Uso de GPUs en cómputo de propósito general. Aplicación a un método de mallado de elementos finitos.

Adriana A. Gaudiani(1), Santiago Montiel (1)
(1) Universidad Nacional General Sarmiento (UNGS), Instituto de Ciencias, Informática, Módulo 1, Campus Universitario, José M. Gutiérrez 1613, Los Polvorines, Buenos Aires, Argentina, agaudi,smontiel@ungs.edu.ar

Resumen

NVIDIA abrió la arquitectura de los procesadores gráficos de sus placas de video (GPUs) para ser utilizados en aplicaciones de propósito general y desarrolló el lenguaje de programación CUDA permitiendo a los programadores crear funciones con explícito paralelismo de datos, brindando una plataforma de cómputo de alto rendimiento y paralelismo a gran escala.

Nuestro trabajo presenta la aplicación de las capacidades de cómputo paralelo provisto por las GPUs, como coprocesadores de la CPU, para mejorar el tiempo de ejecución de un método de mallado de elementos finitos creado por P. Persson y G. Strang para MATLAB. Dado que las GPUs poseen decenas de unidades de procesamiento, son adecuadas para el procesamiento de datos matriciales, siendo esto una ventaja para nuestro algoritmo.

Desarrollamos una función que toma un conjunto de puntos provenientes de la discretización de un dominio sobre el cual se desea el mallado, mediante el método de Delaunay, y calcula el movimiento de cada punto según el método de Persson-Strang. Esta función forma parte de un proceso iterativo propio del método, cuyo objetivo es mejorar la calidad del mallado.

Nuestro trabajo consigue mejoras en la ejecución de la función del 62 % comparado con el tiempo de ejecución dentro de la función *distmesh2d* de MATLAB.

Palabras claves Algoritmos Paralelos, Procesadores Gráficos, GPU, Cómputo de Alto Rendimiento, Paralelismo de datos.

1. Introducción

La programación de computadoras ha ido evolucionando con el tiempo acompañando la evolución de la tecnología de computadoras y el aumento de la potencia del hardware. Tradicionalmente, en la “Era del Código Serial”, los desarrolladores escribían las aplicaciones y estas corrían más rápido cada 18 meses. IBM, AMD,

Intel, Motorola, se encargaban de lograrlo con plataformas de hardware que rondaban alrededor de un procesador sin modificar ninguna línea de código. Esto dejó de ser así desde hace algunos años y actualmente los programadores no pueden esperar el aumento de la frecuencia del reloj.

Herb Sutter, en su artículo para Dr. Dobbs [8] nos decía "Your free lunch will soon be over", alertándonos sobre el fin de esta era, la del código secuencial. Esto se debe a muchas razones conocidas, como ser la física de los semiconductores y el diseño de los chips, el cual ha migrado a múltiples cores. Aseguraba que la ganancia a obtener con un solo procesador sería cada vez menor. Los beneficios a obtener con esta nueva tecnología multi-core sería enorme, pero sólo con esfuerzo. Anunciaba a los desarrolladores la llegada de uno de los más grandes cambios en desarrollo de software desde la programación orientada a objetos: la programación concurrente como la nueva tendencia. El esfuerzo al que hacía referencia era respecto a pensar y diseñar el software mediante concurrencia. [1]

La sabiduría convencional nos dice que la programación en paralelo es difícil. A pesar de esto, el modelo escalable de programación que ofrece CUDA[2] ha demostrado que en poco tiempo los desarrolladores pudieron comprender su modelo de procesamiento en paralelo. Considerando que NVIDIA liberó el código a partir de 2006, los desarrolladores sólo necesitaron un año de gestación y rápidamente crearon un extenso rango de programas paralelos usando CUDA. La capacidad de cómputo de las GPUs permitió lograr rendimientos asombrosos en programas de cómputo intensivo, no necesariamente de computación gráfica, sino también de propósito general. Estos GP-GPUs (General Purposse GPUs) han ofrecido a los desarrolladores una arquitectura unificada gráfica y de cómputo, junto al modelo de programación CUDA, fuertemente escalable. [5] CUDA no deja de ganar clientes en los campos científicos e ingenieriles. El trabajo duro para los desarrolladores está en analizar la aplicación y descomponer los datos logrando la mejor configuración de grids y bloques de threads. Para lograr su propuesta se requiere un cambio en el pensamiento, en lugar de enfocarse en el algoritmo que maneja los datos, es necesario concentrarse en la naturaleza de los datos y en su organización durante el cómputo.[3] Según Jack Dongarra, "Las arquitecturas informáticas del futuro serán sistemas híbridos con GPUs compuestas por núcleos de procesamiento paralelo que trabajarán en colaboración con las CPUs multinúcleo."

2. Modelo de Programación de CUDA

CUDA representa los multiprocesadores como una unidad capaz de manejar múltiples grupos de threads con una extención mínima del lenguaje C/C++. Para ello ofrece un conjunto de abstracciones claves: jerarquía en grupos de threads, memorias compartidas y barreras de sincronización. El programador escribe el

[1] Michael Wrinn- Intel Articles - http://software.intel.com/en-us/articles/is-the-free-lunch-really-over-scalability-in-many-core-systems-part-1-in-a-series/

[2] Compute Unified Device Architecture

código serial de kernels que se ejecutarán en paralelo sobre conjuntos de threads paralelos, mediante un modelo SIMT (Single Instruction Multiple Threading) análogo a SIMD.

Mejorar los resultados obtenidos requiere conocer la arquitectura subyacente y específicamente la sobrecarga de transferencia, conflictos en bancos de memoria el impacto del control de flujo deben considerarse al programar. Una manera de reducir la sobrecarga de cómputo es pensar los algoritmos desde la perpectiva de los GPUs y utilizar como elemento de ayuda la CPU.[1]

El programador organiza los threads en bloques dentro de una jerarquía de *grids*{*de bloques*{*de threads*}}. Estos threads concurrentes pueden cooperar en el cómputo usando barreras de sincronización y accediendo a memoria compartida en un área privada del bloque. Al momento de invocar un kernel, el programador maneja la cantidad de grids, bloques y threads, tratando de hacerlo de la manera más eficiente según los recursos con los que cuenta. La eficiencia se mide con el grado de ocupación de los GPUs. (La relación entre los warps activos y la máxima cantidad de warps soportada por un multiprocesador del GPU, durante la ejecución de un Kernel).[5]

Cada multiprocesador posee un conjunto de N registros disponibles para los threads en ejecución. Por ejemplo, la medida de N en una G80 es de 8192 registros de 32-bits por multiprocesador. Estos registros, compartidos por los bloques de threads, es optimizado por el compilador CUDA asignando la cantidad mínima posible a cada thread y maximizando el número de threads activos simultáneamente. [3]

Es imprescindible comprender el modelo de memoria de CUDA al momento de programar. Cada thread accede a tres niveles de memoria: su propia **memoria local** donde almacena sus variables locales, la **memoria compartida** por bloques, donde los threads declaran variables mediante el calificador *__shared__* y la **GPU DRAM**, visible para todos los threads de un kernel, mediante el uso del calificador *__device__*. [2]

3. Problema de Aplicación.

3.1. Método de Persson

Per-Olof Persson y Gilbert Strang, desarrollaron un código generador de mallas para MATLAB, llamado *DistMesh*, el cual está a disposición del público[4]. Este código fue escrito con el objetivo de brindar un método simple y de muchas menos líneas que otras técnicas de mallado. Persson-Strang toman el mallado que provee el algoritmo de Delaunay y, según se explica a continuación, utilizan una técnica que le permite mejorarlo y obtener un mallado de más calidad.

La idea de este algoritmo es utilizar una simple analogía mecánica entre una

[3] http://news.developer.nvidia.com/2007/03/cuda_occupancy_.html

[4] http://math.mit.edu/ persson/mesh/

malla de triángulos o tetraedros, y una estructura de resortes. Cualquier conjunto de puntos en el plano puede ser triangulado por el algoritmo de Delaunay. Los puntos de la malla son nodos de una estructura armada con "barras" o "resortes" que se encuentran comprimidos dentro del dominio. Se trata de dejar evolucionar el sistema mediante un proceso iterativo hasta llegar algún equilibrio establecido para el método, por ejemplo cuando la sumatoria de fuerzas en cada nodo sea cero, a menos de una tolerancia. La fuerza resultante en cada iteración mueve los nodos determinando la nueva posición de éstos haciéndolo iterativamente usando el algoritmo de Euler. Si algún nodo es reubicado fuera del dominio, entonces debe ser reproyectado hacia la superficie. En el caso que los puntos se separen en más de alguna tolerancia preestablecida para el método, entonces se vuelve a utilizar Delaunay para generar un nuevo mallado, y se sigue adelante con el método general.

Este código MATLAB fue vectorizado para evitar los ciclos-for y lograr mayor eficiencia aunque aun podría ser mejorado según explican los autores en [6].

3.2. Mallado de Delaunay

Delaunay es un antiguo e importante concepto en Geometría Computacional. Existen muchos algoritmos desarrollados y que son muy eficientes en cómputo sobre CPU. Este método genera mallados no estructurados mediante triángulos en 2D y tetraedros en 3D, con un una garantía de calidad aceptable para problemas en los que la geometría del problema cambia con el tiempo. El algoritmo de Delaunay es muy popular en la generación de mallados de diversas aplicaciones, como ser en interpolación, rendering, sistemas de información geográfica y solución de sistemas de ecuaciones en derivadas parciales. Esta última es una de las áreas de más demanda.

Para este trabajo hemos seleccionado una implementación del algoritmo de Delaunay, del tipo Divide y Vencerás escrito por Geo Leach en lenguaje C[5]. Este algoritmo es de complejidad O(n logn), la mejor complejidad lograda, siendo n la cantidad de puntos del dominio. El mismo fue optimizado como explica su autor en [4].

3.3. Descripción del Trabajo

El primer paso de nuestro trabajo consistió en transcribir el método de mallado del algoritmo *Distmesh* a lenguaje C++ con el objetivo de utilizar el lenguaje CUDA para acelerar el cómputo de algunas partes del código. A continuación se muestra una breve descripción general del algoritmo, aprovechando la notación de Matlab:

deltat: paso utilizado en el algoritmo de Euler (para evolución de la ecuación del resorte en el tiempo).

[5]Copyright ©2008 RMIT CS

h0: distancia entre los puntos de la distribución inicial (dominio).
P: estructura que contiene las coordenadas de los puntos del dominio.
epsilon: si los movimientos totales de los nodos en una iteración son menores a este valor el algoritmo termina (ya convergió).
tolerancia: controla cuanto pueden moverse los nodos antes de que ocurra una retriangulación de Delaunay.

Mientras (maximo $(\sqrt{\sum((deltat * FTOT_i)^2)}/h0) > epsilon)\{$
Comienza el ciclo principal de Persson. Este mejora iterativamente la posición inicial de los N Puntos cuyas coordenadas están almacenadas en la estructura P.

Si $(maximo(\sqrt{(P - P_{old})^2} - h0) > tolerancia)\{$
$P_{old} = P;$
$TRI = Delaunay(P);$
Cálculo de los centroides de cada triángulo;
Conserva sólo los triángulos interiores;
$\}$
Cada vez que algún movimiento de los puntos, respecto a la última configuración, es mayor que "tolerancia", recalcula la triangulación usando Delaunay y deja en BARS los puntos de los extremos de cada lado de los triángulos.
FinSi

$BARS = CrearBarras(TRI);$
$FTOT = CalcularFuerzas(P, BARS);$
$P = P + deltat * FTOT;$
Para todo P fuera del dominio $\{$
$P = P + d(P) * (-\nabla d);$
$\}$
Utilizando los datos de TRI, se crea BARS para guardar los puntos extremos de cada barra. Con la configuración de puntos P y BARS se calcula FTOT, la sumatoria de fuerzas en cada punto. Luego se calcula la posición nueva de cada punto usando FTOT. Todo punto que se movió fuera del dominio es reproyectado hacia el punto más próximo del borde mediante el cálculo del gradiente numérico.
$\}$
FinMientras

Nuestro trabajo consistió en mejorar una parte de este algoritmo, la etapa que más tiempo de ejecución consume. Consiste en el cálculo de la fuerza total sobre cada punto y el posterior movimiento de los puntos del mallado.

Por lo general hay dos aproximaciones al uso de GPUs en cómputo general. Una es utilizando cada pequeño procesador de la GPU como procesadores independientes en paralelo, como el modo tradicional de utilizar los procesadores de una máquina paralela. Otra es realizando el cómputo directamente sobre la memoria de texturas, modo que mapea adecuadamente a la arquitectura de la GPU pero requiere desde el cómputo manejar las direcciones de los pixeles y suele utilizar comunicación entre ellos.[7] Nosotros utilizamos la primera aproximación.

Nuestro problema se resuelve desde la CPU y ésta recurre a la GPU como

coprocesador y proveedor de poder de cómputo paralelo. El algoritmo escrito serialmente para implementarse en la CPU utiliza ciclos para recorrer los datos almacenados en arreglos. En el código escrito en CUDA, se reemplazan estos ciclos por *kernels* que serán ejecutados sobre la GPU cuyos *threads* se encargarán de procesar cada elemento del arreglo de manera concurrente siguiendo las instrucciones del código. O sea, el modo de pensar el algoritmo del kernel obliga a pensar exclusivamente en el manejo de los datos, en el modo de representarlos para lograr un óptimo rendimiento de la arquitectura de los GPUs.

El algoritmo presentado anteriormente está pensado para ser ejecutado por la CPU, excepto al llegar a los pasos:
$BARS = CrearBarras(TRI)$;
$FTOT = CalcularFuerzas(P, BARS)$;
$P = P + deltat * FTOT$;

Estos pasos fueron encapsulados en una función escrita con CUDA para ser ejecutado por la GPU. Escribir esta función de manera eficiente demandó la mayor cantidad de tiempo invertido en este trabajo. Uno de los objetivos buscados al escribir esta función fue evitar el pasaje de grandes cantidades de datos entre la memoria de la unidad gráfica y la RAM. Esta operación es realizada con la instrucciones *cudaMemcpy* y los parámetros *cudaMemcpyHostToDevice* o *cudaMemcpyDeviceToHost* según se envíen o reciban los datos, entendiéndose como *Host* el sistema CPU-RAM y como *Device* las GPUs. Si no se controla su uso consume gran cantidad de tiempo y degrada el rendimiento general.

Los pasos de la función desarrollada se pueden resumir en los siguientes:

1. Recibe como datos los arreglos con las barras, los puntos y algunos parámetros propios del problema.

2. Recibe la cantidad de bloques y threads adecuada según el modelo de la placa de video.

3. Calcula las longitudes de la barras, la componente x e y de la distancia entre los puntos de los extremos de las barras y el valor escalar de las fuerzas aplicadas a cada barra.

4. Calcula las componentes x e y de la sumatoria de fuerzas aplicada en cada punto, tomando como datos los puntos de los extremos de cada barra y los datos calculados anteriormente.

5. Calcula la sumatoria de los movimientos a aplicar a cada punto en sus componentes x e y.

6. finalmente se devuelve a la RAM los puntos modificados.

Nuestra función solamente devuelve un arreglo con la nueva posición de los puntos de la malla Fue necesario recurrir a arreglos auxiliares que permitieron

eliminar las alternativas en el código, para lo cual se implementaron en la GPU dos arreglos con dimensiones NxM, siendo N cantidad de lados de los triángulos y M cantidad de puntos, evitando de esa manera otro factor de degradación del rendimiento general. Dichos arreglos son direccionados en la memoria global de la GPU mediante la función *cudaMalloc* pero no son devueltos a la RAM. El paso 5 genera dichas estructuras empleando una matriz dispersa de dimensiones (Cantidad de barras X Cantidad de puntos), dónde cada fila de la matriz contiene los movimientos de cada punto proveniente de cada barra con la que está conectado. En el paso 6 se recurre a las posibilidades de programación que ofrece CUDA como es el manejo de la sincronización de los threads y al uso de la memoria compartida por los bloques de threads, para la creación de arreglos auxiliares. NVIDIA ofrece un detallado ejemplo sobre el uso de la memoria *_SHARE* en su manual del Programador.

4. Resultados Experimentales

Implementamos nuestro algoritmo en una PC INTEL DualCore, 1.5GHz, con 2 GB RAM, con una placa NVIDIA GForce 8400M GS. El programa fue desarrollado usando Microsoft Visual C++ 2008 Express y compilado con todas las opciones de optimización posibles. El código GPU está escrito y compilado con NVIDIA CUDA 2.3. Para correr la función Distmesh se utilizó Matlab 6.1. Esta placa de video se caracteriza por tener 2 Multiprocesadores de 16 cores cada uno, memoria compartida por bloque de 16 KBytes, 8192 registros por bloque, 32 threads por Warp y un máximo de 512 threads por bloque.[6]

La ejecución del algoritmo se hizo utilizando la función *clock()* para la medición de los tiempos de ejecución. Se midió el tiempo correspondiente a la ejecución del kernel y se comparó con el tiempo de ejecución en Matlab de las instrucciones de Distmesh que ejecutan las mismas tareas. Se reprodujo en ambos casos el mismo entorno de ejecución para poder comparar los tiempos obtenidos. El dato que determina la cantidad de cómputo, y en consecuencia el tiempo de ejecución, es la cantidad de puntos del dominio sobre el que se quiere hacer el mallado.

Puntos	Tiempo CPU (msg)	Tiempo GPU (msg)	SpeedUp	Ganancia
80	31,75	23,50	1,35	26%
170	42,64	27,52	1,55	35%
200	63,13	26,80	2,36	58%
360	73,31	28,12	2,61	62%

Figura 1: Tiempos de las experiencias

Según se ve en la figura 1. el tiempo de ejecución del kernel en la CPU no tiene mucha variación, esto se debe a que se asigna un thread para procesar cada

[6] http://www.nvidia.es/page/geforce_8400M.html

elemento de la matriz de movimientos y esto es así aunque se aumente la cantidad de datos, ya que se cuenta con threads disponibles para asignar a cada cómputo individual. Este comportamiento propio de la programación SIMT (*Single Instruction Multiple Threading*) en la GPU es explotado en beneficio del rendimiento obtenido, permitiendo alcanzar un speedup de 2.6 con la mayor cantidad de puntos experimentado. Esta ganancia se puede visualizar en la figura 2.

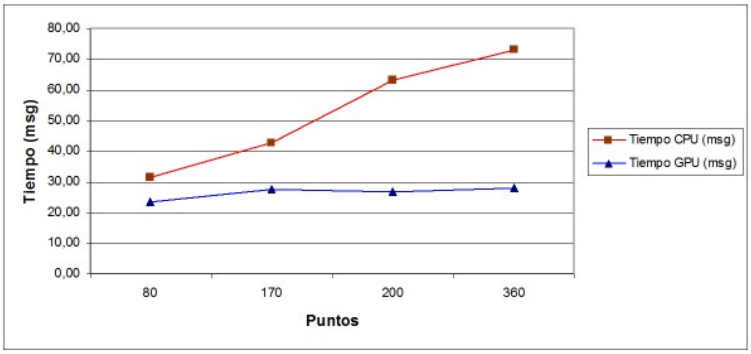

Figura 2: Ganancia Obtenida

5. Conclusiones

La ganancia obtenida mediante la integración CPU + GPU, resultó significativa, pero aun es posible mejorar el rendimiento del algoritmo. Este trabajo nos brindó una prueba del poder de cómputo que ofrecen las GPUs con sus procesadores gráficos en beneficio del cómputo de propósito general. Analizando el uso hecho de la arquitectura de NVIDIA, es necesario seguir trabajando para optimizar los parámetros de cómputo como cantidad de threads, bloques y registros asignados al momento de la ejecución, lo cual es posible entendiendo cada vez más sobre el modelo de programación y de memoria de CUDA. Desde el punto de vista de la programación se debe mejorar aun el algoritmo para evitar al máximo posible el pasaje de datos entre CPU y GPU, y logra así crear una función de tipo MEX para ser incluida entre las funciones de Matlab, como es el caso de Distmesh.

6. Agradecimientos

Los autores quieren agradecer la inestimable ayuda de Emmanuel Frati, para enfrentar incontables problemas de programación, y a Gabriel Rodríguez Acosta quién hizo posible este trabajo. Esta investigación está parcialmente soportada por el Proyecto PICT-00910-CONICET-Argentina.

Referencias

[1] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Parallel Distrib. Comput.*, 68(10):1370–1380, 2008.

[2] Michael Garland. Sparse matrix computation on manycore gpu´s. *Annual ACM IEEE Design Automation Conference. DAC '08: Proceedings of the 45th annual Design Automation Conference.*, pages 2–6, 2008.

[3] Tom R. Halfhill. Parallel processing with cuda. nvidia´s high-performance computing platform uses massive multithreading. *Microprocessor Report*, January 2008.

[4] Geoff Leach. Improving worst-case optimal delaunay triangulation algorithms. In *In 4th Canadian Conference on Computational Geometry*, page 15, 1992.

[5] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. *Scalable Parallel Programming*. ACM QUEUE, April 2008.

[6] Per-Olof Persson and Gilbert Strang. A simple mesh generator in matlab. *SIAM Review*, 46:329–345, 2004.

[7] Guodong Rong, Tiow-Seng Tan, Thanh-Tung Cao, and Stephanus. Computing two-dimensional delaunay triangulation using graphics hardware. In *I3D '08: Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 89–97, New York, NY, USA, 2008. ACM.

[8] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), Marzo 2005.