TECNIACUSTICA Cáceres · 2011

42º CONGRESO ESPAÑOL DE ACÚSTICA
ENCUENTRO IBÉRICO DE ACÚSTICA
EUROPEAN SYMPOSIUM ON ENVIRONMENTAL ACOUSTICS
AND ON BUILDINGS ACOUSTICALLY SUSTAINABLE

TÉCNICAS GPGPU PARA ACELERAR EL MODELADO DE SISTEMAS ACÚSTICOS

PACS: 43.35.Zc

D. Romero-Laorden; O. Martínez-Graullera; C.J. Martín-Arguedas; M. Pérez-López;
L. Gómez-Ullate
Grupo de Evaluación por Ultrasonidos
Centro de Acústica Aplicada y Evaluación No Destructiva (CAEND UPM-CSIC)
Carretera de Campo Real, km 0.200. La Poveda.
Arganda del Rey, Madrid, E-28500 (España)
Tel: (+34) 91 871 19 00
Fax: (+34) 91 871 70 50
E-mail: dromerol@caend.upm-csic.es

ABSTRACT

The modeling of the acoustic field is a common practice in the field of ultrasound. However, it is a task due to its complexity demands a high computing capacity and processing time, which limits their use in systems which require real time. The use of graphics hardware (GPUs) can significantly speed up the calculations, due to the high degree of parallelism and performance involved in this type of devices. This paper shows how the use of these technologies reduces the computation time in more than one order of magnitude over the CPU implementation.

RESUMEN

El modelado del campo acústico es una práctica habitual en el campo de los ultrasonidos. Sin embargo, es una tarea que debido a su complejidad, demanda una gran capacidad de cómputo y tiempo de procesamiento, lo que limita su uso en sistemas que demandan tiempo real. El uso de hardware gráfico (GPUs) puede acelerar significativamente los cálculos, gracias al alto grado de paralelismo y el rendimiento que presentan este tipo de dispositivos. En este trabajo se muestra cómo el uso de este tipo de tecnologías permite reducir el tiempo de cómputo en más de un orden de magnitud con respecto a la implementación en CPU.

1. INTRODUCCIÓN

Actualmente las técnicas de Evaluación No Destructiva (END) juegan un rol importante tanto en la caracterización como en la detección de defectos en materiales y estructuras. Su objetivo es usar algún tipo de fenómeno que al interaccionar con el objeto de interés permita medir algunas propiedades en el mismo sin alterarlo. Las técnicas de END basadas en ultrasonidos se basan en un principio básico: un sistema electrónico excita uno o más transductores ultrasónicos que generan pulsos de ondas mecánicas de alta frecuencia que se introducen y propagan en el material a inspeccionar. Del análisis de los cambios que se producen sobre esta misma onda o

sobre sus reflexiones es posible extraer diversas características como la presencia de discontinuidades, que pueden ser identificadas como defectos [1, 2].

Hoy en día se tiende a inspeccionar estructuras de geometría más compleja, mediante agrupaciones de transductores (arrays) que pueden contener cientos de elementos. Para resolver las dificultades asociadas al análisis de los datos de la evaluación se precisa del apoyo de técnicas de simulación que permitan obtener una referencia del resultado de la misma. En este sentido, las técnicas de END basadas en ultrasonidos hacen uso de distintos modelos de simulación [3, 4, 5] tanto para la planificación de inspecciones como para la obtención de "huellas sónicas" que son usadas como base a la hora de determinar el estado del componente bajo análisis.

En los últimos años, una de las tendencias dominantes en las arquitecturas de microprocesadores ha sido el continuo incremento del paralelismo a nivel de chip. Hoy día, es común trabajar con ordenadores (CPUs) formados por varios núcleos. Las unidades de procesamiento gráfico o GPUs también han visto incrementado su nivel de paralelismo en torno al chip y pueden contener actualmente hasta 240 núcleos [7]. Estos dispositivos gráficos, que originariamente estaban exclusivamente centrados en el mercado de los videojuegos y el 3D ahora pueden ser programados directamente en C usando lenguajes de programación como CUDA [8, 10] u OpenCL. En este trabajo se presentan métodos para el modelado de aplicaciones ultrasónicas en END usando técnicas GPGPU (Computación de Propósito General en Unidades de Procesamiento Gráfico) [11] con el objetivo de reducir el alto coste asociado a los algoritmos y facilitar su implementación en sistemas de tiempo real.

2. MODELADO EN END

Para poder evaluar las capacidades y ventajas de la computación GPGPU, se ha simulado un ejemplo de cálculo del campo generado en emisión y recepción con un array lineal emitiendo en un medio homogéneo (agua). El array tiene $N = 32$ elementos con una separación entre elementos (pitch) de $d = \lambda/2$. El ancho de los elementos es $a = d$ y su altura es $b = 16\lambda$. Los elementos emiten pulsos de 50% de ancho de banda. Se ha calculado el máximo de presión en 12900 puntos pertenecientes a un plano situado frente al array. La expresión general de la presión acústica generada por un array que trabaja en emisión y recepción viene dada por [6]:

$$p(\vec{x},t) = \frac{1}{c^2} * \frac{\partial^2 v(t)}{\partial t^2} * h_E(\vec{x},t) * h_r(\vec{x},t) \tag{1}$$

donde h_E y h_R son, respectivamente, las respuestas al impulso en emisión y recepción del array:

$$h_E(\vec{x},t) = \sum_{i=1}^{N} a_i^E h_i(\vec{x}, t - T_i^E) \tag{2}$$

$$h_R(\vec{x},t) = \sum_{i=1}^{N} a_i^R h_i(\vec{x}, t - T_i^R) \tag{3}$$

En la expresión anterior, $h_i()$ es la respuesta al impulso del i-ésimo elemento y a_i es su peso asociado. T_i^E y T_i^R son los retardos de focalización en emisión y recepción respectivamente, que para un punto $p(\vec{x})$ vienen dados por la expresión:

$$T_p(i) = \frac{|\vec{x} - \vec{x_i}| - |\vec{x}|}{c} \tag{4}$$

donde c es la velocidad del sonido en el medio inspeccionado, y $\vec{x_i}$ es el vector de posición del i-ésimo elemento. Para poder calcular el campo acústico se ha seguido el método de Piwakowsky [3], que hace una computación directa de la integral de Rayleigh por medio de muestras temporales a intervalos Δt y discretiza las superficies del transductor en celdas

cuadradas de área ΔS. La contribución de cada elemento a la respuesta al impulso en el instante dado t_S se obtiene añadiendo las contribuciones de cada una de las celdas contenidas entre dos ondas esféricas concéntricas de radio inferior a $c\Delta t_S$, que están separadas por el intervalo de discretización $c\Delta t$ [3]. Así, la contribución a la respuesta al impulso correspondiente al i^{th} elemento del array en el instante $t = t_S$ es:

$$h_i(t = t_S) = \frac{1}{\Delta t}\sum_j b_j \quad , t_S - \frac{\Delta t}{2} \le t_j \le t_S + \frac{\Delta t}{2} \quad donde \quad b_j = \frac{a_i \Delta S}{2\pi R_j} \quad y \quad t_j = \frac{R_j}{c} + T_i \qquad (5)(6)$$

donde T_i denota el retardo de focalización en emisión o recepción y R_j/c es el tiempo de propagación.

2.1 Algoritmo Computacional

El algoritmo de Piwakowsky que se ha presentado en la sección anterior se muestra esquemáticamente en la figura 1. Como se observa, seis son las etapas necesarias para obtener el valor del campo en cada punto espacial *P:*

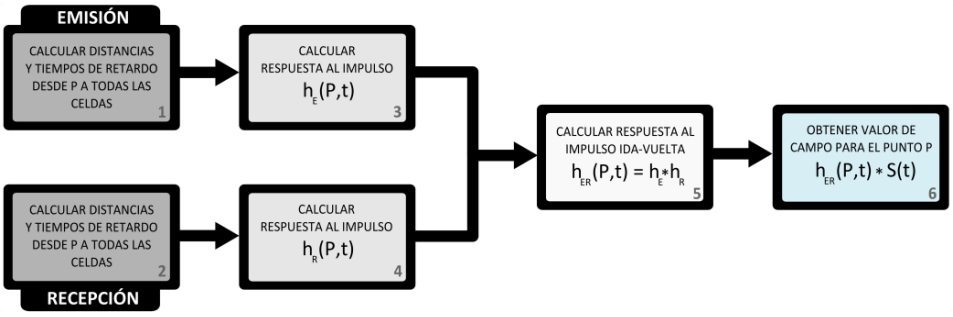

Figura 1. Algoritmo iterativo de Piwakowsky. El cálculo del campo en emisión y recepción requiere de seis etapas

1. En el primer paso se calculan las distancias desde todas las celdas pertenecientes a la apertura en emisión a cada punto del espacio *P*. Además, se calcula otro tipo de información necesaria como son los tiempos de retardo de focalización (ec. 4) y los pesos de la apodización o filtro espacial, y se asocian a sus respectivas celdas (ec. 6).
2. En el segundo paso, si es necesario, se lleva a cabo un procedimiento similar como el del primer paso pero aplicándolo a la apertura en recepción.
3. Después, se calcula la respuesta al impulso del array en emisión $h_E(P,t)$ (ec. 2 y 5).
4. En el cuarto paso, se calcula la respuesta al impulso del array en recepción $h_R(P,t)$ de manera similar al método del paso anterior (ec. 3 y 5).
5. A continuación, se hace la convolución entre h_E y h_R, obteniendo la respuesta al impulso en ida y vuelta del array $h_{ER}(P,t)$ (ec. 1).
6. Finalmente, se obtiene el campo acústico en ida y vuelta, realizando la convolución entre $h_{ER}(P,t)$y la señal emitida (ec. 1).

Es importante remarcar que el algoritmo es iterativamente aplicado al conjunto completo de puntos de campo.

3. COMPUTACIÓN PARALELA SOBRE GPUS

Antes de discutir el diseño de nuestros algoritmos, se comentarán brevemente algunos detalles sobre la arquitectura NVIDIA GPU y el modelo paralelo de programación CUDA. Como ya

hemos comentado, las GPUs nacieron con el objetivo de acelerar la representación de gráficos (sobretodo en interfaces de programación como OpenGL o DirectX). Debido al gran paralelismo inherente en los gráficos, éste tipo de hardware es una máquina paralela muy potente para ser utilizada en ámbitos científicos. Son por tanto, procesadores multi-núcleo muy flexibles, altamente paralelos, multi-hilo y con un alto poder computacional y ancho de banda a memoria.

CUDA (de las siglas *Compute Unified Device Arquitecture*) es una interfaz de programación desarrollada para facilitar la programación de este tipo de dispositivos en entornos diferentes. Fundamentalmente, una aplicación se organiza como un programa secuencial en el PC que es capaz de ejecutar programas paralelos, llamados *kernels.* A partir de aquí el programador se encarga de organizar los distintos flujos de ejecución (especificando el número de hilos en ejecución) y particionando apropiadamente el problema en particular. Se proporcionan mecanismos lógicos como los bloques de hilos y las rejillas de bloques como ayuda adicional en la programación. Un bloque es un grupo de hilos en el que existen mecanismos de sincronización (es decir, los hilos pueden cooperar entre sí para llegar a un objetivo). En este sentido, CUDA proporciona a los desarrolladores los medios necesarios para ejecutar programas paralelos en la GPU. Una completa descripción sobre la arquitectura y el modelo de programación puede consultarse en distintas fuentes [8, 9].

3.1 Consideraciones GPU

Las notables diferencias entre las arquitecturas CPU y GPU hacen necesario comentar brevemente algunas consideraciones. Es importante enfatizar que las transacciones entre CPU y GPU son bastante lentas, por lo que deben ser minimizadas lo máximo posible para mejorar el rendimiento global. Además, la memoria del dispositivo es limitada por lo que en muchos casos puede ser necesario tener que reducir el volumen de datos que se debe procesar en paralelo. El tiempo de lectura desde memoria global es lento y es recomendable por tanto usar algunos otros mecanismos, como la memoria de textura o la memoria compartida siempre que sea posible. Finalmente, simplificar las operaciones por hilo y homogeneizar el tiempo de ejecución de todos los hilos puede resultar muy beneficioso así como minimizar las escrituras a memoria de la GPU. Desafortunadamente, la implementación directa de los algoritmos originalmente diseñados para CPU no suele ser la mejor opción como podrá comprobarse en la siguiente sección.

4. ALGORITMOS PARALELOS

Los candidatos perfectos para la computación paralela son aquellas funciones que son ejecutadas muchas veces independientemente sobre diferentes datos y en este sentido, las características propias de los algoritmos utilizados en END normalmente implican operar con un gran volumen de datos y el cálculo de distintos resultados de manera repetitiva, lo que les hace ser unos candidatos idóneos para computación paralela. El mayor problema de la computación en GPU es encontrar la mejor estrategia de paralelización de los algoritmos para poder obtener el máximo beneficio.

En este trabajo se han evaluado distintas alternativas. Debido al hecho de que el algoritmo se aplica a cada punto del espacio, la paralelización más obvia sería por puntos, usando un hilo por punto y calculándolos todos simultáneamente. Pero hay otras posibilidades que pueden ser tomadas en cuenta. Las diferentes etapas del algoritmo implican diferentes requisitos computacionales, así que quizás podría intentarse una paralelización por celdas espaciales de la apertura, muestras de las respuestas en emisión-recepción u otras. El primer aspecto a tener en cuenta es que se debe lanzar el cómputo por grupos de puntos, adaptando el volumen de datos almacenado a la memoria de la GPU.

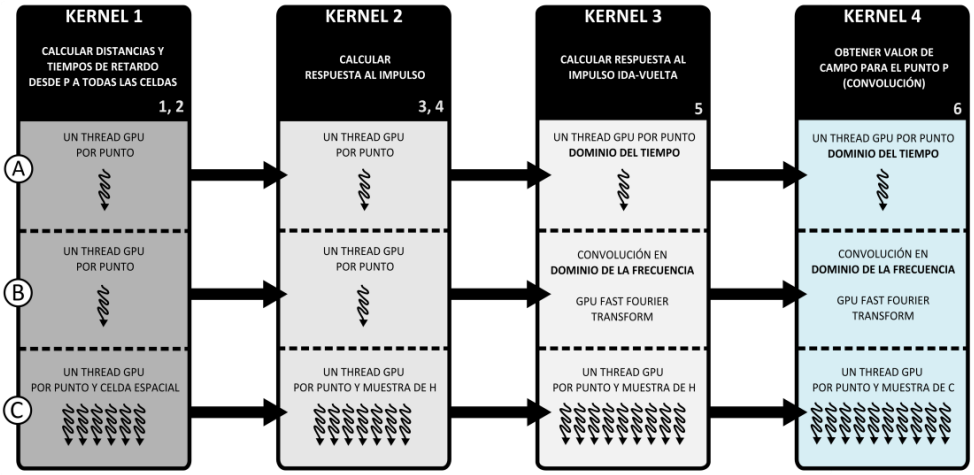

Figura 2. Esquemas de las tres estrategias estudiadas en este trabajo: (A) un hilo por punto de campo y convolución en el dominio del tiempo; (B) un hilo por punto de campo y convolución en el dominio de la frecuencia; (C) hilos adaptados a cada etapa

Para el desarrollo de los distintos algoritmos paralelos se han diseñado 4 *kernels* bien diferenciados que cubren cada una de la etapas definidas anteriormente: (1) *kernel* encargado de calcular las distancias y tiempos de retardo desde cada punto espacial a todas las celdas (en emisión y recepción - pasos 1 y 2); (2) *kernel* que computa la respuesta espacial al impuso con una implementación paralela del método de Piwakowsky (en emisión y recepción - pasos 3 y 4); (3) *kernel* para realizar la convolución en ida y vuelta (paso 5); y (4) kernel para obtener el valor final de campo (paso 6). A continuación se analizan las tres estrategias seguidas y los *kernels* definidos, que están representados esquemáticamente en la figura 2. En todos los casos, los datos relativos al conjunto de puntos son copiados a memoria de textura para beneficiarse de las ventajas que ofrece este tipo de memoria (caché fundamentalmente).

4.1 Primera Estrategia

La primera estrategia **(A)** es una implementación directa del algoritmo. Consiste en lanzar un hilo (dibujado como una flecha ondulada en la figura 2) para cada punto del campo y paso del algoritmo. La configuración de bloques (en este caso común a todos los *kernels*) tiene un tamaño de bloque $t_b = 128$ distribuidos en una rejilla de dimensiones $ceil(n_{puntos}/t_b) \times 1$. Cada resultado parcial (esto incluye las distancias, retardos y respuestas al impulso) es transferido a memoria de textura para su uso posterior. A pesar de doblar la velocidad obtenida en CPU, los requisitos de memoria eran muy grandes y se comprobó que era demasiado lento debido al gran tiempo consumido por las funciones de convolución de las dos últimas etapas.

4.2 Segunda Estrategia

La siguiente estrategia **(B)**, surge como solución directa al problema encontrado en la primera solución. Se optimizó por tanto el cálculo de las convoluciones, haciendo uso de una librería optimizada para el cálculo de FFTs (NVIDIA CUFFT). Respecto al resto de detalles, se continúa usando la misma configuración descrita en el punto anterior, realizando las convoluciones de los pasos 5 y 6 en el dominio de la frecuencia con FFTs. En este caso el problema surgido fue que las FFTs requerían de una gran cantidad de memoria, más de la disponible, y eso provocaba una disminución del rendimiento general. Además, la homogenización que exige las FFTs es ineficiente porque las respuestas espaciales al impulso tienen longitudes diversas.

4.3 Tercera Estrategia

Como consecuencia de los problemas encontrados en las dos primeras soluciones, se trató de mejorar el uso de memoria y acelerar el tiempo de cómputo de cada *kernel*, explotando el

poder de la GPU y diseñando una solución específica para cada paso acompañada del desarrollo de nuevas estructuras de datos que permitieran incrementar el uso de recursos del hardware:

- Kernel 1. Se modificó la estrategia de paralelización por puntos por otra que consiste en paralelizar por punto y por celda como se observa en la figura 2. El tamaño de bloque se ha fijado en 128 dando lugar a un rejilla de tamaño $ceil(n_{puntos}/t_b) \times n_{celdas}$. En ese sentido, las distancias y retardos se calculan muy rápido (raíz cuadrada, sumas y multiplicaciones) que se almacenan en vectores de tuplas (distancia, retardo) en memoria de textura.
- Kernel 2. Con el objetivo de reducir la memoria requerida para almacenar las respuestas al impulso en los pasos 3 y 4, el segundo *kernel* se divide en dos: el primero utiliza un algoritmo de ordenación bitónica y de suma, que tiene como objetivo ordenar las distancias de menor a mayor en función del primer campo de la tupla y sumar los retardos cuando las distancias son iguales, generando una estructura definida como $proto_{respuesta}$, que contiene sólo la información esencial de h (por ejemplo se excluyen los valores nulos). En este sentido, se calcula la $proto_{respuesta}$en emisión rápidamente; el segundo se encarga de calcular la respuesta completa en recepción lanzando un hilo por punto y por muestra de la h y almacenando el resultado en memoria de textura igualmente.
- Kernel 3. Para realizar la primera de las convoluciones se hace uso de las $proto_{respuestas}$ en emisión y las respuestas en recepción calculadas anteriormente. En este caso se ha implementado un algoritmo de convolución de tipo Input Side [13] paralelo, y se lanza un hilo por punto y por muestra de la estructura que almacena la respuesta al impulso en ida y vuelta que permite obtenerla de manera eficaz. Esto es así en parte gracias a que los vectores están ordenados lo que permite calcular directamente el valor correspondiente para la muestra.
- Kernel 4. Para el último paso, se ha desarrollado un algoritmo paralelo optimizado basado en la convolución de tipo Ouput Side [13] para hacer las convoluciones con la onda que permite obtener el resultado final para esa muestra al final de la ejecución del kernel (no es necesario almacenar el resultado, sólo interesa obtener el máximo). Se lanza un hilo por punto y por muestra de la convolución en ida-vuelta, computa el valor (minimiza los accesos de lectura y escritura a memoria) y lo guarda en memoria compartida. Una vez todos los hilos del mismo bloque han terminado su tarea, se obtiene el máximo por bloque y se realiza una reducción del conjunto de máximos por bloque para cada punto, calculando el máximo global y almacenándolo en el vector de resultados en memoria global.

5. RESULTADOS

Para testear los algoritmos paralelos desarrollados, se ha utilizado una de las últimas tarjetas, una NVIDIA GeForce GTX 295. Esta tarjeta dispone de dos GPUs en su interior, aunque por el momento sólo se ha utilizado una de ellas. Está formada por 240 cores con 1GB de memoria global. Se ha instalado en un PC Core 2 Quad y procesador Intel Q9450 2.66 GHz con 4GB de RAM. En resumen, los recursos disponibles en GPU superan los recursos de CPU en aproximadamente 60 veces.

Se ha llevado a cabo una comparación entre el tiempo de cómputo usando GPU y CPU, considerando un array lineal con 32 elementos separados $\lambda/2$ y con $16/\lambda$ de alto. Se han calculado 12900 puntos de campo. Las resoluciones espaciales y temporales se han variado desde 16 a 4096 celdas por elemento en el primer caso, y desde $\lambda/32c$ a $\lambda/128c$en el segundo caso. Los cálculos se han realizado utilizando precisión simple.

Se han desarrollado dos implementaciones en CPU. Ambas implementaciones presentan optimizaciones referentes al cálculo de convoluciones. La primera implementación CPU_TEMP realiza las operaciones de convolución en el dominio temporal basados en el mismo principio que los algoritmos diseñados para GPU, esto es, las convoluciones Input y Output Side. En la segunda implementación CPU_FREC se hace uso de FFTW, una librería conocida de cálculo de FFTs [12] que hace uso de las extensiones SSE y realiza un cálculo rápido de las FFTs. Se comparan únicamente con la implementación de la tercera estrategia GPU_TEMP por ser la que ofrece los resultados de mayor interés.

Resolución espacial		CPU_TEMP			CPU_FREC			GPU_TEMP		
Tamaño de la celda	Celdas por elemento	$\frac{\lambda}{32c}$	$\frac{\lambda}{64c}$	$\frac{\lambda}{128c}$	$\frac{\lambda}{32c}$	$\frac{\lambda}{64c}$	$\frac{\lambda}{128c}$	$\frac{\lambda}{32c}$	$\frac{\lambda}{64c}$	$\frac{\lambda}{128c}$
$\lambda/2$	16	71,8	143,6	287,2	23,8	24,8	29,3	0,64	1,04	2,26
$\lambda/4$	64	267,7	533,2	1070,2	102,6	104,1	106,7	2,08	2,72	4,62
$\lambda/8$	256	1117,9	2235,8	4471,4	259,1	260,2	265,3	8,36	9,32	11,94
$\lambda/16$	1024	4833,1	9665,3	19330,1	968,4	970,3	984,8	37,13	39,06	42,97
$\lambda/32$	4096	21364,3	42728,5	85456,2	3935,7	3919,4	3890,1	226,2	229,4	234,6

Tabla 1. Tiempo en segundos para calcular el campo acústico en CPU y GPU

La tabla 1 muestra los tiempos obtenidos usando la CPU y la GPU. En las dos primeras columnas se presentan los resultados para los dos algoritmos diferentes. Con respecto a la primera implementación CPU_TEMP, se trata obviamente de la más lenta. El tiempo de cómputo (debido fundamentalmente a las convoluciones temporales) se incrementa de manera lineal con la resolución temporal. En el segundo caso CPU_FREC no hay tantas diferencias en la variación de los intervalos de muestreo temporal debido al hecho de que los algoritmos en CPU hacen uso de FFTs y normalizan las longitudes de h al valor más cercano potencia de dos. Sin embargo, si el número de celdas por elemento incrementa, se produce un significativo incremento del tiempo de cómputo. Los tiempos obtenidos usando la GPU se muestran en la tercera columna donde se aprecian grandes diferencias en la velocidad de cómputo con respecto a las implementaciones en CPU. Todos los tiempos incluyen las transferencias a memoria global y la copia de resultados al host. En este caso, no se usan FFTs en GPU debido a los problemas explicados en la sección anterior y la relación entre las muestras temporales decrece a medida que el número de celdas por elemento se incrementa, a causa de un mayor uso del poder de la GPU.

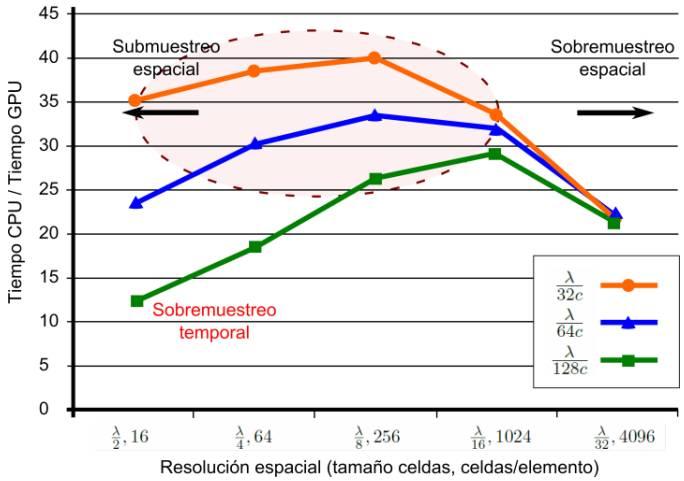

Figura 3. Ganancia en velocidad conseguida en el ejemplo de cálculo de campo acústico

Finalmente, la figura 3 muestra la ganancia en velocidad conseguida. La parte derecha del gráfico se corresponde con un alto número de celdas por elemento (sobremuestreo espacial) que produce un ligero decremento en la eficiencia de método. Por otro lado, en la parte

izquierda de la figura, que corresponde con un submuestreo espacial, la eficiencia en GPU no es óptima. Respecto a la resolución temporal, la línea con cuadrados corresponde a intervalos de tiempo muy cortos (sobremuestreo temporal), demandando una mayor cantidad de memoria y aumentando por tanto el tiempo de cómputo. Las líneas con triángulos y círculos corresponden a valores más lógicos. Los valores más adecuados para las resoluciones espaciales y temporales están incluidos dentro del área punteada, y como podemos observar, para estos valores el rendimiento del algoritmo en GPU es cerca de 40 veces más rápido que el rendimiento en CPU. Sin embargo, cabe destacar que en todos los casos, la eficiencia en GPU está más de un orden de magnitud por encima de la CPU.

6. CONCLUSIONES

Este trabajo se ha centrado en explorar el paralelismo de las GPUs para algoritmos de modelado en END. Utilizando una tarjeta gráfica sencilla equipada con tecnología NVIDIA CUDA, nuestros resultados experimentales demuestran que el tiempo de cálculo para el campo acústico se ve drásticamente reducido con respecto a implementaciones en CPU, consiguiendo en algunos casos una ganancia 40 veces más rápida que los mismos algoritmos ejecutados en PCs convencionales (8 segundos frente a más de 5 minutos). ésta es una solución económica con muy buena relación entre poder computacional y precio, escalable y que puede ser utilizada para acelerar muchos de los problemas de modelado en END diseñando estrategias similares para aplicarlas a más casos de estudio e incrementar el rendimiento de los mismos, facilitando implementaciones en tiempo real.

AGRADECIMIENTOS

Este trabajo está apoyado por el Ministerio de Ciencia e Innovación de España a través del proyecto DPI2010-19376 y la beca BES-2008-008675.

BIBLIOGRAFÍA

[1] B. D. Steinberg, Principles of Aperture and Array System Design, Wiley-Interscence,1976
[2] G. S. Kino, Acoustic Waves, devices, imaging and analog signal processing, Prentice,1987
[3] B. Piwakowsky and K. Sbai, A new approach to calculate the field radiated from arbitrary structured transducer arrays, IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control 46, 422 (Marzo 1999)
[4] Joon-Hyun Lee Sang-Woo Choi, A parametric study of ultrasonic beam profiles for a linear phased array transducer, IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control 47, 644 (Mayo 2000)
[5] Pinar Crombie, Peter A.J.Bascom and Richard S.C.Cobbold, Calculating the pulsed response of linear arrays Accuracy versus computational efficiency, IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control 44, 997 (Septiembre 1997)
[6] J.A. Jensen, A model for the propagation and scattering of ultrasound in tissue, J. Acoust. Soc. Am. N. 89 (1), 182 (1991)
[7] E.Lindholm, J.Nickolls, S.Oberman and J. Montrym, NVIDIA Tesla: A unified graphics and computing architecture, IEEE Micro 28, 39 (Marzo - Abril 2008)
[8] NVIDIA Corporation, NVIDIA CUDA 3.2 Programming Guide, Marzo 2011
[9] Zona CUDA, http://www.nvidia.com/object/cuda_home.html
[10] J. Nickolls, I. Buck, M. Garland and K. Skadron, Scalable parallel programming with CUDA, Queue 6, 40 (Marzo - Abril 2008)
[11] General-purpose computation using graphics hardware http://www.gpgpu.org
[12] Fastest Fourier Transform in the West, http://www.fftw.org
[13] Steven W. Smith, Digital Signal Processing, Analog Devices, páginas 110-123, 1998