.

2.1.1 Administración de threads

La arquitectura de una placa gráfica TESLA está construida mediante un arreglo escalable de multiprocesadores, conocidos como SM (Stream Multiprocessors). La mínima cantidad de SM que pueden encontrarse es dos y cada uno está integrado por ocho SP (Stream Processors) escalares. Cada SM contiene también dos SFU o unidades de funciones especiales, una MTIU o unidad de instrucción multithread y la memoria compartida on-chip [30].

1 SM contiene { 8 SP + 2 SFU + 1 MTIU + SharedMem }

Cada SM crea, administra y ejecuta hasta 768 threads concurrentes en hardware, con cero sobrecarga del cómputo. Para manejar tal cantidad de threads corriendo a la vez TESLA utiliza la arquitectura llamada SIMT (single-instruction, multiple-thread). El SM mapea cada thread sobre un SP escalar y cada uno se ejecuta independientemente.

CUDA define una jerarquía al agrupar los threads: Los threads se agrupan en bloques que contienen igual cantidad de threads. A su vez los bloques se organizan en grids.

Esta organización en bloques y grids es dimensionanada por el programador al comenzar la ejecución del kernel.

En esta abstracción de la arquitectura física, provista por CUDA, cada bloque puede verse como organizado en estructuras tipo arreglo, de una a tres dimensiones. Lo mismo sucede con los grids, pero con una o dos dimensiones. Esta organización se visualiza gráficamente, para 2D, en la Figura 2.

CUDA soporta hasta 512 threads por bloque, los cuales pueden ser ejecutados en cualquier orden y en diferentes procesadores. Los threads de un bloque se ejecutan en el mismo procesador y **cooperan entre sí durante el cómputo** mediante:

- El acceso a un área de memoria compartida.
- El uso de operaciones atómicas.
- El uso de barreras de sincronización.

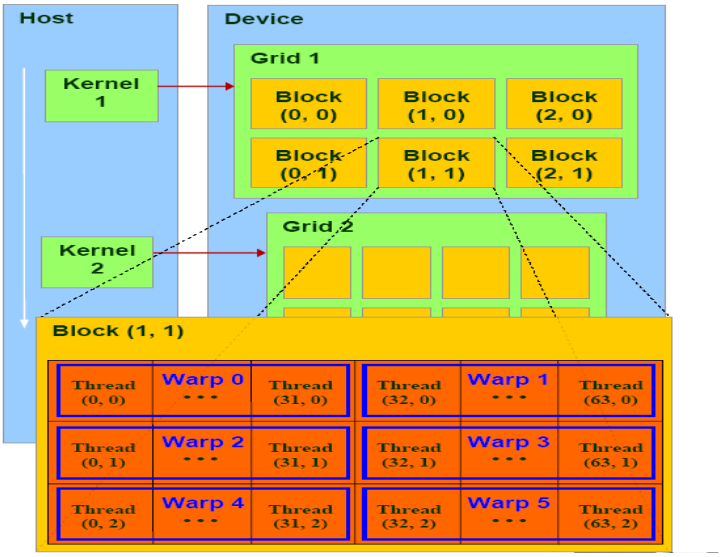

Figura 2: Organización jerárquica de threads. Obtenido de CUDA Programming Guide

CUDA permite manejar esta jerarquía de threads en ejecución desde el código del kernel. Como todos los threads en un grid están ejecutando la misma función o kernel, es necesario identificarlos para asignarles el dato apropiado a cada uno. Cada bloque tiene un identificador, *blockIdx* – es una variable de dos componentes –, cada thread tiene su identificador, *threadIdx* – es una variable de tres componentes – y lo mismo sucede con los grids, su identicador es *grimDim*. Todas son variables (*built-in*) que ofrece CUDA como extensiones del ANSI C. La Figura 3 muestra un kernel en ejecución dentro del bloque i-ésimo, provocando que cada thread lea su dato, como también que genere y almacene su resultado, en la memoria compartida de dicho bloque.

Una función de kernel se invoca indicando la configuración de su ejecución. Un ejemplo sería:

```
__global__ void KernelFuncion(…);  //  declara la función KernelFuncion
dim3 DimGrid(100, 50) ;              // se dimensiona un grid de 5000 bloques
dim3 DimBlock(4, 8, 8) ;             //  se dimensiona cada bloque con 256 threads
Size_t ShareMemBytes = 64 ;          // 64 bytes de memoria compartidaKernelFuncion<<<DimGrid, DimBlock,SharMemBytes>>>(…)  //se invoca el kernel
        …
```

Las llamadas a los kernels son asíncronas, por lo tanto requerirán sincronización explícita, si fuese necesario.

¿Qué es un Warp? Un warp es la cantidad de threads que pueden efectivamente correr concurrentemente en una unidad MP. Como se indicó,

cada MP contiene 8 procesadores SP, que procesan mediante un pipeline y la instrucción más rápida toma 4 ciclos. Cada SP puede tener 4 instrucciones en su pipeline, con un total de (8 x 4) 32 instrucciones ejecutadas concurrentemente dentro de un MP. Dentro de un warp es posible ejecutar 32 threads por vez. Un warp sólo puede subdividirse en mitades de 16 threads.

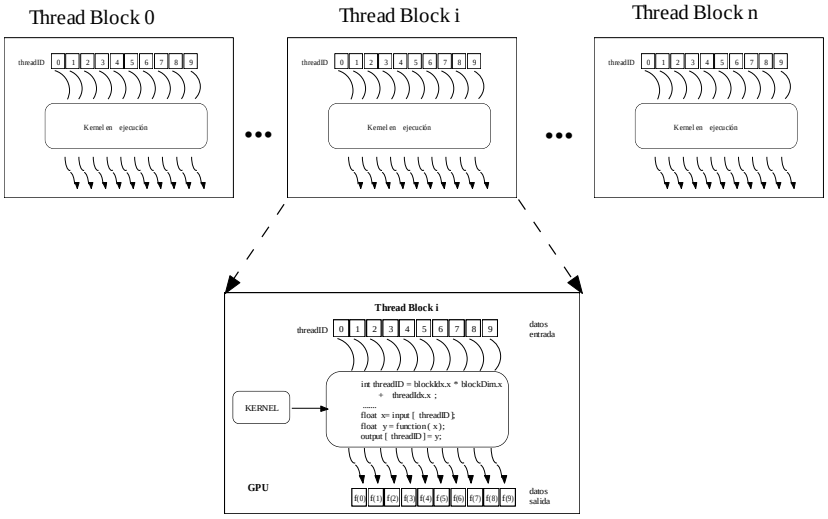

Figura 3: Organización de threads en CUDA

El rendimiento logrado en la GPU, depende directamente de la homogeneidad de los threads en un warp. Si todos ejecutan la misma instrucción entonces todos los SPs en un MP ejecutan la misma instrucción en paralelo. Pero, si uno o más threads del warp ejecutan una instrucción diferente al resto, el warp debe ser particionado en grupos de 16 threads (o sea la mitad de un warp) y cada grupo se ejecutará uno después del otro por cada thread que diverge, serializando el cómputo y degradando notablemente el rendimiento del sistema. Esta situación se da si el código del kernel tiene instrucciones de bifurcación del tipo *if-then-else*, o estructuras cíclicas en las cuales algunos threads realizan más ciclos que otros. Esta situación se conoce como *divergencia* de la ejecución de los threads. Se encuentra más información en [31] [7] [32].

2.1.2 Organización y administración de la memoria

Para soportar una arquitectura heterogénea CPU + GPU, cada una con su propio sistema de memoria, CUDA permite copiar datos entre la memoria de ambas unidades de cómputo.

Los threads pueden acceder a datos en los diferentes niveles de memoria de la GPU durante su ejecución. Cada thread tiene acceso a los siguientes niveles de memoria:

1. **Memoria local**: utilizada por CUDA cuando las variables privadas no entran en los registros.
2. **Memoria compartida**: visible a todos los threads del bloque y se mantienen los datos durante la vida del bloque, o sea mientras se ejecuta el kernel. Esta es una pequeña memoria on-chip de muy baja latencia.

3. **Memoria global**: accesible por los threads de todos los bloques activos y su permanencia es durante la ejecución de la aplicación. Reside en la DRAM de la tarjeta gráfica.

Estos tres niveles de memoria están en diferentes espacios físicos de la GPU. Un adecuado uso de la memoria compartida deriva en una significativa mejora del rendimiento de la aplicación, en comparación con hacer uso tan solo de la memoria global. Esta organización puede verse en la Figura 4.

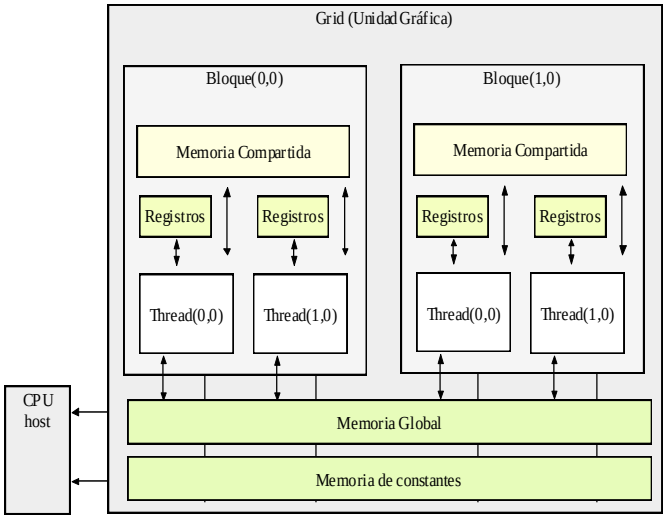

Figura 4: Jerarquía de memoria que administra CUDA

2.2 Uso de GPUs para mejorar el rendimiento de un algoritmo de mallado.

Las GPUs están compuestas por cientos de unidades de procesamiento en paralelo y, al correr aplicaciones paralelas que aprovechan el paralelismo de datos, ofrecen un rendimiento en el cómputo superior al de los procesadores multinúcleo. Son una solución de alto rendimiento, no solo para el procesamiento gráfico para el que fueron diseñadas, sino también para el procesamiento no gráfico conocido como GPGPU (General Purpose – GPU).

La aplicación seleccionada, es parte de un trabajo cuyo objetivo es paralelizar y acelerar un método de mallado en elementos finitos en 2D y 3D con aplicaciones a problemas elípticos.

A continuación se expone las características del algoritmo de mallado seleccionado, y su paralelización utilizando GPU+CPU como arquitectura de cómputo paralelo. Luego se describen las mejoras obtenidas al disminuir significativamente su tiempo de ejecución.

2.2.1 Métodos numéricos de mallado

La generación de un mallado es un paso clave en representación gráfica y uso del método de elementos finitos. El método de elementos finitos es un método numérico que se aplica a complejos problemas ingenieriles y científicos. Por su naturaleza, este tipo de problemas presenta un alto paralelismo de datos y gran cantidad de operaciones matemáticas, realizadas por igual sobre un gran volumen de datos que crece cuando los dominios de mallado se hacen más complejos.

El paso previo a la implementación de estos métodos es la discretización del dominio del problema. Esta es una etapa muy importante de la solución, pues determinará los recursos de memoria necesarios, la velocidad de generación de la solución y la exactitud de los resultados obtenidos. El dominio se divide en pequeños elementos, cuya geometría se define según el número de dimensiones sobre las que se trabaja y las características del problema. Esta etapa se llama de *generación del mallado*.

Los elementos del mallado varían según la geometría del problema:

- Si el problema es unidimensional, serán segmentos de líneas los elementos que subdividan el dominio.
- Si es bidimensional, el dominio se discretizará con triángulos y, en algunos casos, con cuadriláteros.
- Si es tridimensional, se utilizarán tetraedros, o también pentaedros y hexaedros.

La investigación en métodos de generación de mallados se encuentra con la dificultad de combinar un bajo costo computacional con un mallado de alta calidad, junto a la capacidad de discretizar dominios complejos.

Los métodos se dividen en dos grandes clases:

- *Mallas estructuradas*: son las discretizaciones cuya conectividad sigue un patrón reticular. Exigen fuertes condiciones sobre el dominio y suelen ser de baja calidad.
- *Mallas no estructuradas*: no presentan un patrón de conexión predeterminado. Por ser una discretización arbitraria los hace adaptables a cualquier tipo de dominio. Esto las hace de aplicación más general, pero con costo computacional más elevado. Es el caso del método de *Delaunay*, utilizado en este trabajo.

2.2.2 El Algoritmo de Persson-Strang: *Distmesh*

Per-Olof Persson y Gilbert Strang un código simple y de acceso público para Matlab, llamado *Distmesh*[10] Los autores ofrecen una técnica iterativa basada en una analogía física para disponer los puntos y crear los tetraedros o triángulos de la forma más equilibrada posible. El resultado obtenido es un mallado de alta calidad.

Muchos problemas son definidos usando dominios de formas irregulares. En estos casos, las mallas no estructuradas son mucho más flexibles y adaptables a la física del problema que las estructuradas. El problema que surge es la complejidad, y casi inaccesibilidad, de los códigos de los programas de mallado. La elección del algoritmo

[10] http://persson.berkeley.edu/distmesh/

de Persson y Strang se hace debido a su simplicidad y precisión. A continuación se brinda una breve descripción de su funcionamiento.

Inicialmente se define el objeto a mallar posicionando los nodos dentro del dominio mediante una distribución equidistante. Esta acción funciona muy bien en geometrías sencillas. Se define una función *d*, distancia signada para medir la distancia de cada punto al borde del dominio, siendo negativa dentro y positiva fuera de él. Junto con esta función se utiliza una función *h*, en 2D o 3D, para establecer la resolución de la malla en cada región del dominio, lo cual permite una mejor adaptabilidad a geometrías complicadas, utilizando elementos más pequeños cuando sea necesario.

Los puntos definen la geometría y un mallado creado por el algoritmo de triangulación *Delaunay* determina la topología. El método de *Delaunay* utiliza triángulos en 2D o tetraedros en 3D, para crear la cápsula convexa del dominio de puntos.

El algoritmo consiste en una técnica iterativa que considera los puntos, o nodos, de una estructura de resortes. Las aristas generadas por *Delaunay*, son los lados de los triángulos o tetraedros y se corresponden con barras o resortes que ejercen fuerzas sobre los nodos, como si todos los resortes estuvieran comprimidos. En cada iteración todo el "sistema de resortes" evoluciona buscando una posición de equilibrio. En cada iteración se obtiene la nueva posición de los puntos mediante el cálculo de las ecuaciones de equilibrio para las fuerzas, calculadas con el método de Euler. Si algún punto se desplaza fuera del dominio, es reproyectado hacia la superficie. Cuando los puntos se separan más de lo tolerado, una nueva triangulación de Delaunay ajusta la topología. Para una explicación más detallada sobre el método original de Persson se puede consultar en [33].

Distmesh es un algoritmo muy eficiente en Matlab, el cual fue completamente vectorizado para evitar las estructuras cíclicas que degradan el rendimiento. Sus autores utilizan una matriz dispersa para calcular el movimiento de los nodos, cuyas dimensiones está determinada por la cantidad de nodos (n) y cantidad de barras (m) de la malla. La versión Matlab de *Distmesh* invoca la función *Delaunay*, también de Matlab, para determinar la topología de la malla. En la versión secuencial de este trabajo, se seleccionó el programa de código abierto, escrito en C por Geoff Leach, que reproduce exactamente la función *Delaunay* de Matlab. Su autor utilizó el algoritmo de Guibas-Stolfi y lo mejoró logrando un speedup de 4-5, con una complejidad O(n log(n)). Se puede encontrar una detallada explicación del programa y del método utilizado en [34]

2.2.3 Características de la versión secuencial de *Distmesh*

Inicialmente fue necesario crear una versión C++ del algoritmo *Distmesh*, para esto se tomaron en cuenta algunas características importantes de su versión original, las cuales se detallan a continuación:

- Una función distancia *d* determina la geometría del dominio mediante una función distancia signada, la cual es negativa dentro de la región. Los autores definen ésta como una decisión esencial. La función es calculada en todo el conjunto de puntos y posteriormente, se utiliza al calcular la distancia de los nodos al punto más cercano del borde para su reproyección.
- Esta implementación usa una función lineal para definir la fuerza repulsiva y no permite fuerzas atractivas. Su expresión es:

$$f(l, l_0) = \begin{cases} k(l - l_0) & if\ \ l < l_0 \\ 0 & if\ \ l \geq l_0 \end{cases}$$

- La fuerza resultante *Ftot* es la sumatoria de los vectores fuerza que concurren a cada nodo. Cada barra ejerce una fuerza $f(l, l_0)$ dependiendo de su longitud actual l y su longitud de relajación l_0.
- La longitud de relajación l_0 es constante para mallados uniformes y es necesario que $f = 0$ para $l = l_0$. Los creadores de *Distmesh* eligen l_0 apenas más grande que una longitud estimada… El cálculo depende de la suma total de la longitud de cada barra.
- El paso de tiempo del método de Euler es el parámetro Δt. El parámetro *geps* es usado para calcular la tolerancia al evaluar la geometría y la cantidad de movimiento total en cada iteración como también para decidir si un punto está fuera del dominio.
- Todos los puntos que salen del dominio luego de actualizar sus posiciones son proyectados al contorno. El gradiente numérico de *d* da la dirección de movimiento del punto.

A continuación se exponen las partes que integran la versión secuencial del algoritmo, pensando en la futura versión paralela para GPUs.

Entrada de datos: En un primer paso, se crea una distribución uniforme de puntos dentro de una geometría que es dato del problema. Estos son los nodos del mallado que son distribuidos de manera regular a una distancia h_0 de sus vecinos cercanos.

Triangulación: Un paso significativo es la triangulación de *Delaunay*. En cada iteración se compara la posición actual de los puntos con la posición dada por la última triangulación. Cuando el máximo desplazamiento de los puntos es mayor que cierta tolerancia, se invoca nuevamente una triangulación de *Delaunay* para redefinir la posición de los puntos y garantizar las propiedades de *Delaunay*. Se esquematiza en los pasos 2 a 10 de la Figura 1.

Actualización: Las longitudes de las barras son utilizadas para calcular las componentes de las fuerzas. La fuerza resultante en los nodos es la sumatoria de fuerzas ejercidas por las barras que concurren a cada uno; el cálculo de este valor se usa para actualizar la posición de los nodos. Se esquematiza en los pasos 11 a 15 de la Figura 1.

Proyección: El proceso de actualización puede ubicar algunos puntos fuera de la geometría. Al detectarse son proyectados al contorno, en respuesta a una fuerza normal. Esto se realiza usando el gradiente numérico de la función distancia para calcular la dirección del movimiento hacia el punto más cercano del contorno. Se esquematiza en los pasos 16 a 18 de la Figura 5.

El paso de actualización mueve los puntos a su siguiente posición, usando los valores escalares de la fuerza ejercida por cada barra. Esta es la operación con más carga computacional y utiliza una gran matriz de movimientos *M*. Cada elemento de la matriz *M* va acumulando en cada iteración los movimientos de los puntos.

Es posible asegurar que esta versión serial es fácil de entender, brinda una versión optimizada y facilita escribir la versión paralela.

Persson_Mesh_Secuencial(Pact, Pold, Bars, Tri)

{Proceso iterativo del método de Persson – Versión C ++ secuencial}
{El método recibe la ubicación de los puntos y los triángulos dados por Delaunay}
{y la ubicación inicial de los puntos del dominio}

```
1   While ( MovePoints( Pact, Pold)  >  eps )
2       If  ( MovePoints(Pact, Pold) < tolerancia)
3           {Si el movimiento es grande se retriangula con Delaunay}
4           Pold = Pact
5           Tri = Delaunay(Pact)
6           for  cada triangulo t / t ε Tri
7           {solo mantienen los triángulos interiores al dominio}
8               c = Centroide( t )
9               if  (c esta fuera del dominio D)
10                  EliminaTriangulo ( t )
11      {Se actualiza el valor de las barras y la sumatoria de fuerzas que
        convergen en cada punto}
12      Bars = CrearBarras( Tri )
13      Ftot = CalcularFuerzas( Pact, Bars )
14      {Se actualiza la posición de cada punto}
15      Pact = Pact + deltat * Ftot
16      for   cada punto p de Pact y ubicado fuera del dominio D
17          move =  Gradiente ( Pact, D)
18          p = p + move
```

Figura 5: Algoritmo secuencial de Persson

2.3. Distmesh Paralelo: entorno GPU + CPU

La mayoría de las operaciones realizadas por la versión secuencial del algoritmo de Persson se pueden mapear fácilmente sobre los multiprocesadores de las GPUs. El paralelismo de datos propio del algoritmo Distmesh es la propiedad más importante que facilita este proceso. Las operaciones que se realizan sobre las estructuras de datos son manejadas por cuatro kernels, los cuales reparten los datos a bloques de threads que realizan en simultáneo las operaciones del kernel activo.

La Figura 6 presenta un esquema de alto nivel de la versión paralela para GPUs, del algoritmo de Persson. La GPU funciona como un co-procesador paralelo de la CPU, trabajando juntos de manera colaborativa. La fase inicial se realiza en la CPU generando una primera triangulación del dominio usando Delaunay, de esta forma se crean las estructuras de datos que contienen las coordenadas de los puntos, los puntos extremos de cada barra y los puntos que definen cada triángulo.

Una vez que las estructuras de datos se transfieren desde la memoria RAM a la memoria global de la GPU, la CPU lanza la ejecución de los kernels y comienza en la GPU el proceso que genera el mallado: *Persson_Mesh_Paralelo.*

Durante el proceso que se da dentro de la unidad gráfica, no hay transferencia de datos entre la memoria de la CPU (RAM) y la GPU, sólo al concluir el trabajo en la GPU se inicia la transferencia de datos final. La transmisión de datos incluye, tan solo, la posición final de los datos y los puntos de los triángulos, estos últimos para fines de graficación de los resultados. Los datos acerca de los puntos, las barras, los triángulos y los movimientos permanecen en la memoria de la GPU durante el trabajo de los kernels.

La sobrecarga del cómputo debido al pasaje de datos entre datos entre RAM y GPU es muy poco significativa en esta aplicación ya que sólo se realiza al inicio y al final del

proceso. A continuación se detallan las etapas diseñadas para ejecutar la versión paralela de la aplicación.

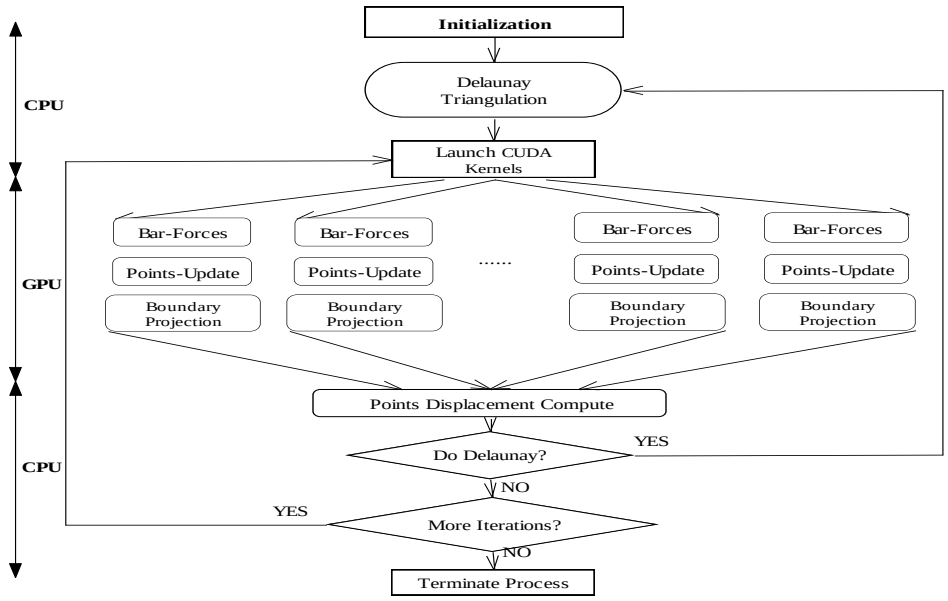

Figura 6: Algoritmo Persson_Mesh_Paralelo para CPU-GPU

2.3.1 Kernels que se ejecutan en la GPU

El primer paso para obtener una versión paralelizable en CUDA del algoritmo es identificar bloques del algoritmo secuencial que realizan el mismo cómputo sobre un conjunto de datos, y que se puedan pensar como funciones independientes de tareas concurrentes.

Los nodos del mallado permanecen fijos mientras se realiza el cómputo de las fuerzas y de la cantidad de movimiento de cada uno. Al tener estos datos se procede a mover los puntos de manera independiente uno de otro, permitiendo paralelizar el cómputo y distribuir el procesamiento entre los threads. En la Figura 7 se puede visualizar gráficamente la relación entre las estructuras de datos y los kernels. A continuación se detalla el comportamiento de cada uno.

- **Longitud de Barras** (*Kernel_Length*): Mapea cada barra de la triangulación a un thread, de manera que todos en paralelo calculan su longitud. Los datos están almacenados en el arreglo *Bars* y genera el arreglo *Length_Bar*. Posteriormente es necesario sumar las distancias para calcular un parámetro del método. El resultado es un dato de entrada con uno de salida, lo cual es ideal en este modelo de programación. En el ítem siguiente se explica cómo se maneja esta situación, no muy conveniente para el cómputo.
- **Fuerzas Escalares** (*Kernel_Forces*): Se lanza un thread por elemento del arreglo *Length_Bar* residente en memoria global. Nuevamente, cada thread realiza sus operaciones sobre cada dato de entrada y calcula la fuerza escalar que produce cada barra a sus extremos. El resultado se almacena en un vector Move, un elemento por cada barra.

- **Movimiento de los puntos** (*Kernel_Movements*): Cada thread lee la fuerza de cada barra y su longitud. Con estos datos calcula el movimiento en cada coordenada, de los puntos del mallado, afectado por esa barra en particular. Cada thread acumula su resultado en el arreglo puntos (Points) en el elemento correspondiente a los puntos de sus extremos. Todos los threads acceden a esa estructura de datos durante el cómputo. Esto fue posible utilizando funciones especiales de CUDA que son explicadas en el próximo ítem.

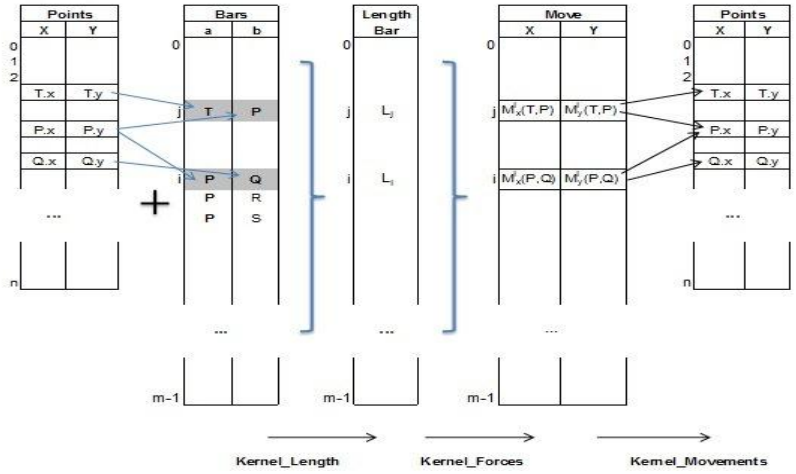

Figura 7: Estructuras de Datos creadas por los Kernels

2.3.2 Optimización del cómputo en la GPU

El próximo paso es escribir el programa completo en CUDA y C, para ello se debe manejar eficientemente la transferencia de datos entre la memoria RAM de la CPU y la memoria global de la GPU.

Latencia de Memoria: Según se dijo anteriormente, la transferencia de datos en esta aplicación se realiza dos veces, antes de lanzar la ejecución de los kernels y cuando éstos finalizan. En este caso no se produce el famoso cuello de botella de la transferencia de datos, situación que se presenta cuando hay frecuentes transferencias entre ambas unidades de cómputo y entre una y otra ejecución de los kernels [35]. En la Figura 8 se visualiza gráficamente que la transferencia de datos representa el 5.7% del tiempo total de ejecución de todo el proceso. Dicho gráfico fue obtenido usando la herramienta *Compute Visual Profiler 4.0* distribuida por NVIDIA [36]. La ejecución de los kernels anteriores están sujetos a la velocidad de distribución de las tareas de la GPU, o sea el proceso en las GPUs está limitado por el cómputo y no por la transferencia de datos.

Conflictos de acceso a memoria: Para mejorar el rendimiento, se optimizó el acceso a memoria accediendo memoria compartida desde los kernels. El uso de esta memoria local a los bloques, evita las cargas de datos redundantes, aunque es necesario evitar que los threads de un mismo warp accedan a bancos de memoria compartidos distintos [31].

Para optimizar la memoria compartida se debe tener en cuenta que si todos los threads de un medio warp acceden a distintos bancos de memoria o todos acceden a la misma

dirección (broadcast) no hay conflictos y el proceso es muy rápido. Pero, si múltiples threads de un medio warp acceden al mismo banco, el proceso se serializa y se degrada el rendimiento. Si cada banco tiene un ancho de banda de 32 bits, sucesivas palabras de 32 bits son asignadas a consecutivos bancos. Un problema de rendimiento surge si los elementos que se cargan no son de 32 bits. Otro factor a tener en cuenta con el uso de memoria es asegurar lecturas alineadas (*coalescing*) y evitar múltiples accesos simultáneos a una dirección de memoria [37].

Evitar estos conflictos requiere múltiples intentos al estructurar los datos y lograr una organización que los minimice. En esta aplicación fue modificado el arreglo de movimientos de los puntos, el cual inicialmente era una matriz esparcida que provocaba un mal uso de la memoria compartida.

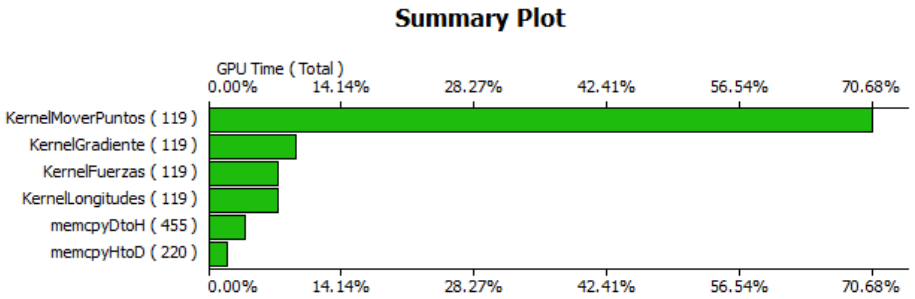

Figura 8: Tiempos de ejecución y de transferencia de datos

Conflictos en el cómputo: El cómputo de los valores de las fuerzas, en las barras de la triangulación, requiere la suma de los elementos de arreglos, cuyos valores son calculados en paralelo por los threads. Surge el problema que la suma no es trivialmente paralelizable. Esto tiene varias soluciones, en este caso se utilizaron funciones atómicas.

Las funciones atómicas como CUDA Atomic Add son el núcleo de las operaciones aritméticas necesarias para la suma de arreglos y para calcular fuerzas y movimientos de los puntos. Aseguran que las lecturas de cada thread se harán sin ninguna interferencia de otros threads. CUDA asegura la sincronización de múltiples threads en diferentes bloques, en el caso que accedan a la misma variable para realizar operaciones de lectura-escritura [38]

A continuación se detallan algunas de las técnicas utilizadas para optimizar la versión paralela del algoritmo, teniendo en cuenta los conflictos que se detallaron anteriormente:

1. **Optimización del Kernel_Length**: En una versión inicial, la longitud de relajación de las barras (se recuerda la analogía con resortes) l_0, era calculada por un solo thread y los otros permanecían ociosos mientras tanto. Para aprovechar el modelo de programación y disminuir el tiempo de ejecución, se copió en memoria compartida el nuevo arreglo de longitudes de la barras y se usó *Atomic_Add* en memoria compartida para sumar los elementos del arreglo.

2. **Optimización del Kernel_Movements:** Inicialmente se utilizó una enorme matriz de datos, siguiendo el método implementado en Matlab para el algoritmo de Persson. En esa matriz se guardaron los valores de los movimientos de los

puntos, para cada coordenada. Debido a los conflictos con los bancos de memoria, esta matriz esparcida fue reemplazada con el arreglo de movimientos Kernel_Movements. El arreglo con las ubicaciones de los puntos se utilizó como acumulador, guardando en éste la sumatoria de movimiento que sobre cada punto ejercen las barras que convergen en él. Se utilizó la función Atomic_Float_Add en este cómputo con los datos en memoria compartida, obteniendo con ambas mejoras una importante mejora en el rendimiento.

2. 4 Resultados Experimentales y Conclusiones

Las experiencias se hicieron en una plataforma integrada por un microprocesador Intel Xeon Dual-core con 4 GB de memoria RAM y 3.2 GHz, conectada a una placa NVIDIA Tesla C2070 con el driver CUDA versión 4.0. La GPU comprende 14 MPs de 32 SPs cada uno, teniendo un total de 448 procesadores SPs. La capacidad de cómputo de CUDA es 2.0.

La Figura 9 compara los tiempos totales de ejecución en CPU y GPU del algoritmo de mallado en función de la cantidad de puntos del mallado. Estos incluyendo los tiempos de transferencia de datos entre ambas unidades. Los valores de Versión 1 corresponden a correr el programa con operaciones atómicas en memoria global y en Versión 2 se utilizan en memoria compartida. La mejora en el segundo caso es notable, confirmando la conveniencia de hacer la mayor cantidad de cálculo posible en la memoria compartida de cada bloque. Los tiempos de Versión 2 fueron los utilizados para el cálculo del Speedup que se muestra en la Tabla I.

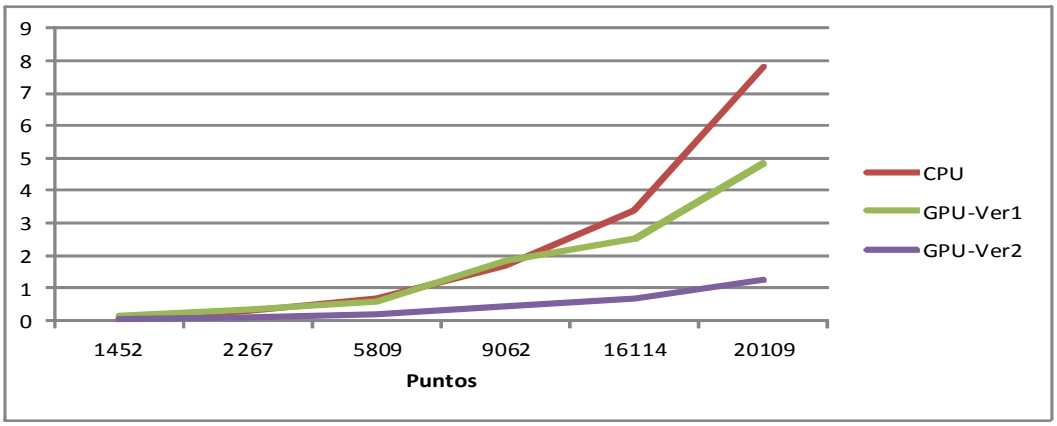

Figura 9: Comparación tiempos de ejecución

Una medida que ayuda a conocer si las versiones secuencial (CPU) y paralela (CPU+GPU) del algoritmo son comparables entre sí, es contar la cantidad de iteraciones realizadas del método principal y contar cuántas veces se computa la función Delaunay en cada caso. Podemos afirmar que la cantidad de trabajo fue la misma y se pueden comparar. La Tabla II muestra la cantidad de veces que se realizó cada método, obteniendo resultados satisfactorios, ya que las diferencias en cada caso son mínimas.

TABLA I

Tiempo de Ejecución (sg.)

Puntos del Mallado	*Tiempo CPU*	*GPU Versión 1*	*GPU Versión 2*	*SpeedUp Completo*
1452	0.035	0.130	0.037	0.95
2267	0.290	0.320	0.112	2.59
5809	0.687	0.580	0.183	3.75
9062	1.689	1.820	0.425	3.97
16114	3.388	2.510	0.666	5.09
20109	7.786	4.860	1.239	6.28

La Tabla II, también muestra el Speedup alcanzado, al considerar sólo los kernels de la versión paralela y el cómputo equivalente a ellos en el método secuencial. Esto significa considerar sólo el tiempo de ejecución de Persson_Mesh_Secuencial mostrado en la Figura 5, que es el núcleo del cómputo. Se logra un speedup de 53X en el momento en que la configuración de threads y bloques en el grid permite el mayor nivel de ocupación la GPU.

TABLA II

Iteraciones Persson y Delaunay

Puntos del Mallado	*CPU*		*GPU*		*Speedup Kernels*
	Delaunay	*Persson*	*Delaunay*	*Persson*	
1261	25	265	23	253	4.77
2827	39	365	30	313	6.81
5025	45	381	35	381	4.94
7851	49	487	38	487	5.09
11313	65	658	44	616	12.08
15395	150	1056	134	983	12.10
20109	231	2083	212	2405	17.35
31409	728	7764	719	6980	53.38

El uso de la herramienta *Compute Visual Profiler* permitió medir el nivel de ocupación de los procesadores de los SMs, dando como resultado 65%. Esto indica que hay tiempo

ocioso de los procesadores durante el cómputo y que no sería posible alcanzar un speed-up mayor que el logrado. Era esperable que el tiempo de ejecución medido exclusivamente en los kernels sobre la GPU, lograra un rendimiento mayor sabiendo que se está procesando con 448 procesadores escalares SP. A continuación se detallan algunas de los motivos que afectaron el rendimiento general de esta aplicación en las GPUs y sobre los que se trabajan actualmente.

a) Una de las razones que provocan el degradamiento del rendimiento es el uso de estructuras de control *if-then-else* en el código de los kernels, como puede verse en el Anexo al final de este trabajo. En esos casos, el cómputo se serializa y los warps deben ejecutar aquellos threads que siguen uno de los caminos primero, y el otro camino al finalizar. Esto se realiza siempre en grupos de, a lo sumo, medio warp (16 threads).
b) Por otro lado, el modo de acceso a memoria determina la explotación de la localidad. El uso de memoria compartida puede evitar conflictos en los bancos de memoria cuando threads contiguos acceden a direcciones tales que (*dir % 16)* coinciden. Si no es posible cambiar el patrón de acceso a memoria para evitarlo es preferible utilizar memoria compartida con aquellos datos que tienen un solo acceso. Se sacrifica un poco la memoria compartida y se evita en parte el problema evitando parte de la serialización. Esta situación se da en el kernel *KernelLongitudes* al declarar en memoria compartida el arreglo *longi*.
c) Los accesos a memoria global (aunque sea para leer los datos que se llevarán a la memoria compartida) se hacen a nivel de medio warp y un acceso a memoria global puede provocar hasta 16 transacciones. Por lo tanto, cada warp puede provocar desde 2 hasta 32 transacciones de acceso a memoria durante su ejecución dependiendo de los segmentos distintos de memoria en que se encuentran los datos. El objetivo que aumenta el rendimiento general es lograr accesos unificados a segmentos de memoria, llamados *coalesced*. El tipo de dato abstracto **persson*, utilizado en el código, contiene estructuras de datos que superan los 16 bytes por elemento. Este tipo de dato permitió construir un código muy fácil de interpretar y modificar, pero debe ser modificado para evitar los accesos a memoria *no-coalesced*.
d) La latencia debida a las barreras de sincronización (*syncthreads()*) puede ocultarse si hay tres o más bloques activos en cada multiprocesador. Durante la ejecución fue posible tener hasta dos bloques activos por vez.

Es destacable, a pesar de los inconvenientes anteriores, lo apropiada que resulta ser la tecnología de las GPUs para acelerar algoritmos como los que calculan un mallado sobre un dominio 2D o 3D. Este trabajo muestra un tipo de aplicación de cómputo de propósito general y de alto paralelismo de datos, que se ve favorecida por la arquitectura paralela de alto rendimiento que proveen la combinación CPU+GPU más el modelo de programación que brinda CUDA.

Aunque se vería ampliamente potenciada si se contara con una arquitectura híbrida, como ser un *cluster de CPUs multicore + la tecnología many-core de la GPU*. En este caso el modelo de programación lo daría *Open MP + Open MPI + CUDA*, y es el paso siguiente de este desarrollo que necesitará trabajar con grandes volúmenes de datos a medida que se necesitan dominios de mallado más extensos o complejos.

2.5 Trabajo futuro

Este trabajo continúa en las siguientes direcciones:

- En primer lugar se trata de optimizar al máximo posible el cómputo en GPUs de la versión 2D ya presentada y utilizar una placa FERMI con el objetivo de medir la escalabilidad del algoritmo.
- Continuar con el estudio de los límites que se imponen en el cómputo con dominios complejos a mallar, los cuales son más costosos en volúmenes de datos.
- Aplicar las mejoras a dominios 3D de formas complejas.
- Finalizar con una versión para arquitecturas híbridas, en particular para la arquitectura BLADE que está disponible en el III-LIDI de la Facultad de Informática de la UNLP.

Bibliografía

[1] NRC-HECC, «The potential Impact of high-end capability computing on four illustrative fields of science and engineering,» National Research Council of the National Academies Press, Washington D.C., 2008.

[2] R. Stevens, T. Zacharia y H. Simon, «Modeling and simulation at the exascale for energy and the environment,» *Report on the Advanced Scientific Computing Research Town Hall Meetings*, p. 174, 2008.

[3] P. M. Kogge, «ExaScale Comuting Study: Technology Challenges in Achieving Exascale Systems,» DARPA IPTO, Departmeno of Defense, EEUU, 2008.

[4] S. Chaudhry, P. Caprioli, S. Yip y M. Tremblay, «High-performance throughput computing,» *Micro, IEEE*, vol. 25, nº 3, pp. 32-45, 2005.

[5] S. Moore, «Multicore is bad news for supercomputers,» *Spectrum, IEEE*, vol. 45, nº 11, pp. 15-15, 2008.

[6] H. Sutter, «Design for Manycore Systems,» *Dr. Dobb's Report*, 2009.

[7] D. Kirk y W.-m. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Burlington, USA: Morgan Kaufmann, 2010.

[8] H. Sutter, «The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software,» *Dr. Dobb's Journal*, vol. 30(3), 2005.

[9] G. E. Moore, «Cramming more circuits on chips,» *Electron*, vol. 19, pp. 114-117, 1965.

[10] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack y J. Shen, «Coming Challenges in Microarchitecture and Architecture,» *Proceeding of the IEEE*, vol. 89, nº 3, pp. 325-340, 2001.

[11] M. Stojcev, T. Tokic y I. Milentijevic, «The Limits of Semiconductor Technology and Oncomming Challenges in Computer Microarchitectures and Architectures,» *FACTA UNIVERSITATIS. SER: ELEC. ENERG.*, vol. 17, pp. 285-312, 2004.

[12] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, A. Miller y M. Upton, «Hyper-Threading Technology Architecture and Microarchitecture,» *Intel Technology Journal Q1*, vol. 6, nº 1, pp. 4-15, 2002.

[13] T. Mowry y A. Gupta, «Tolerating Latency Through Software-Controlled Prefetching,» *Journal of Parallel and Distributed Computing*, vol. 12, pp. 87-106, 1991.

[14] R. Murphy, «On the Effects of Memory Latency and Bandwidth on Supercomputer Application Performance,» *IEEE 10th. International Symposium on Workload Characterization*, pp. 35-43, 2007.

[15] H. W. Meuer, «The TOP500 Project: Looking Back Over 15 Years of Supercomputing,» *Informatik Spektrum*, vol. 31, nº 3, pp. 203-222, 2008.

[16] L. Silva y R. Buyya, «Parallel Programming Paradigms,» de *High Performance Cluster Computing: Programming and Applications*, NJ, USA, Prentice Hall, 1999, pp. 4-27.

[17] T. Sterling, How to Build a Beowulf, Cambridge, MA: MIT Press, 2001.

[18] J. Dongarra, T. Sterling, H. Simon y E. Strohmaier, «High-performance computing: clusters, constellations, MPPs, and future directions,» *Journal of Computing in Science & Engineering*, vol. 7, nº 2, pp. 51-59, Marzo-Abril 2005.

[19] T. Sterling, «An Introduction to PC Clusters for High Performance Computing,» *The International Journal of High Performance Computing Applications*, vol. 15, nº 2, pp. 92-101, 2001.

[20] M. Baker y R. Buyya, «Cluster Computing at a Glance,» de *High Performance Cluster Computing: Architectures and Systems*, NJ, USA, Prentice Hall, 1999, pp. 3-47.

[21] M. Baker y R. Buyya, «Cluster Computing: The Commodity Supercomputer,» *SOFTWARE—PRACTICE AND EXPERIENCE*, vol. 29, nº 6, pp. 551-576, 1999.

[22] G. Bell y J. Gray, «What's Next in High Performance Computing?,» *Communications of the ACM*, vol. 45, nº 2, pp. 91-95, 2002.

[23] D. A. Bader, Petscale Computing: Algorithms and Applications, USA: Chapman & Hall/CRC, 2008.

[24] T. Wilkie, «From mobile pphone to supercomputers?,» *Scientific Computing World*, vol. Febrero/Marzo, pp. 25-30, 2012.

[25] I. G.-t.-M. Services, «Heterogeneous Computing: A new paradigm for the Exascale Era,» *IDC- Executive Brief*, nº www.idc.com, 2011.

[26] J. Earl y S. Conway, «HPC End-User study of Processor and Accelerator Trends in Technical Computing,» *IDC- Sponsored by NVIDIA Corporation*, vol. #228098, 2008.

[27] NVIDIA_Corporation, «NVIDIA,» 2010. [En línea]. Available: www.nvidia.es/object/LO-tesla-brochure-12-lr.html. [Último acceso: Marzo 2012].

[28] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone y J. Phillips, «GPU Computing,» *Proceedings of the IEEE*, vol. 96, nº 5, pp. 879-899, 2008.

[29] M. Piccoli, Computación de alto desempeño en GPU, Primera ed., La Plata: Universidad Nacional de La Plata, 2011.

[30] E. Lindholm, J. Nickolls, S. Oberman y J. Montrym, «NVIDIA Tesla: A Unified Graphics and Computing Architecture,» *IEEE Micro*, vol. 28, pp. 39-55, 2008.

[31] NVIDIA, Optimization. NVIDIA CUDA C Programming. Best Practice Guides, Nvidia Corporation, 2009.

[32] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 2.3, Nvidia Corporation, 2009.

[33] P.-O. Persson y G. Strang, «A Simple Mesh Generator in Matlab,» *SIAM Review*, vol. 46, pp. 329-345, 2004.

[34] G. Leach, «Improving Worst-Case Optimal Delaunay Triangulation Algorithms,» *In 4th Canadian Conference on Computational Geometry*, 1992.

[35] G. Chris y K. Hazelwood, «Where is the data?,» *IEEE Transaction* , pp. 136-144, 2011.

[36] NVIDIA, «Compute Visual Profiler,» *User Guide*, 2011.

[37] J. Nickolls, I. Buck, M. Garland y K. Skadron, «Scalable Parallel Programming with CUDA,» *ACM Queue*, pp. 42-53, 2008.

[38] NVIDIA, «NVIDIA CUDA Programming Guide,» *NVIDIA Corporation,* vol. Version 2.3, 2009.

[39] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee y K. Skadron, «Rodinia: A benchmark suite for heterogeneous computing,» *IEEE Workload Characterization Symposium*, pp. 44-54, 2009.

[40] J. Dongarra y D. Walker, «The Quest for Petascale Computing,» *Computing in Science & Engineering*, vol. 3, nº 3, pp. 32-39, 2001.

[41] J. Brodman, B. Fraguela, M. Garzarán y D. Padua, «New Abstractions for Data Parallel Programming,» de *Proceedings of the First USENIX conference on Hot topics in parallelism*, Berkeley, California, 2009.

[42] I. Raicu, I. Foster y Y. Zhao, «Many-Task Computing for Grids and Supercomputers,» de *IEEE Workshop on Many-Task Computing on Grids and Supercomputers*, 2008.

ANEXO: Código de los kernels

1. Kernel que calcula las longitudes de las barras

```
__global__ void KernelLongitudes(SimuladorDeFuerzasCUDA *persson)
{
	SimuladorDeFuerzasCUDA &p = *persson;
	int i = blockIdx.x * blockDim.x + threadIdx.x;
	__shared__ float longi[1024];
	__shared__ float L2Local;

	//Calcula las longitudes de las barras
	if (i < p.cantidadDeBarras)
	{
		Point2D delta = p.puntosActuales_d[p.barras_d[i].a] –
                                                    p.puntosActuales_d[p.barras_d[i].b];
		p.longitudes_d[i]  = delta.radio(); // longitud de la barra
		longi[threadIdx.x] = delta.radio(); // longitud de la barra

// se determina si la capacidad de cómputo es 2.0 o menor.
#if (CUDAVERSION < 200)
		__syncthreads();
		if (threadIdx.x == 0)
		{
			float L2Temp = 0;
			for (int j = 0; j < blockDim.x && i < p.cantidadDeBarras; j++, i++)
				L2Temp += longi[j] * longi[j];
			atomicFloatAdd(&p.L2, L2Temp);
		}
#else
		if (threadIdx.x == 0)
			L2Local = 0;
		__syncthreads();
		atomicAdd(&L2Local,  delta.radio() * delta.radio());
		__syncthreads();
		if (threadIdx.x == 0)
			atomicAdd(&p.L2,  L2Local);
		//atomicAdd(&p.L2,  p.longitudes_d[i] * p.longitudes_d[i]);
#endif
	}

}
```

2. Kernel que suma las longitudes de las barras

```
__global__ void KernelFuerzas(SimuladorDeFuerzasCUDA *persson)
{
    SimuladorDeFuerzasCUDA &p = *persson;
```

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < p.cantidadDeBarras)
     {
          float Fscale=1.2f;
          float L0 = 1 * Fscale * sqrt(p.L2/p.cantidadDeBarras);

          /// Calcula las fuerzas
          p.fuerzas_d[i] = L0 - p.longitudes_d[i];
          if (p.fuerzas_d[i] < 0)
                    p.fuerzas_d[i]=0;

          float aux = 1.0f;
          aux = aux * p.fuerzas_d[i];
          // poniendo las fuerzas en los lugares que indica la barra

           Point2D delta = p.puntosActuales_d[p.barras_d[i].a] -
          p.puntosActuales_d[p.barras_d[i].b];

          p.movimientosx_d[i]  =  (aux * delta.x * p.deltat) / p.longitudes_d[i]; ///lo guarda en un
          solo lugar

          p.movimientosy_d[i]  =  (aux * delta.y * p.deltat) / p.longitudes_d[i]; /// 3 con 5 y no 5
          con 3


          if (i < p.cantidadDePuntos)
          {
                    p.puntosSiguientes_d[i] = p.puntosActuales_d[i];
          }

      }
}
```

3. Kernel que mueve los puntos en función de las fuerzas ejercidas en las barras

```
    __global__ void KernelMoverPuntos(SimuladorDeFuerzasCUDA *persson)
    {
          SimuladorDeFuerzasCUDA &p = *persson;
          int i = blockIdx.x * blockDim.x + threadIdx.x;
          if (i < p.cantidadDeBarras)
          {
                    ATOMIC_FLOAT_ADD(&p.puntosSiguientes_d[p.barras_d[i].a].x,
                    p.movimientosx_d[i]);
                    ATOMIC_FLOAT_ADD(&p.puntosSiguientes_d[p.barras_d[i].a].y,
                    p.movimientosy_d[i]);
                    ATOMIC_FLOAT_ADD(&p.puntosSiguientes_d[p.barras_d[i].b].x, -
                    p.movimientosx_d[i]);
                    ATOMIC_FLOAT_ADD(&p.puntosSiguientes_d[p.barras_d[i].b].y, -
                    p.movimientosy_d[i]);

            }
          if (i == 0)
                  p.L2 = 0;

  }
```

4. Kernel que calcula el gradiente y luego reproyecta

```
 __global__ void KernelGradiente(SimuladorDeFuerzasCUDA *persson)
{
        SimuladorDeFuerzasCUDA &p = *persson;
        int i  = blockIdx.x * blockDim.x + threadIdx.x;
        if (i == 0)
                p.L2 = 0;
        if (i < p.cantidadDePuntos)
        {
                Point2D punto   = p.puntosSiguientes_d[i];
                float distancia = 0;
                p.dext_d[i]       = distancia;
                Point2D puntomasxh(punto.x + p.deps, punto.y);
                Point2D puntomasyh(punto.x         , punto.y + p.deps);
                p.gradientes_d[i].x = (p.region_d->distancia(puntomasxh) - distancia) / p.deps;
                p.gradientes_d[i].y = (p.region_d->distancia(puntomasyh) - distancia) / p.deps;
                if (p.dext_d[i] > -p.geps)  //fuera del dominio
                {
                    p.puntosSiguientes_d[i].x = p.puntosSiguientes_d[i].x - p.dext_d[i] *
                                                p.gradientes_d[i].x;
                    p.puntosSiguientes_d[i].y = p.puntosSiguientes_d[i].y - p.dext_d[i] *
                                                p.gradientes_d[i].y;
                    p.dext_d[i] = 0.f;
                }
                else    //if(dext_h[p]<=-geps) - dentro del dominio
                {
                    Point2D diff = p.puntosActuales_d[i] - p.puntosSiguientes_d[i];
                    p.dext_d[i]  = diff.radio() / p.h0;
                }
          //~ *  Calculo de los desplazamientos en comparacion con los puntos
          //~ *  usados para el ultimo Delaunay.
                Point2D diff = p.puntosUltimoDelaunay_d[i] - p.puntosSiguientes_d[i];
                p.desplazamientos_d[i] = diff.radio() / p.h0;
        }
 }
```

5. Llamado a los kernels desde la CPU

```
void lanzarKernels(SimuladorDeFuerzasCUDA *persson)
{
    int threadsPorBloque = 512;
    int cantidadDeThreads = ss->persson_d->m_mayado.m_cantidadDeBarras;
    int cantidadDeBloques = (cantidadDeThreads / threadsPorBloque) + 1;

    KernelLongitudes<<<cantidadDeBloques , threadsPorBloque>>>(persson);
    cudaThreadSynchronize();
```

```
  KernelFuerzas<<<cantidadDeBloques, threadsPorBloque>>>(persson);
  cudaThreadSynchronize();

  KernelMoverPuntos<<<cantidadDeBloques, threadsPorBloque>>>(persson);
  cudaThreadSynchronize();

  KernelGradiente<<<cantidadDeBloques, threadsPorBloque >>>(persson);
  cudaThreadSynchronize();

}
```