Acelerando los momentos de Zernike sobre Kepler

Antonio Ruiz, Manuel Ujaldón
Departamento de Arquitectura de Computadores
Universidad de Málaga, España
E-mail: {antruiz,ujaldon}@uma.es

***Abstract*—Este trabajo analiza las características más avanzadas de la arquitectura Kepler de Nvidia, principalmente el paralelismo dinámico para el lanzamiento de kernels desde la GPU y la planificación de hilos con Hyper-Q. Posteriormente, ilustra diversas formas de aprovecharlas en un código que computa los momentos de Zernike, y que admite formulaciones directa y recursiva. Hemos podido así contrastar las posibilidades que ambas ofrecen para maximizar rendimiento en las nuevas GPUs, la primera desplegando todo el paralelismo, y la segunda aumentando la intensidad aritmética gracias a la amortización de resultados procedentes de iteraciones previas. Esto nos ha permitido aumentar los factores de aceleración que ya logramos anteriormente con arquitecturas Fermi frente a la versión C ejecutada en una CPU multicore de su misma generación. Logramos también identificar la carga de trabajo crítica que necesita un código para mejorar su ejecución en las nuevas plataformas dotadas de seis veces más núcleos computacionales, y cuantificar la sobrecarga introducida por los nuevos mecanismos de programación dinámica en CUDA.**

I. Introducción

El procesamiento de algoritmos de propósito general en procesadores gráficos (GPGPU) comenzó hace apenas diez años y desde entonces ha revolucionado el mundo de la computación de altas prestaciones (HPC) con extraordinarios factores de aceleración. Las GPUs, diseñadas con millares de núcleos de procesamiento pequeños y eficientes, permiten desplegar paralelismo a múltiples niveles, ayudando a las CPUs a procesar aquellas partes de una aplicación que requieren gran capacidad computacional y/o permiten beneficiarse del paralelismo de datos a escala masiva.

La irrupción de modelos de programación como CUDA o OpenCL ha acercado la GPUs a los programadores de aplicaciones muy diversas. Sin embargo, aún es necesario conocer a un nivel básico el nuevo paradigma de programación de estos procesadores para poder rediseñar las aplicaciones y, a un nivel avanzado, si se pretende explotar todas las prestaciones de cada nueva arquitectura.

Este trabajo pretende analizar la arquitectura Kepler de Nvidia, prestando atención a los dos puntales estrella de sus multiprocesadores SMX: paralelismo dinámico y planificación Hyper-Q. Para ello hemos utilizado un algoritmo que computa los momentos de Zernike para la caracterización de imágenes, aplicado en diversas áreas científicas como la biomedicina, y que recientemente hemos implementado sobre la generación anterior de GPUs Fermi de Nvidia.

Las funciones de momentos se expresan con cálculos integrales que, aplicadas a imágenes digitales compuestas por píxeles, se transforman al dominio discreto. Aunque la operación realizada puede ser interpretada como una máscara de convolución, los momentos pueden tener atractivas características como la invarianza a la translación de la imagen, al cambio de escala y la rotación [1]. Estas propiedades, junto con la de ortogonalidad, otorgan protagonismo a los momentos de Zernike como descriptores de imagen independientes y mínimamente redundantes [2].

Aunque la compatibilidad de las aplicaciones desarrolladas en CUDA está asegurada en sus nuevas versiones, el rendimiento puede mejorar notablemente si el código se optimiza para una arquitectura concreta, labor que requiere un buen conocimiento de las mejoras software y hardware para tomar las decisiones más acertadas que maximicen el rendimiento. En este sentido, los cambios arquitecturales no son revelados en su completitud por parte de los fabricantes. Nuestro trabajo aquí consiste en aportar nuevos elementos de juicio en base a investigación experimental que permitan guiar al programador en esta ardua labor.

Los contenidos de este artículo están estructurados de la siguiente forma: La sección II describe los momentos de Zernike y el estado del arte de su desarrollo computacional. La sección III presenta el modelo de programación CUDA, haciendo énfasis sobre los mecanismos que más utilizaremos. La sección IV proporciona una visión detallada de la arquitectura Kepler y las diferencias principales respecto a su predecesora. En la sección V mostramos la implementación de partida para los momentos de Zernike con sus peculiaridades para explotar el paralelismo de la GPU. La sección VI introduce las estrategias a seguir para conseguir un mejor rendimiento con la arquitectura Kepler. Los resultados experimentales y las conclusiones que se derivan de los mismos se exponen en las dos secciones finales.

II. Momentos de Zernike

A. Formulación matemática

Los momentos de Zernike son un conjunto de funciones ortogonales y complejas con unos coeficientes que tienen la propiedad de invarianza a la rotación de la imagen sobre la que se computan. También satisfacen la propiedad de ortogonalidad, esto es, la contribución de cada coeficiente de un momento en particular de la imagen es único, evitando así la redundancia entre ellos. Estas características los señalan como buenos descriptores de imagen. El conjunto de momentos ortogonales de Zernike para una imagen representada por la

intensidad de sus píxeles $f(r,\theta)$ con orden p y repetición q se define como sigue [2]:

$$Z_{pq} = \frac{p+1}{\pi} \int_0^{2\pi} \int_0^1 f(r,\theta) V_{pq}^*(r,\theta), r dr d\theta, \tag{1}$$

donde $V_{pq}^*(r,\theta)$ son los conjugados complejos de los polinomios de Zernike $V_{pq}(r,\theta)$ cuya representación describen el círculo unidad definido como

$$V_{pq}(r,\theta) = R_{pq}(r) e^{jq\theta} \tag{2}$$

siendo p un entero que cumple

$$\begin{gathered} p \geq 0, 0 \leq |q| \leq p, p - |q| = par, j = \sqrt{-1} \\ \theta = \tan^{-1}(y/x), 0 \leq \theta \leq 2\pi \end{gathered} \tag{3}$$

Los polinomios radiales $R_{pq}(r)$ son definidos como

$$R_{pq}(r) = \sum_{k=0}^{(p-|q|)/2} (-1)^k \frac{(p-k)!}{k!(\frac{p+|q|}{2}-k)!(\frac{p-|q|}{2}-k)!} r^{p-2k} \tag{4}$$

La formulación anterior está expresada en el dominio continuo y en coordenadas polares. La aproximación de la ecuación 1 en el dominio discreto para una función imagen $f(x,y)$ de tamaño NxN queda tal que

$$Z_{pq} = \frac{p+1}{\gamma N} \sum_{i=0}^{N-1} \sum_{k=0}^{N-1} f(x_i, y_k) V_{pq}^*(x_i, y_k) \Delta x_i \Delta y_k \tag{5}$$

donde $x_i^2 + y_i^2 \leq 1$ y γN es una componente de normalización que corresponde con el número de píxeles que comprenden el círculo unitario cuyas coordenadas están representadas por

$$x_i = \frac{2i+1-N}{N}, y_k = \frac{2k+1-N}{N} \tag{6}$$

Si añadimos a la imagen $f(x,y)$ una rotación con un ángulo α, el momento de zernike V'_{pq} de la imagen es

$$V'_{pq} = V_{pq} e^{-jq\alpha} \tag{7}$$

Como la rotación añadida sólo modifica la fase de los momentos de Zernike, el valor absoluto es invariante a la rotación. Además, lo mismo ocurre cuando las imágenes son modificadas para ser centradas o escaladas, siendo también invariante a las transformaciones lineales. Estas características tan apreciadas en el procesamiento de imágenes hace que los momentos de Zernike sean más adecuados que otros [3], como pueden ser los momentos de Legendre, aunque a cambio de un mayor coste computacional [4].

```
FUNCTION RadialPolynomial(ρ, n, m)
radial = 0
for s = 0 to (n–m)/2
  c = (-1)^s (n-s)! / ( s! ((n+|m|)/2 - s)! ((n-|m|)/2 - s)! )
  radial = radial + c * ρ^(n-2s)
end for
return radial

FUNCTION ZernikeMoments(n, m)
z_r = 0
z_i = 0
cnt = 0
for y = 0 to N–1
  for x = 0 to N–1
    ρ = sqrt( (2x - N + 1)^2 + (N - 1 - 2y)^2 ) / N
    if ρ ≤ 1
      radial = RadialPolynomial(ρ, n, m);
      theta = tan^-1( (N - 1 - 2y) / (2x - N + 1) )
      z_r = z_r + f(x, y) * radial * cos(m*theta)
      z_i = z_i + f(x, y) * radial * sin(m*theta)
      cnt = cnt + 1
    end if
  end for
end for
return (n+1)/cnt (z_r + j z_i)
```

Fig. 1. Pseudocódigo para el cálculo de los momentos de Zernike.

B. *Técnicas de computación*

La figura 1 muestra el pseudocódigo para el cálculo de los momentos de Zernike de un orden n y una repetición m acorde a la ecuación 5, y asumiendo un tamaño de imagen de NxN. En la figura 1, la parte real e imaginaria de los momentos de Zernike están expresadas como z_r y z_i, respectivamente.

Los métodos para computar los momentos de Zernike han ido evolucionando en la búsqueda de reducir la complejidad [5]. Desde una perspectiva computacional, los métodos existentes se pueden clasificar en dos vertientes:

1) Métodos directos: Los métodos directos son aquellos que calculan individualmente un momento de Zernike para un orden y una repetición dados, a través de la ecuación 5. Este método es el más usado cuando se precisa de órdenes y repeticiones concretos, tal y como ocurre en la clasificación y segmentación de imágenes, donde sólo ciertos momentos son relevantes para la discriminación [6].

2) Métodos recursivos: Esta metodología se usa en aplicaciones que necesitan toda la serie de momentos de Zernike para un orden y/o repetición dados, tal y como sucede en los algoritmos de reconstrucción de imágenes. Con estos métodos, el cálculo de momentos aislados no pueden ser procesados sin la previa realización de un barrido de procesamiento para todos los momentos de un orden o repetición. Los nuevos

coeficientes se calculan en base a los obtenidos previamente, y a través de formulas matemáticas que permiten amortizar parte de la computación ya realizada, ya que el punto de partida para cada orden y repetición corresponde con el resultado del coeficiente anterior.

Chong y cols. [7] proponen un particular método recursivo al igual que exponen las técnicas existentes para realizar una comparativa. Además, otros autores han aportado ideas para acelerar el procesamiento de los momentos de Zernike a través de la simetría del espacio de coordenadas polares [8].

III. Modelo de Programación GPU

Esta sección presenta los elementos de paralelismo específicos de CUDA (*Computer Unified Device Architecture*) [9] que permiten a la GPU desplegar paralelismo masivo y a diferentes niveles.

Cada GPU se estructura en multiprocesadores, compuestos de núcleos de computación SIMT (*Single Instruction Multiple Threads*) que comparten la unidad de control y una pequeña memoria compartida tan rápida como una caché. La memoria DRAM de vídeo se denomina memoria global y se habilita como el único espacio accesible por todos los multiprocesadores, a mucha mayor latencia pero gran ancho de banda.

El modelo de programación CUDA define los siguientes elementos básicos:

- **Hilos** (*threads*): Constituyen las unidades de ejecución básica, mapeándose sobre los núcleos del hardware.
- **Bloques**: Grupos de hilos asignados a un multiprocesador y que se ejecutan lógicamente en paralelo (físicamente sólo cuando la disponibilidad de recursos físicos lo permite, esto es, los registros y memoria compartida que comparten). Los bloques planifican su ejecución en forma de **warps**, o unidades mínimas de trabajo sobre el multiprocesador al que han sido asignados. Hasta la fecha, el tamaño del warp es 32.
- **Malla** (*grid*): Conjunto de bloques de trabajo homogéneos en que se descompone la ejecución de un *kernel* CUDA.
- **Kernel**: Porción de código que delimita una función que se pretende acelerar en GPU. Es ejecutada por todos los hilos definidos para la malla, cada uno de ellos sobre un área de datos diferente gracias a los identificadores de bloque e hilo unívocos que se establecen en tiempo de ejecución.
- **Stream**: Flujo de ejecución paralelo con otros *streams*. Los *kernels* no asociados a streams deben respetar una ejecución en serie, esto es, sólo puede haber un kernel en ejecución en cada momento, que acapara todos los recursos de la GPU. Con *streams* podemos paralelizar la ejecución de tantos *kernels* como *streams* existan, utilizando concurrentemente los multiprocesadores disponibles.

Con todos estos elementos, el programador declara explícitamente el número de bloques y el tamaño del bloque necesario para ejecutar cada kernel en la GPU.

TABLA I
PRINCIPALES RASGOS CUDA DE LAS GPU FERMI Y KEPLER.

Generación de GPU	Fermi	Kepler
Modelo hardware	GF100	GK110
Hilos por warp	32	32
Máximo de warps por multiprocesador	48	64
Bloques activos por multiprocesador	8	16
Máximo tamaño de bloque (en hilos)	1024	1024
Máximo número de hilos por multiprocesador	1536	2048
Máximo número de registros por hilo	63	255
Dimensión máxima de la malla de hilos	$2^{16}-1$	$2^{32}-1$
Paralelismo dinámico	No	Sí
Hyper-Q	No	Sí

IV. Arquitectura Kepler

Las plataformas gráficas evolucionan a una velocidad inusitada, habiéndose desarrollado tres generaciones desde el nacimiento de CUDA a finales de 2006: Tesla (2008), Fermi (2010) y Kepler (2012)[10]. Este trabajo se centra en la última de ellas, que describiremos con cierto detalle, y compararemos frente a su predecesora Fermi.

Fermi extendió el número de núcleos de GPU hasta los 512, y los de doble precisión hasta los 256. También introdujo las cachés L1 y L2 (la primera de ellas configurable en tamaño junto a la memoria compartida), popularizó la corrección de errores ECC en la memoria DRAM, mejoró los cambios de contexto y agilizó las operaciones atómicas.

Kepler, por su parte, presentó un nuevo multiprocesador SMX con 192 núcleos para computación entera y de simple precisión, y 64 núcleos para doble precisión. Inicialmente se comercializaron versiones con 13 y 14 SMXs (K20 y K20X, respectivamente), ampliándose la gama en Noviembre de 2013 a la versión de 15 SMXs (K40, dotada de 2880 núcleos). SMX incorpora dos grandes novedades: Paralelismo dinámico y planificación Hyper-Q [1], que representan el principal objeto de nuestro estudio en este artículo. Antes de detenernos en ellas, la tabla I realiza una comparativa entre Fermi y Kepler en base a los parámetros que mejor definen las capacidades de computación en CUDA. El aumento del número de *warps* y bloques que se pueden ejecutar simultáneamente en la nueva arquitectura constituyen la vía de mejora para las aplicaciones regulares y masivamente paralelas. Si éstas carecen de alguna de estas cualidades, el paralelismo dinámico y la planificación Hyper-Q ofrecen gran potencial como alternativa, aunque eso sí, de forma más exigente para el programador.

A. Paralelismo dinámico

En un sistema híbrido CPU-GPU, la ejecución eficiente de aplicaciones con elevado grado de paralelismo depende en gran medida de la versatilidad de los mecanismos que permitan distribuir el trabajo entre ambos aprovechando las mejores cualidades de cada uno de estos procesadores. Hasta 2013,

[1]Una versión preliminar de Kepler, comercializada como K10, no incluía ninguna de estas dos prestaciones.

CUDA postulaba a la GPU como un coprocesador esclavo de la CPU que recibía sus encargos y contribuía con gran aceleración, pero leve autonomía. El paralelismo dinámico permite a la GPU lanzar sus propios *kernels*, crear los eventos e hilos necesarios para controlar las dependencias, sincronizar los resultados y controlar la planificación de tareas, todo ello sin intervención alguna de la CPU, que ahora puede dedicarse de forma más eficiente a sus propias tareas. Por su parte, la GPU aporta un procesamiento más directo de bucles anidados y algoritmos recursivos, y en general, una computación más natural de código dinámico y estructuras de datos irregulares.

Por ejemplo, ahora es posible determinar en tiempo de ejecución el número de hilos encargados de procesar los nuevos kernels creados, pudiendo establecerse una configuración inicial con un paralelismo más conservador que evite cálculos innecesarios en zonas livianas, para aumentar gradualmente el paralelismo en aquellas zonas que vayan requiriendo una computación más exigente.

B. Hyper-Q

La búsqueda de un planificador óptimo que administre la carga de trabajo en GPU cuando ésta procede de diferentes *streams* es uno de los retos más difíciles de su arquitectura. Fermi permite una ejecución concurrente de hasta 16 *streams*, pero la existencia de una sola cola de trabajo obliga a multiplexar los *streams*, y por tanto, a su serialización. Aunque esta dependencia puede ser aliviada en una primera fase reordenando los *kernels* de cada *stream*, la tarea empieza a ser complicada y el rendimiento disminuye conforme la complejidad de los programas aumenta. Hyper-Q habilita hasta 32 colas de trabajo entre el host y el distribuidor de trabajo de CUDA en GPU, dotando de gran flexibilidad al conjunto para lograr grandes mejoras sin modificar la implementación. Ahora, cada *stream* se gestiona desde su propia cola *hardware* de trabajo, sin interferir las dependencias con otros *streams*, que pueden proceder del mismo programa CUDA u otros ubicados en diferentes procesos MPI o hilos POSIX (más conocidos como *p-threads*).

De esta forma, la concurrencia es natural y no requiere preprocesamiento. Además, este mecanismo resulta más potente a medida que aumentamos el número de núcleos de cada multiprocesador, erigiéndose en uno de los pilares para la escalabilidad de las futuras generaciones de GPUs.

V. IMPLEMENTACIÓN DE LOS MOMENTOS DE ZERNIKE

Para la optimización de los momentos de Zernike en la arquitectura Kepler tomaremos como punto de partida la implementación más reciente desarrollada para GPU [6]. Además, de cara a unificar las comparativas con las diferentes versiones de la bibliografía, mediremos los tiempos distinguiendo entre métodos directos y recursivos.

La implementación base defiende el empleo de la metodología directa por ser más apta para la GPU al evitar la dependencia de datos y por ser más eficiente en las aplicaciones donde no se necesitan todos los momentos de un orden o repetición. Además, en su implementación pueden aprovecharse un par de optimizaciones:

- **Paralelismo.** Nuestro algoritmo aplica a cada pixel el mismo conjunto de operaciones de forma independiente, permitiendo un uso natural del paralelismo de datos SIMD utilizado en CUDA.
- **Simetría.** Una de las fases computacionalmente más pesadas de los momentos de Zernike son los cálculos trigonométricos del circulo unidad que se aplican para la transformación del espacio polar a un espacio de coordenadas cartesianas. La simetría de los cuadrantes e incluso de los optantes permite reutilizar muchos de los cálculos [8], logrando aceleraciones de hasta 8x en GPU.

Nuestra implementación en GPU consta de cinco *kernels* que aceptan una imagen de intensidades o escala de grises y devuelven el cálculo de los momentos de Zernike según el método directo. Adaptando el proceso al modelo de programación CUDA, tenemos:

1) **Trigonometría del circulo unidad.** En una primera fase se procesa todo el espacio de coordenadas cartesianas con sus valores trigonométricos (seno y coseno) y su distancia, distinguiendo además los píxeles que quedan dentro y fuera del círculo unidad. Ésta es la fase que aprovecha la optimización de simetría.
2) **Polinomios de Zernike.** Con las distancias y senos/cosenos procedentes de la fase anterior, se calculan los polinomios de Zernike para cada píxel. El sumatorio de la ecuación 5 se realiza sobre un espacio de memoria compartida para evitar accesos reiterados a memoria global de CUDA.
3) **Aplicación a la imagen de entrada.** El resultado del espacio conseguido en la fase anterior se multiplica con la imagen de entrada.
4) **Sumatorio de píxeles.** Se contabiliza la suma de los píxeles comprendidos dentro de círculo unitario a través de un algoritmo de reducción.
5) **Sumatorio de las componentes de cada píxel.** Al igual que en la fase previa, sumamos la componente que cada píxel aporta al momento de Zernike, fusionándola en un único valor mediante un algoritmo de reducción.

Presentada la implementación base en CUDA, la tabla II describe el *hardware* sobre el que realizaremos todos los experimentos. Contaremos con dos GPUs de generaciones diferentes: Una Fermi GF100 que sitúa los tiempos de referencia para la arquitectura antecesora, y otra Kepler GK110 que permite cuantificar las mejoras logradas en los nuevos procesadores SMX con paralelismo dinámico y Hyper-Q.

VI. OPTIMIZANDO ZERNIKE SOBRE KEPLER

En esta sección analizaremos las distintas partes del código en las que se pueden aplicar las características de la arquitectura Kepler, aunque como veremos no todas ellas revertirán en mejoras productivas.

A. Recursividad

Comenzamos describiendo la implementación del método recursivo en GPU, a pesar de que pueda parecer más desafiante que el directo para lograr una ejecución eficiente [6].

Dentro de los métodos recursivos, *q-recursive* es el más actual y eficiente [7] para computar todas las repeticiones de los momentos de Zernike que corresponden a un orden concreto. Los dos primeros momentos calculados corresponden a las dos repeticiones más altas, y a partir de ahí, se obtienen todas las repeticiones progresivamente más bajas a partir de unas expresiones estáticas que involucran a los dos momentos de repetición inmediatamente superiores.

Esta metodología resulta acertada en CPU cuando el objetivo es computar varios momentos pertenecientes a un mismo orden, aunque para ello es necesario cambiar algunos aspectos de la implementación base que comentamos en la sección V.

La codificación de este método se va a llevar a cabo siguiendo un proceso iterativo que calcula en cada paso los polinomios de Zernike para una repetición dada a partir de los momentos previamente almacenados. El espacio de memoria aumentará en el mismo factor que el número de repeticiones a calcular, pero la complejidad del algoritmo disminuye y los *kernels* que no dependen de la repetición se pueden amortizar para todas las iteraciones.

Los *kernels* numerados como 1 y 4 en la sección V permanecen intactos, mientras los demás sufren los cambios que se detallan a continuación:

- El *kernel* 2 que aplica los polinomios de Zernike a cada píxel debe escindirse en dos, ya que la naturaleza del algoritmo iterativo impide enlazar el procesamiento de los polinomios de Zernike con su aplicación al punto del espacio de coordenadas cartesianas. Ahora tenemos:
  - 2.1 Polinomios de Zernike en forma recursiva. Procesa los polinomios de Zernike recursivamente. Este *kernel* se ejecuta tantas veces como repeticiones haya para los momentos de Zernike de un orden específico. Cada valor procesado usa su propio espacio de memoria.
  - 2.2 Aplicación al espacio cartesiano. Los polinomios de Zernike se aplican sobre todo el espacio, junto con las funciones trigonométricas que le anteceden.
- Los *kernels* 2.1, 3 y 5 aumentan su carga de trabajo en el mismo factor que el número de repeticiones, recayendo este trabajo sobre cada hilo, que distingue de forma unívoca la partición del espacio de memoria al que necesita acceder gracias a su identificador de bloque e hilo dentro de éste.

B. Paralelismo dinámico

El paralelismo dinámico puede aplicarse de varias formas a los momentos de Zernike según la carga de trabajo que se traslade desde la CPU a la GPU. A continuación se exponen diferentes estrategias de paralelismo dinámico que se podrían combinar, y lo que cada una de ellas puede aportar:

1) **Lanzar *kernels* desde la GPU.** Las llamadas de los cinco *kernels* del método directo se trasladan al ámbito de la GPU, de forma que inicialmente se lanza un *kernel* de un único hilo y bloque. Este *kernel* inicial o raiz realiza las llamadas correspondientes al código de los momentos de Zernike desde la GPU al igual que lo hacía la versión convencional desde la CPU.
2) **Calcular una repetición desde cada hilo.** En este caso se aprovecha el paralelismo dinámico para calcular todos los momentos de un orden dado. En la versión convencional y siguiendo el método directo, sería necesario aplicar un bucle que itere tantas veces como repeticiones existan. Aplicando paralelismo dinámico, la implementación sería equivalente a la del punto anterior, con la salvedad de que el kernel no tendría un solo hilo, sino uno por cada repetición. Para evitar redundancia en los cálculos, los *kernels* en común para todas las repeticiones serían procesados desde un mismo hilo.
3) **Paralelizar el bucle de los polinomios de Zernike.** El bucle *for* que se requiere en el cálculo de los polinomios de Zernike por el método directo (ver figura 1), es buen candidato para aprovechar el paralelismo dinámico. Cada píxel precisa de dicho cálculo, así que cada uno de ellos lanzará un nuevo *kernel* que procese de forma concurrente el trabajo de ese bucle.

C. Hyper-Q

El aprovechamiento de Hyper-Q requiere establecer un proceso con varios flujos de ejecución concurrente que no presenten dependencias. En los momentos de Zernike se consigue, al igual que el segundo punto del anterior apartado, cuando necesitamos calcular todas las repeticiones de los momentos de Zernike para un orden específico.

El aprovechamiento de Hyper-Q es transparente al programador. La implementación se lleva a cabo usando *streams* que puedan beneficiarse de las múltiples colas de ejecución concurrente. El aumento del número de colas de 16 a 32 resulta también clave en la nueva arquitectura para aprovechar al máximo el creciente número de procesadores CUDA, ya que ahora, con un número cercano a los tres mil, resulta más probable que un kernel sólo pueda ocupar una fracción de éstos. Distribuyendo los *kernels* en *streams* siempre que sea posible, conseguiremos que el remanente de procesadores que haya dejado libre la malla de hilos definida para un primer kernel en ejecución, pueda ser aprovechada desde otros procedentes de *streams* adicionales.

VII. Resultados experimentales

La Tabla II resume las prestaciones de las GPUs utilizadas durante la evaluación experimental de nuestras optimizaciones.

Hemos empleado un tipo de datos de simple precisión y tomado tamaños de imagen progresivos desde 64x64 hasta 2Kx2K píxeles. El orden máximo de los momentos que se calcula es de 34 debido al límite impuesto por el hardware para el cálculo de los factoriales que aparecen en los polinomios de Zernike.

TABLA III
TIEMPOS DE EJECUCIÓN (EN MILISEGUNDOS) PARA PROCESAR TODAS LAS REPETICIONES DE UN ORDEN A TRAVÉS DEL MÉTODO DIRECTO DE LOS MOMENTOS DE ZERNIKE. LAS IMÁGENES DE ENTRADA SON CUADRADAS Y DE DIMENSIONES POTENCIA DE DOS EN UN RANGO ENTRE 64 Y 2048.

Momentos	GPU Fermi						GPU Kepler						Factor de mejora	
de Zernike	64	128	256	512	1024	2048	64	128	256	512	1024	2048	Mínimo	Máximo
$A_{4,*}$	0,12	0,17	0,37	1,11	4,08	15,75	0,18	0,19	0,33	0,61	1,91	7,17	0,67x	2,20x
$A_{8,*}$	0,20	0,32	0,74	2,36	8,83	34,71	0,28	0,31	0,46	1,16	3,86	14,74	0,71x	2,35x
$A_{12,*}$	0,29	0,50	1,21	4,08	15,32	60,41	0,41	0,44	0,72	1,88	6,42	24,67	0,71x	2,45x
$A_{16,*}$	0,38	0,72	1,82	6,14	23,49	92,05	0,53	0,58	1,00	2,72	9,55	36,93	0,72x	2,49x
$A_{20,*}$	0,51	0,97	2,50	8,68	33,37	130,93	0,67	0,74	1,33	3,75	13,28	51,56	0,76x	2,54x
$A_{24,*}$	0,61	1,27	3,31	11,76	45,39	176,51	0,80	0,90	1,70	4,92	17,64	68,59	0,76x	2,57x
$A_{28,*}$	0,74	1,59	4,23	15,20	58,20	229,20	0,97	1,09	2,11	6,23	22,52	87,87	0,76x	2,61x
$A_{32,*}$	0,87	2,00	5,27	18,89	73,30	288,76	1,14	1,29	2,58	7,70	28,04	109,53	0,76x	2,64x
$A_{34,*}$	0,95	2,16	5,89	20,96	81,29	319,76	1,22	1,39	2,81	8,48	31,01	121,23	0,78x	2,64x

TABLA II
RESUMEN DE LAS CARACTERÍSTICAS DE LAS GPUS QUE HEMOS UTILIZADO A LO LARGO DE NUESTRA EVALUACIÓN EXPERIMENTAL.

Generación de GPU	Fermi	Kepler
Modelo comercial	Tesla C2075	Tesla K20c
Tipo de multiprocesador	SM (32 núcleos)	SMX (192 núcleos)
Número de multiprocesadores	14	13
Número total de núcleos	448	2496
Frecuencia	1.15 GHz	710 MHz
Rendimiento pico	1030 GFLOPS	3.52 TFLOPS
Frecuencia de la memoria	2x 1566 MHz	2x 2600 MHz
Anchura del bus de memoria	384 bits	320 bits
Ancho banda de la memoria	148 GB/s.	208 GB/s.
Tamaño memoria (GDDR5)	6 Gbytes	5 Gbytes
Bus desde/a CPU	PCI-e x16 2.0	PCI-e x16 2.0

A. Cambio de arquitectura: SMX

En primer lugar, cuantificamos las mejoras producidas por el cambio de arquitectura sin realizar modificaciones en el código. La tabla III recoge los tiempos de ejecución sobre Fermi y Kepler para el algoritmo de partida de la sección V, a la vez que se comparan con el factor de ganancia.

El factor de mejora para el cambio de plataforma oscila entre 0.67x y 2.64x. La cota mínima de 0.67x, que supone un aumento de tiempo de ejecución de 1.49x, se produce para el tamaño más pequeño de la imagen, y de igual forma, la aceleración máxima de 2.64x llega cuando la carga de trabajo alcanza las dosis más elevadas.

La GPU como procesador, y su modelo de paralelismo de datos como forma de programación, se lucen más a medida que crecen las imágenes de entrada, lo que explica que la migración a Kepler se luzca más conforme aumenta el número de pídeles a procesar. El tamaño de imagen más pequeño consta de 4096 píxeles que se distribuyen en bloques repartidos equitativamente entre todos los multiprocesadores disponibles. El multiprocesador SM de Fermi puede procesar hasta 2 *warps* de forma concurrente, lo que da un total de 896 píxeles a procesar en nuestra plataforma (2x32x14). El SMX de Kepler llega hasta 4 *warps* para un total de 1664 píxeles a la vez para aprovechar los recursos. Con estos datos, la imagen pequeña de 64 x 64 píxeles aprovecharía sólo el 32.6% y 18.9% de los recursos para Fermi y Kepler, respectivamente. En este caso, aunque la arquitectura haya sido mejorada, sus recursos son infrautilizados y es la frecuencia, my superior en Fermi, la que resulta más determinante en el tiempo de ejecución.

Con un tamaño de 128 x 128 píxeles, la carga de trabajo ya satura a la máxima que puede procesar concurrentemente Fermi, mientras que la GPU Kepler se queda en un 75.7%. En este tamaño de imagen, el rendimiento tiende a igualarse en ambas arquitecturas, ya que aunque la GPU Kepler no aproveche el 100% de sus recursos, puede planificar todo el cálculo de una sola vez, mientras que la GPU Fermi necesita una segunda tanda para planificar el remanente de bloques que están a la espera.

A partir de aquí, para imágenes de tamaño superior, la mejora de rendimiento en Kepler aumenta hasta alcanzar su cota máxima con el 100% de los recursos ocupados en esta nueva arquitectura.

En un análisis vertical, el aumento del orden en los momentos de Zernike supone una ejecución adicional del algoritmo por cada dos órdenes. Esta carga adicional aumenta el tiempo de cómputo a la vez que beneficia ligeramente a la GPU.

B. Configuración de la carga de trabajo

El incremento del número de núcleos del multiprocesador SMX en un factor de seis respecto a SM predispone a reflexionar sobre si la configuración óptima del tamaño del bloque de hilos puede haber variado respecto a la implementación base. En la GPU Fermi dotada de 32 núcleos, un valor entre 128 y 256 hilos era el óptimo para lograr un factor de ocupación del 100% mientras no hubiera restricciones respecto al uso de los registros y la memoria compartida. La GPU Kepler, cuyo multiprocesador SMX cuenta con 192 núcleos, merece un análisis más pormenorizado a este respecto.

La figura 2 desvela los tiempos de ejecución para el *kernel* que procesa los polinomios de Zernike, siendo éste el más critico en cuanto a recursos. Los tiempos, según la escala de la derecha, se contrastan frente al valor teórico para bloques con diferentes configuraciones de hilos, lo que desemboca

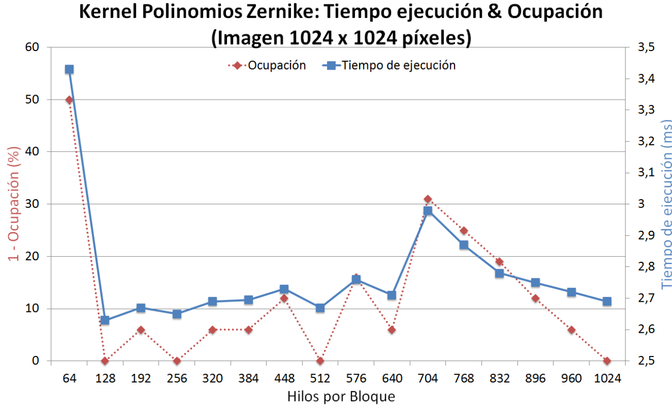


Fig. 2. Tiempo de ejecución en Kepler para evaluar el rendimiento (escala derecha) en función del tamaño de bloque. Estos valores se contrastan con los teóricos de ocupación para la arquitectura (escala izquierda).

en el factor de ocupación según la escala de la izquierda. Aunque teóricamente el mejor rendimiento se consigue para potencias de dos a partir de 128 hilos por bloque, el resultado ha sido levemente mejor para una configuración exacta de 128 hilos, la más baja entre las candidatas. Sin embargo, la mayor divergencia entre la teoría y la práctica se produce cuando el número de hilos por bloque no es suficiente para aprovechar los recursos, o también cuando al añadir un nuevo bloque se sobrepasa el límite de hilos por multiprocesador y, por tanto, hay que sacrificar un bloque en la ejecución concurrente. Ambas zonas se pueden diferenciar:

1) Si el bloque tiene menos de 128 hilos, la limitación para no llegar al número de hilos máximo por multiprocesador la impone el máximo número de bloques permitido. En la GPU Kepler, se pueden ejecutar hasta 16 bloques activos en cada multiprocesador SMX, y la fórmula de la ocupación tiene la siguiente expresión:

$$Ocupaci\acute{o}n = \frac{Hilos * 16}{2048} \tag{8}$$

2) Cuando el número de hilos alojado en un multiprocesador se acerca a su valor máximo, la inclusión de un nuevo bloque puede sobrepasar el máximo y, por tanto, el número permitido de bloques no consigue aprovechar todos los recursos. El peor caso (69% de ocupación) se da con una configuración de 704 hilos por bloque cuando se aplica la siguiente expresión:

$$Ocupaci\acute{o}n = \frac{Hilos * Bloques_{max}}{2048} \tag{9}$$

C. Paralelismo dinámico

Las estrategias de paralelismo dinámico descritas en la sección VI-B van a permitir conocer cuándo es beneficiosa su aplicación. En la tabla IV se muestra que la aplicación más sencilla, denominada “Lanzar *kernels* desde la GPU”, y la estrategia “Cálcular una repetición desde cada hilo” no resultan positivas. Estos resultados se han tomado para momentos con orden y repetición específicos para, además de comparar los resultados con paralelismo dinámico, dar una idea sobre los tiempos de ejecución para momentos individuales.

TABLA IV
TIEMPOS DE EJECUCIÓN Y FACTORES DE ACELERACIÓN LOGRADOS PARA LAS DIFERENTES ESTRATEGIAS QUE EXPLOTAN EL PARALELISMO DINÁMICO CON IMÁGENES CUADRADAS DE DIFERENTES TAMAÑOS PARA MOMENTOS DE ZERNIKE CONCRETOS EN EL PRIMER CASO, Y TODAS LAS REPETICIONES DE CADA ORDEN EN EL SEGUNDO.

(a) Tiempos de ejecución en milisegundos sin/con paralelismo dinámico.

Momento	Sin P. Dinámico			Con P. Dinámico			Ganancia	
de Zernike	64	256	1024	64	256	1024	Min.	Max.
$A_{0,0}$	0,08	0,10	0,56	0,16	0,20	0,88	0,48x	0,64x
$A_{6,2}$	0,08	0,13	0,93	0,16	0,22	1,26	0,53x	0,74x
$A_{12,0}$	0,09	0,16	1,43	0,16	0,27	1,79	0,58x	0,80x
$A_{25,13}$	0,09	0,17	1,52	0,16	0,27	1,88	0,59x	0,81x
$A_{34,0}$	0,12	0,28	3,07	0,18	0,38	3,51	0,65x	0,88x
$A_{34,18}$	0,10	0,19	1,82	0,16	0,29	2,20	0,60x	0,83x
$A_{34,34}$	0,08	0,10	0,63	0,16	0,20	0,95	0,49x	0,66x

(b) Factores de aceleración logrados con paralelismo dinámico.

Todos los momentos	Tamaño de imagen					
para un orden dado	64	128	256	512	1024	2048
$A_{4,*}$	0,73	0,70	0,78	0,70	0,71	0,72
$A_{8,*}$	0,82	0,76	0,71	0,75	0,76	0,76
$A_{12,*}$	0,89	0,82	0,76	0,79	0,79	0,79
$A_{16,*}$	0,91	0,84	0,78	0,80	0,81	0,81
$A_{20,*}$	0,93	0,86	0,81	0,82	0,82	0,82
$A_{24,*}$	0,93	0,87	0,82	0,84	0,84	0,84
$A_{28,*}$	0,95	0,89	0,84	0,85	0,85	0,85
$A_{32,*}$	0,96	0,91	0,85	0,86	0,86	0,86
$A_{34,*}$	0,96	0,90	0,85	0,86	0,86	0,86

En el primer caso, los tiempos de ejecución son mayores en un factor entre 1.15x y 2.15x (ver tabla IV(a)). Esta varianza depende de la carga de trabajo que supone cada llamada a un nuevo *kernel* desde la GPU. Las imágenes de tamaño superior sufren menos penalización al igual que los momentos computacionalmente más exigentes, que son aquellos en los que la diferencia entre el orden y la repetición es mayor.

Para la segunda estrategia, el rendimiento empeora hasta en un factor de 1.4x en el mismo sentido que la estrategia anterior (ver tabla IV(b)). En este caso, el rendimiento es mejor para las imágenes pequeñas debido a que se procesa un conjunto de momentos que consiguen aprovechar los recursos, mientras que para momentos específicos no sería posible. En resumen, el rendimiento para imágenes pequeñas es sensible a dos aspectos: el propio de lanzar los *kernels* internamente desde GPU, y el que permite paralelizar los cálculos de los momentos que forman parte de la cadena $A_{n,*}$.

La tercera estrategia, denominada “Paralelizar el bucle de los polinomios de Zernike”, es muy pretenciosa por su naturaleza dinámica. Sin embargo, las pruebas experimentales dejan ver rápidamente que su rendimiento es catastrófico. La implementación requiere que cada hilo lance un nuevo *kernel*, lo que dispara el tiempo hasta factores 1000x rápidamente. Hemos medido el tiempo que consume una nueva llamada

a GPU, resultando entre 5 y 16 $\mu sg.$, mientras que en la CPU consume alrededor de 3 $\mu sg.$. No parece lógico asumir que el lanzamiento de un *kernel* introduzca mayor sobrecarga cuando se lanza desde circuitería mucho más cercana como la propia GPU, y pensamos que esta rémora se solventará en versiones más maduras de los *drivers* y/o implementaciones más maduras de los multiprocesadores SMX.

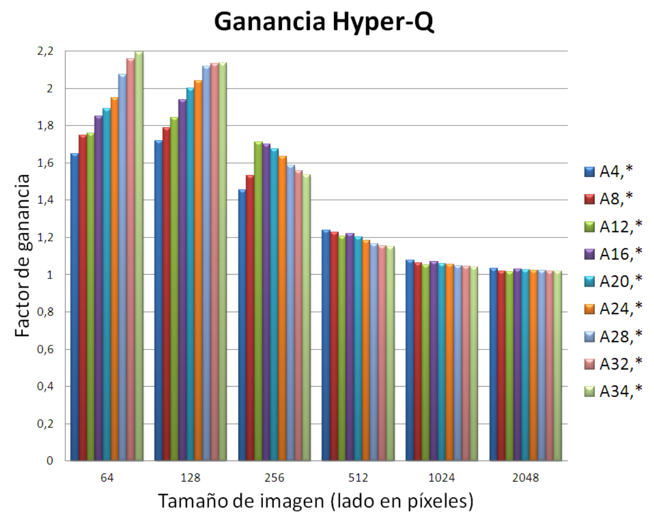
No obstante, el paralelismo dinámico está orientado a aplicaciones con una filosofía “divide y vencerás” y debe concebirse en la misma línea que la computación con GPUs: Potenciando pocas llamadas a *kernels* con gran cantidad de datos. Otra característica, que limita el ámbito de aplicación del paralelismo dinámico, es la restricción de que cada hilo no puede acceder a la memoria compartida de los *kernels* padres. La información a compartir entre *kernels* padres e hijos queda relegada al uso de la memoria global, con la penalización de rendimiento que esto supone.

D. Hyper-Q

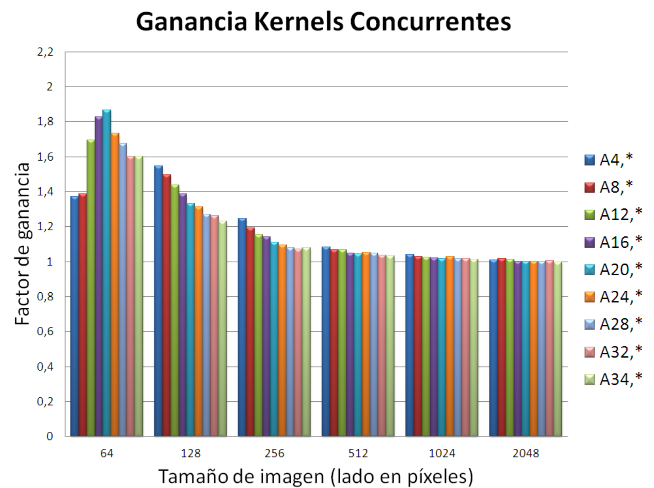
Para aprovechar Hyper-Q en nuestro algoritmo definiremos un *stream* por cada repetición cuando se persigue calcular todas las repeticiones de un orden. La figura 3(a) compara el factor de ganancia cuando entra en escena Hyper-Q para este supuesto, variando el orden de los momentos de Zernike y el tamaño de la imagen de entrada sobre la que se aplica.

Mientras que la ganancia máxima obtenida es de 2.2x, la ganancia mínima queda a la par con la implementación básica. Esta situación de paridad se produce ya para una carga de trabajo superior a la que abastece la totalidad de recursos de la arquitectura. Si la imagen es de tal tamaño que la carga de trabajo generada mantiene a la GPU completamente ocupada, no quedan remanentes para ser aprovechados desde *streams* adicionales mediante Hyper-Q, y cada nuevo *stream* acabará serializándose en la cola de trabajo. Por otro lado, en los tamaños de imagen más pequeños, la ganancia aumenta proporcionalmente al número de repeticiones a calcular por cada momento. Este escenario es el ideal para el aprovechamiento de Hyper-Q, ya que la poca carga de trabajo suministrada para cada imagen se compensa dando entrada al procesamiento de otras imágenes en paralelo. Por tanto, el rendimiento máximo se consigue procesando la imagen más pequeña y el momento de Zernike para el orden más alto.

Como el uso de Hyper-Q no supone ningún cambio para el desarrollador y la gestión de colas para procesar los *streams* es transparente, la misma aplicación ejecutada en la arquitectura Fermi nos proporciona una visión de la mejora que supuso la inclusión de *Kernels* Concurrentes. La figura 3(b) muestra, al igual que se hizo en Kepler, el factor de ganancia obtenido por la incorporación de esta técnica en Fermi. Los valores son similares cualitativamente con la peculiaridad que, para el tamaño de imagen inferior donde la ganancia es mayor, la GPU llega a su plena ocupación con órdenes de momento inferiores. Concretamente, el valor máximo 1.86x se consigue para 11 *streams*, situación que corresponde para el cálculo del momento de orden 20 (A_{20}).



(a) Mejoras logradas con Hyper-Q en Kepler.



(b) Mejoras logradas con *Kernels* concurrentes en Fermi.

Fig. 3. Ganancia obtenida con el uso de *streams* para distintos tamaños de imagen cuando aumentamos el conjunto de momentos de Zernike a procesar.

Hyper-Q ha supuesto en los momentos de Zernike una aceleración máxima de 1.74x respecto a lo ya conseguido con *Kernels* Concurrentes en Fermi.

El análisis anterior engloba conjuntamente la aplicación de Hyper-Q y el uso pleno de los recursos de los multiprocesadores. Para aislar la aceleración correspondiente a Hyper-Q, hemos realizado un experimento con una imagen de 16 x 16 píxeles (que corresponde con nuestro tamaño del bloque de hilos en CUDA). De esta forma, la ejecución convencional de todas las repeticiones de un mismo orden se realiza secuencialmente y sin aprovechar todos los recursos al enviar un único bloque por iteración. Cuando se usa Hyper-Q o *Kernels* concurrentes, el uso de los recursos se incrementa por la ejecución de las distintas repeticiones en paralelo.

La figura 4 muestra los resultados de la comparativa para Fermi y Kepler. Los tiempos cuando no se habilita

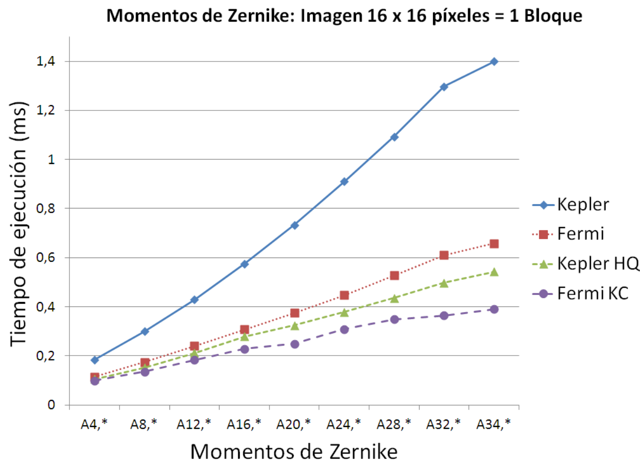


Fig. 4. Análisis comparativo del beneficio conseguido en la GPU cuando la imagen es de 16 x 16 píxeles, correspondiente a un bloque de hilos CUDA, con objeto de aislar la aceleración atribuida a Hyper-Q y *Kernels* Concurrentes.

la paralelización con *streams* favorecen a la GPU Fermi al tratarse de una imagen de entrada muy pequeña, tal y como ya nos ocurrió y explicamos en la sección VII-A para imágenes de 64x64 píxeles. Cuando se habilita Hyper-Q en Kepler o *Kernels* Concurrentes en Fermi, la ganancia obtenida es superior en el primer caso (un factor 1.7x), aunque el tiempo de ejecución sigue quedando por encima del obtenido para Fermi. Esto induce a pensar que Hyper-Q resulta un gran complemento para explotar el incremento de núcleos de procesamiento, pero que no aporta tanto a los pilares que despliegan el paralelismo más tradicional de CUDA, léase el que se cimenta sobre los bloques de hilos a nivel intra-multiprocesador y las mallas de bloques concurrentes a nivel inter-multiprocesador. Esta conjetura se consolidará tras la aportación proveniente de la variante recursiva a continuación.

E. Recursividad

Hasta ahora hemos tratado de aprovechar paralelismo dinámico y Hyper-Q para procesar todas las repeticiones de un orden concreto de los momentos de Zernike. Ésta es la manera más sencilla de aumentar la cantidad de datos a computar esquivando las dependencias de datos. Sin embargo, la sección VI-A introdujo algoritmos que realizan estos mismos cálculos aplicando recursividad. De optar por este camino, tendremos decididamente un algoritmo más eficiente en CPU, pero menores oportunidades de paralelismo en GPU.

La tabla V muestra los tiempos de ejecución obtenidos al aplicar el método *q-recursive* en las GPU Fermi y Kepler, contrastándolos en las columnas de ganancia respecto al método directo que computamos anteriormente, donde las oportunidades para desplegar paralelismo son muy superiores.

Los resultados favorecen a la implementación recursiva prácticamente en todos los casos, con la única excepción del orden del momento más bajo para la imagen más pequeña. La ganancia crece conforme aumenta el orden de los momentos y el tamaño de la imagen, y también es superior en Fermi.

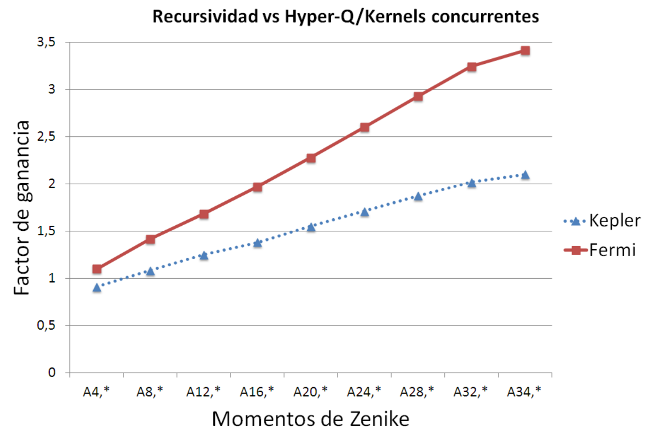


Fig. 5. Comparativa entre la variante recursiva frente a Hyper-Q en el método directo. Valores superiores a 1 suponen una mejora de la alternativa recursiva.

Dos razones puedes justificar este resultado aparentemente sorprendente: Primero, la implementación recursiva reduce la complejidad de los cálculos y, por tanto, la carga de trabajo exigida a la GPU, lo que perjudica a la plataforma Kepler. Segundo, Fermi parte de un tiempo de referencia bastante más elevado, lo que le otorga un mayor potencial de mejora.

F. Recursividad frente a Hyper-Q en métodos directos

Las dos subsecciones anteriores tienen como objetivo el cálculo del conjunto de momentos con un orden específico desde dos perspectivas diferentes: Mediante métodos directos que permitan aplicar un mayor grado de paralelismo y Hyper-Q, y utilizando métodos recursivos que sacrifiquen estos mecanismos al introducir dependencias de datos. Una comparativa final entre ambas estrategias determinará si para el cálculo de los momentos de Zernike resulta mejor explotar todos los recursos de la GPU u optar por un algoritmo más hostil al paralelismo pero computacionalmente más eficiente.

La figura 5 muestra la esperada comparativa entre ambas, una suerte de confrontación entre la fuerza bruta y la calidad de la computación. Tanto en Fermi como en Kepler, la implementación recursiva resulta vencedora en GPU, con una tendencia prácticamente lineal respecto al orden del momento y escalable para el rango de órdenes considerado (hasta 34).

Además, los tamaños de imagen más grandes favorecen a la GPU Kepler. Esto se debe a que los multiprocesadores están empleando un mayor número de núcleos en una distribución convencional a través de la malla de hilos en CUDA, y alcanzado el volumen de datos necesario para alimentar a todos los núcleos, la sobrecarga que introduce la gestión más sofisticada de Hyper-Q queda claramente amortizada.

La principal conclusión que podemos extraer de este análisis es que el despliegue del paralelismo inherente a los algoritmos debe centrarse en primer lugar en explotar los niveles de hilos por bloque (entre los núcleos del multiprocesador) y bloques concurrentes (entre los multiprocesadores de la GPU), dejando el tercer nivel de kernels concurrentes y su Hyper-Q como baza adicional cuando la carga de trabajo no permita

TABLA V
TIEMPOS DE EJECUCIÓN (EN MILISEGUNDOS) PARA IMÁGENES CUADRADAS DE DIFERENTE TAMAÑO CUANDO PROCESAMOS TODAS LAS REPETICIONES DE UN ORDEN ESPECÍFICO A TRAVÉS DEL MÉTODO *q-recursive* EN LAS GPU FERMI Y KEPLER. LA GANANCIA DE LAS 4 ÚLTIMAS COLUMNAS SE CALCULA RESPECTO AL TIEMPO OBTENIDO EN CADA CASO MEDIANTE EL MÉTODO DIRECTO.

Momentos	GPU Fermi						GPU Kepler						Ganancia Fermi		Ganancia Kepler	
de Zernike	64	128	256	512	1 K	2 K	64	128	256	512	1 K	2 K	Min	Max	Min	Max
$A_{4,*}$	0,08	0,12	0,29	0,90	3,31	13,07	0,11	0,12	0,24	0,59	2,01	7,76	1.21	1.50	0.92	1.62
$A_{8,*}$	0,11	0,17	0,45	1,50	5,78	22,53	0,14	0,16	0,32	0,91	3,27	12,73	1.53	1.92	1.16	2.05
$A_{12,*}$	0,12	0,23	0,61	2,12	8,12	32,12	0,17	0,20	0,42	1,25	4,54	17,73	1.88	2.35	1.39	2.34
$A_{16,*}$	0,14	0,28	0,79	2,74	10,58	41,66	0,20	0,24	0,52	1,58	5,82	22,79	2.21	2.67	1.62	2.58
$A_{20,*}$	0,17	0,34	0,95	3,36	13,13	51,35	0,23	0,29	0,62	1,91	7,11	27,87	2.54	3.07	1.85	2.88
$A_{24,*}$	0,19	0,39	1,12	3,98	15,59	60,88	0,26	0,32	0,72	2,24	8,38	32,93	2.90	3.26	2.08	3.06
$A_{28,*}$	0,21	0,45	1,28	4,60	17,97	70,56	0,28	0,36	0,82	2,58	9,66	38,03	3.24	3.56	2.31	3.42
$A_{32,*}$	0,23	0,50	1,45	5,31	20,41	80,19	0,32	0,41	0,93	2,92	10,95	43,15	3.56	4.02	2.54	3.52
$A_{34,*}$	0,24	0,52	1,54	5,55	21,64	85,00	0,33	0,44	0,98	3,09	11,59	45,70	3.76	4.12	2.65	3.66

saturar los núcleos de computación disponibles. Esta situación se producirá con asiduidad en el contexto de aplicaciones irregulares en las que el volumen de datos se encuentre repartido en un conjunto ingente de procesos de tal forma que cada uno de ellos involucre a estructuras de datos de tamaño reducido, pongamos inferior al millar de elementos de entrada. También será un supuesto más probable en las futuras generaciones hardware dotadas de un mayor número de cores, lo que indica que sólo estamos comenzando a aprovechar el potencial que Hyper-Q esconde como recurso.

VIII. CONCLUSIONES

Este trabajo analiza las posibilidades que las GPUs ofrecen para acelerar el cálculo de los momentos de Zernike, otorgando protagonismo a la arquitectura Kepler y su multiprocesador SMX. El cambio de arquitectura supone un gran avance en rendimiento, resultando una mejora de hasta un 265% con el mismo código, y muy superior si se aprovechan técnicas como el paralelismo dinámico y Hyper-Q. El paralelismo dinámico no ha resultado beneficioso por la sobrecarga detectada en el lanzamiento de *kernels* desde la GPU, que esperemos quede solventada en versiones futuras. Por su parte, Hyper-Q consigue aumentar el rendimiento hasta un 220% respecto a la implementación base en Kepler del método directo, resultando mayor beneficio cuando la carga de trabajo procedente de un solo kernel no satura el número de procesadores disponibles en GPU. Esta ganancia puede mejorarse en códigos paralelos que usan hilos POSIX o MPI.

La versión recursiva de los momentos de Zernike también se mejora en la arquitectura Kepler aunque haya que sacrificar buena parte del paralelismo y Hyper-Q. Comparada con el método directo, sólo resulta peor hasta un orden seis, trasladando a partir de ahí mejoras que alcanzan un 350%.

En última instancia, el mejor método para el cálculo de los momentos de Zernike está supeditado a las necesidades de nuestra aplicación. En muchas de ellas, como la caracterización de imágenes y texturas, sólo se utilizan unos pocos momentos puntuales (aquellos que presentan mayor poder de discriminación tras un análisis previo). En otras aplicaciones, como las pertenecientes al campo de la compresión y reconstrucción de imágenes, es necesario computar, además del conjunto de repeticiones para un orden dado, un número elevado de órdenes en los que la formulación recursiva puede incluso desplegarse en una segunda dimensión. Este trabajo estudia la situación intermedia que considera el cálculo de todas las repeticiones de un mismo orden, y establece las bases para que en cualquiera de las numerosas aplicaciones de los momentos de Zernike, siempre podamos encontrar una implementación ventajosa en GPU respecto a CPU: Ya sea por la vía directa, gracias a la riqueza de los mecanismos de paralelización de CUDA, o por la vía recursiva, que a pesar de ser una formulación más competitiva en CPU, ha logrado factores de aceleración superiores en GPU. Y todo ello, favorecido por el tamaño de la imagen de entrada y el número de procesadores disponibles, dos factores que tienen mucho recorrido al alza tanto para las aplicaciones software como para las plataformas hardware venideras.

REFERENCES

[1] Y. Bin and P. Jia-Xiong, "Invariance analysis of improved zernike moments," *Journal of Optics A: Pure and Applied Optics*, vol. 4, no. 6, p. 606, 2002.

[2] M. R. Teague, "Image analysis via the general theory of moments," *J. Opt. Soc. Am*, vol. 70, no. 8, pp. 920–930, 1980.

[3] C.-H. Teh and R. T. Chin, "On image analysis by the methods of moments," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 10, no. 4, pp. 496–513, 1988.

[4] R. Mukundan and K. Ramakrishnan, "Fast computation of legendre and zernike moments," *Pattern recognition*, vol. 28, no. 9, pp. 1433–1442, 1995.

[5] M. Al-Rawi, "Fast zernike moments," *Journal of Real-Time Image Processing*, vol. 3, no. 1-2, pp. 89–96, 2008.

[6] M. J. Martín-Requena and M. Ujaldón, "Leveraging graphics hardware for an automatic classification of bone tissue," in *Computational Vision and Medical Image Processing*. Springer, 2011, pp. 209–228.

[7] C.-W. Chong, P. Raveendran, and R. Mukundan, "A comparative analysis of algorithms for fast computation of zernike moments," *Pattern Recognition*, vol. 36, no. 3, pp. 731–742, 2003.

[8] S.-K. Hwang and W.-Y. Kim, "A novel approach to the fast computation of zernike moments," *Pattern Recognition*, vol. 39, no. 11, pp. 2065–2076, 2006.

[9] N. CUDA, "Parallel programming and computing platform," http://www.nvidia.es/object/cuda_home_new_es.html, Jun. 2013.

[10] NVIDIA, "The kepler architecture," http://www.nvidia.com/object/nvidia-kepler.html, Jun. 2013.