Proyecto de Sistemas Informáticos.
Curso 2006-2007.

Registro de Imágenes utilizando Hardware Gráfico

Componentes del grupo:
Luis Rafael Canet Salazar
Rodrigo González Alberquilla
Salvador de la Puente González

Director del proyecto:
Christian Tenllado van der Reijden

Facultad de Informática.
Universidad Complutense de Madrid.

Registro de Imágenes Utilizando Hardware Gráfico

Memoria de proyecto fin de carrera presentada por Luis Rafael Canet Salazar, Rodrigo González Alberquilla y Salvador de la Puente González en la Universidad Complutense de Madrid, realizado bajo la dirección de Christian Tenllado van der Reijden.

Madrid, a 3 de Julio de 2007.

Prefacio

En este proyecto, estudiamos el registro de imágenes sobre hardware gráfico. Con esta técnica nos referimos a encontrar una transformación que haga posible el solapamiento entre dos imágenes.

En el Capítulo 1 realizaremos una breve introducción acerca del registro de imágenes, explicando en qué consiste y cuáles son sus aplicaciones, y discutiremos diferentes estrategias para llevarlo a cabo.

A continuación, describiremos el algoritmo del que partimos para realizar el registro, basándonos en el trabajo de I. De Falco *et al.* [FMS+07] y aportando nuestras propias mejoras y optimizaciones. Hablaremos también de su implementación y analizaremos el rendimiento en CPU.

A la vista de los resultado, propondremos una nueva plataforma de ejecución, la GPU, y detallaremos su estructura interna.

Seguidamente, estableceremos las bases del Modelo de Procesamiento de Flujos y trasladaremos el algoritmo de registro a este modelo.

Para finalizar, comprobaremos gráficamente los resultados obtenidos tanto en CPU como en GPU, observando una mejora importante en el rendimiento del algoritmo.

Abstract

In this project, we study the image registration on graphical hardware. This task consists in finding a suitable transformation to match two images.

In Chapter 1 we introduce image registration, explaining how it works and its applications, and showing different approaches to the solution.

After, we describe the algorithm based on the work of I. De Falco *et al.* [FMS+07], adding our own improvements and optimizations. We write about its implementation and its CPU performance.

With these results in mind, we introduce a new framework and its behavior, the GPU. Then, we make another approach to the image registration algorithm, following the guidelines of the Stream-Processing Model.

In the end, we examine the results between both models, showing how promising is the GPU approach.

Luis Canet, Salvador de la Puente y Rodrigo González autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, la memoria desarrollada en este proyecto.

Madrid, a 3 de Julio de 2007.

Luis R. Canet Salvador de la Puente Rodrigo González

Lista de palabras

GPGPU, registro de imágenes, imagen hiperespectral, GPU, información mutua, evolución diferencial, scattering, gathering, encaje de imágenes, histograma

A mi familia, que son lo que más quiero.

Luis.

A mamá, papá, mi hermana Paula, Bea y a mis amigos de aquí y de allí.

Salva.

A mis padres, Estela y todo ASCII.

Rodrigo.

Agradecimientos

Damos las gracias a Christian Tenllado, nuestro director de proyecto, por ayudarnos a llevarlo a término y a la gente del DACYA por darnos un espacio donde trabajar, y facilitarnos un framework base. En especial a Enrique de la Torre, por mantener los equipos preparados para realizar las pruebas y a Manuel Prieto por ser el referente en imágenes hiperespectrales.

Luis quiere dar las gracias a Edgardo Mejía, por su ayuda y consejos; y a Wenceslao Canet, por ser su mejor referente.

A Salvador le gustaría agradecer el apoyo de todos sus amigos, en especial el de Beatriz y su familia; y a sus compañeros de proyecto por aguantar su carácter.

Rodrigo quiere agradecer a Christian de nuevo y a Javier Setoain por ser oídos y respuesta a sus dudas. A Estela por el apoyo prestado en los momentos más duros. A Alberto Ferreiro por descubrirnos sus secretos a la hora de que la búsqueda converja.

Índice general

Capítulo 1

Introducción

1.1. El Registro de Imágenes

En el campo de la visión por computadora, cuando se toman imágenes de una misma escena u objeto, en diferentes instantes de tiempo, o desde diferentes posiciones, obtenemos información que se encuentra en diferentes ejes de referencia. Sabemos que es información de un mismo sujeto o lugar, pero no podemos hacer una comparación fiable entre los diferentes datos obtenidos, al estar tomados en diferentes condiciones. Además, esa diferencia de condiciones puede ser una característica de nuestro estudio, como seria el caso de imágenes de un terreno tomadas con un año de diferencia, para comparar la evolución de la vegetación, o estudiar fenómenos meteorológicos.

El registro de imágenes consiste en el estudio e integración de esa información en un único eje de referencia como se observa en la figura 1.1, esto es, una vez tomadas las imágenes, se analizan las similitudes existentes entre las mismas, y se busca una transformación que haga que podamos superponer

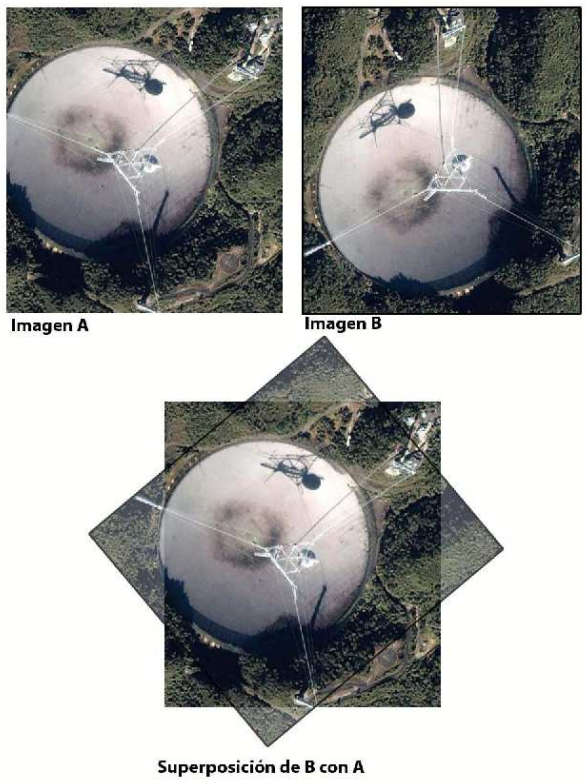

FIGURA 1.1: El registro de imágenes intenta encontrar la transformación que traslade al mismo marco de referencia ambas imágenes.

una imagen a la otra, teniendo la certeza de que la información de un punto superpuesto con otro en la imagen final, corresponde con el mismo lugar en la realidad.

Obteniendo esta nueva imagen, fruto de la superposición de ambas, podemos observar los cambios producidos entre unas condiciones y otras, por ejemplo con respecto al tiempo. También podemos coger una serie de imágenes de un mismo lugar, y crear una nueva imagen que incluya a todas, uniendo esas imágenes por sus lugares comunes. Esto nos permite obtener una imagen mas grande de un lugar, sin apreciar corte alguna entre las diferentes partes.

Teniendo estas posibilidades, se nos plantea el uso de imágenes con mayor

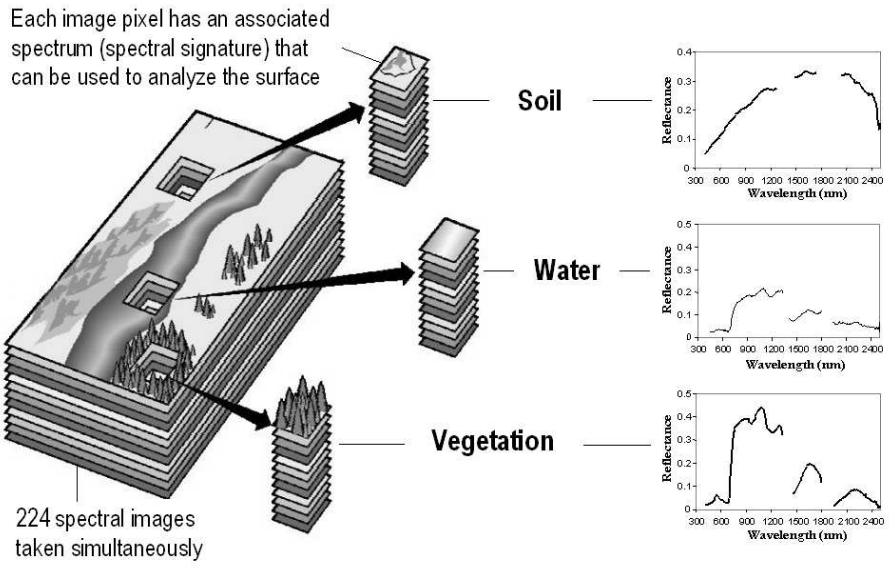

FIGURA 1.2: Representación de una imagen hiperespectral

tamaño que una imagen común, y que albergan mucha mas información que la percibida por el ojo humano, las imágenes hiperespectrales.

1.2. Imágenes Hiperespectrales

Las imágenes hiperespectrales, son imágenes de gran tamaño, con mayor información que una imagen común. Al tomar una de estas imágenes, no solo se almacena la información bidimensional típica de una imagen multiespectral, sino que además se añade un conjunto de canales a lo largo del espectro visible, infrarrojo, y ultravioleta.

Esta información es tomada por un espectrómetro, que recoge cientos de bandas espectrales, resultado de la reflexión de la luz en las diferentes regiones u objetos de la imagen. Esto nos permite estudios mucho mas relevantes, como por ejemplo, los materiales que presenta un terreno, o de los que esta hecho

un objeto, como puede comprobarse en la figura 1.2.

En el caso que nos ocupa, trabajamos con imágenes de porciones de terreno tomadas desde un satélite. Con esta información, podemos identificar las diferentes regiones que componen el paisaje, y estudiar la evolución de las mismas. Por ejemplo, la erosión que produce el agua en el cauce de un río, o el estado de una formación rocosa.

1.3. Algoritmos de Registro de Imágenes

Para enfrentarnos al registro de imágenes, tenemos diferentes formas de encontrar la transformación que unifica los ejes de referencia de las dos imágenes [Bro92].

- El mapeo de puntos, propone la creación de una serie de puntos de control, llamados *landmarks*, en ambas imágenes, que representan una misma característica. Partiendo de esas referencias, se intenta entonces emparejarlos. Una vez emparejados, se utiliza esa transformación para todos los píxeles de la imagen.

- El ajuste de superficies, valora las superficies en vez de los puntos de control. Utilizando varios algoritmos de segmentación, puede localizar satisfactoriamente superficies de alto contraste. Si dos superficies pueden ser identificadas por este método en diferentes imágenes, entonces se produce el ajuste de ambas. Este se utiliza especialmente en imágenes medicas, por ejemplo con la superficie epidérmica.

- Medidas de similitud de *Voxels* [1]: Estos métodos de registro utilizan la imagen entera sin reducirla o segmentarla a lo largo de todo el proceso, por lo que resultan más fiables que los basados en puntos. Sin embargo, dado que manejan una cantidad enorme de datos, tienen el inconveniente del gran coste computacional que ello supone. La idea de estos métodos es definir los parámetros del registro como una función de alguna medida de similitud entre las dos imágenes, e intentar maximizar esta medida de similitud [yJMV07].

- El método basado en la intensidad, utiliza las intensidades de la imagen como método de registro, de modo que no se requiere una segmentación explícita de cada imagen. Sin embargo, para la transformación global de la imagen el número de parámetros que han de ser optimizados suele ser enorme y llevarlo a la práctica puede resultar imposible [yJMV07].

- Método basado en la Información Mutua (*Mutual Information*, MI): Este método esta basado en maximizar la cantidad de información compartida por las dos imágenes. No se ayuda de ningún agente externo, o punto de control en la imagen, y tampoco precisa de una manipulación previa de la misma.

Viendo todas estas posibilidades, nos decantamos en nuestro estudio por la MI, puesto que es la que presenta el menor numero de restricciones. Esto supone un espacio de búsqueda muy grande, ideal para búsquedas aleatorias como lo son los algoritmos evolutivos. [FMS+07]

[1] Unidad cubica que compone un objeto tridimensional

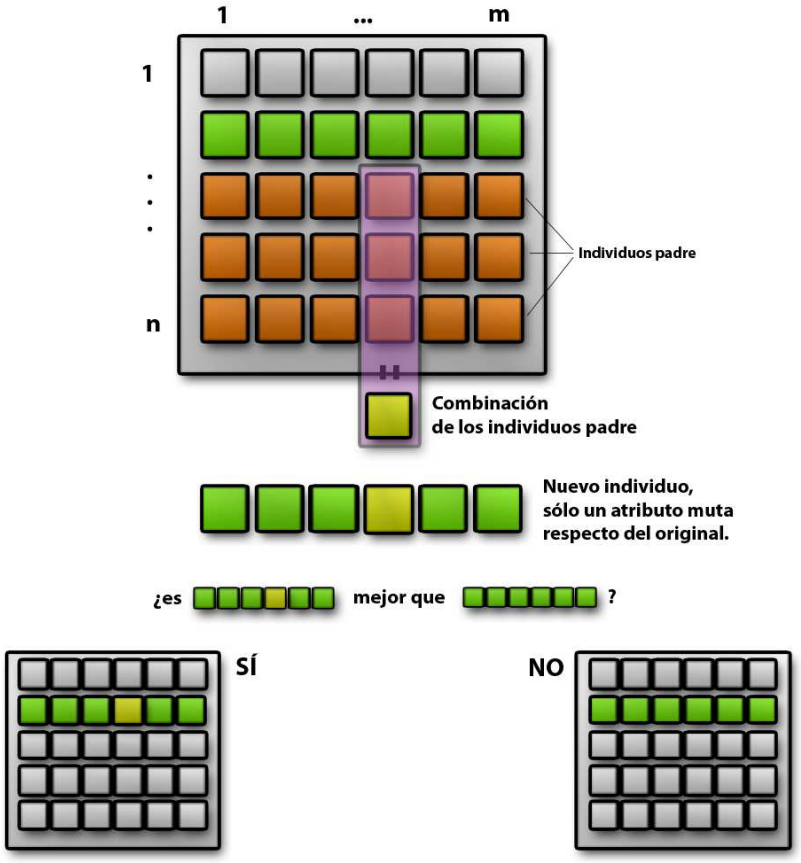

FIGURA 1.3: Los algoritmos evolutivos se rigen por las mismas leyes que la biología

Los algoritmos evolutivos son métodos de búsqueda en un espacio de soluciones. El nombre se debe, a que al igual que en las reglas de la evolución biológica, las soluciones mas aptas tienen mas posibilidades de sobrevivir. Dada una población inicial de soluciones potenciales a un problema, el algoritmo evolutivo selecciona un conjunto de individuos padre, con los que se generan nuevas soluciones hijo que extienden la población, si superan una función de aptitud. Los supervivientes son soluciones mas adecuadas como se observa en la figura 1.3.

En concreto, vamos a utilizar el algoritmo de Evolución Diferencial (*Differential Evolution*, DE), como recomienda el trabajo de I. De Falco *et al.* [FMS+07], en el cual, precisamente, se ofrece un método de registro de imágenes basado en MI.

1.4. GPGPU

Cómputo de Propósito General en Unidades de Proceso Gráfico (*General-Purpose Computing on Graphics Processing Units*,GPGPU) se puede definir como el paradigma que promueve la utilización de la Unidad de Proceso de Gráficos como una unidad de proceso de propósito general. Este nuevo punto de vista, nace a raíz de la inclusión en la tarjeta gráfica de módulos programables.

1.4.1. Orígenes y evolución

Las primeras tarjetas gráficas surgieron para aliviar la carga del proceso gráfico a la CPU. Esto permitía una mejora en el rendimiento general del sistema, aunque la funcionalidad de las tarjetas gráficas era bastante limitada. Pasado un tiempo, se incluyo la posibilidad de programar ciertas partes de la tarjeta gráfica, en concreto, los procesadores de fragmentos y vértices. Esto supuso una mejora importante, ya que entonces los desarrolladores podían explotar esta característica, creando nuevos efectos gráficos y nuevos modelos de iluminación.

Muchos de estos efectos explotaban el paralelismo de las unidades programables, y su rendimiento superior de calculo en coma flotante, respecto de

las CPUs. Todo ello, sumado al abaratamiento de las tarjetas gráficas en el mercado, permitió que los desarrolladores mas innovadores tuvieran acceso a equipos con múltiples unidades de procesamiento en paralelo.

Además, con el paso del tiempo aparecieron ciertos problemas de rendimiento al operar sobre vectores y matrices. Las CPUs proporcionaron una solución mediante un conjunto de instrucciones nuevo. Aun así, las CPUs seguían siendo demasiado generalistas, y el conjunto de instrucciones demasiado limitado. Se necesitaba mayor rendimiento, y mayor especialización.

Las GPUs parecían la solución indicada, sin embargo, aun era necesario una serie de técnicas que permitiesen traducir los algoritmos diseñados para CPU, en algoritmos equivalentes que explotasen las características únicas de las tarjetas gráficas. Estos nuevos algoritmos encajan en el modelo de procesamiento de flujos, cuyos detalles ampliaremos en el Capítulo 4. Basta decir que el modelo de flujos permite aplicar un programa a un conjunto de datos, operando sobre los mismos en paralelo.

1.4.2. Perspectiva futura

La GPU NVIDIA 8800 es la primera tarjeta gráfica en introducir un modelo unificado de hardware totalmente programable, donde se elimina la visión clásica de un conjunto de módulos programables independientes, y ofrece como característica única un lenguaje de programación orientado al desarrollo de software general sobre GPU, CUDA.

Teniendo estas posibilidades, no seria de extrañar que las compañías ofrezcan productos de alto rendimiento, basados en sus soluciones gráficas.

1.5. Motivación

Múltiples aplicaciones en el mundo de la medicina [MV98], como el estudio del desarrollo de un tumor [SMH+07], el deterioro oseo de una persona [HBHH01], o el análisis detallado de la retina del ojo [ret02].

También es utilizado para el estudio del efecto del paso del tiempo en diversos campos, como las construcciones antiguas o excavaciones arqueológicas [JEE77], o las formaciones montañosas de un paisaje [rid00].

El principal problema de esta técnica, es el tiempo de computo. Al procesar las imágenes, se estudian píxel a píxel. Para una imagen de tamaño medio (800x600), tenemos que procesar 480000 píxeles. Es una cantidad muy elevada, y el procesamiento se hace secuencialmente, por lo que es muy lento.

Por otro lado, en este algoritmo existe una característica que podemos explotar, y es que el estudio de cada píxel es independiente de los demás, por lo que podríamos procesar varios píxeles en paralelo sin llegar a una solución errónea. De esta forma, disminuiría mucho el tiempo de registro de las imágenes, proporcionalmente al numero de píxeles en paralelo que podríamos computar.

1.6. Objetivos

Partiendo de la idea del registro de imágenes hiperespectrales expuesto anteriormente, y los avances en las GPUs actuales, se propone el desarrollo de una aplicación para el registro automático de imágenes, prestando especial atención a la optimización de sus algoritmos desde un punto de vista compu-

tacional. Para ello, se abordara el problema del registro siguiendo el algoritmo expuesto en el Capítulo 2, basado en la Evolución Diferencial, y se plantea la utilización de unidades de procesamiento gráfico (GPUs) programables para acelerar aquellas fases más costosas de la aplicación susceptibles de explotar las capacidades de este tipo de plataformas.

1.7. Organización del resto de este documento

- En el Capítulo 2 describimos el algoritmo del que partimos para realizar el registro de imágenes, y su implementación y análisis de rendimiento en la CPU.
- En el Capítulo 3, entramos en el ámbito de las GPUs, analizando la evolución de esta arquitectura, y describiendo el *pipeline* gráfico de la misma.
- En el Capítulo 4, describimos el modelo de procesamiento de flujos sobre GPU, e implementamos el registro de imagen haciendo uso de este modelo. que partes del algoritmo son abordables por los diferentes módulos de la arquitectura.
- En el Capítulo 5, comparamos los resultados de CPU y GPU.
- En el Capítulo 6, desarrollamos las conclusiones a las que hemos llegado y planteamos algunas mejoras.

Capítulo 2

Algoritmo para el registro de Imágenes Hiperespectrales

En el Capítulo 1 hemos hablado sobre el registro de imágenes y elegido la estrategia a seguir, basada en la MI, descrita en la sección 1.3.

A lo largo de este capítulo desarrollaremos esta estrategia y los algoritmos que la integran tomando como referencia el trabajo de I. De Falco *et al.* [FMS+07].

Comenzaremos analizando el algoritmo de búsqueda DE[1], y la prueba de aptitud basada en MI[2].A continuación propondremos algunas mejoras y terminaremos con los resultados de la implementación sobre CPU.

[1]Differential Evolution
[2]Mutual Information

2.1. Algoritmo de Evolución Diferencial

Los algoritmos evolutivos son sistemas de resolución de problemas de optimización o búsqueda que se rigen por las leyes de la evolución. Trabajan variando una población de soluciones, y evaluando la aptitud de los individuos, como expresa el siguiente esquema:

Algorithm 1 Algoritmo Evolutivo

```
Inicializar_Poblacion(P)
for i = 1 to generaciones do
  for each Individuo i in P do
    i_mutado = Mutar(i)
    if es_Mejor(i_mutado, i) then
      i = i_mutado
    end if
  end for
end for
solucion = Mejor(P)
```

Las diferentes formas de Algoritmo Evolutivo dependen de como se comporten las funciones $Mutar(Individuo)$ y $es\_Mejor(Individuo, Individuo)$. La función $Mutar(Individuo)$ actúa sobre un conjunto de individuos *padre* seleccionados aleatoriamente, y el individuo *hijo* es generado en función de sus padres. Por otro lado $es\_Mejor(Individuo, Individuo)$ devolverá *true* si la medida de similitud (*Measure Of Match*, MOM) de a es superior a la MOM de b según el criterio que estemos estudiando.

Por ejemplo, en un problema de maximización, $es\_Mejor(n_1, n_2)$ sería equivalente a $n_1 > n_2$.

En DE, dado un problema de maximización con m parámetros reales, y eligiendo un tamaño de población de n individuos, cada individuo representará

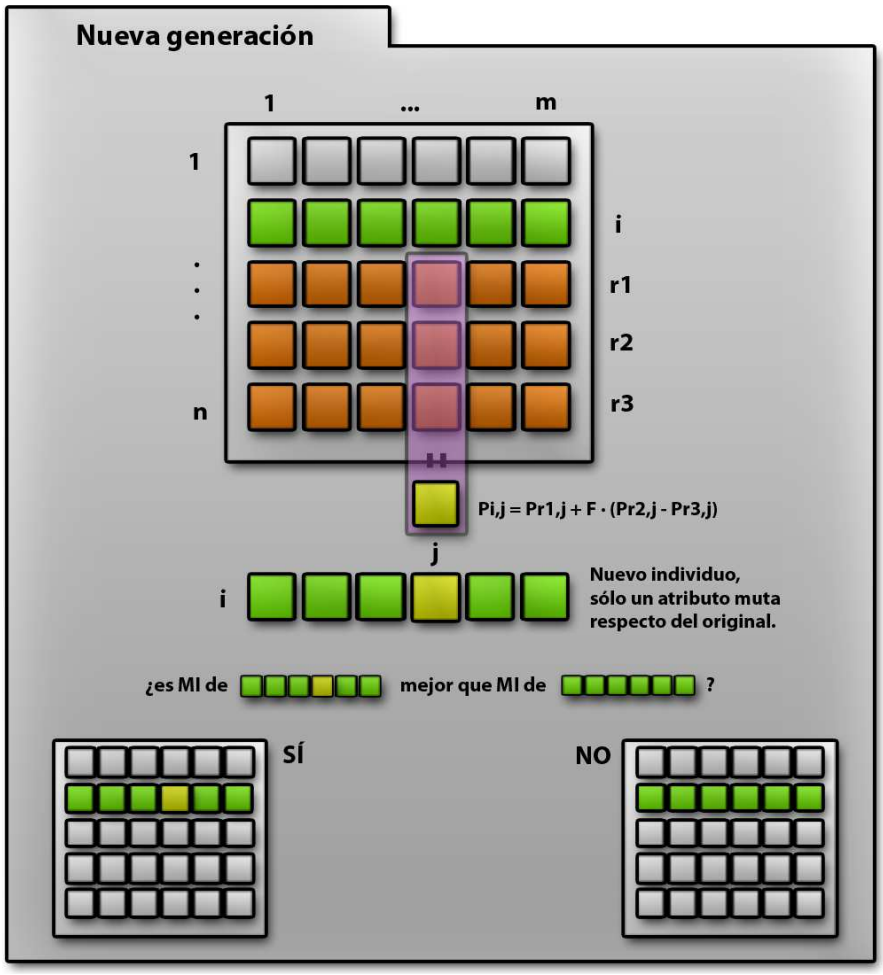

FIGURA 2.1: Cómo se genera un nuevo individuo en una generación

una solución potencial como un vector de m valores reales. Como se observa en la figura 2.1, la función $Mutar(Individuo)$ afecta a un sólo parámetro j de un individuo cualquiera i. Para ello, primero selecciona tres individuos *padre* por la posición de los mismos en la población (r_1,r_2 y r_3), todas distintas entre sí. A continuación, genera el nuevo parámetro mediante la expresión:

$$P_{i,j} = P_{r_3,j} + F \cdot (P_{r_1,j} - P_{r_2,j}) \tag{5}$$

F es un parámetro del algoritmo en [0,0 . . . 1,0] que expresa la magnitud con la que la diferencia $(P_{r_1,j} - P_{r_2,j})$ se aplica a $P_{r_3,j}$. La DE debe su característica *diferencial* precisamente al hecho de aplicar una diferencia entre dos de los padres con una magnitud F.

Sin embargo, la función $Mutar(Individuo)$ sólo sucede si se da alguna de las siguientes condiciones:

- Si un número real aleatorio p en [0,0 . . . 1,0] es menor que la tasa de cambio (*Changue Rate*, CR) introducida en el algoritmo como parámetro.
- Si un número aleatorio k en $[1, m]$ es exactamente j, es decir, coincide con el número de parámetro seleccionado para mutar.

Si ninguno de estos requisitos se verifica entonces el parámetro se mantiene intacto.

Este individuo mutado que designaremos *candidato* es comparado contra el individuo original. Si resulta mejor solución según el criterio de la función $es\_Mejor(Individuo, Individuo)$ reemplaza al mismo en la próxima población. De otra forma, es el original el que sobrevive y pasa a la nueva población. El mismo patrón se repite un máximo de g generaciones.

En nuestro caso, los individuos son vectores de 6 elementos $(rot_1, rot_2, rot_3, rot_4, dx, dy)$ que representan las transformaciones afines que pueden ser solución del problema de registro de imágenes como se explicó en la sección 1.1. Los cuatro primeros elementos, $(rot_1, rot_2, rot_3, rot_4)$, forman una rotación bidimensional y los dos últimos, (dx, dy), un vector desplazamiento en el plano XY. Nuestro criterio de aptitud, es decir, la base de la función $es\_Mejor(Individuo, Individuo)$ es la MI como ya vimos en la sección 1.3.

2.2. Cálculo de la información mutua

En teoría de la probabilidad, y en teoría de la información, la información mutua de dos variables aleatorias es una cantidad que mide la dependencia mutua de las variables.

Esta será nuestra MOM y se calcula en función de la expresión 2.1:

$$I(Y,Z) = \sum_{y,z} P_{Y,Z}(y,z) \cdot \log \frac{P_{Y,Z}(y,z)}{P_Y(y) \cdot P_Z(z)} \tag{2.1}$$

donde $P_Y(y)$ y $P_Y(z)$ son las funciones de masa de probabilidad marginal y $P_{Y,Z}(y,z)$ es la función de masa de probabilidad condicionada.

La MI esta relacionada con las entropías como sigue:

$$I(Y,Z) = H(Y) + H(Z) - H(Y,Z) \tag{2.2}$$

siendo $H(Y,Z)$ la entropía conjunta y $H(Y)$ y $H(Z)$ las entropías de Y y Z respectivamente. La definición de estas entropías es:

$$H(Y) = -\sum_{y} P_Y(y) \cdot \log P_Y(y), \qquad H(Z) = -\sum_{z} P_Z(z) \cdot \log P_Z(z) \tag{2.3}$$

$$H(Y,Z) = -\sum_{y,z} P_{Y,Z}(y,z) \cdot \log P_{Y,Z}(y,z) \tag{2.4}$$

Para calcular las probabilidades, debemos utilizar el histograma conjunto del par de imágenes h. Este se define como una función de dos variables Y y Z con la intensidad de grises de las dos imágenes como podemos ver en la figura

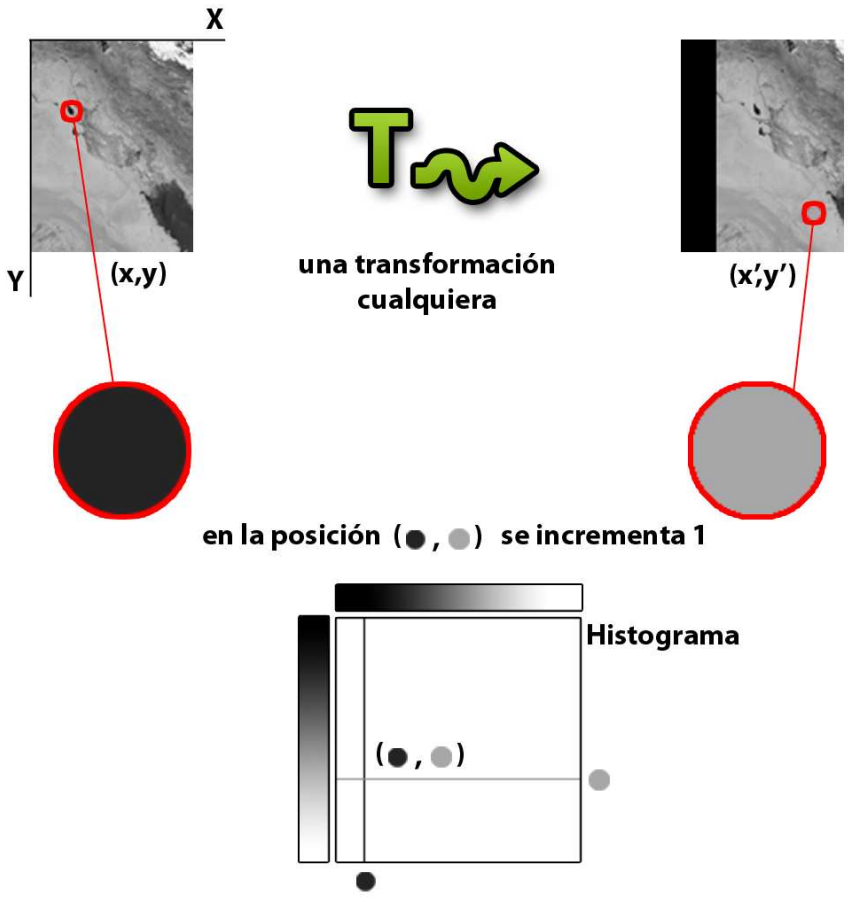

FIGURA 2.2: Elaboración del histograma

2.2. Su valor en la coordenada (Y, Z) es el numero de pares correspondientes teniendo nivel de grises Y en la primera imagen y nivel de grises Z en la segunda. La función conjunta de masa de probabilidad se obtiene normalizando el histograma conjunto del par de imágenes:

$$P_{Y,Z}(y,z) = \frac{h(y,z)}{\sum_{y,z} h(y,z)} \tag{2.5}$$

$$P_Y(y) = \sum_z P_{Y,Z}(y,z), \qquad P_Z(z) = \sum_y P_{Y,Z}(y,z) \tag{2.6}$$

El registro de imágenes utilizando la MI establece que un par de imágenes están alineadas según una transformación geométrica T cuando $I(Y(x), Z(T(x)))$ es máxima. Por lo tanto debemos buscar la transformación T que maximice I.

2.3. Optimizaciones a los algoritmos

Dos estrategias empleadas para favorecer la convergencia propuestas por Alberto Ferrería Blanco en [Bla07] son:

1. Para calcular la función de masa conjunta $P_{Y,Z}(y, z)$ lo que hacemos es dividir cada valor $h(y, z)$ entre el sumatorio del histograma, como se muestra en la ecuación 2.5 y a continuación se reduce para calcular P_Y y P_Z. $\sum_{y,z} h(y, z)$ es el tamaño del área solapada. La mejora propuesta es en vez de usar este valor, usar el tamaño de las imágenes para normalizar de manera que las ecuación quedaría:

$$P_{Y,Z}(y, z) = \frac{h(y, z)}{ancho(Y) \cdot alto(Y)} \tag{2.7}$$

   De esta manera cuanto menor sea el área solapada menores serán las probabilidades en comparación con el resultado de no aplicar esta mejora. Con este cambio estamos penalizando a aquellas soluciones que tienen un área de solapamiento pequeño con el objetivo de evitar casos degenerados en los que la aparición de nubes en los bordes puede resultar problemática, ya que la población tendería en ese caso a encajar zonas pequeñas que al estar cubiertas por nubes tienen unos colores similares

y obtendría un valor de MI elevado.

2. El uso cada un cierto número de generaciones de una estrategia de máxima pendiente, esto es, intentar adivinar que dirección debería tomar el mejor individuo para incrementar su MI. El procedimiento es, cada 5 generaciones tomamos al mejor individuo. Calculamos el gradiente de la función $I(Y_T, Z)$,donde Y_T representa la imagen Y transformada según los valores del individuo, siendo cada componente del gradiente la derivada parcial en cada uno de los parámetros de la transformación, lo que nos da la dirección (teórica) de mayor crecimiento de la MI: $\nabla I(Y_T, Z)$. Generamos un nuevo individuo sumándole esa cantidad al mejor individuo y lo evaluamos. Si es mejor, entonces sustituye al que era mejor individuo de nuestra población. Si no, es desechado. De esta manera conseguimos guiar la búsqueda hacia zonas donde debería encontrarse el máximo.

Para la implementación de estas mejoras creamos tres versiones de la aplicación, una primera que no incluye ninguna mejora, una segunda que incluye la mejora 1, y una última que incluye las mejoras 1 y 2.

2.4. Descripción de Implementación CPU

En esta sección vamos a mostrar diagramas de flujo y el pseudocódigo que describen como hemos implementado los algoritmos discutidos anteriormente.

2.4.1. Algoritmo de Evolución Diferencial

A continuación mostramos el flujo de ejecución de la DE en la figura 2.3, junto con su pseudocódigo, algoritmo 2, según esta descrito en la sección 2.1.

FIGURA 2.3: Implementación del algoritmo de evolución diferencial

Algorithm 2 Algoritmo de Evolución Diferencial

```
for i = 0 to tamPoblacion do
    Crear_Individuo()
    Calcular_MI_Individuo()
end for
for i = 0 to numGeneraciones do
    Calcular_Siguiente_Generacion() {Detallado mas adelante}
end for
```

En detalle, cada nueva generación se calcula como se indica gráficamente en la figura 2.4, acompañada de su pseudocódigo (algoritmo 3).

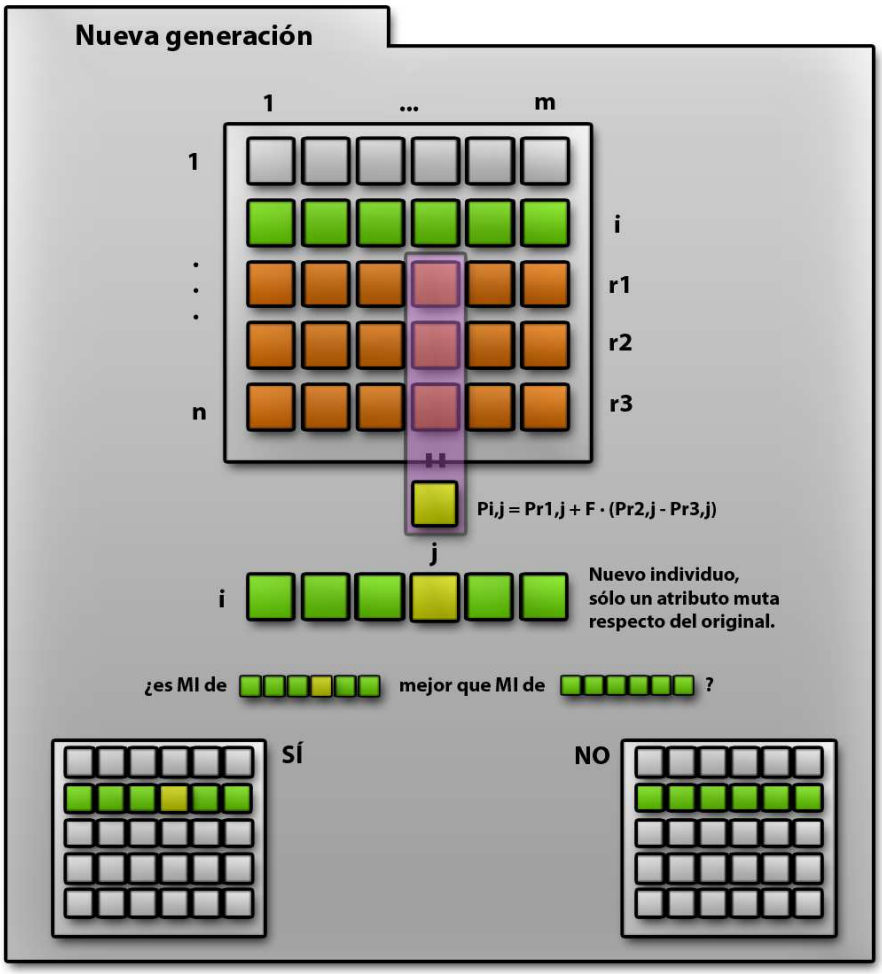

FIGURA 2.4: Cómo se genera un nuevo individuo en una generación

Algorithm 3 Nueva generación

```
for i = 0 to tamPoblacion do
    {Para cada individuo de la Poblacion}
    Generar_Individuos_Aleatorios()
    X = poblacin[i]
    candidato = poblacin[i]
    mutar(candidato, individuos_aleatorios)
    Calcular_MI(candidato)
    if candidato.MI > X.MI then
        X = candidato
    end if{Nos quedamos con el que tenga mejor MI}
end for{Al terminar tenemos una nueva generacion}
```

2.4.2. Cálculo de la MI

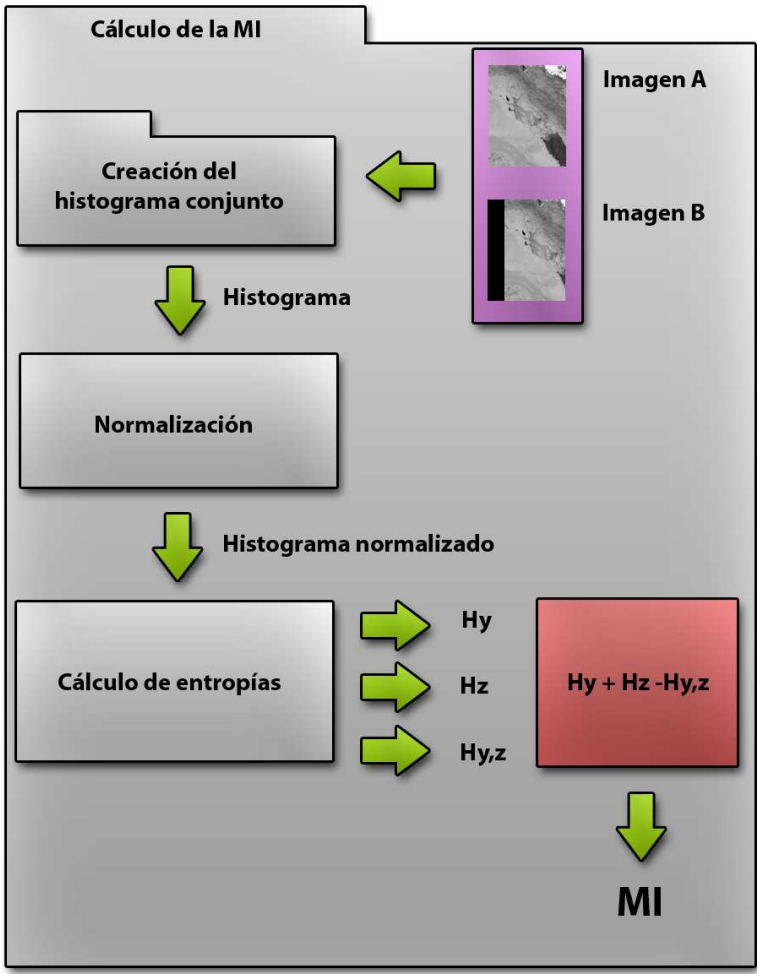

FIGURA 2.5: Implementación del cálculo de la MI en CPU

Implementar la información mutua sobre CPU consiste únicamente en aplicar las fórmulas descritas en 2.2 como puede comprobarse en la figura 2.5, acompañado del algoritmo 4. Como optimización hemos decidido precomputar en distintas matrices los valores de P_Y, P_Z y $P_{Y,Z}$ así como H_Y, H_Z y $H_{Y,Z}$ con el fin de reutilizar cómputos.

Algorithm 4 Calculo de la MI

Creacion_Histograma_Conjunto() {Detallado mas adelante}
Normalizacion_Histograma_Conjunto() {Calculo de P_Y, P_Z y $P_{Y,Z}$}
Calculo_Entropias() {Calculo de H_Y, H_Z y $H_{Y,Z}$}
$MI = Entropia\_de\_Y + Entropia\_de\_Z - Entropia\_Conjunta\_Y\_Z$
{Obtenemos así la MI de un individuo}

El funcionamiento del calculo del Histograma se muestra en la figura 2.6, y el pseudocódigo puede verse en 5

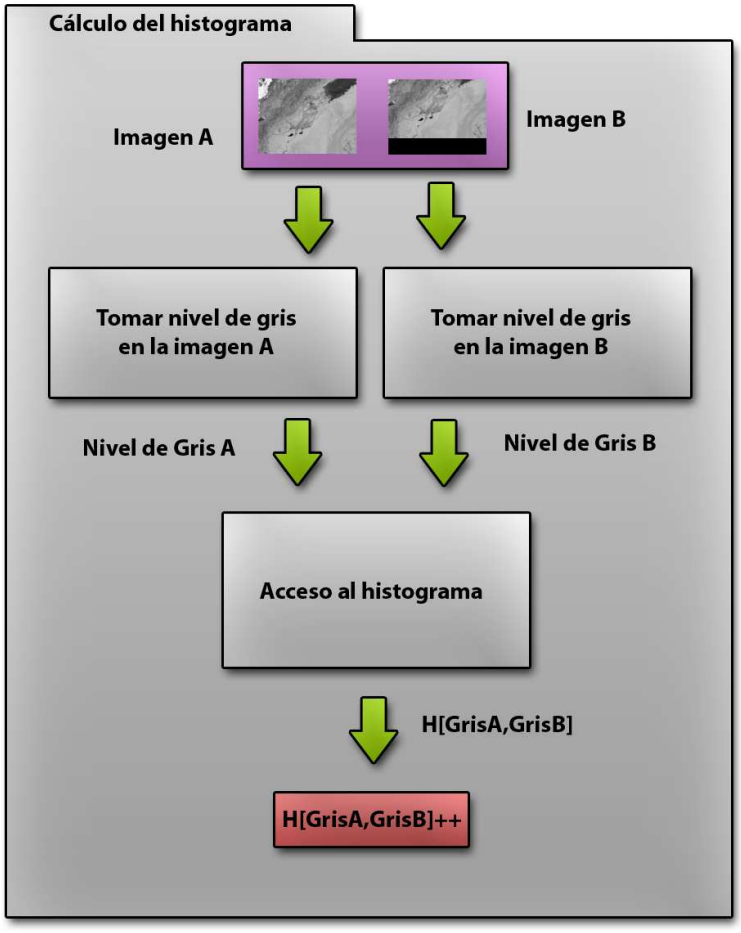

FIGURA 2.6: Implementación CPU del histograma

Algorithm 5 Cálculo del Histograma

for $i = 0$ to $tamao\_imagen$ **do**
  $i\_transformada = T \cdot i$
  $ImA = Valor\_De\_Gris(ImagenA, i)$
  $ImB = Valor\_De\_Gris(ImagenB, i\_transformada)$
  $Histograma[ImA, ImB] + +$
end for

2.5. Resultados obtenidos

En esta sección vamos a analizar los resultados obtenidos en CPU. Primero haremos una somera descripción de la plataforma experimental usada y los datos de entrada. Después presentaremos tablas que muestran los resultados de la ejecución del programa. A continuación analizaremos qué número de generaciones y de individuos necesita la evolución diferencial para garantizar que encontrará buenas soluciones. Y para terminar mostraremos las tablas de tiempos obtenidas para varias ejecuciones del algoritmo con diferentes datos de entrada y tamaños de histograma. Más adelante, en la sección 4.4 mostraremos los tiempos obtenidos en la ejecución en GPU, y en el capítulo 5 se hace una comparativa y un análisis de los tiempos en ambas plataformas.

2.5.1. Plataformas de ejecución

Las ejecuciones se han llevado a cabo sobre procesadores Intel, la tabla 2.1 describe los aspectos principales de la plataforma experimental. Las implementaciones han sido compiladas utilizando GNU-C/C++ compiler (version 4.1.2, con los flags de optimización *-O3 -msse*). Además, la implementación en ha sido optimizada para aprovechar al máximo las localidades de la cache para

un mayor rendimiento.

TABLA 2.1: Descripción técnica de la CPU

Características	**Core 2 Duo**
Año	2006
FSB	1066 MHz
ICache L1	32KB
DCache L1	32KB
L2 Cache	4M compartida
Memoria	2 GB
Reloj	2.4 GHz

2.5.2. Imágenes de prueba

Para las pruebas han sido utilidadas imágenes sintéticas de tamaños 128x128, 256x256, 512x512, 1024x1024 y 2048x2048 extraídas de una fotografía tomada con un satélite al telescopio Ikonos de Puerto Rico e imágenes hiperespectrales de los sensores Ali e Hyperion.

2.5.3. Resultados con imágenes sintéticas

Las tablas 2.2 y 2.3 recogen los resultados obtenidos de promediar 5 ejecuciones del algoritmo para distintos tamaños de población y número de generaciones de evolución de dichas poblaciones.

Hay que tener en cuenta que el hecho de aplicar la normalización mejorada hace que el valor de la información mutua se reduzca, en función del tamaño del área solapada. En este caso la información mutua máxima es, calculada con la normalización 'tradicional' 2.3619, y para la normalización mejorada 1.8177. Vemos como el valor máximo se consigue para 300 generaciones con 200 individuos en cualquiera de las versiones.

200 generaciones			
Mejora	**100 individuos**	**150 individuos**	**200 individuos**
Sin mejoras	2.3564	2.3603	1.9063
Normalizacion	1.8172	1.8171	1.7988
Normalizacion+Gradiente	1.6322	1.2383	1.7957
250 generaciones			
Mejora	**100 individuos**	**150 individuos**	**200 individuos**
Sin mejoras	1.4189	1.4866	2.361
Normalizacion	1.8175	1.8176	1.8174
Normalizacion+Gradiente	1.8176	1.8176	1.8176
300 generaciones			
Mejora	**100 individuos**	**150 individuos**	**200 individuos**
Sin mejoras	2.3619	2.3619	2.3619
Normalizacion	1.8176	1.8177	1.8177
Normalizacion+Gradiente	1.8177	1.8176	1.8177

TABLA 2.2: Información mutua máxima obtenida tras 200, 250 y 300 generaciones

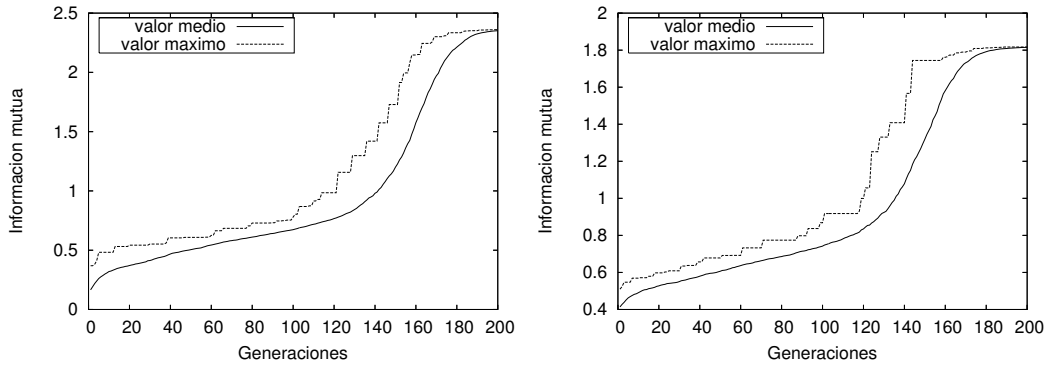

FIGURA 2.7: MI media y máxima para las versiones: sin mejoras y con la normalización mejorada con imágenes sintéticas.

Las figuras 2.7 y 2.8 muestran las gráficas de la información mutua media y máxima para la convergencia más rápida de cada una de las versiones.

2.5.4. Número de generaciones

El número de generaciones durante las que va a evolucionar una población nos indica cuanto vamos a afinar la solución, esto es, con un numero demasiado bajo de generaciones, nuestros individuos tenderán a la solucion pero no se

200 generaciones			
Mejora	**100 individuos**	**150 individuos**	**200 individuos**
Sin mejoras	200	192	200
Normalizacion	180	184	200
Normalizacion+Gradiente	200	200	200
250 generaciones			
Mejora	**100 individuos**	**150 individuos**	**200 individuos**
Sin mejoras	250	250	233
Normalizacion	218	207	221
Normalizacion+Gradiente	189	207	195
300 generaciones			
Mejora	**100 individuos**	**150 individuos**	**200 individuos**
Sin mejoras	227	212	202
Normalizacion	250	191	208
Normalizacion+Gradiente	194	235	204

TABLA 2.3: Generacion de convergencia con tope 200, 250 y 300 generaciones

acercarán lo suficiente. A medida que aumentamos el número de generaciones los individuos se van refinando, hasta que llega un punto en el que sin importar cuanto más aumentemos el numero de generaciones los individuos no pueden mejorar, y es ese punto el que debemos estimar.

Viendo los datos de la tabla 2.2, ese punto lo podemos estimar en 250 generaciones, ya que para este valor, se consiguen resultados máximos para las versiones mejoradas. En el caso de la version sin mejoras habría que llegar a las 300 generaciones.

Si bajamos el numero de generaciones de 250 nos arriesgamos a no conseguir una solución como vemos que pasa en la tabla 2.2, para el caso de 200 generaciones con las dos mejoras, que con 100 y 150 individuos obtiene resultados demasiado bajos o el el resultado de 250 generaciones en la versión sin mejoras que tiene dos malos resultados. Elevarlo por encima de 350 va a consumir tiempo de cómputo sin aportar mejoras a nuestra solución.

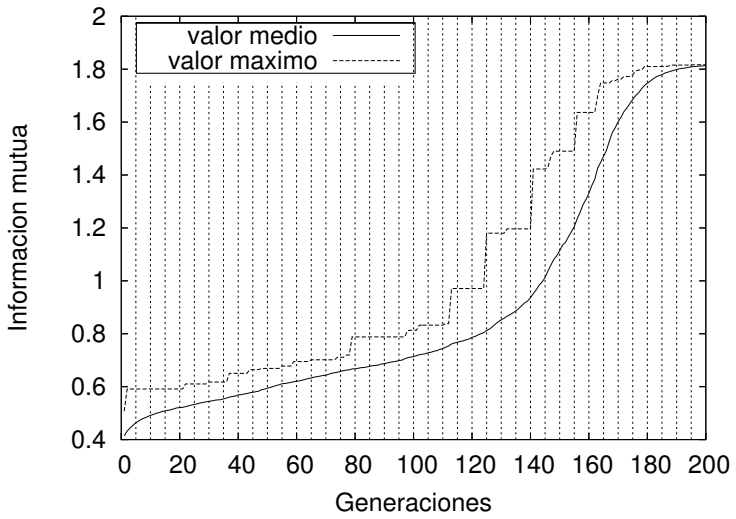

FIGURA 2.8: MI media y máxima para la versión con la normalización mejorada y el uso del gradiente con imágenes sintéticas.

2.5.5. Tamaño de la población

El tamaño de población nos determina la variabilidad con la que vamos a contar, cuantos más individuos tenga nuestra población más exhaustivamente buscaremos en nuestro espacio. Si tenemos un numero pequeño de individuos buscaremos en entornos reducidos, o repartidos de manera dispersa por el espacio de búsqueda, si, en cambio, tenemos un numero demasiado grande, las zonas de búsqueda se solaparán, lo que quiere decir que buscaremos varias veces en el mismo sitio, que de nuevo es desperdiciar recursos. Por eso debemos estudiar cual es el rango de tamaño ideal para una población.

Observando la generación en la que ha convergido el algoritmo, entendiendo como momento de convergencia aquel en el que la distancia entre el individuo más apto de la población y la media es menor del 1 %, mostrada en la tabla 2.3, podemos estimar que el tamaño ideal se encuentra entre 150 y 200 individuos,

ya que para sólo 100 individuos se suele alcanzar el máximo de generaciones, lo que indica que el individuo no ha convergido, o ha convergido a un máximo local.

Si usásemos menos de 150 individuos, puede que ninguno contuviese una buena solución en su entorno, y por tanto se estancarían en máximos locales como podemos apreciar en la tabla 2.2-250 generaciones, en el caso de la versión sin mejoras para 100 y 150 individuos, que a pesar de vivir 250 generaciones los resultados son bastante pobres. Si tenemos más individuos de 350 las zonas se solaparán y nuestra solución no mejorará, ya que vemos que en los casos en los que tenemos al menos 250 generaciones, con este número de individuos es suficiente.

En el caso de elegir que una población tenga una vida de 200 generaciones, habría que aumentar el número de individuos, pues tienen mucho menos tiempo para mezclarse, y por lo tanto hay que dotar a la población de una mayor variabilidad.

2.5.6. Mejoras a la convergencia

Para estudiar las ventajas de las mejoras nos fijaremos en las figuras 2.7 y 2.8. Podemos ver cómo la versión sin mejoras tiene un ascenso más suavizado, esto es debido a que hay individuos que se benefician de que la zona de solape es demasiado pequeña y si son similares el resultado de MI es alto pero todos sus hijos tenderán a tener una mala MI por lo que el progreso es más lento.

En cambio en el caso de la normalización mejorada vemos como se producen saltos significativos en un espacio de pocas generaciones esto es debido a

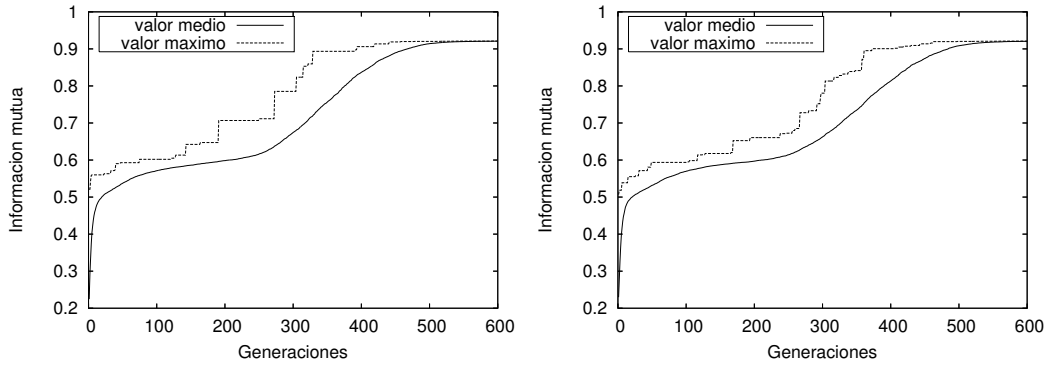

FIGURA 2.9: MI media y máxima para las versiones con la normalización mejorada y con la normalización mejorada y el uso del gradiente, con imágenes reales.

que dado un individuo, cuando se genere su hijo, si este tiene un menor solape, deberá ser mucho más parecido para ser aceptado, esto hace que se desechen varias mutaciones antes de aceptar una, por eso se tienen varias generaciones sin apenas mejora, y repentinamente hay un salto.

En el caso del uso del gradiente se puede ver cómo en las generaciones 125, 140 y 155 podemos apreciar un salto significativo, producido probablemente por el uso del gradiente.

A continuación mostramos un estudio simplificado realizado con las imágenes reales de Ali e Hyperion para una población con 400 individuos durante 600 generaciones.

2.5.7. Resultados con imágenes reales

Habiendo realizado tres ejecuciones con cada nivel de mejora llegamos a la conclusión de que no tiene sentido la ejecución sin mejoras, pues en ninguno de los casos dio solución, ni siquiera aproximada, por lo que la discusión se centrará en la mejora de la normalización (nivel 1) y esta misma con el uso

del gradiente (nivel 2).

La media de las MI obtenidas para el caso del nivel 1 es 0,921094333, y el máximo es 0.921146. En el caso del nivel 2 de mejora, la media obtenida es, 0,921089333 y el máximo está en 0.921198.

Para estudiar la generación de convergencia tomaremos dos criterios

1. El algoritmo converge cuando el valor máximo supera al medio en menos de un 1 %. En este caso las generaciones de convergencia son, en media, para el nivel 1: 509, nivel 2: 516.667.

2. El algoritmo converge cuando el valor máximo dista del máximo final en menos de un 1 %. En este caso las generaciones de convergencia son, en media, para el nivel 1: 447.333, nivel 2: 442.

Adoptar estos dos criterios es necesario porque el uso del gradiente puede verse en cierta manera como 'anticonvergencia' porque se incrementa el valor de MI únicamente del individuo con MI máxima, lo cual acrecenta la distancia que va en contra del primer criterio, en cambio vemos como con el segundo criterio sí se nota cierta mejoría, y ya que en todas las ejecuciones se encuentra un valor final aceptable el criterio es aplicable.

2.5.8. Tiempos obtenidos

Las ejecuciones test han sido llevadas a cabo durante 200 generaciones para 100 individuos que son suficientes para que se obtenga una solución con imágenes sintéticas. Los tiempos de ejecución para ejecuciones con un tamaño de histograma fijo, 256 colores, para imágenes sintéticas de 128x128, 256x256, 512x512, 1024x1024, 2048x2048 se muestran en la tabla 2.4.

Tamaño de imagen	128x128	256x256	512x512	1024x1024	2048x2048
Tiempo de ejecución	27.1183	75.1726	295.887	1136.72	4617.41

TABLA 2.4: Tiempos medios de 5 ejecuciones para un tamaño de histograma 256 sobre CPU para una población de 100 individuos durante 200 generaciones

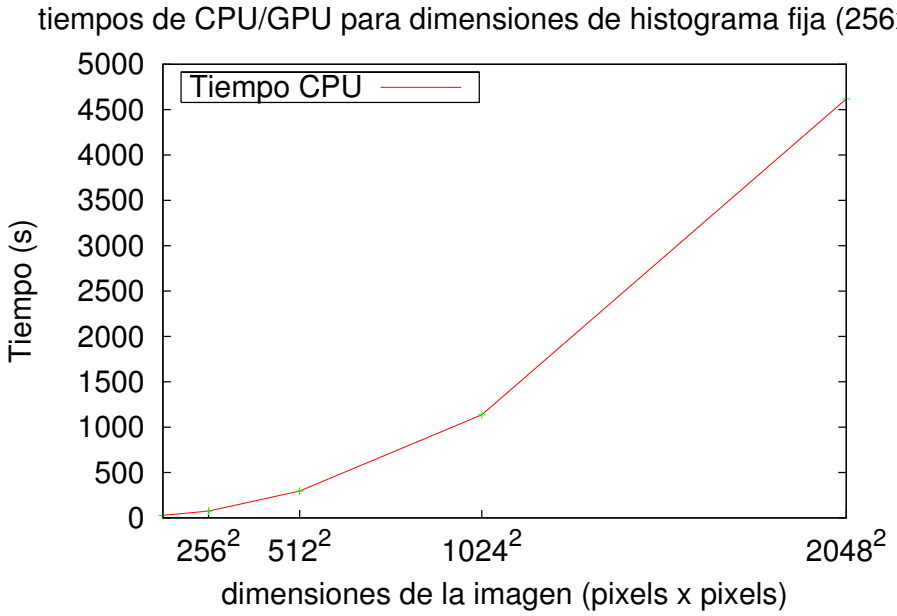

FIGURA 2.10: Tiempos de ejecución para tamaño de histograma fijo

En esta tabla podemos ver como el tiempo crece de manera aproximadamente cuadrática con el ancho de la imagen, es decir, es lineal en el número de píxeles de cada imagen, como cabría esperar. Esto se aprecia mejor en la figura 2.10.

Tamaño de histograma	128	256	512	1024	2048
Tiempo de ejecución	61.648	75.1726	108.872	266.878	943.524

TABLA 2.5: Tiempos medios de 5 ejecuciones para un tamaño de histograma 256 sobre CPU para una población de 100 individuos durante 200 generaciones

En la tabla 2.5 podemos ver los tiempos que han tardado las ejecuciones variando el tamaño del histograma, con imágenes de tamaño 256x256. Se

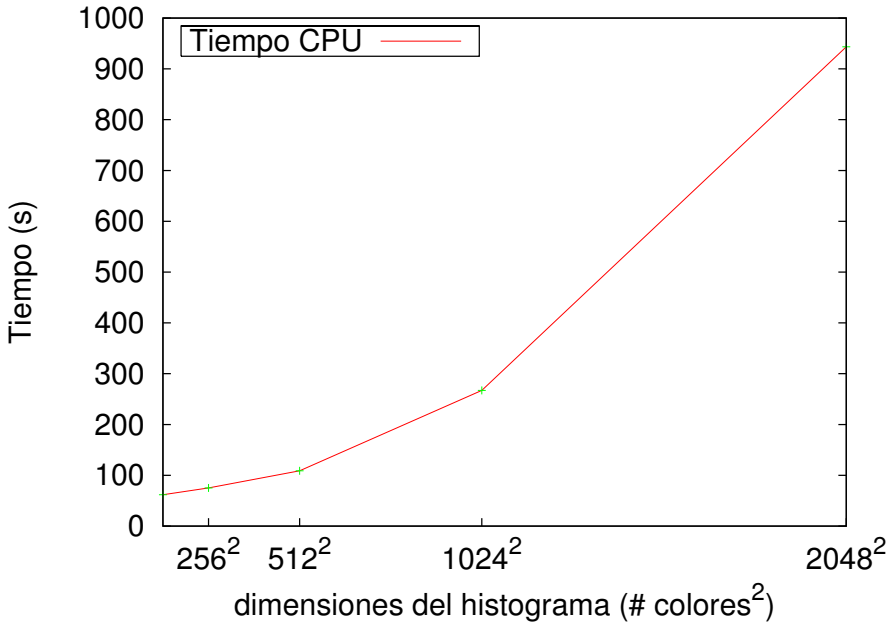

FIGURA 2.11: Tiempos de ejecución para tamaño de imagen fijo

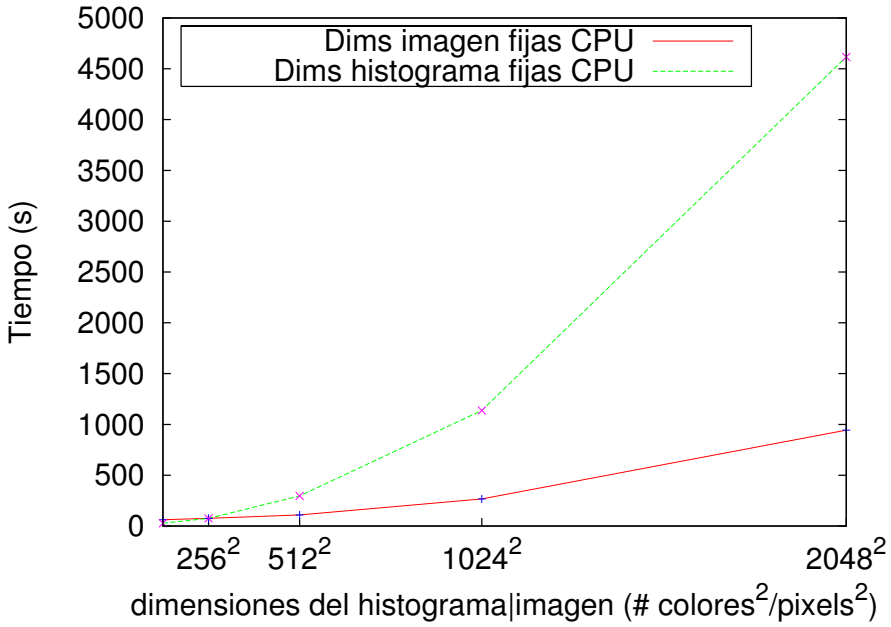

FIGURA 2.12: Tiempos de ejecución para CPU

puede ver que el crecimiento del tiempo es mucho más lento que en el caso anterior, es decir que aunque aumentemos el tamaño del histograma en la misma proporción que el tamaño de imagen, el incremento del tiempo de cómputo necesario es menor. Es más facil de apreciar en las figuras 2.11 y 2.12 donde podemos ver las dos curvas y cómo la que se refiere a la variación en función del tamaño de histograma crece de manera más suave.

Capítulo 3

Arquitectura de la GPU

GPU son las siglas de *Graphic Processing Unit o Unidad de Procesamiento Gráfico* [PF05]. El término fue introducido por la compañía NVIDIA para designar el nuevo tipo de hardware que venía sustituyendo las tradicionales controladoras de monitores VGA. Como vimos en la sección 1.4.1, estos procesadores fueron diseñados para aliviar la carga de trabajo de las CPUs.

Una primera aproximación entre las arquitecturas GPU y CPU resulta del estudio de las instrucciones que operan sobre múltiples datos al mismo tiempo (*single instruction on multiple data*, SIMD). Para poder operar simultáneamente necesitan ejecutarse sobre una ALU especial que permita tratar datos en paralelo. Las instrucciones SIMD se soportan por el hardware directamente. En un principio, nacieron para acelerar algunos algoritmos que precisaban ejecuciones sobre paquetes de datos tales como vectores o matrices y es por ello que no es de extrañar que los primeros procesadores en implementar este conjunto de instrucciones fuesen llamados procesadores vectoriales.

Como veremos en la sección 3.1 y a lo largo de la evolución tecnológica en

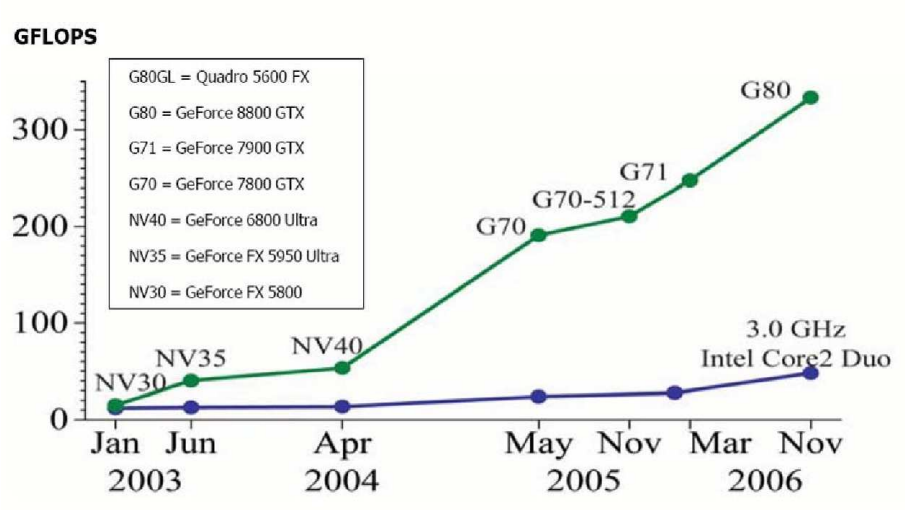

FIGURA 3.1: Comparación de potencia (en GIGAFLOPS) entre CPUs y GPUs

la sección 3.2, los módulos más importantes de una GPU no son más que ALUs especializadas en un conjunto determinado de operaciones que tienen que ver con la transformación de gráficos tridimensionales. Estos módulos poseen un conjunto de instrucciones SIMD orientadas a tales propósitos. Además, para acelerar todavía más su potencial de cálculo, una GPU utiliza estos módulos en paralelo, compartiendo una memoria común.

A lo largo de su evolución, las GPUs se han aprovechado de las mejoras técnicas en fabricación de componentes para abaratar su coste y han sido fuertemente influenciadas por la evolución de dos APIs importantes, OpenGL y Microsoft DirectX. Como muestra la figura 3.1, la potencia bruta de las GPUs supera con creces la de los procesadores más modernos y cada generación resulta muy superior a la anterior.

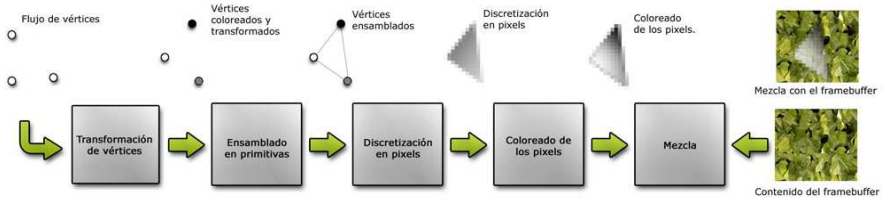

FIGURA 3.2: Distintas etapas del renderizado

3.1. El pipeline gráfico

Los ingenieros de *Silicon Graphics*, con su API OpenGL, fueron los primeros en establecer el conjunto de etapas necesarias para interpretar un conjunto de vértices tridimensionales como una imagen bidimensional. Como se observa en la figura 3.2, el proceso general puede resumirse en: transformación de vértices, ensamblado de los mismos en primitivas, discretización en píxeles, coloreado de cada uno de ellos y mezcla a realizar con el contenido de memoria, normalmente alguna operación de reemplazo [PF05]. A la implementación de estas etapas se la conoce por el nombre de *graphic pipeline* o tubería gráfica puesto que los resultados de una etapa alimentan la siguiente como si de una serie de tuberías en cadena se tratara.

Partiendo de la figura 3.2 explicaremos las distintas etapas del pipeline.

La primera información que recibe una GPU desde el software es la lista de vértices que conforman los distintos objetos tridimensionales. Estos vértices son procesados en el procesador de vértices (*vertex processor*). Aquí, cada uno de ellos es transformado desde su posición inicial hasta el espacio de visualización y coloreado teniendo en cuenta una serie de factores como son las posiciones relativas de objeto, cámara y luces. Como cada vértice es independiente de los demás, las transformaciones de varios de ellos pueden ejecutarse

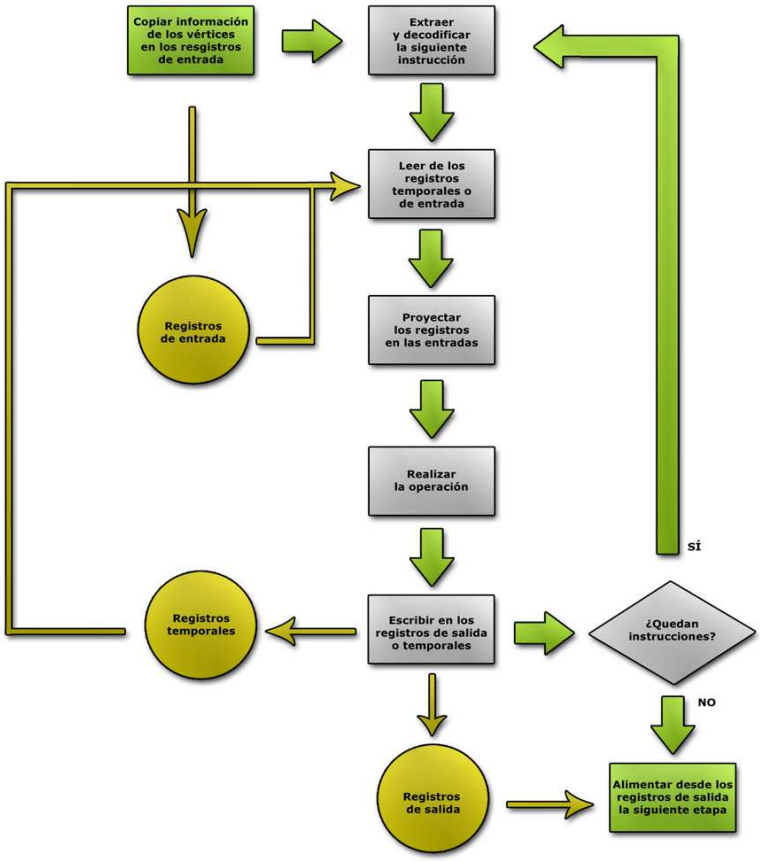

FIGURA 3.3: Esquema de trabajo de un procesador de vértices

al mismo tiempo por distintos procesadores en paralelo. Las APIs OpenGL y DirectX ofrecían distintos efectos como el conocido *ojo de pez* y otros. Cada uno de ellos podía conseguirse configurando apropiadamente la API para luego transformar consecuentemente los vértices haciéndolos atravesar una ruta de ejecución distinta a la clásica. Así pues, los finitos efectos soportados por las APIs suponían los finitos modos de ejecución de los procesadores de vértices.

La figura 3.3 muestra el flujo de trabajo de los procesadores de vértices desde el punto de vista del desarrollador de software.

Detalladamente, primero, el procesador copia los atributos del vértice en una memoria de sólo lectura. A continuación decodifica la instrucción, lee de los registros temporales, proyecta los registros sobre las entradas oportunas y ejecuta la instrucción. Los resultados intermedios pueden escribirse en los registros temporales y también es posible escribir en los registros (sólo escritura) de salida. Una vez finalizado el proceso, este se repite hasta consumir todas las instrucciones. El proceso se repite una vez por vértice y genera un flujo de datos de salida que alimentarán la entrada de la siguiente fase. Por tanto, la salida contendrá los vértices transformados además de otros atributos.

El procesador de vértices tiene capacidad para manipular operandos en forma de vectores y matrices, normalmente con elementos en coma flotante con precisión simple. Debido a que gran parte de su labor es transformar el espacio de coordenadas, las instrucciones típicas contempladas son productos escalares y vectoriales; suma y multiplicación de matrices y operaciones combinadas como multiplicación y adición.

Los vértices transformados de la etapa anterior pasan al ensamblador de geometría y al rasterizador. El primero determina las primitivas formadas por los vértices de la etapa anterior conforme lo indiquen las reglas de ensamblado que acompañan a los mismos. Además descarta aquellos vértices que no podremos ver debido a que se sitúan fuera del espacio de visualización actual (*clipping*). Por otra parte, el rasterizador determina qué píxeles (sus posiciones finales en el framebuffer) van a ser cubiertos por cada una de las primitivas generadas en el ensamblador de geometría, qué caras de los mismos deben dibujarse y cuales no, para ello ha de transformar las coordenadas del marco de clipping a coordenadas normalizadas de dispositivo. A este proceso se le

denomina *culling*[1]. El rasterizador emite entonces una lista con los píxeles de pantalla que van a ser cubiertos acompañándolos de información extra sobre ese punto. A la suma píxel más información se la denomina fragmento.

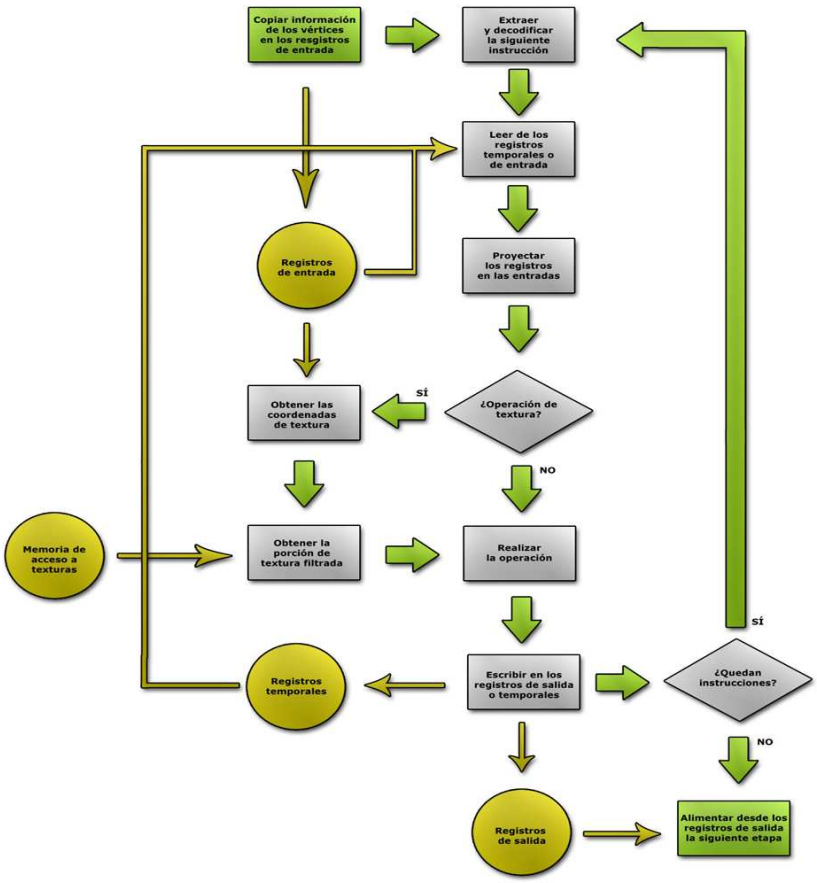

FIGURA 3.4: Esquema de trabajo de un procesador de fragmentos

Entonces, cada fragmento alcanza el procesador de fragmentos (*fragment processor*) que es otra unidad de procesamiento paralelo la cual tan sólo determinará el color final de cada fragmento proyectando sobre ellos alguna textura, lo que determinará el color final del pixel.

[1]Una traducción al castellano podría ser selección.

Como en el caso de los procesadores de vértices, cada API ofrecía aquí distintos efectos que podían conseguirse mediante rutas de ejecución alternativas.

El proceso de ejecución no es muy diferente al del procesador de vértices. El conjunto de funciones es esencialmente el mismo con las extensiones necesarias para utilizar texturas. La figura 3.4 presenta el esquema de trabajo de estos procesadores.

Parece lógico pensar que en una tarjeta haya más procesadores de fragmentos que de vértices, pues las primitivas como los triángulos tienen sólo 3 vértices pero pueden llegar a generar gran cantidad de fragmentos, de 100 a 1000 veces más. De hecho, así ha venido ocurriendo a lo largo de estos años para cambiar radicalmente ahora, con las llegada de la nueva generación de tarjetas.

Para terminar, los píxeles pasarán una serie de test propios de cada API que permitirán decidir si han de dibujarse o no pudiendo actualizar de esta manera el espacio de dibujo o framebuffer. Por último se aplicará una operación de fundido (*blending*) que decidirá como ha de mezclarse el fragmento con el pixel ya existente.

3.2. Evolución tecnológica de los procesadores gráficos

A lo largo de los años, pueden distinguirse cinco generaciones de GPUs desde las primeras controladoras CRT hasta nuestros días. En lo que a funcionalidad se refiere, el pipeline de renderizado de imágenes tridimensionales se

mantuvo esencialmente invariable hasta *la quinta*. Sin embargo, es la cuarta generación la más amplia y versátil y la que sentaría las bases de una nueva forma de entender el renderizado de gráficos 3D.

3.2.1. Primera generación

Las primeras GPU como la TNT2 de NVIDIA, la ATI Rage o la 3dfx Voodoo3 permitían realizar todas las etapas en la GPU excepto la transformación de vértices y su ensamblado. Las transformaciones se modelan mediante matrices 4x4 y modifican los vértices mediante sucesivas multiplicaciones por estas matrices, por ello gran parte de la carga matemática seguía ejecutándose sobre la CPU. Además implementaban el conjunto de características de la API de Microsoft DirectX 6 por lo que soportaban efectos de niebla, compresión de texturas, estados de mezcla, etc.

3.2.2. Segunda generación

La segunda generación comprende los modelos NVIDIA GeForce 256 y GeForce2, ATI 7500 y S3 Savage3. Implementaban al completo el proceso de renderización contando con algunas operaciones de transformación de vértices rápidas. Posteriormente, con las nuevas versiones de OpenGL y DirectX 7, se añadieron algunos efectos como los mapas cúbicos de textura (*cube map textures*) o las operaciones matemáticas con signo.

Como cabía esperar, el hardware implementa fielmente cada una de las etapas del pipeline distinguiéndose casi los mismos bloques.

La posibilidad de configurar los procesadores de vértices y de fragmentos

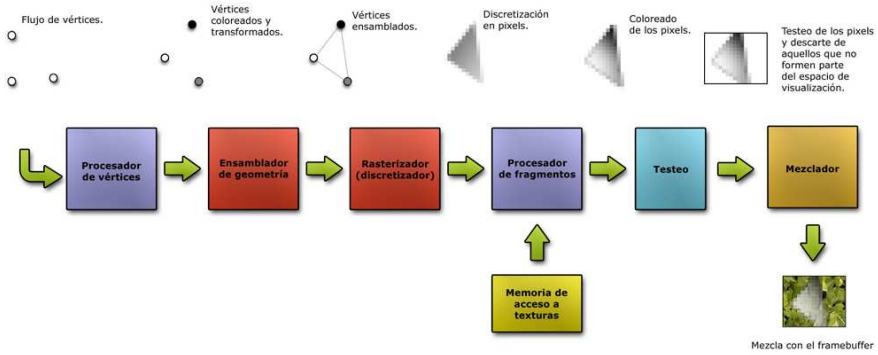

FIGURA 3.5: Pipeline estándar de segunda generación

son los primeros pasos hacia su programación. Sin embargo, tendremos que esperar a la tercera generación de GPU para advertir este cambio.

3.2.3. Tercera generación

Camino de la programación total, la tercera generación de GPU (modelos GeForce3/4, Xbox y Radeon 8500) es considerada transitoria puesto que permitía únicamente la programación de los procesadores de vértices pero no la de los procesadores de fragmentos. DirectX 8 y las extensiones ARB_vertex_program de OpenGL permitan dictar una serie de órdenes al procesador de vértices pero las extensiones para píxeles de ambas APIs no eran más que opciones de configuración algo avanzadas.

3.2.4. Cuarta generación

Por fin, la cuarta generación de CPU formada por la series GeForce FX de NVIDIA y Radeon 9700 de ATI, implementaban procesadores de vértices

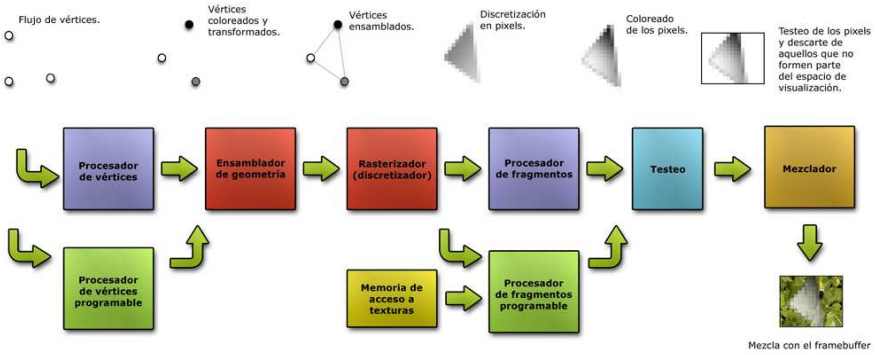

FIGURA 3.6: Pipeline programable de cuarta generación

y fragmentos programables. Este hecho favoreció la aparición de lenguajes de programación específicos como GLSL, HLSL, Cg, etc. En el contexto de la industria del videojuego, a los programas escritos bajo estos lenguajes se los conoce como sombreadores (*shaders*); al vertex processor como *vertex shader unit* y al fragment processor como *pixel shader unit*. De la misma manera, a los lenguajes de programación se los llama lenguajes de sombreado (*shading languages*).

3.2.5. Hacia la quinta generación

La evolución más reciente se produjo con la llegada, en Noviembre de 2006, del chip G80 y la serie GeForce 8800 de NVIDIA. Actualmente ATI cuenta también con sus propias GPU de nueva generación, la serie 2900 XT.

Probablemente, esta generación de tarjetas supongan la segunda transición evolutiva por introducir un nuevo modelo de ejecución. La *quinta generación*, influenciada en gran medida por el trabajo de Microsoft en DirectX 10, supo-

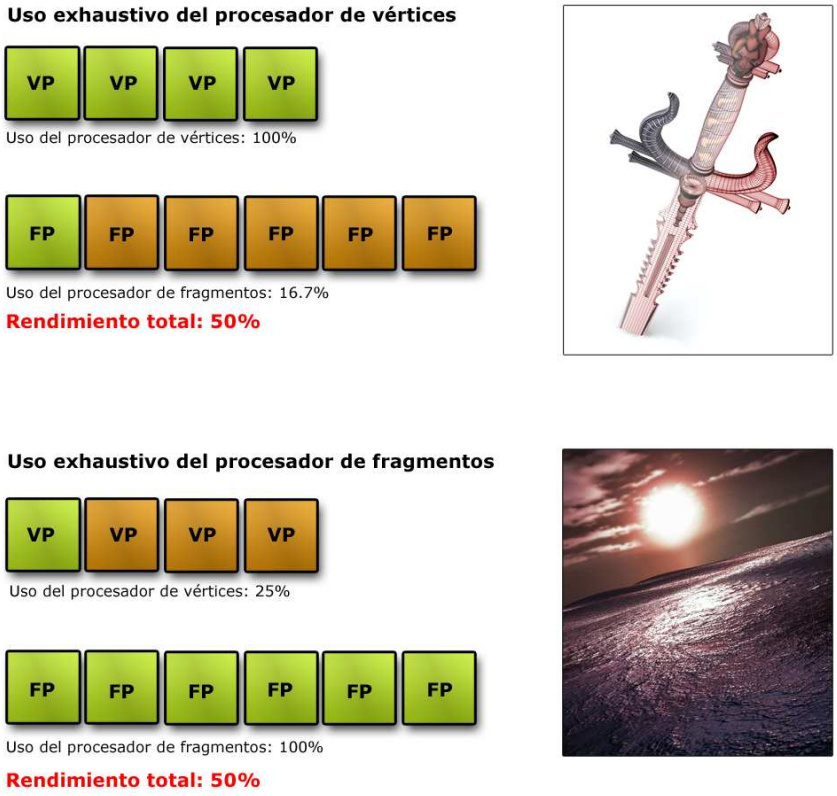

FIGURA 3.7: Rendimiento de una escena sobre una arquitectura de shaders no unificada

ne la reinvención del pipeline gráfico. Aunque conceptualmente el proceso se mantiene invariable, la nueva tecnología unifica los conceptos de procesadores programables integrando en las mismas unidades las funcionalidades de los antiguos vertex y fragment processors. Ahora, la salida de una etapa realimenta de nuevo los mismos procesadores que ejecutarán la siguiente etapa del pipeline.

La figura 3.7 refleja dos modelos de distribución de carga sobre las GPU actuales, en uno de ellos la carga sobre el procesador de vértices dada la alta

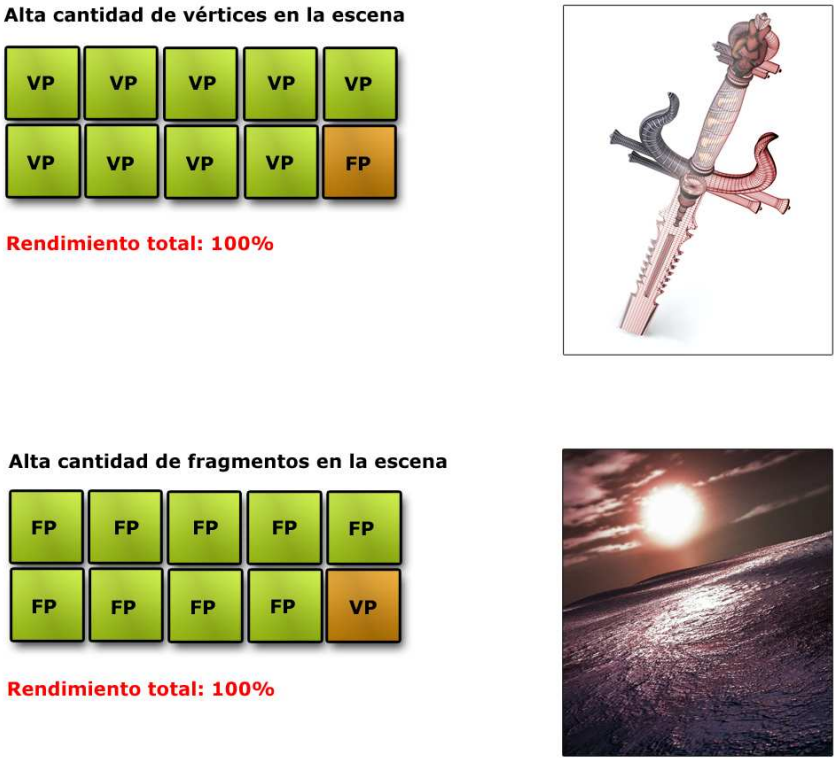

FIGURA 3.8: Rendimiento de una escena sobre una arquitectura de shaders unificada

poligonación del objeto y en la otra, la carga sobre el procesador de fragmentos debido a los efectos sobre la superficie oceánica. En ambos el rendimiento total de las unidades de cálculo es del 50 % mientras que sin embargo, cada una de ellas satura respectivamente los procesadores de vértices y fragmentos.

Haciendo uso de la tecnología es posible aprovechar hasta el último de los procesadores libres para maximizar el rendimiento de la GPU.

3.3. Conclusiones

Gracias a las GPUs y a sus módulos programables contamos con una plataforma versátil capaz de explotar diversas formas de paralelismo. El paralelismo de tares, nos servirá en el Capítulo 4 para ejecutar programas distintos que implementen etapas del problema distintas al mismo tiempo. Además, algunos problemas como los que estudiaremos a continuación procesan grandes cantidades de datos independientes unos de otros y esta naturaleza encaja a la perfección con el modelo de procesamiento de los fragment y vertex processors.

Por último, como se indicó al comienzo de este capítulo, la tecnología SIMD confiere a estas unidades una potencia muy elevada a la hora de tratar vectores o matrices. Y como veremos, estas construcciones se presentan muy a menudo en el ámbito del registro de imágenes.

Por tanto, en el próximo capítulo trataremos de aprovechar las características de las GPUs para el cálculo de propósito general introduciendo un nuevo modelo de software y haciendo uso de las técnicas GPGPU.

Capítulo 4

Registro de imágenes en GPU

Hablamos en el capítulo 1 de la GPGPU como un paradigma. Sin embargo, para lograr GPGPU son necesarias una serie de técnicas que permitan aprovechar las características de las GPUs como procesadores de flujo. Una vez conocidas estas técnicas, el desarrollador encapsulará la utilidad de la tarjeta gráfica en un framework[1] con el que trabajar más cómodamente, ofreciendo una visión más general del hardware, más orientada hacia el modelo de procesamiento de flujos que describiremos en la sección 4.1.

4.1. El modelo de procesamiento de flujos

En programación de algoritmos podemos encontrar en muchas ocasiones construcciones del tipo indicado en el algoritmo 6 que pueden reescribirse utilizando otra construcción más abstracta como la del listado 7. Si además

[1]En programación, un framework es una estructura básica dónde pueden resolverse problemas complejos. Normalmente aportan clases o librerías para encapsular funcionalidades o prestar algún servicio.

sabemos que el cómputo de cada individuo en el conjunto de datos no depende de otros individuos del mismo conjunto, o dicho de otra manera, si el orden en el que recorramos la colección no importa en absoluto, podríamos simplificar más aun el algoritmo escribiendo algo como lo mostrado en el pseudocódigo 8.

Algorithm 6 Procesamiento de los datos de una colección

```
for i = 1 to n do
  for j = 1 to m do
    valor[i][j] := FuncionDeInteres(i, j)
  end for
end for
```

Algorithm 7 Procesamiento de todos los datos de una colección mediante bucle genérico

```
for all d in datos do
  FuncionDeInteres(d)
end for
```

Algorithm 8 Procesamiento en paralelo de los datos de una colección

```
AplicarFuncionADatos(datos, FuncionDeInteres)
```

La función $AplicarFuncionADatos(conjuntoDatos, f)$ aplica f a cada uno de los elementos de *conjuntoDatos* en paralelo. Presentamos de esta manera un nuevo modelo de procesamiento de datos basado en la aplicación de un programa, llamado (*kernel*) en la terminología de procesamiento de flujos, a todos los elementos dentro de una colección o flujo, llamado (*stream*).

La salida será pues otro flujo de datos que podría alimentar etapas posteriores como refleja la figura 4.1. A todo ello se lo conoce como modelo de flujos y la unidad capaz de ejecutar un programa de estas características se la llama procesador de flujos, (*stream processor*).

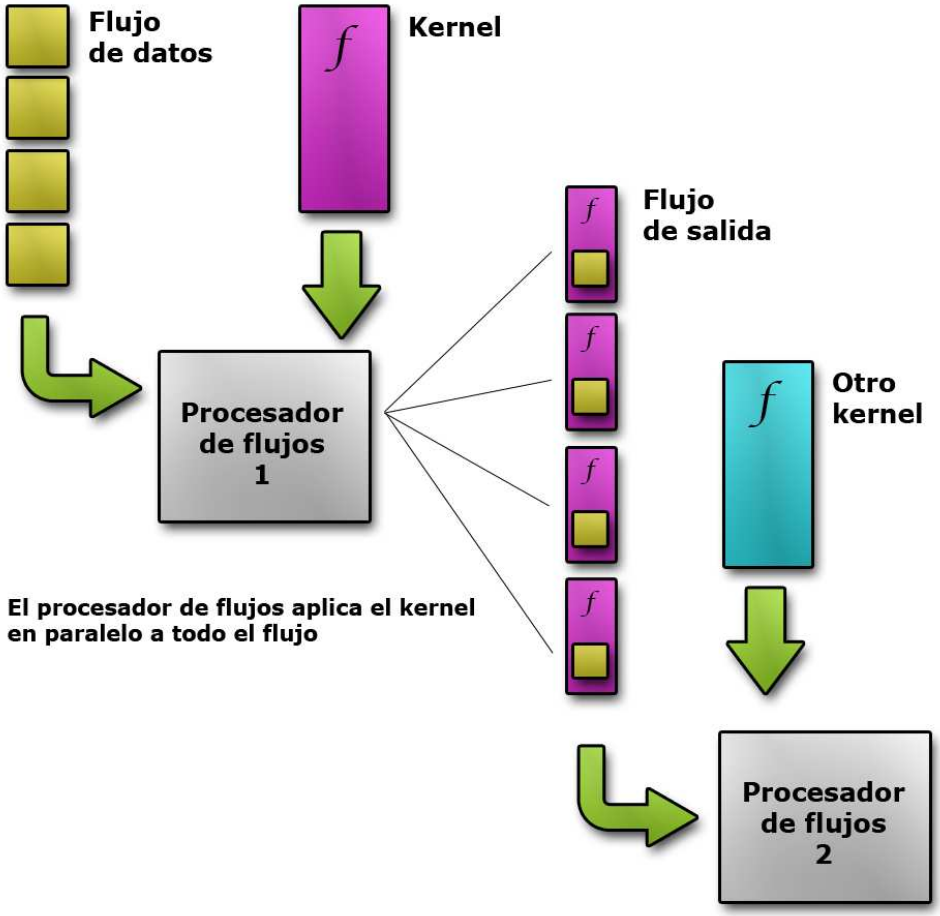

FIGURA 4.1: Esquema de trabajo de un procesador de flujos

4.1.1. Gathering y scattering

Por su comportamiento, los algoritmos pueden clasificarse en algoritmos de dispersión (*scattering*) y acumulación (*gathering*) [PF05].

Al scattering se le conoce también como escritura indirecta como puede apreciarse en la ecuación 4.1 y consiste en calcular la posición de escritura en función de aquello que vamos a escribir.

$$i = f(x)a[i] = x \tag{4.1}$$

El gathering, por contra, conocido también como lectura indirecta (tal y como vemos en la expresión 4.2) supone conocer los valores origen en función de la posición de escritura.

$$a[x] = f(v_1, ..., v_n)dondev_i = f_i(x) \tag{4.2}$$

4.1.2. El modelo de flujos en la GPU

Recordamos del capítulo 3 que una GPU opera sobre un conjunto de vértices de entrada aplicando varios programas a lo largo del pipeline gráfico. Este modelo de ejecución se corresponde con modelo de flujos recién descrito. Así, tenemos dos conjuntos de stream processors, los vertex processors y los fragment processors; un flujo de datos independientes entre si atravesando los mismos, los streams de vértices; y unos shaders que aplicar sobre los datos, los kernels.

Como los vertex processors pueden variar la posición de escritura dentro del framebuffer, pueden realizar operaciones de scattering. Además, las GPUs

modernas permiten acceso a las texturas desde los vertex processors, por tanto también pueden realizar operaciones de gathering. Por otro lado, los fragment processors no pueden alterar la posición de escritura, porque la posición ya ha sido calculada por el rasterizador, sin embargo, al poder leer de textura también soportan operaciones de gathering.

Para el scattering por medio del vertex processor consideramos a como el framebuffer, x como un valor precalculado en función de la entrada e i como una posición dentro del framebuffer, que depende de la entrada.

Para el gathering a través del fragment processor tomaremos a como el framebuffer, i como una posición para esa textura y x la posición del framebuffer calculada por el rasterizador, y v_i serán valores leídos de una textura.

Por último, para el gathering a través del vertex processor tomaremos las mismas consideraciones que para el gathering por píxel processor con la única diferencia de que x puede no estar determinada de antemano, y a es el flujo de salida. El procesador de vértices nos permite pues operaciones como la expresión 4.3.

$$x = a[i]; \qquad o[j] = x; \qquad j = f(i) \tag{4.3}$$

4.1.3. Lenguaje de programación

Hemos elegido Cg (C for graphics) de la compañía NVIDIA como lenguaje de sombreado para programar los kernel. Este lenguaje al estilo C puede compilarse bajo distintos perfiles para luego ser cargado en las unidades programables por medio de la API.

Los perfiles son una medida de la potencia del lenguaje y por tanto, de la potencia necesaria de la GPU [FK03]. En nuestro caso fue necesario emplear el perfil *VP40* para procesadores de vértices y el perfil más moderno que soporte el hardware para los procesadores de fragmentos. La elección de *VP40* es necesaria porque es el primer perfil que permite hacer lectura de texturas a los vertex processor, que es algo necesario para realizar el histograma.

4.1.4. Streams y Texturas

Los flujos de datos que llegan hasta los procesadores de vértices y fragmentos son colecciones de vectores de 4 elementos que codifican las propiedades de los vértices: posición, color, normal... Las texturas[2] son análogos a los *arrays* de hasta 3 dimensiones de la programación clásica. En ellos podemos almacenar los datos de entrada y los resultados intermedios de nuestro algoritmo.

Además del flujo inicial de entrada, los procesadores de vértices y de fragmentos emiten otros dos flujos que alimentan las etapas posteriores del pipeline.

4.1.5. El rasterizador

Se puede pensar en el rasterizador como el generador de los threads paralelos, ya que emite muchos más fragmentos que contendrán atributos interpolados a partir de los que poseían los vértices, generando los índices para una textura-array.

El rasterizador puede emitir una serie de posiciones no interpoladas si se le

[2]Una textura no es más que una región de memoria a la que podemos acceder por medio de índices

enviaron puntos como primitivas. O también puede interpolar los límites del rango de una matriz para obtener los índices internos de la misma mediante el envío de una primitiva quad[3] que tenga como coordenadas las equinas de la matriz. Puede verse en detalle en la figura 4.2

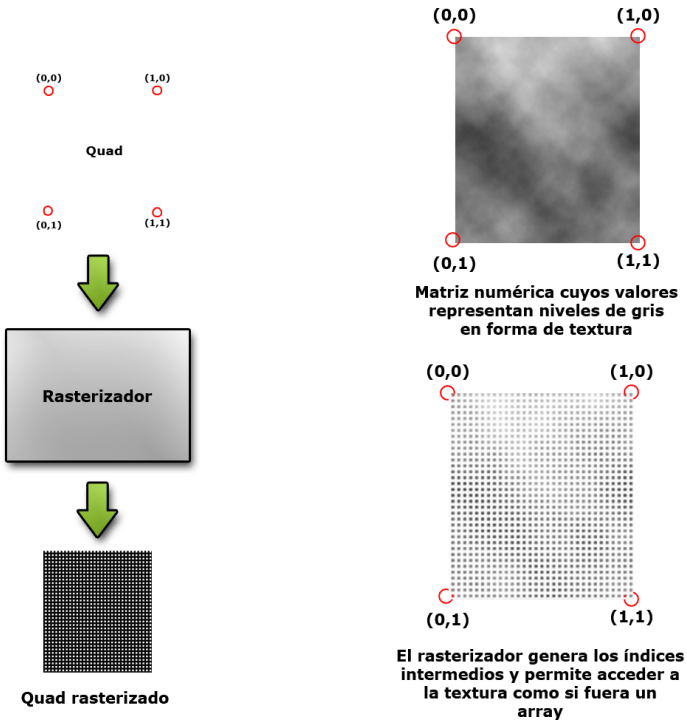

FIGURA 4.2: El rasterizador como un generador de índices

4.1.6. Retroalimentación

Los cuando un fragment processor emite su salida, el framebuffer como se indicó en el capítulo anterior, este puede ser mostrado en pantalla o bien escribirse a una textura (*render to texture*) en la memoria de la tarjeta. Este mecanismo resulta muy útil pues permite la realimentación de los datos de

[3]Un quad es un polígono de cuatro vértices

salida hacia el comienzo del pipeline de renderizado para una etapa de cálculo con esos datos.

4.1.7. Memoria de vídeo

Cada elemento de la memoria de vídeo o téxel, tanto en las texturas, como en el framebuffer, es una cuaterna de elementos, ya que en un principio ha sido diseñada para codificar la información de los tres canales RGB y la transparencia. Hemos decidido usar esta característica para que cada texel empaquete cuatro elementos de una misma fila, por lo que una matriz de dimensiones mxn ocupará $mx\frac{n}{4}$ texels, y la columna i-ésima de texels empaquetará las columnas $4 * i$, $4 * i + 1$, $4 * i + 2$ y $4 * i + 3$.

4.2. Implementación del modelo de procesamiento de flujos: el framework

Para el desarrollador de GPGPU la tarjeta gráfica ha de presentarse como un procesador de flujos y no como un sistema de renderizado de gráficos tridimensionales. Para este propósito puede desarrollarse un framework que oculte la visión de aceleradora 3D y resalte los aspectos más importantes del procesamiento de flujos. Así, necesitamos alguna manera de cargar los kernel en los procesadores de flujo, de enviar un stream al comienzo del pipeline, de recuperar el stream de salida y de comunicar parámetros a los kernel. Estos han sido los objetivos de nuestra implementación.

Para ayudarnos en el desarrollo del framework utilizamos el conjunto de

herramientas *Cg Toolkit* de NVIDIA. Cg Toolkit ofrece una primera abstracción del las extensiones de OpenGL que permiten la comunicación con los módulos programables. Sus características más importantes incluyen el ofrecer los tipos necesarios para enlazar variables del lenguaje con parámetros o entradas de los kernels y contar con su propio compilador Cg en tiempo de ejecución. El compilador en tiempo de ejecución libera al desarrollador de tener que compilar explícitamente para el último perfil de fragmentos, referido en la sección 4.1 puesto que es capaz de decidir qué características de Cg son soportadas por la GPU usada, y elegir el perfil más avanzado en el momento de la compilación.

4.3. Implementación del registro de imágenes

El problema de registro de imágenes está constituido principalmente por la búsqueda de la transformación y por el cálculo de su aptitud como vimos en el capítulo 2. En el caso del algoritmo de evolución diferencial, la existencia de variables aleatorias y la dependencia de unos individuos con otros no presentan un buen escenario para su implementación sobre el modelo de flujos.

Sin embargo, la valoración de la aptitud, es decir, el cálculo de la información mutua resulta un problema mucho más propicio. Veamos por qué.

Dividiremos el problema en el cálculo del histograma y la obtención de la información mutua descritos en la sección 2.2.

4.3.1. Cálculo de histograma

En el Capítulo 2 explicamos qué era un histograma. Vimos que en él se contabilizaban los distintos pares de colores (o atributos a analizar) de ambas imágenes y por tanto, la función de interés en la creación del histograma puede resumirse en la expresión 4.4

$$H[ImagenA[x][y], ImagenB[x'][y']] += 1 \tag{4.4}$$

En ella, $ImagenA[x][y]$ representa el atributo en la posición dada por el par (x, y) sobre la imagen A e $ImagenB[x'][y']$, el atributo en (x', y') sobre la imagen B. Visto así, no resulta difícil encontrar semejanzas directas con el modelo combinado de gathering - scattering propio de los procesadores de vértices.

Como advertimos hace unas líneas en la sección 4.1, H será el framebuffer, $ImagenA$ e $ImagenB$ se almacenaran como sendas texturas, el par (x, y) será parte del flujo de entrada y el par (x', y') se calculará a partir de (x, y) aplicando la transformación T de la que se desea conocer su aptitud. Todo el procedimiento puede observarse en la figura 4.3.

El kernel que se ejecuta en el procesador de fragmentos sólo tendrá que dejar pasar el flujo de fragmentos provenientes del rasterizador y por último, la operación *suma 1* será ejecutada por las unidades de blending que han sido sido configuradas apropiadamente.

Para minimizar el tráfico entre CPU y GPU, el conjunto de pares (x, y), que se mantendrá invariante durante toda la ejecución, puede almacenarse en un búfer de vértices (*vertex buffer object*). Lo búferes de vértices almacenan en

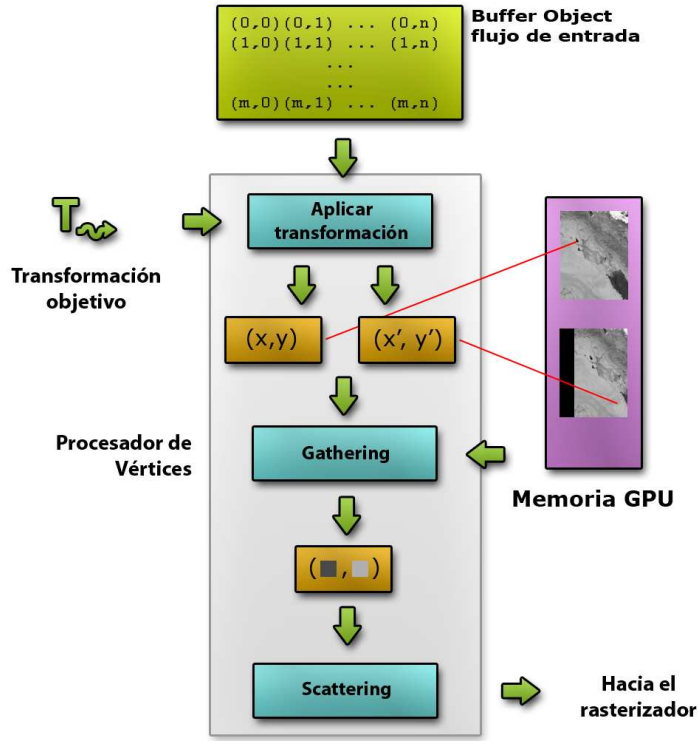

FIGURA 4.3: El modelo de ejecución del histograma sobre vertex processor

la memoria de la GPU conjuntos de vértices y permiten su carga en el pipeline en mucho menos tiempo del que se tardaría en transferirlos desde la memoria de la CPU.

4.3.2. Cálculo de la Información Mutua

El cálculo de la información mutua supone, como se introdujo en el Capítulo 2, una serie de transformaciones sobre el histograma de la forma indicada en la ecuación 4.5.

$$a[i][j] = f(b[i][j]) \tag{4.5}$$

No es difícil darse cuenta de la similitud entre la expresión 4.5 y el modelo de gathering. Es precisamente por eso por lo que implementaremos esta segunda parte del problema sobre los procesadores de fragmentos.

Esta cadena de transformaciones será llevada a cabo en sucesivas pasadas a través del pipeline, cada uno de ellos con su propio kernel de transformación.

Para esta segunda parte el kernel cargado en los procesadores de vértices deja pasar los datos sin hacerles nada más que las transformaciones necesarias para que escriban en el framebuffer, por lo que a partir de ahora sólo se referirán los kernels de fragmentos. La primera pasada consiste en aplicar un kernel de reducción en varios pasos, que en cada paso reduce el tamaño del histograma a un cuarto (la mitad en cada dimensión), de manera que en $log_2(n)$ pasadas habremos reducido a un escalar una matriz de dimensiones nxn. A continuación se hace una pasada con un kernel que realiza un paso de reducción en filas del histograma, que almacena en la mitad superior de la matriz de

salida, y en paralelo aplica un paso de reducción en columnas y traspone el resultado, escribiéndolo sobre la mitad inferior de la matriz de salida. De esta manera, el cálculo de las probabilidades marginales es reducir únicamente esta nueva matriz en filas hasta que queden dos filas. La superior será la reducción en filas, y la segunda será la reducción en columnas traspuesta. La siguiente pasada se realiza por separado, primero al histograma y luego a las reducciones. En ella se aplica un kernel que normaliza cada valor según el resultado obtenido en la primera pasada de esta segunda parte, y a continuación lo multiplica por su $-log$ para calcular la entropía.

Para terminar se aplica una reducción en columnas a la reducciones, para obtener $H(Y)$ y $H(Z)$, y una reducción bidimensional a la matriz como en el primer paso para obtener $H(Y, Z)$.

Llegados a este punto ya podemos aplicar la ecuación 2.2, esta vez en CPU y así obtener la MI.

4.4. Resultados obtenidos

Análogamente a como hicimos en la sección 2.5 vamos a analizar los resultados obtenidos al ejecutar el algoritmo en GPU. Primero haremos una breve descripción de la plataforma experimental usada y los datos de entrada. Después presentaremos tablas que muestran los resultados de la ejecución del programa. En este caso no analizaremos el algoritmo de búsqueda, pues como no varía de una implementación a otra, los resultados expuestos en las secciones 2.5.4 y 2.5.5 son igualmente válidos para la implementación en GPU. En el capítulo 5 se hace una comparativa y un análisis de los tiempos en ambas

plataformas.

4.4.1. Plataformas de ejecución e imágenes de entrada

Las ejecuciones se han llevado a cabo sobre GPUs NVIDIA 8800GTX en computadores con microprocesadores Intel, las tablas 4.1 y 2.1 describen los aspectos principales de la plataforma experimental, GPU y CPU respectivamente. Las implementaciones han sido compiladas utilizando GNU-C/C++ compiler (versión 4.1.2, con los flags de optimización *-O3 -msse*) para los archivos de C++ y el compilador proporcionado por NVIDIA, cgc (versión 1.5.0019), para los kernels escritos en Cg.

TABLA 4.1: Descripción técnica de la GPU

Características	**8800 GTX**
Año	2006
Arquitectura	G80
Bus	PCI Express
Memoria de Vídeo	768MB
Reloj del núcleo	575 MHz
Reloj de los sombreadores	1350 MHz
Reloj de memoria	900 MHz GDDR3
Interfaz con memoria	384-bit
Ancho de banda con memoria	86.4 GB/s
#Procesadores de flujos	128
Tasa de relleno de texturas	36800 MTexels/s

Las imágenes de prueba usadas han sido las mismas que para la ejecución en CPU descritas en la sección 2.5.2.

4.4.2. Tiempos obtenidos

Las ejecuciones test han sido llevadas a cabo durante 200 generaciones para 100 individuos que son suficientes para que se obtenga una solución con

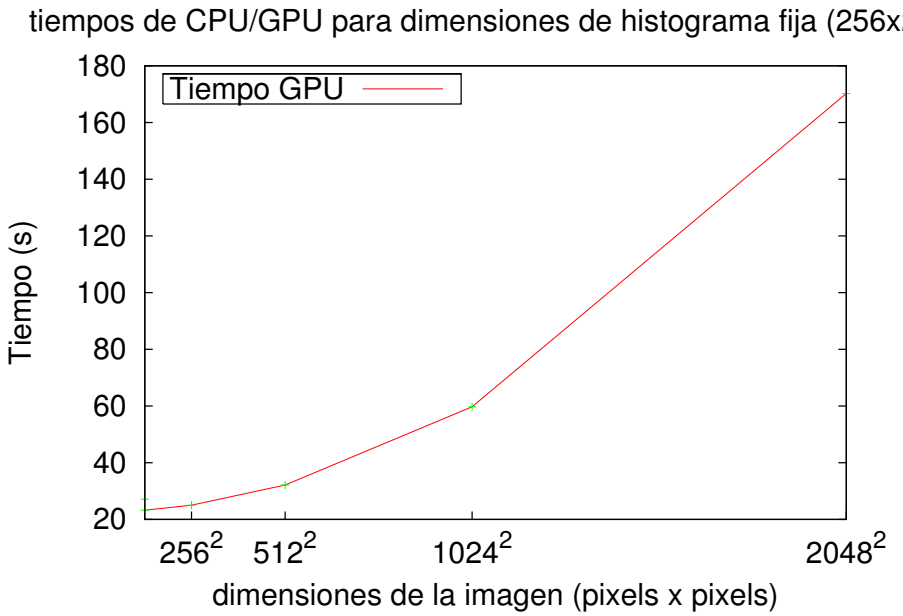

FIGURA 4.4: Tiempos de ejecución para tamaño de histograma fijo

imágenes sintéticas. Los tiempos de ejecución para ejecuciones con un tamaño de histograma fijo, 256 colores, para imágenes sintéticas de 128x128, 256x256, 512x512, 1024x1024, 2048x2048 se muestran en la tabla 4.2.

Tamaño de imagen	128x128	256x256	512x512	1024x1024	2048x2048
Tiempo de ejecución	23.2273	25.0018	32.1242	59.7452	170.218

TABLA 4.2: Tiempos medios de 5 ejecuciones para un tamaño de histograma 256 sobre GPU para una población de 100 individuos durante 200 generaciones

En esta tabla podemos ver como el tiempo crece de manera polinómica, pero de manera muy muy lenta, es decir, si duplicamos el ancho de la imagen de entrada el tiempo se multiplica por $f * X^n$ con $f << 1$ y $n > 2$. Esto se aprecia mejor en la gráfica 4.4.

En la tabla 4.3 podemos ver los tiempos que han tardado las ejecuciones variando el tamaño del histograma, con imágenes de tamaño 256x256. Se puede

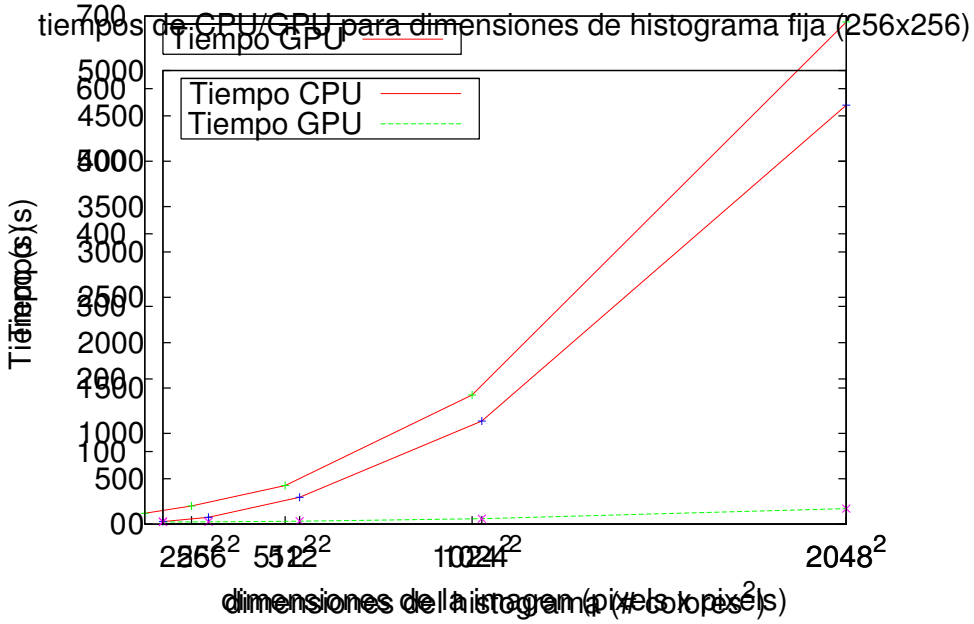

FIGURA 4.5: Tiempos de ejecución para tamaño de imagen fijo

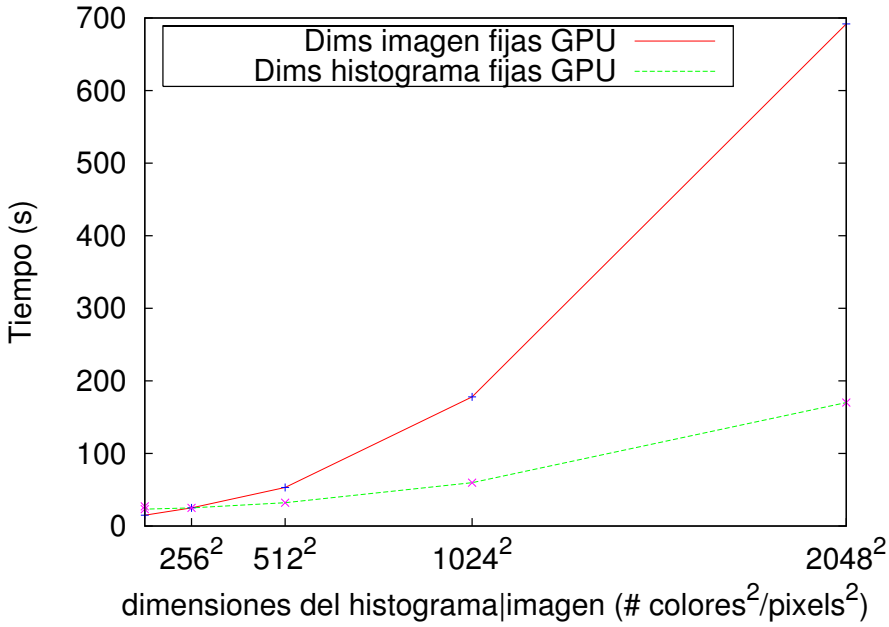

FIGURA 4.6: Tiempos de ejecución para GPU

Tamaño de histograma	128	256	512	1024	2048
Tiempo de ejecución	14.9987	25.0018	51.1589	178.002	691.813

TABLA 4.3: Tiempos medios de 5 ejecuciones para un tamaño de histograma 256 sobre GPU para una población de 100 individuos durante 200 generaciones

ver que el crecimiento del tiempo es mucho más rápido que en el caso anterior, es decir que si aumentemos el tamaño del histograma en la misma proporción que el tamaño de imagen, el incremento del tiempo de cómputo necesario es mayor. Es más fácil de apreciar en las figuras 4.5 y 4.6 donde podemos ver las dos curvas y cómo la que se refiere a la variación en función del tamaño de histograma crece de manera más acusada.

Capítulo 5

Comparación de rendimiento

En este capítulo vamos a comparar los tiempos obtenidos en la ejecución sobre CPU y GPU mostrados en los apartados 2.5.8 y 2.5.8, tablas 2.4, 2.5, 4.2, 4.3 para ver el speedup obtenido en el proceso total, así como en diferentes partes. Debido a que las figuras de estas secciones están en diferente escala. En la figura 5.1 presentamos los tiempos de manera conjunta.

5.1. Speedup

En esta sección vamos a estudiar el speedup conseguido en función del tamaño de imagen cuando el tamaño del histograma es fijo, y en función del tamaño del histograma cuando el tamaño de imagen es fijo. La figura 5.2 muestra las dos curvas obtenidas en las ejecuciones de test de 100 individuos durante 200 generaciones.

En la figura podemos ver cómo la curva de que representa las ejecuciones con tamaño de imagen fijo es decreciente. Esto se debe a que como comentamos

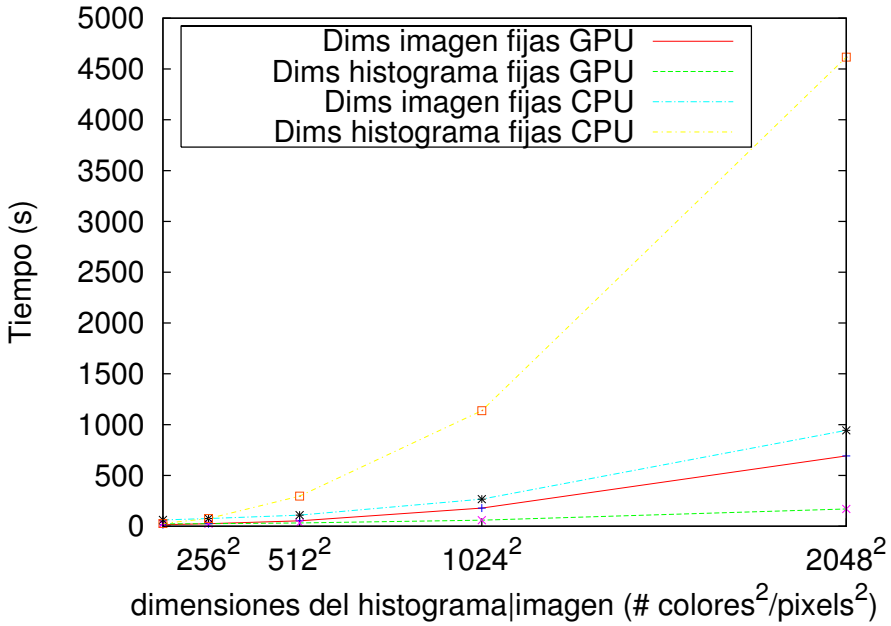

FIGURA 5.1: Tiempos de ejecución sobre CPU y GPU para variaciones de tamaño de paleta e Imagen

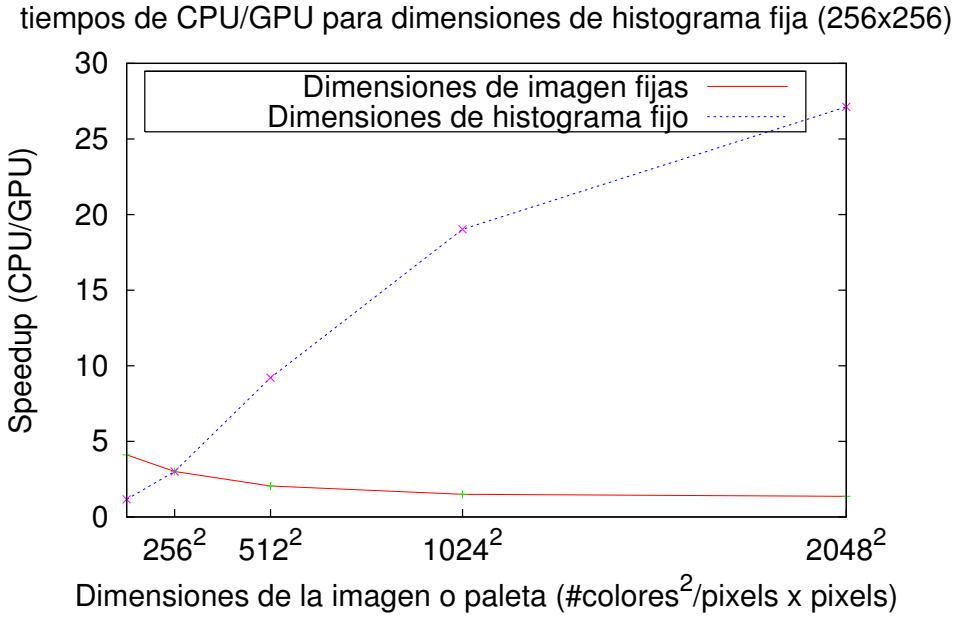

FIGURA 5.2: Speedup conseguido variando de manera separada el tamaño de imagen y el tamaño de histograma

en la sección 4.3.2 las reducciones se hacen en varias pasadas, esto hace que el cálculo no sea tan paralelo como cabría desear, y es por esta barrera que el speedup decrece.

En cambio la curva que representa las ejecuciones con tamaño de imagen fijo es creciente. Esto se debe a que la mayoría del tiempo en CPU se dedica al cálculo del histograma que es la parte qué más aprovecha las características de la GPU para ejecutarse a mayor velocidad.

5.2. Tiempo de calculo del histograma

La primera acción que se lleva a cabo en el algoritmo de cálculo de la MI es la realización del histograma conjunto de dos imágenes. Este histograma tiene un tamaño que viene determinado por el tamaño de la paleta, de forma que el número de filas y el número de columnas del histograma es igual al número de colores que tiene esta. La cantidad de información que hay en la paleta es proporcional al tamaño de las imágenes: cuanto mayores sean las imágenes más píxeles habrá en el área de solapamiento y por lo tanto mayores serán los valores contenidos en el histograma. Computacionalmente esto significa que tendremos que hacer un numero de operaciones de incremento igual al tamaño de la primera imagen, que tendrán que escribir en una matriz cuadrada de tamaño igual al tamaño de la paleta. Esto significa que cuanto mayor sea el tamaño de la paleta, si asumimos una distribución uniforme de la probabilidad de cada nivel de color, más homogéneo será el histograma, lo que hace que la cache no sea efectiva.

Nótese también que aunque la primera imagen es accedida por filas, debido

a que la segunda está transformada, ésta es accedida, en la mayoría de los casos de manera oblicua, lo cual también hace ineficaz la caché de datos de la CPU. Estos dos problemas para la CPU son resueltos por la GPU pues tanto el espacio de escritura como las unidades que se encargan de hacer el incremento están divididos en bancos lo que hace que las escrituras tengan mayor localidad en cada banco y además la memoria de texturas tiene una organización 2D en vez de unidimensional, lo cual hace que el acceso en oblicuo no suponga ningún problema.

Por estos motivos vemos que el aumento del tamaño de imagen apenas es acusado por la GPU, en cambio ralentiza el proceso en CPU y podemos ver cómo el speedup crece con el tamaño de las imágenes de entrada.

Tras varias ejecuciones hemos visto que para nuestra aplicación, un histograma de 256x256 da buenos resultados. Nos centraremos en este tamaño de histograma para compara los tiempos de cálculo del histograma y de la MI para diferentes tamaños de imagen. Las tablas 5.1 y 5.2 muestran estos tiempos.

Tamaño de imagen	128x128	256x256	512x512	1024x1024
Tiempo de ejecución CPU	0.006871	0.030168	0.13078	0.525011
Tiempo de ejecución GPU	0.003724	0.004694	0.00804	0.021718

Tabla 5.1: Tiempos medios de cálculo de un histograma 256x256

Tamaño de imagen	128x128	256x256	512x512	1024x1024
Tiempo de ejecución CPU	0.006403	0.005037	0.015186	0.030125
Tiempo de ejecución GPU	0.007545	0.007592	0.007931	0.007733

Tabla 5.2: Tiempos medios de cálculo de la MI para un histograma 256x256

Mirando las tablas 5.1 y 5.2 vemos cómo el tiempo de cálculo del histograma en CPU crece mucho más rápido que en GPU, además observamos que

aunque cabría esperar que el cálculo de la MI tardase un tiempo constante para un mismo tamaño de histograma como ocurren en el cálculo sobre GPU, en el caso de el cálculo sobre CPU no es así, ya que como el tamaño de imagen es mayor, hay más probabilidad de que cada posición del histograma sea distinta de cero, lo que hace que haya que calcular más logaritmos[1].

5.3. Tiempo de cálculo del MI

A partir de un histograma conjunto las operaciones que tenemos que hacer son reducciones, productos y logaritmos. Estas operaciones son llevadas a cabo más rápidamente por la GPU como podemos observar en la figura 5.1 y en la tabla 5.2. Lo que ocurre es que según aumenta el tamaño del histograma, aumenta el número de operaciones que hay que realizar. Como el código está organizado en bucles que iteran un número de veces igual al tamaño del histograma, esto hace que el pipe de la CPU esté continuamente lleno y sin paradas, lo que aumenta el rendimiento en CPU. Además, tal y como se describe en 4.3.2 las reducciones se llevan a cabo en varias pasadas. Estas pasadas las ordena la CPU por lo que cuantas más pasadas haya que hacer más tráfico entre CPU y GPU habrá lo que ralentiza el cálculo si aumenta el tamaño del histograma.

[1]Esto se debe a que cuando una posición es cero, su entropía vale $0 * \infty = 0$, por lo que en ese caso nos ahorramos el cálculo de un logaritmo, porque sabemos el resultado de antemano. En la GPU esto no se nota ya que las construcciones condicionales se transforman en ejecuciones predicadas, por lo que el *log* se calcula siempre, sólo que se guarda o se desecha según la condición

5.4. Diferencia entre la versión con mejoras o sin mejoras

En la sección 2.3 hemos hablado de unas mejoras a la convergencia, y hemos visto como afectan teóricamente al tiempo. En la tabla 5.3 podemos ver cómo varia el tiempo de ejecución para la media de 5 ejecuciones en cada versión para 600 generaciones con 300 individuos con el cálculo de la MI sobre GPU.

Tipo de mejora	**Tiempo(s)**
Sin Mejoras	183.568
Normalización mejorada	168.1414
Normalización mejorada y uso de gradiente	172.38

TABLA 5.3: Tiempos según las mejoras para una ejecución con 300 individuos durante 400 generaciones sobre GPU

5.5. Reparto del tiempo de cómputo

En esta sección hacemos un estudio de cómo se divide el tiempo de cómputo entre las diferentes partes del algoritmo promediado para 5 ejecuciones con poblaciones de 50 individuos durante 200 generaciones, tanto en CPU como en GPU.

En la tabla 5.4 vemos cómo en ambas plataformas la parte con más peso es el cálculo del histograma, y que los tiempos de preparación y el control de la búsqueda son despreciables. En CPU la reducción del histograma tiene coste 0 porque se hace a la vez que el histograma, aunque en las versiones mejoradas esta reducción es innecesaria. Como en GPU tenemos el problema de que las

Tarea	**% Tiempo para CPU**	**% Tiempo para GPU**
Lectura y carga de las imágenes	0 %	0 %
Realización del histograma	94.76 %	62.59 %
Reducción del histograma	0 %	10.65 %
Cálculo de las probabilidades	1.66 %	7.06 %
Cálculo de las entropías marginales	0.02 %	0.61 %
Cálculo de la entropía conjunta	3.3 %	0.59 %
Reducción de las entropías	0.07 %	17.8 %
Control de la búsqueda	0.19 %	0.7 %

TABLA 5.4: Reparto del tiempo de cómputo

reducciones se hacen en varias pasadas vemos cómo se llevan bastante tiempo (un 10.65 % más un 18.8 %).

El dato más importante a extraer tiene que ver con la ley de Amdahl y el speedup máximo esperado. Viendo el reparto de tiempos y sabiendo que las partes no paralelizables son las reducciones y el control de la búsqueda, llegamos a que la parte secuencial del algoritmo representa únicamente un 0.26 % por lo que en el extremo ideal de que la parte paralelizable se hace en tiempo $0s$ el speedup sería del orden de 400. En el caso real vemos que las reducciones en GPU son del orden de 10 veces más lentas que en CPU, lo que nos llevaría a esperar un speedup máximo teórico, que aún sigue siendo ideal, de 40.

5.6. Conclusiones

Hemos visto cómo para los tamaños de histograma e imágenes que usamos podemos esperar un speedup en torno a 15, lo que hace que un registro que podía tardar 2 horas se lleve a cabo en unos 10 minutos. Este speedup está por debajo de la mitad del máximo teórico, que es una cifra bastante aceptable.

Capítulo 6

Conclusiones y trabajo futuro

Las GPUs son un hardware que cada vez ofrece mayor potencia de cómputo a menor coste. Los fabricantes de GPU están empezando a dar soporte para el GPGPU, el mejor ejemplo de esto es CUDA.

Ésto es motivo suficiente para intentar adaptar algoritmos que tradicionalmente se ejecutan en CPU a GPU pues su bajo coste y alto rendimiento la hace idónea para algoritmos con paralelismo en datos y un acceso a memoria irregular o con una topología 2D.

A lo largo de este trabajo hemos discutido un ejemplo de tales algoritmos, y hemos aportado datos que muestran cómo el rendimiento obtenido en GPU en el cálculo de la MI acelera la ejecución del algoritmo de búsqueda entre 2 y 20 veces.

Habiendo ejecutado el algoritmo para imágenes reales hemos observado que un tamaño de paleta de 256 colores se hace suficiente, por lo que el factor de aceleración esperado, para el tamaño de imagen usado, 256x3352, se sitúa en torno a 15, lo que deja un tiempo de ejecución para 300 individuos durante

600 generaciones, lo que nos garantiza encontrar una buena solución, entorno a los 10 minutos.

Conseguir el mismo rendimiento paralelizando en un *cluster* tendría un coste mucho más elevado, además que el coste de las comunicaciones es mucho mayor

Como posibles continuaciones para seguir mejorando el rendimiento proponemos las siguientes dos opciones, que no son excluyentes:

- Distribución del algoritmo en poblaciones que se reparte en un *cluster* con topología de toroide de manera que cada cierto número de generaciones cada procesador hace un envío de su mejor individuo a sus cuatro vecinos que es incorporado a las poblaciones, tal y como proponen en [FMS+07] pero calculando la MI en GPU.

- Usar el lenguaje CUDA y las herramientas que provee de manera que los pasos innecesarios del pipeline gráfico no sean ejecutados, por lo que previsiblemente se obtenga una nueva mejora de rendimiento.

Como último apunte, recalcar el alto rendimiento obtenido por la GPU en el registro de imágenes usando el algoritmo de evolución diferencial teniendo como función de aptitud la información mutua.

Bibliografía

[Bla07] A. Ferreiro Blanco. Registro de imágenes hiperespectrales, 2007.

[Bro92] Lisa G. Brown. A survey of image registration techniques. *ACM Computing Surveys*, 24(4):325–376, 1992.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FMS+07] I. De Falco, D. Maisto, U. Scafuri, E. Tarantino, and A.Della Cioppa. Distributed differential evolution for the registration of remotely sensed images. In *PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 358–362, Washington, DC, USA, 2007. IEEE Computer Society.

[HBHH01] D. L. Hill, P. G. Batchelor, M. Holden, and D. J. Hawkes. Medical image registration. *Phys Med Biol*, 46(3), March 2001.

[JEE77] Larry R. Tinney John E. Estes, John R. Jensen. The use of historical photography for mapping archaeological sites. In *Journal*

of Field Archaeology, Vol. 4, No. 4 (Winter, 1977), pp. 441-447, 1977.

[MV98] J. B. Maintz and M. A. Viergever. A survey of medical image registration. *Med Image Anal*, 2(1):1–36, March 1998.

[PF05] Matt Pharr and Randima Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005.

[ret02] *Automatic retinal image registration based on blood vessels feature point*, volume 4, 2002.

[rid00] *On ridges and valleys [image registration]*, volume 4, 2000.

[SMH+07] Oliver Schmitt, Jan Modersitzki, Stefan Heldmann, Stefan Wirtz, and Bernd Fischer. Image registration of sectioned brains. *International Journal of Computer Vision*, V73(1):5–39, June 2007.

[yJMV07] Pablo Amil Crespo y Juan Martínez Vázquez. Registro de imágenes. Disponible en la pagina web http://varpa.lfcia.org/Docencia/VAFiles/temasVA0607.html, 2007.

Índice de figuras

Índice de tablas