XI Congreso Nacional de Teledetección, 21-23 septiembre 2005.
Puerto de la Cruz. Tenerife.

Recuperación de fase de frente de onda atmosférico usando hardware gráfico

José G. Marichal–Hernández[1], *Jose M. Rodríguez–Ramos*[1] *y Fernando Rosa*[1]

[1] Dep. Física Fundamental y Exp., Electrónica y Sistemas. Universidad de La Laguna. Tenerife. Avda. Francisco Sánchez s/n, 38200 La Laguna, Tenerife, Canarias. {jmariher, jmramos, frosa}@ull.es

Resumen

Se estudia el problema de la recuperación de la fase del frente de onda de un sistema de óptica adaptativa usando un sensor Shack–Hartmann. Con el objetivo de realizar ese cómputo dentro del tiempo de estabilidad atmosférica para un número de subpupilas de $32x32$ o superior, aplicamos una tecnología innovadora: las *Unidades de Procesamiento Gráfico* (GPU's). Los resultados obtenidos reflejan una aceleración del cálculo, frente a la implementación en CPU, de 2x para el algoritmo de Hudgin y de 10x para el cálculo de los centroides.

1. Introducción

Las técnicas de la Óptica Adaptativa (AO) en Astronomía persiguen obtener el límite de difracción del telescopio empleado. El número de actuadores de los sistemas AO se espera que crezca desde cientos hasta decenas de miles en el futuro, debido al aumento de diámetro de los nuevos telescopios y a nuevas aplicaciones de alta resolución de los ya existentes. La óptica adaptativa implica varios pasos: la detección, la recuperación del frente de onda, la transmisión de comandos a los actuadores y la ejecución de su mecánica. Es en este contexto en el que se entiende que un reconstructor de fase de frente de onda más eficiente puede ser de gran relevancia, para no incumplir el límite temporal impuesto por el tiempo característico de la atmósfera, 10 ms en el visible. Nuestro objetivo no será seleccionar el mejor algoritmo de reconstrucción de fase del frente de onda, sino explorar posibilidades para obtener un reconstructor más eficiente. Estudiaremos un algoritmo zonal que encaja bien en los modelos de cómputo paralelo: el algoritmo de Hudgin[1].

2. Recuperación de la fase del frente de onda

El sensor de frente de onda Shack–Hartmann muestrea la amplitud compleja del campo en la pupila del telescopio para obtener, tras el adecuado procesamiento, el mapa de fase del frente de onda, $\Phi_{j,k}(u,v)$, donde u, v son las coordenadas del plano en la pupila y j, k son los índices de las subpupilas. Estudiaremos el problema de la reconstrucción para un número de subpupilas de $32x32$, lo que está en el límite de lo que el estado del arte de la óptica adaptativa puede lograr en tiempo real. Los restantes parámetros elegidos en los datos simulados que recuperaremos son similares a los de los telescopios GTC y Keck.

Las diferencias de fase para un objeto puntual se pueden obtener comparando los centroides de la intensidad medida en cada subpupila –vistos como un todo conforman una imagen de patrón de puntos, *spots pattern image*– con los de una referencia. El algoritmo de recuperación se aplica sobre las diferencias de fase para obtener el mapa de fase.

Los algoritmos zonales iterativos se adaptan mejor que los modales al modelo abstracto de *Stream processor* y se adecúan mejor a un enfoque paralelo. El algoritmo elegido será el de Hudgin. Si no hacemos consideraciones sobre los errores de medida de las diferencias de fase, una implementación recursiva que recupere la fase se puede expresar como:

$$\begin{aligned}\Phi'_{j,k} = \quad & \tfrac{1}{4}(\Phi_{j+1,k} + \Phi_{j-1,k} + \Phi_{j,k+1} + \Phi_{j,k-1}) \\ & + \tfrac{1}{4}(S^1_{j,k} - S^1_{j-1,k} + S^2_{j,k} - S^2_{j,k-1}),\end{aligned} \tag{1}$$

donde $\Phi'_{j,k}$ será en el próximo paso de refinamiento el valor de $\Phi_{j,k}$, y

$$S^1_{j,k} = \Phi_{j,k} - \Phi_{j+1,k}$$
$$S^2_{j,k} = \Phi_{j,k} - \Phi_{j,k+1}$$

son las diferencias de fase que conectan las subpupilas en cada dirección.

Reformulamos este algoritmo para introducir la existencia o no de valores vecinos a los de la subpupila considerada y así poder tener en cuenta el borde del dominio de medida, una pupila anular. Para ello definimos N_u, N_d, N_r and N_l, variables de vecindad:

$$N_u(j,k) = \begin{cases} 1 & \text{, if } \exists\ S^1_{j,k} \ \wedge\ \exists\ \Phi_{j+1,k} \\ 0 & \text{, i.o.c.} \end{cases}$$

donde N_u representa al vecino encima, *"neighbour up"*, y de forma análoga N_d se define para *"down"*, N_l para *"left"* y N_r para *"right"*.

Si sumamos las expresiones de S^1 y S^2 donde $\Phi_{j,k}$ aparece relacionada con las fases de sus subpupilas vecinas, la expresión original se transforma en:

$$\begin{aligned} N = \quad & N_u(j,k) + N_d(j,k) + \\ & N_r(j,k) + N_l(j,k) \end{aligned}$$

$$\begin{aligned} \Phi'_{j,k} = \tfrac{1}{N} \quad & (N_u(j,k)(S^1_{j,k} + \Phi_{j+1,k}) + \\ & N_r(j,k)(S^2_{j,k} + \Phi_{j,k+1}) + \\ & N_d(j,k)(-S^1_{j-1,k} + \Phi_{j-1,k}) + \\ & N_l(j,k)(-S^2_{j,k-1} + \Phi_{j,k-1})) \end{aligned} \tag{2}$$

Este nuevo algoritmo converge, al menos en ausencia de errores en la medida.

3. GPU.

En los últimos años se ha introducido el concepto de hardware gráfico programable[2]. Las operaciones a realizar sobre cada elemento del *pipeline* gráfico se expresan en lenguajes de alto nivel y se permite un alto grado de libertad. De tal forma que las GPU's se han convertido en procesadores en sí mismos, en los que el objetivo inicial para el que fueron diseñados puede ser obviado para pasar a enfatizar las capacidades de la máquina de cómputo que encierran. Así la GPU se ha aplicado a tareas de cómputo de propósito genérico tales como motores de trazado de rayos[3] o el algoritmo de transformada rápida de Fourier[4].

Las capacidades, restricciones y el modo en que se adapta un algoritmo a la arquitectura peculiar de una GPU han de ser tenidas en cuenta si se quiere hacer un uso adecuado de ellas. Desde un punto de vista conceptual, una GPU puede ser entendida como una máquina que ejecuta un algoritmo para procesar un *stream* de datos de entrada y generar un *stream* de datos de salida. Un *stream* no es otra cosa que un conjunto de datos uniformes. Por cada elemento del *stream* de entrada habrá un elemento en el *stream* de salida, el algoritmo es el mismo para todos los elementos del *stream* y los cómputos sobre cada uno es independiente de los del resto. Durante la ejecución del algoritmo se puede leer pero no modificar datos en memoria. El algoritmo que se ejecuta recibe el nombre de *kernel* o *shader*.

Las salidas de un programa de la GPU no dejan de ser imágenes en color, ahora bien, cada pixel de color consta de cuatro componentes (rojo, verde, azul, alfa; RGBA) y las GPU's actuales permiten expresar

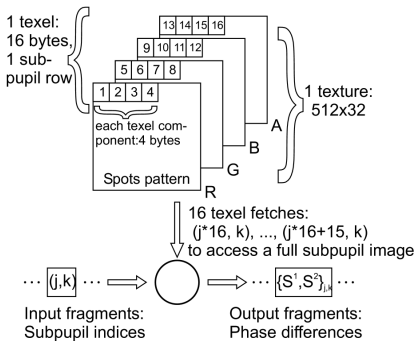

Figura 1: Algoritmo de centroides adaptado a la GPU.

cada una de esas componentes como números en punto flotante de 32 bits. Así, internamente, cada instrucción de máquina de la GPU trabaja sobre registros de 128 bits, operando a la vez sobre las 4 componentes, esto es, una arquitectura SIMD de 4 vías.

Se exige independencia en el cómputo para cada pixel para poder disponer múltiples unidades funcionales corriendo en paralelo de tal forma que el *pipeline* con sus diversas etapas puede considerarse replicado y con capacidad de dar salida a varios elementos de un *stream* a la vez. A esta forma de paralelismo unida al paralelismo SIMD a nivel de instrucción, se la designa como SIMD multi-hebra.

3.1. Implementación del cómputo de centroides en GPU

Las siguientes consideraciones han de ser tenidas en cuentas a la hora de adaptar el cómputo de centroides al modelo de *stream* de la GPU. Véase Fig. 1.

El patrón de puntos no es en sí el *stream* de entrada, sino que es accedido como memoria de sólo lectura. En su lugar los elementos de los *streams*, de entrada y salida, están relacionados con las subpupilas. La información relevante a la entrada son los índices de posición de las subpupilas: j, k. A la salida se espera obtener las diferencias de fase en esa subpupila: $S^1_{j,k}$ y $S^2_{j,k}$.

Cada una de las cuatro componentes, RGBA, de un elemento de textura (texel) tiene 32 bits y puede contener empaquetados 4 bytes, de tal modo que dentro de un registro interno de la GPU de 128 bits se dispone de dieciséis (4 componentes $\times$ 4 bytes por componente) elementos de 8 bits. Cuando la imagen pasa a la GPU, se manipula como 512×32 elementos de textura RGBA. De este modo un único elemento de

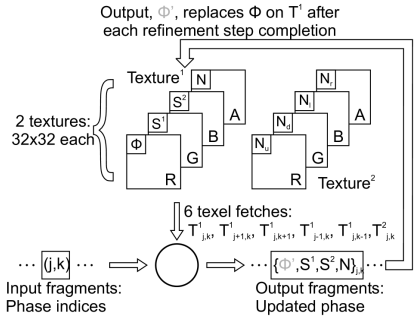

Figura 2: Algoritmo de Hudgin adaptado a la GPU.

textura es consultado (*fetched*, en terminología GPU), a la hora de procesar una fila de una subpupila. Al operar los datos se desempaquetan y se acumula los valores de intensidad pesados en función de su posición en la subpupila. De este modo se aprovecha la capacidad de la GPU de leer 128 bits de una vez y de operar con 4 componentes por instrucción.

El procesado de filas se repite 16 veces para completar una subpupila. Si suponemos que los puntos se sitúan en las filas intermedias de las subpupilas, podemos acelerar el cálculo reduciendo el procesamiento a la mitad de las filas.

3.2. Implementación del algoritmo de Hudgin en GPU.

Los índices de subpupilas (j, k) y los valores de fase ($\Phi'_{j,k}$) son los elementos individuales de los *streams* de entrada y salida respectivamente. El kernel del algoritmo se puede expresar directamente como un programa de fragmento, siguiendo la ecuación 2.

Los arrays Φ, S^1, S^2 y la información de vecindad son los valores necesarios para computar un nuevo valor de la fase. Se sitúan en dos texturas RGBA como se ilustra en la Fig. 2. La primera contiene datos fijos y variables [$\Phi_{j,k}$, $S^1_{j,k}$, $S^2_{j,k}$, $(N_u + N_d + N_l + N_r)(j, k)$]. La segunda permanece constante y contiene la información de los vecinos [$N_u(j, k)$, $N_d(j, k)$, $N_l(j, k)$ y $N_r(j, k)$].

Solo el primer componente de la primera textura se actualiza en cada paso. En ese componente se almacena la fase recuperada hasta el paso actual. Cuando se ha calculado todos los valores de Φ', ésta reemplaza a Φ en la textura que sirve de memoria de sólo lectura. El proceso se repite realizando otro paso de refinamiento de la solución.

4. Resultados obtenidos y análisis.

La tarjeta gráfica empleada para llevar a cabo este trabajo ha sido una Chaintech SA5900X. Tiene un motor gráfico nVidia GeForce FX 5900 XT (nv35) con 128 MB DDR SDRAM. Hace unos pocos meses nVidia ha lanzado la GeForce 6x00. Esta nueva serie de chips duplica y triplica en rendimiento al que hemos usado.

El sistema gráfico y el sistema huésped, un Pentium IV a 3 GHz con 1 GB de memoria y chipset i865, se comunican sobre un bus AGP 8x. La versión de los drivers Linux nVidia usada fue 1-6106 (actualizada a 30 Junio de 2004). Estos drivers soportan OpenGL 1.5, que fue la API de gráficos empleada. El compilador del lenguaje de shaders Cg fue el cgc versión 1.2.1001.

4.1. Centroides.

Cuadro 1: Tiempos de cómputo para el algoritmo de centroides.

Ejecuciones	Tiempo en milisegundos			
	GPU		CPU	
1	21	38	13	14
10	29	53	128	137
100	108	206	1253	1370
1000	901	1730	16108	17552
	half	full	half	full

El Cuadro 1 muestra los tiempos consumidos al ejecutar repetidas veces un cómputo de centroides. Aunque sólo se necesita realizar un cómputo por cada recuperación de fase, esta tabla ilustra el hecho de que la medida de rendimiento de la GPU falla cuando el *pipeline* no funciona a pleno rendimiento, lo cual sucede para pocas ejecuciones. Cuando el sistema de AO está funcionando al completo, el *pipeline* se mantiene lleno. Las medidas de rendimiento son consistentes cuando la relación entre ejecuciones y tiempo requerido se estabiliza, lo cual en este caso sucede por encima de las 100 ejecuciones. En esas condiciones un único paso consume aproximadamente 0'9 (*half*) y 1'7 (*full*) ms. Designamos *"full"* o *"half"* a los algoritmos de acuerdo a la porción de filas consideradas en una subpupila. La aceleración conseguida es de 10 veces para el caso *"full"* y casi 20 para el *"half"*.

Cuadro 2: Tiempos de cómputo para Hudgin en CPU y GPU.

subp.	tiempo milisegundos			
	32		64	
pasos	GPU	CPU	G	C
1	**0'62**	**0'04**	**0'68**	**0'23**
10	**0'88**	**0'40**	**1'63**	**2'18**
50	**2'10**	**2'01**	**5'84**	10'99
100	**3'60**	**4'02**	11'06	21'80
200	**6'62**	**8'05**	22'29	43'88

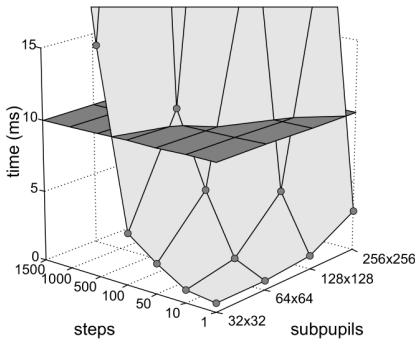

Figura 3: Tiempos de cómputo para la implementación en GPU de Hudgin. El plano horizontal subraya donde se supera el límite de 10 ms.

El Cuadro 2 muestra los tiempos que consume la implementación del algoritmo de Hudgin modificado para algunas combinaciones de pasos/subpupilas. Gráficamente se muestra en la Fig. 3 y la reducción del error en la Fig. 4.

Si se compara el rendimiento de nuestra implementación en GPU frente al rendimiento en CPU se observa una aceleración de 2x en las combinaciones de pasos/subpupilas salvo para aquellos casos donde el *pipeline* está infrautilizado. Esta relación se mantiene para tamaños mayores del problema, no incluidos en la tabla por falta de espacio.

5. Conclusiones.

Una fase de frente de onda se puede recuperar de un sensor Shack–Hartmann usando como único recurso computacional una GPU. Nuestra implementación demuestra ser, cuando menos, equivalente en rendimiento a una implementación sobre una

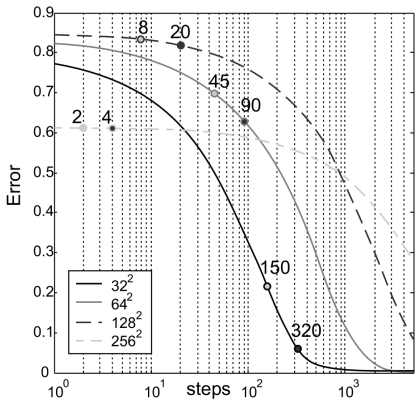

Figura 4: Evolución del error en el algoritmo de Hudgin para tamaños de subpupila de 32×32, 64×64, 128×128 y 256×256. Los círculos representan el número de pasos ejecutados en 5 y 10 ms.

CPU de gama alta. En la determinación de los centroides se obtuvo una aceleración de 10x, mientras que en la recuperación de la fase es de 2x. Con nuestra actual implementación zonal la recuperación de fases de $64x64$ o $128x128$ aún supera el límite de los 10ms.

6. Agradecimientos

Este trabajo ha sido parcialmente financiado por el "Gobierno Autónomo de Canarias" (PI2002/186), y por "Becas Cajacanarias-ULL de Investigación para Doctorandos (2004)".

7. Referencias

[1] R. Hudgin, "Wave-Front Reconstruction for Compensated Imaging," J. Opt. Soc. Am. A **67,** 375–378 (1977).

[2] E. Lindholm, M. J. Kilgard, and H. Moreton, "A user-programmable vertex engine," in *Proc. Comp. graph. and interactive tech.,* (ACM Press, 2001), pp. 149–158

[3] T. J. Purcell, *Ray tracing on a stream processor,* Ph. D. Thesis, (Stanford Univ., California, 2004).

[4] K. Moreland and E. Angel, "The FFT on a GPU," in *Proc. ACM SIGGRAPH Graph. hardware,* (Eurographics Association, 2003), pp. 112–119