CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Elsevier - Publisher Connector

SciVerse ScienceDirect
Disponible en www.sciencedirect.com

Revista Iberoamericana de Automática e Informática industrial 9 (2012) 282–289

www.elsevier.es/RIAI

Técnicas GPGPU para acelerar el modelado de sistemas ultrasónicos

D. Romero-Laorden*, O. Martínez-Graullera, C.J. Martín-Arguedas, M. Pérez, L.G. Ullate

Centro de Acústica Aplicada y Evaluación No Destructiva (CAEND CSIC-UPM) Arganda del Rey, Madrid, España.

Resumen

El desarrollo de sistemas de simulación de campo acústico en tiempo real para aplicaciones de Evaluación no Destructiva ultrasónica constituiría una herramienta muy útil tanto para la planificación de las inspecciones como para la interpretación de los resultados de evaluaciones in-situ. Sin embargo, son algoritmos que requieren una alta capacidad de cómputo, no tanto por su complejidad sino por el gran número de puntos a analizar, lo que limita su uso al laboratorio sobre estaciones de trabajo de altas prestaciones. Los recursos de paralelización que actualmente ofrecen los sistemas informáticos, como son los procesadores multicore o las técnicas GPGPU, constituyen una oportunidad muy interesante para el desarrollo de este tipo de aplicaciones. Este trabajo analiza el modelo de paralelización de ambas alternativas con objeto de desarrollar un sistema portable de simulación de campo ultrasónico para tiempo real. Se describen por tanto los cambios en el algoritmo de cálculo de campo acústico para adaptarlo a una estrategia GPGPU y se valora el coste computacional de ambas implementaciones. *Copyright © 2012 CEA. Publicado por Elsevier España, S.L. Todos los derechos reservados.*

Palabras Clave:
algoritmos paralelos, sistemas ultrasónicos, GPGPU

1. Introducción

Actualmente las técnicas de Evaluación No Destructiva o 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 (Steinberg, 1976; Kino, 1987).

Hoy en día se tiende a inspeccionar estructuras de geometría cada vez más compleja, mediante agrupaciones de transductores (conocidos como 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 (Piwakowsky and Sbai, 1999; Choi, 2000; Pinar Crombie, 1997) 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. Sofisticados paquetes de software como el CIVA (Cea, 2003) han sido desarrollados con este objetivo. Sin embargo su uso esta principalmente limitado al laboratorio, ya que la simulación del campo acústico es un proceso que requiere de una gran potencia de cálculo y tiene un coste computacional alto. No tanto por la complejidad del modelo de simulación sino por el gran número de puntos sobre los que se debe calcular y por los altos requerimientos de resolución que exige la zona de campo cercano. Este hecho limita la eficacia de los modelos de simulación como herramienta de campo ya que el evaluador no puede hacer uso de ella durante la inspección, donde le permitiría adaptar la planificación a las circunstancias reales o facilitaría la interpretación de los resultados. Se han propuesto distintos modelos basados en aproximaciones que reducen el tiempo de computación (Jensen, 1991) pero los resultados aún están fuera del tiempo real. Sin embargo, la mayoría de los métodos de modelado en END presentan características comunes: el algoritmo computacional utilizado se aplica de manera iterativa a una gran cantidad de puntos, aumentando el tiempo de ejecución y haciendo por tanto muy difícil su uso en sistemas que demandan tiempo real. Sin

*Autor en correspondencia.
Correo electrónico: david.romero@csic.es (D. Romero-Laorden)

© 2012 CEA. Publicado por Elsevier España, S.L. Todos los derechos reservados
http://dx.doi.org/10.1016/j.riai.2012.05.002

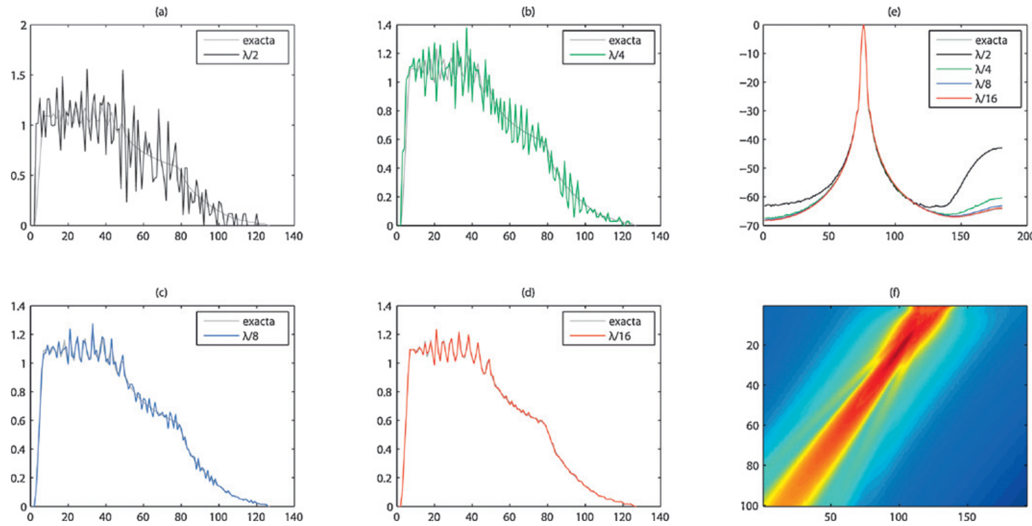

Figura 1: Ejemplo de la discretización de la apertura. Comparativa de la respuesta exacta con (a) la respuesta discreta con resolución espacial $\frac{\lambda}{2}$ (b) la respuesta discreta con resolución espacial $\frac{\lambda}{4}$ (c) la respuesta discreta con resolución espacial $\frac{\lambda}{8}$ (d) la respuesta discreta con resolución espacial $\frac{\lambda}{16}$ (e) valor del campo acústico para todos los casos expuestos (f) campo de presión generado sobre un plano situado en frente del array. Resolución temporal en todos los casos es de $\frac{\lambda}{32c}$.

embargo, estas características los hacen idóneos para la computación paralela.

La progresiva popularización de sistemas que explotan el paralelismo como vía para aumentar la capacidad de procesamiento, como los procesadores multicore o las técnicas GPGPU, suponen alternativas interesantes para el desarrollo de este tipo de problemas. Ambas plataformas plantean soluciones para aplicaciones que precisen alta capacidad de procesamiento a bajo coste. En nuestro caso nos permiten desarrollar soluciones para cálculo en tiempo real sacando la simulación del laboratorio y llevándolo a las zonas de trabajo, aumentando la capacidad del evaluador para determinar la gravedad del problema detectado. Sin embargo, debido a la naturaleza de ambos procesadores los estilos de paralelización son radicalmente distintos e involucran modelos de programación muy diferentes.

El modelo multicore basado en la integración de procesadores de altas prestaciones en un sólo chip, cada uno con su propia memoria cache L1 y una serie de recursos compartidos entre todos (cache L2, etc). Estos sistemas, diseñados para el acceso aleatorio a memoria, permiten implementar estructuras de datos complejas y un modelo de desarrollo que explota el paralelismo a nivel de *thread* (thread-level parallelism), que en nuestro caso es fácil desarrollar a nivel de punto espacial.

En el modelo GPGPU (*General Purpose Computation on Graphics Processing Units*) donde el número de procesadores es mucho mayor, pudiendo llegar a centenares, pero con unos recursos mucho más reducidos (pocos registros internos, limitaciones aritméticas) y particulares modos de acceso a memoria (E.Lindholm and Montrym, 2008), la paralelización no es un proceso sencillo. Para adaptar los accesos a memoria y las estructuras de datos a las características de la GPU es necesario un análisis complejo del problema a paralelizar. Aunque ya existen herramientas que facilitan tanto el desarrollo como la portabilidad del código (Nvidia, 2012b; J. Nickolls and Skadron, 2008), un conocimiento de esta arquitectura permite una mejor optimización del algoritmo.

Pese a que cualquier algoritmo que es implementable en una CPU lo es también en una GPU, éstas implementaciones no son igual de eficientes. Este trabajo analiza ambos modelos de desarrollo para el problema de cálculo de campo acústico, basándonos en el modelo de la respuesta espacial al impulso desarrollada por Piwakowsky (Piwakowsky and Sbai, 1999). Se analiza en profundidad la solución para GPU, por sus peculiaridades y por el cambio que supone frente al modelo de paralización convencional. El objetivo es reducir el alto coste asociado a los algoritmos y facilitar su implementación en sistemas que permitan a la larga el desarrollo de soluciones de tiempo real, lo que de alguna forma permitan desarrollar aplicaciones para operar en campo.

2. Modelado en END

La expresión general de la presión acústica generada por un array que trabaja en emisión y recepción viene dada por (Jensen, 1991):

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

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

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

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

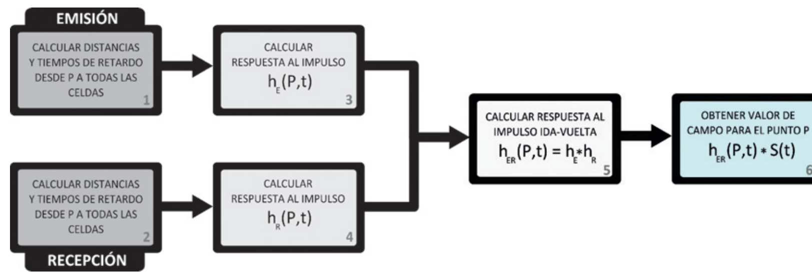

Figura 2: Algoritmo iterativo de Piwakowsky. El cálculo del campo acústico en emisión y recepción requiere seis pasos

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 (Piwakowsky and Sbai, 1999), que hace una computación directa de la integral de Rayleigh por medio de muestras temporales a intervalos Δt y discretiza las superficie 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 $c\Delta t$ (Piwakowsky and Sbai, 1999). Así, la contribución a la respuesta al impulso correspondiente al i-ésimo elemento del array en el instante $t = t_s$ es:

$$h_i(t = t_S) = \frac{1}{\Delta t} \sum_j b_j \qquad t_S - \Delta t/2 \leq t_j \leq t_S + \Delta t/2 \tag{5}$$

$$b_j = \frac{a_i \Delta S}{2\pi R_j} \text{ y } t_j = \frac{R_j}{c} + T_i \tag{6}$$

donde T_i denota el retardo de focalización en emisión o recepción y $\frac{R_j}{c}$ es el tiempo de propagación. De este modo, el valor del campo resultante viene dado por la siguiente ecuación:

$$P(\vec{x}) = max_t(abs(p(\vec{x}, t))) \tag{7}$$

A pesar de ser un modelo aproximado, el método de Piwakowsky tiene la peculiaridad de poder aplicarse a cualquier tipo de aperturas y geometrías. El error de cómputo entre los patrones discretos y la solución analítica exacta ha sido estudiado en (Piwakowsky and Sbai, 1999), concluyendo que hay una alta precisión cuando tanto la resolución espacial como temporal son del orden de $\frac{\Delta x}{\lambda} = 0{,}01$ y $\Delta t f = 0{,}01$ respectivamente, donde f es la frecuencia central de la onda ultrasónica.

Los efectos acerca del problema de la discretización para el caso de un array de 32 elementos (tabla 1) pueden apreciarse en la figura 1. En concreto, las figuras (a), (b), (c) y (d) muestran las diferencias existentes entre las respuestas obtenidas en el cálculo de la respuesta al impulso por el método de Piwakowsky para una resoluciones de $\frac{\lambda}{2}$, $\frac{\lambda}{4}$, $\frac{\lambda}{8}$ y $\frac{\lambda}{16}$ respectivamente con respecto a la respuesta propuesta por Piwakowsky calculadas sobre un punto situado a una profundidad de 10mm para una focalización deflectada 15° en elevación. Como vemos, el incremento de la resolución espacial ayuda a reducir el error introducido por la discretización, aunque al mismo tiempo el coste computacional aumenta sustancialmente. La mayoría de los errores introducidos por la discretización son ruido de alta frecuencia, y por tanto son eliminados por la onda al actuar ésta como un filtro. Este efecto de filtro se hace patente sobre los resultados de la figura 1, donde se muestra que una resolución espacial de $\frac{\lambda}{16}$ es adecuada y ofrece una solución de compromiso entre el número de puntos en la apertura y la exactitud del resultado. Esto es así ya que la propuesta de Piwakowsky es fuertemente restrictiva al definir estos límites para determinar la exactitud sobre la respuesta espacial al impulso y no sobre la onda de presión resultante. Por otro lado, en la figura 1(e) se muestra el valor de campo para todas las resoluciones en decibelios donde los errores de la discretización se hacen evidentes principalmente para el caso de $\frac{\lambda}{2}$. Por último, el campo de presión sobre un plano situado en frente del array se muestra en la figura 1(f) como resultado de la simulación.

2.1. *Algoritmo computacional*

La figura 2 describe esquemáticamente el algoritmo computacional utilizado para el cálculo del campo acústico usando el método de Piwakowsky. Más en detalle, se muestran las etapas necesarias para calcular el valor del campo en cada punto espacial P, siendo por tanto, un proceso iterativo que se aplica sobre todo el conjunto de puntos del espacio. A continuación se describen cada una de las etapas:

- En los pasos 1 y 2, se calculan las distancias desde todas las celdas pertenecientes a la apertura a cada punto del espacio P en emisión y recepción respectivamente. Además, se calcula información adicional 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).

- Después, en los pasos 3 y 4, se calcula la respuesta al impulso del array en emisión $h_E(P,t)$ y en recepción $h_R(P,t)$ (ecs. 2 y 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).
- Finalmente, se obtiene el campo acústico en ida y vuelta, realizando la convolución entre $h_{ER}(P,t)$ y la señal emitida (ec. 1).

Por último, cabe remarcar un aspecto importante. La longitud de las respuestas al impulso h depende fundamentalmente de la distribución de los elementos del array en el tiempo. Estos valores pueden ser corregidos mediante la aplicación de lentes de focalización, de tal manera que acortarán la longitud de las respuestas al impulso en el foco mientras que se verán alargadas fuera de él.

2.2. *Modelo de cálculo*

Para poder evaluar las capacidades y ventajas de la computación paralela, se ha simulado un ejemplo de cálculo del campo generado en emisión y recepción con un array lineal emitiendo en un medio homogéneo (en este caso, agua). Las características del modelo están resumidas en la tabla 1. Como vemos, el array tiene $N = 32$ elementos con una separación entre elementos (*pitch*) $d = \frac{\lambda}{2}$. El ancho de los elementos es $a = d$ y su altura es $b = 10\lambda$. Los elementos emiten pulsos de 50 % de ancho de banda. Para la evaluación de los resultados se ha calculado el máximo de presión en 19300 puntos pertenecientes a un plano situado frente al array.

Tabla 1: Parámetros del modelo de simulación

Medio	Agua
Velocidad del medio	1540 m/s
Tamaño del array	32 elementos
Ancho de los elementos (*pitch*)	$\frac{\lambda}{2}$mm
Altura de los elementos	10λmm
Frecuencia central	3 MHz
Ancho de banda	50 %
Número de puntos de campo	19300 (48.25mm×101mm)
Resolución espacial	(res_x, res_z)=(0.25, 1.01)

3. Infraestructuras de cómputo paralelo

Antes de discutir el diseño de nuestro algoritmo, es necesario comentar brevemente algunos detalles relativos a las arquitecturas multicore en CPU como en GPU. En estos últimos años, una de las tendencias dominantes en las arquitecturas de microprocesadores ha sido el continuo incremento del paralelismo a nivel de chip. Hoy en día, vemos algo común trabajar con CPUs compuestas de varios núcleos de proceso (*cores*) y probablemente la tendencia seguirá incrementando aún más el paralelismo con nuevos *chips* formados por múltiples núcleos. Un procesador multicore CPU, está compuesto por dos o más procesadores independientes, encargados de leer y ejecutar las instrucciones del programa. Estas instrucciones son las instrucciones típicas de una CPU (como sumar, mover datos o instrucciones condicionales) pero con la ventaja de que los distintos núcleos son capaces de ejecutar varias instrucciones al mismo tiempo aumentando considerablemente la velocidad general de los programas que son susceptibles de computación paralela.

En cambio, las unidades de procesamiento gráficos o GPUs (*Graphics Processor Units*) son arquitecturas muy poderosas computacionalmente hoy día que pueden estar formadas por centenares de *cores*. De hecho, es común tratar con GPUs de 128, 240, 256 o 512 *cores* (arquitectura Fermi) aunque las actuales NVIDIA GPUs (Nvidia, 2012a) ya pueden contener hasta 1358 *cores* (arquitectura Kepler) y éstas pueden ser programadas directamente en código C usando las tecnologías CUDA (Nvidia, 2012b; J. Nickolls and Skadron, 2008) u OpenCL.

Con el objetivo de obtener el mayor provecho posible de una GPU, se ha considerado apropiado describir las características generales de la arquitectura hardware de una tarjeta gráfica para su programación en CUDA. Para tal fin, una comparativa entre las arquitecturas de una CPU multicore y una GPU se muestra esquemáticamente en la figura 3. Como vemos, una GPU se compone de una serie de multiprocesadores (MPs) (desde 1 a 30 dependiendo de la arquitectura considerada) que a su vez están compuestos por varios procesadores escalares (SPs) (varían entre 8 a 192 según el tipo de GPU). Además, cada MP contiene una memoria on-chip (memoria compartida) con muy baja latencia y alto ancho de banda, similar a una cache L1 de una CPU. Las diferencias son apreciables y tanto la CPU como la GPU se encargan de gestionar su propia memoria, siendo la primera conocida como memoria del *host* (que es directamente la memoria RAM del PC) mientras que la segunda, llamada memoria del dispositivo, *device* o global, es la memoria de la tarjeta gráfica que es usada por todos los multiprocesadores de la GPU.

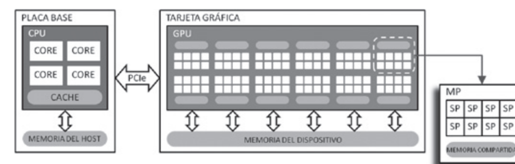

Figura 3: CPU de cuatro-cores frente a una tarjeta gráfica de 240 cores (30 MPs de 8 SPs cada uno)

El concepto fundamental del modelo de programación CUDA es trabajar con cientos de hilos que ejecutan la misma función pero con diferentes datos. Por tanto, una aplicación se organiza como un programa secuencial en la CPU que incluye funciones especiales paralelas, llamadas *kernels*, que son ejecutadas varias veces sobre diferentes datos. El programador organiza los datos a procesar por un *kernel* en estructuras lógicas que consisten en bloques de hilos y rejillas de bloques. Así, una rejilla o *grid* es una estructura 1D, 2D o 3D de bloques y un bloque es una estructura 1D, 2D o 3D de hilos. Únicamente se puede definir un *grid* dentro de un *kernel*. Asociado a cada función paralela existen una serie de parámetros que es necesario conocer si se quiere maximizar el rendimiento, siendo los más

importantes el número de registros utilizados por cada hilo y la ocupación de los MPs. Para una completa descripción sobre la arquitectura y el modelo de programación puede consultarse la extensa documentación disponible (Nvidia, 2012b).

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 la computación paralela.

En este trabajo se evalúan tres implementaciones distintas, siendo dos de ellas desarrolladas exclusivamente en CPU. La primera de las soluciones que se ha desarrollado es la puramente tradicional, que utiliza un único núcleo como cerebro para el cálculo el campo acústico. En todos los casos, los algoritmos se han programado en C y ejecutados en Matlab® por simplicidad para el desarrollo.

4.1. Paralelización multi-core

Trasladar el algoritmo de cómputo de campo a una arquitectura multicore en CPU es un proceso relativamente sencillo. Por ello, la paralelización ha consistido en utilizar un hilo de ejecución en CPU por cada uno de los puntos del espacio, distribuyendo el trabajo entre el máximo de procesadores disponibles en nuestro equipo. Para tal fin, y puesto que todas las librerías de campo habían sido desarrolladas con anterioridad utilizando el entorno de cálculo matemático Matlab®, la paralelización se ha realizado usando las herramientas para cómputo paralelo que éste entorno nos proporciona, consiguiendo de una manera rápida una buena distribución de los recursos disponibles y de la memoria.

4.2. Paralelización en GPU

Desafortunadamente, una implementación directa de los algoritmos diseñados para CPU a GPU no es normalmente la mejor opción por lo que es necesario definir una estrategia de paralelización apropiada de los algoritmos para poder obtener el máximo beneficio. Las notables diferencias que existen entre las arquitecturas CPU y GPU hacen necesario comentar brevemente algunas consideraciones sobre la computación en GPU.

En primer lugar, 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 la memoria del dispositivo es lento y es recomendable por tanto usar algunos otros mecanismos de acceso, como la memoria de textura (que es cacheable y rápida) o la memoria compartida (disponible en cada MP y por tanto accesible únicamente entre hilos de un mismo bloque) 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.

En este trabajo se han evaluado distintas alternativas. Debido al hecho de que el algoritmo se aplica a cada punto del espacio, la paralelización directa sería por puntos al igual que sucede en el caso multicore calculando todos los puntos simultáneamente. Pero la arquitectura es distinta, y tenemos mucho más paralelismo, por lo que 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.

Tabla 2: Configuraciones de los distintos *kernels* para cada estrategia, tamaño de bloque t_b, número de bloques en x e y n_{bx} y n_{by}, número de puntos n_p, número de registros por *kernel* n_{reg}, ocupación de los multiprocesadores $ocup$, número de celdas n_c, número de muestras de las respuestas en ida-vuelta $m_{h_{iv}}$ y número de muestras de la convolución con la onda m_c.

Kernels	A	B
1	$t_b = 128$ $n_{bx} = n_p/t_b$ $n_{by} = 1$ $n_{reg} = 15$ $ocup = 100\,\%$	$t_b = 128$ $n_{bx} = n_c/t_b$ $n_{by} = n_p$ $n_{reg} = 13$ $ocup = 100\,\%$
2	$t_b = 128$ $n_{bx} = n_p/t_b$ $n_{by} = 1$ $n_{reg} = 19$ $ocup = 75\,\%$	$t_b = 64$ $n_{bx} = n_h/t_b$ $n_{by} = n_p$ $n_{reg} = 5$ $ocup = 100\,\%$
3	$t_b = 64$ $n_{bx} = n_p/t_b$ $n_{by} = 1$ $n_{reg} = 17$ $ocup = 93\,\%$	$t_b = 128$ $n_{bx} = m_{h_{iv}}/t_b$ $n_{by} = n_p$ $n_{reg} = 12$ $ocup = 100\,\%$
4	$t_b = 128$ $n_{bx} = n_p/t_b$ $n_{by} = 1$ $n_{reg} = 19$ $ocup = 75\,\%$	$t_b = 128$ $n_{bx} = m_c/t_b$ $n_{by} = n_p$ $n_{reg} = 10$ $ocup = 100\,\%$

Para el desarrollo de los algoritmos paralelos en GPU se han diseñado 4 *kernels* bien diferenciados que cubren cada una de la etapas definidas anteriormente: *Kernel 1*, para calcular las distancias y tiempos de retardo desde cada punto espacial a todas las celdas en emisión y recepción (etapas 1 y 2); *Kernel 2*, que computa la respuesta espacial al impuso con una implementación paralela del método de Piwakowsky en emisión y recepción (etapas 3 y 4); *Kernel 3*, que realiza la convolución en ida y vuelta (etapa 5); y *Kernel 4* obtiene el valor final de campo (etapa 6).

A partir de la definición operativa de estos *kernels*, la paralelización se puede realizar a dos niveles que corresponden con las dos estrategias identificadas en la figura 4. La primera sigue el patrón convencional seguido para la implementación multicore (paralelización por punto espacial), lo que hemos definido como un modelo de grano grueso. La segunda, es de grano más fino y la paralelización se lleva a cabo a partir de las celdas espaciales de la apertura y los tamaños de los vectores. Asimismo, en la tabla 2 se muestran los distintos parámetros asociados a ambos niveles, relativos al tamaño de bloque, número de bloques, la ocupación de los multiprocesadores, registros, etc.

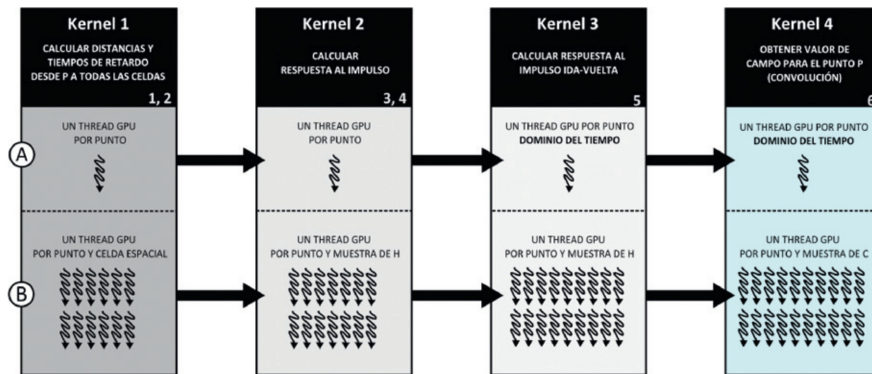

Figura 4: Esquemas de las dos estrategias estudiadas para GPU en este trabajo: (A) un thread por punto de campo, convolución en el dominio del tiempo, (B) solución paralela específica adaptada a cada etapa del algoritmo

4.3. Paralelización de grano grueso

Como se ha comentado antes, está primera estrategia consiste en lanzar un *thread* (simbolizado con una flecha ondulada en la figura 4) para cada punto del campo y cada paso del algoritmo. El tamaño del *grid*, que en este caso es idéntico a todos los *kernels*, tiene unas dimensiones de $(ceil(\frac{n_p}{t_b}) \times 1)$ bloques de tamaño $t_b = 128$. Esto proporciona una buena ocupación de los multiprocesadores pero que no llega a ser del 100 % en todos los casos (ver tabla 2). Cada resultado parcial (distancias, retardos, respuestas al impulso) es transferido a memoria de textura (que es cacheable) para usarlo en las etapas posteriores. De este modo, se optimizan los accesos a memoria y se gana en velocidad. 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 como factores más determinantes.

4.3.1. Paralelización de grano fino

A raíz de los problemas identificados se intentó mejorar el uso de memoria y acelerar el tiempo de cómputo de cada *kernel*, explotando el poder de la GPU, diseñando una solución paralela a más bajo nivel (esto es, para cada etapa del algoritmo de la figura 2.1) y desarrollando nuevas estructuras de datos que permitieran incrementar el uso de recursos del hardware. A continuación se detallan las optimizaciones y cambios llevados a cabo en cada en cada *kernel*:

- *Kernel 1*. La paralelización se hace por punto y por celda como se observa en la figura 4. El tamaño de bloque se ha fijado en $t_b = 128$ en un *grid* de dimensión $(ceil(\frac{n_p}{128}) \times n_c)$. En ese sentido, el *kernel* se ha simplificado hasta la mínima expresión, calculando las distancias y los retardos muy rápido (raíz cuadrada, sumas y multiplicaciones). Los resultados 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 - figura 2.1, el segundo *kernel* tiene dos funciones: la primera, tiene como objetivo ordenar las distancias de menor a mayor en función del primer campo de la tupla (ordenación bitónica) y sumar los retardos cuando las distancias son iguales, generando una estructura que llamaremos $Proto_{respuesta}$ que contiene únicamente la información esencial de h (se excluyen los valores nulos). De esta manera, la $Proto_{respuesta}$ en emisión se calcula muy rápido. A continuación, la respuesta completa en recepción se computa lanzando un *thread* por punto y por muestra de la h. Ambos resultados se almacenan en memoria de textura para beneficiarnos del acceso a memoria más eficiente.

- *Kernel 3*. La primera convolución utiliza las $Proto_{respuesta}$ en emisión y las respuestas en recepción calculadas en la etapa anterior. En este caso, el algoritmo desarrollado sigue el modelo *Input Side* (Smith, 1998) paralelo, donde se lanza un *thread* por punto y por muestra de la estructura que almacena la respuesta al impulso en ida y vuelta. De este modo, la calculamos de manera eficaz, gracias a que los vectores están ordenados, lo que permite acceder directamente el valor correspondiente para la muestra.

- *Kernel 4*. Para el último paso, se ha desarrollado un algoritmo paralelo optimizado en GPU y que está basado en el modelo de convolución *Ouput Side* (Smith, 1998). Calcula por tanto las convoluciones con la onda y permite obtener el resultado final para esa muestra al final de la ejecución del *kernel*, no siendo necesario almacenar el resultado pues sólo nos interesa obtener el máximo. Para ello, se lanza un *thread* por punto y por muestra de la convolución en ida-vuelta, se computa el valor (minimizando los accesos de lectura y escritura a memoria) y se almacena en memoria compartida. Una vez todos los *threads* 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 obteniendo así el valor final del campo.

Tabla 3: Tiempo en segundos para calcular el campo acústico en 19300 puntos de campo para distintas configuraciones en CPU y GPU

Resolución Espacial		**CPU (1 core)**			**CPU (4 cores)**			**GPU**					
Tamaño de la celda	Puntos en la apertura	Resolución Temporal			Resolución Temporal			Resolución Temporal			Iteraciones		
		$\frac{\lambda}{32c}$	$\frac{\lambda}{64c}$	$\frac{\lambda}{128c}$	$\frac{\lambda}{32c}$	$\frac{\lambda}{64c}$	$\frac{\lambda}{128c}$	$\frac{\lambda}{32c}$	$\frac{\lambda}{64c}$	$\frac{\lambda}{128c}$	$\frac{\lambda}{32c}$	$\frac{\lambda}{64c}$	$\frac{\lambda}{128c}$
$\frac{\lambda}{2}$	1280	31.42	32.50	38.12	9.58	11.25	12.32	0.51	1.44	4.94	1	1	1
$\frac{\lambda}{4}$	2560	71.70	74.60	83.10	20.25	21.89	23.18	1.15	2.34	6.55	2	4	4
$\frac{\lambda}{8}$	5120	224.35	231.30	240.14	68.75	69.76	77.62	3.27	5.75	10.66	8	12	12
$\frac{\lambda}{16}$	10240	875.00	889.45	912.36	261.30	268.64	284.28	14.38	15.32	31.80	20	24	28
$\frac{\lambda}{32}$	20480	3241.42	3256.10	3289.51	984.10	1001.80	1019.60	91.12	105.7	178.60	48	56	64

Cabe por último comentar en ambos casos, que los datos correspondientes al modelo de simulación (esto es, el array y los puntos de campo) se almacenan en memoria constante de la GPU para conseguir un acceso lo más rápido posible. Respecto a la tabla 2, como vemos la paralelización de grano fino hace que los multiprocesadores estén al 100 % de actividad, no como sucedía en la solución directa. Y además, el número de registros usados en cada *kernel* también se ha visto reducido, dando como efecto una mejor optimización de cada una de las etapas que se refleja significativamente en el tiempo de cómputo.

5. Discusiones y resultados

Para testear los algoritmos paralelos desarrollados, se ha utilizado una tarjeta NVIDIA Quadro 4000 que contiene 256 cores de proceso con 2GB de memoria global. Se ha instalado en un PC Core 2 Quad y procesador Intel Q9450 2.66 GHz con 4GB de RAM.

Las implementaciones de los diversos algoritmos se han realizado desde el entorno de programación MATLAB® en su versión R2010a gracias a la interoperabilidad que CUDA proporciona con estos entornos de trabajo. Se ha llevado a cabo una comparación entre el tiempo de cómputo usando las tres configuraciones descritas: CPU (1 core), CPU (4 cores) y GPU para el modelo de simulación descrito en la sección 2.2. Se han calculado por tanto 19300 puntos de campo. Las resoluciones espaciales y temporales se han variado desde $\frac{\lambda}{2}$ a $\frac{\lambda}{32}$ y desde $\frac{\lambda}{32c}$ a $\frac{\lambda}{128c}$ respectivamente. Los cálculos se han realizado utilizando precisión simple. Un análisis exhaustivo acerca de la precisión en el cálculo de campo acústico sobre hardware gráfico está descrito en (Romero-Laorden, 2011) concluyendo que no hay diferencias apreciables entre el uso de precisión simple o doble en este tipo de aplicaciones.

La tabla 3 muestra los tiempos obtenidos para la solución monocore, multicore y GPU. Los tiempos obtenidos usando la GPU incluyen las transferencias a memoria global y la copia de resultados al *host*. Además, el límite de memoria impuesto por la GPU Quadro 4000 nos obliga a dividir nuestro cómputo lanzando grupos más pequeños de puntos, puesto que la memoria necesaria para almacenar los resultados parciales puede llegar a ser muy grande. Esto significa que irremediablemente tendremos una o más iteraciones dependiendo del caso como se puede ver en la última columna de la tabla 3.

Con respecto a las dos primeras implementaciones - CPU (1 core) y CPU (4 cores) - la primera es obviamente la más lenta al no existir ningún tipo de paralelización. En general, la tabla 3 indica que los tiempos de la solución multicore dividen entre 3 y 4 veces el tiempo de ejecución de la solución monocore. Para las soluciones que explotan el paralelismo por punto espacial, el aumento del número de puntos en la apertura supone un aumento significativo en el coste mucho más importante que al incrementar la resolución temporal. Esto se debe en parte al hecho de que las FFTs de los algoritmos en CPU trabajan con respuestas al impulso cuyas longitudes son normalizadas al valor más cercano potencia de dos. Sin embargo, esto no se produce en la solución GPU donde el aumento más importante en el coste se produce al incrementar el tiempo, disminuyendo el rendimiento global debido a un mayor número de ejecuciones.

Para los valores de resolución propuestos por Piwakowsky, los tiempos son de más de 7 horas en monocore y más de 2 en el caso multicore. Dado el filtro que introduce el pulso gaussiano de excitación mostrado en la sección 2 para el ejemplo desarrollado, estos requerimientos son excesivos siendo posible obtener un buen resultado para $\frac{\lambda}{16}$ y $t = \frac{\lambda}{32c}$ que corresponde con tiempos de 875.00, 261.30 y 15.32.

Finalmente, las figuras 5(a) y 5(b) muestran la ganancia en velocidad conseguida. La parte derecha de ambos gráficos corresponde con un alto número celdas en la apertura (más de 15000 celdas) que produce un decremento en la eficiencia de método. La causa principal se debe a que el número de iteraciones en estos casos es mayor debido a las limitaciones de memoria. Por otro lado, la parte izquierda de las figuras corresponde con un submuestreo espacial, donde no se obtiene tanto provecho en GPU. Respecto a la resolución temporal, la línea con cuadrados corresponde a intervalos de tiempo muy cortos (sobremuestreo temporal), demandando una mayor cantidad de memoria que aumenta el tiempo de ejecución. Las líneas con triángulos y círculos corresponden a valores más lógicos, siendo los más adecuados para resoluciones de $\frac{\lambda}{8}, \frac{\lambda}{16}$ y $\frac{\lambda}{32c}$ y $\frac{\lambda}{64c}$. Como podemos observar, para estos valores el rendimiento del algoritmo en GPU llega a ser de hasta 20 veces más rápido respecto a una implementación multicore en CPU.

6. Conclusiones

Este trabajo se ha centrado en explorar el paralelismo que se ofrecen en los sistemas de consumo para el desarrollo de algoritmos de modelado en END con vía a la implementación de sistemas de tiempo real. Así, se han comparado tres implementaciones de cálculo de campo acústico para el modelo de la

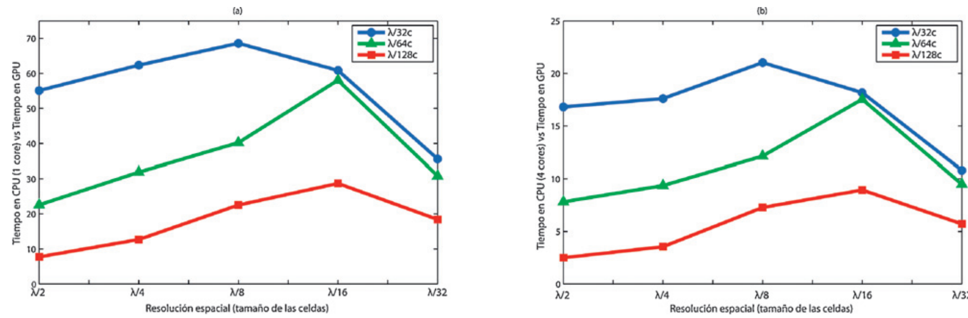

Figura 5: (a) Ganancia en velocidad CPU (1 core) vs GPU (b) Ganancia en velocidad CPU (4 cores) vs GPU

respuesta espacial al impulso basándonos en la solución aproximada de Piwakowsky. La solución convencional que correspondería a un sólo núcleo; la paralelización de cálculo a nivel de puntos de campo que corresponde a un desarrollo multicore; y por último la solución para GPU que explota el paralelismo dentro de las cuatro etapas del algoritmo. La paralelización multicore es particularmente sencilla ya que permite utilizar el algoritmo original respecto al monocore, y produce una mejora directamente proporcional al número de cores involucrados. Utilizando únicamente 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 (monocore y multicore), consiguiendo en algunos casos una ganancia de hasta 70 veces más rápida para el sistema monocore y de 20 para el sistema multicore. Los resultados demuestran que la GPU se adapta particularmente bien a los algoritmos con un alto grado de paralelismo, que no precisan necesidad de estructuras de datos complejas, y que exigen una alta intensidad aritmética. Ésta es una solución económica con muy buena relación entre potencia y coste. Además, es escalable y puede ser utilizada para acelerar muchos otros problemas de modelado en END, facilitando el desarrollo de implementaciones para tiempo real. El incremento en velocidad conseguido por la técnicas GPGPU es especialmente significativo si el número de hilos de ejecución y los recursos asociados se mantienen dentro de los límites físicos de la tarjeta y el número de ejecuciones es reducido.

English Summary

Using GPGPU techniques to accelerate modelling of ultrasonic systems.

Abstract

The development of acoustic field simulation in real time for non destructive ultrasonic evaluation applications would be an useful tool for both the planning and evaluation of inspections in-situ. However, they are algorithms which require high computing power, not due to their complexity but because of the large number of points to be analysed, which limits their use to laboratory workstations for high performance. The parallelization resources currently available in computer systems, such as multicore processors and GPGPU techniques, are a very interesting chance for the development of such applications. This work analyses the parallelization model of both alternatives in order to develop a portable ultrasonic field simulation system for real-time. The changes for both algorithms are described, in order to adapt it to GPGPU philosophy and a estimation of the computational cost of both implementations is given.

Keywords:

parallel algorithms, ultrasonic systems, GPGPU

Agradecimientos

Este trabajo está apoyado por el Ministerio de Economía y Competitividad a través del proyecto DPI2010-19376 y la beca BES-2008-008675.

Referencias

Cea, 2003. Civa: Simulation software for non destructive testing.

Choi, J.-H. L. S.-W., 2000. A parametric study of ultrasonic beam profiles for a linear phased array transducer. IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control 47, 644–650.

E.Lindholm, J.Nickolls, S., Montrym, J., 2008. Nvidia tesla: A unified graphics and computing architecture. IEEE Micro 28 (2), 39–55.

J. Nickolls, I. Buck, M. G., Skadron, K., 2008. Scalable parallel programming with cuda. Queue 6 (2), 40–53.

Jensen, J., 1991. A model for the propagation and scattering of ultrasound in tissue. J. Acoust. Soc. Am. N. 89 (1), 182–190.

Kino, G. S., 1987. Acoustic Waves, devices, imaging and analog signal processing. Prentice-Hall.

Nvidia, 2012a. Gtx680 kepler white paper.

Nvidia, Enero 2012b. Guía de Programación de CUDA 4.1.

Pinar Crombie, e., 1997. Calculating the pulsed response of linear arrays accuracy versus computational efficiency. IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control 44, 997–1009.

Piwakowsky, B., Sbai, K., 1999. A new approach to calculate the field radiated from arbitrary structuredtransducer arrays. IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control 46 (2), 422–439.

Romero-Laorden, David, e., Mar. 2011. Field modelling acceleration on ultrasonic systems using graphic hardware. Computer Physics Communications 182 (3), 590–599.
DOI: 10.1016/j.cpc.2010.10.032

Smith, S. W., 1998. Digital Signal Processing. Analog Devices.

Steinberg, B. D., 1976. Principles of Aperture and Array System Design. Wiley-Interscience.