ESTRATEGIAS DE PARALELIZACIÓN Y OPTIMIZACIÓN DE LOS ALGORITMOS JPEG Y MPEG

Índice

1-Introducción

En la sociedad en la que vivimos, la información juega un papel fundamental. Hemos pasado de una vida tranquila y disipada en el campo, que tenían nuestros abuelos a una contrarreloj continua en la que nos cruzamos con miles de personas y cosas nuevas al día. Existen estudios en los que se compara la cantidad de información que recibimos hoy con la que recibían hace sesenta años. Las cifras son sorprendentes, diariamente nos llega el equivalente a lo que en el pasado recibían durante seis meses. Esto ni es magia ni es casualidad, pues en apenas cincuenta años ha habido una revolución tecnológica, que ha cambiado radicalmente la forma en que los seres humanos tratamos la información.

Este panorama que no es nuevo para ninguno, esta tan presente en nuestras vidas que muchas veces ni nos detenemos a pensarlo. Teléfonos móviles, televisión, ordenadores, PDAS,… y un largo sin fin de artilugios tecnológicos expendedores de litros y litros de información nos acompañan a todos lados. Están ahí siempre dispuestos, para hacernos llegar la información que necesitemos o no, en cada momento. Pueden ser simples llamadas. *mails*, música, fotos, video… que viajan a través de una red para llegar a su destino final.

Toda esta tecnología que hemos citado está soportada por una red, que no es más que una serie de puntos conectados. Es la conexión el factor clave, haciendo posible la comunicación entre dos puntos cualesquiera de una misma red. Si por ejemplo disponemos de un teléfono móvil, este se conecta a una antena que a su vez se conecta a otras antenas y teléfonos haciendo posible realizar y recibir llamadas a través de él.

Otro factor clave, íntimamente relacionado con la información, es el tiempo que tarda en llegar desde la fuente hasta nosotros. Todos hemos hablado alguna vez por teléfono, con alguien que esta lejos, en otro país, quizá en otro continente, y seguramente hemos notado como nuestra voz tarda un poco en llegarle. Este hecho refleja claramente lo importante que es la velocidad cuando de lo que se trata es de enviar y recibir información. Esta velocidad depende fundamentalmente de dos cosas: la velocidad de la red, también conocida como ancho de banda; y el tamaño de la información que queremos enviar. En cuanto a la velocidad de la red se refiere, existen multitud de elementos importantes que van desde el tipo de soporte utilizado (cable, ondas de radio, infrarrojos,…) hasta las capas software sobre las que operan. Es por tanto tarea de los ingenieros diseñar redes más rápidas. En la otra cara de la moneda se encuentra el tamaño de los datos que se transmiten, no ocupa lo

mismo una carta a un amigo que una enciclopedia entera, siempre podríamos encoger la enciclopedia…. Pero en el mundo digital, en el que las cosas se representan con bits (0s y 1s), se puede hacer que ocupen menos tamaño sin recurrir a métodos tan complicados.

La acción de reducir el tamaño de la información se llama compresión, y es ampliamente utilizado por todas las tecnologías que trabajan con datos digitales. Hay dos formas de comprimir, una sirve para cualquier tipo de información, partiendo de la base de que toda se representa con bits fácilmente manipulables. La otra es más específica y depende del tipo de datos, pretende conseguir una reducción mayor del tamaño, fundamentada en que datos del mismo tipo seguirán patrones similares. Como contrapartida esta forma de compresión, no puede aplicarse sobre todo tipo de datos, y suele ser más lenta, es decir el proceso de comprimir/descomprimir lleva más tiempo. Aquí es donde entra en juego nuestro trabajo, en el que estudiaremos esta manera de comprimir, investigando formas de hacerla más rápida.

Como ya hemos dicho antes, el método de compresión es específico del tipo de datos, nosotros concretamente estamos interesados en la compresión de video e imágenes. Al ser este contenido multimedia, seguramente muchos habrán tenido alguna vez la posibilidad de ver imágenes o videos; en cámaras digitales, en el móvil o en el ordenador, y podemos asegurar, sin temor a equivocarnos, que casi el cien por ciento de estos estaban comprimidos. Estas imágenes y video muchas veces se comprimen directamente en el dispositivo que las obtiene, un ejemplo claro de esto son las cámaras digitales, que comprimen automáticamente las fotos, justo después de ser tomadas. Otras veces se hace por una persona desde un ordenador; si alguna vez hemos pasado un video al ordenador para editarlo y montarlo, es posible que también lo hallamos comprimido para que ocupase menos. Los que lo hallan hecho habrán notado que el proceso de comprimir un video no es ni mucho menos una cosa inmediata, sino que lleva bastante tiempo, pudiendo durar incluso horas, si el video es largo.

Actualmente todos disponemos de al menos un ordenador en nuestra casa, tener ordenador se ha vuelto como tener coche o televisión. Este *nuevo electrodoméstico* esta compuesto por una serie de componentes electrónicos más comúnmente llamados hardware. Como toda la tecnología, el hardware se encuentra en continúa evolución, y muchas veces trabajamos con programas que no explotan todas las posibilidades que este nos brinda. Una de las cosas que más rápido está evolucionando en este mundo, son los procesadores. En poco tiempo hemos pasado de lentos y arcaicos 486, a potentes procesadores duales como el nuevo Intel Core 2 Duo o el AMD Athlonx2. Si bien el hardware ha evolucionado de manera

vertiginosa, el software muchas veces no ha sido capaz de seguir este ritmo, creándose un nuevo problema: la mala gestión e inutilización de los recursos disponibles. Por ello es tarea de los investigadores, buscar y experimentar formas de hacer que el software utilice adecuadamente lo que el hardware ofrece.

Este estudio emplea las nuevas mejoras del hardware para conseguir hacer la compresión de videos e imágenes más rápida. Para ello hemos explotado las posibilidades de dos componentes: el procesador y la tarjeta gráfica.

Los procesadores han cambiado mucho en este ultimo tiempo, el gran aumento de tamaño de la memoria caché, acompañado del aumento y replicación de muchas partes del procesador, permiten realizar cálculos mucho más rápido, si a esto le añadimos el hecho de que ya se fabrican varios procesadores dentro de un mismo chip, tenemos una máquina casi el doble de potente. Estas tecnologías duales benefician mucho al software programado con threads, pero mucho menos al que se ejecuta en un solo proceso. En el caso que nos atañe los codificadores (compresores), nos encontramos con que se están usando muchas implementaciones, que no usan el potencial de los procesadores sobre los que se ejecutan. Nosotros pretendemos reflejar la mejora de rendimiento que se puede producir si se programan pensando en las nuevas características del hardware.

El segundo elemento con el que trabajaremos es la tarjeta gráfica, un dispositivo exclusivamente pensado para realizar dibujos tridimensionales en la pantalla, y que dispone de una potencia de cálculo muy superior a la del procesador. El interés por este componente viene de que mientras el usuario no ejecute programas que realicen cálculos tridimensionales, el procesador de la tarjeta gráfica permanece completamente parado. Esta infravaloración de los recursos, ha hecho que mucha gente comience a intentar aprovechar la potencia de la tarjeta para ejecutar otro tipo de programas no gráficos.

En el presente trabajo se estudian las formas de mejorar el rendimiento de los algoritmos de compresión de imágenes JPEG y de compresión de videos MPEG, intentando explotar el paralelismo existente dentro de los nuevos procesadores, y el existente entre el procesador y el procesador de la tarjeta gráfica.

Los objetivos que persigue este trabajo de investigación son:

- Adquirir un conocimiento claro y profundo acerca del funcionamiento de los algoritmos de compresión de imágenes y video, JPEG y MPEG.

- Estudiar las posibilidades de los procesadores gráficos como hardware de propósito general.
- Estudiar y experimentar con técnicas de paralelización de algoritmos en JPEG y MPEG
- Diseñar e implementar distintas optimizaciones sobre estos algoritmos, haciendo uso de las técnicas y elementos antes citados.
- Obtener conclusiones claras acerca de la utilidad de todas estas tecnologías en el campo de la compresión de imágenes.

Los resultados obtenidos reflejan como las mejoras hardware aumentan significativamente el rendimiento en los algoritmos, si son utilizadas correctamente. También corroboran que la tarjeta gráfica puede ser usada para desarrollar software no gráfico, obteniéndose un incremento significativo del rendimiento y sin coste añadido.

En el capítulo dos se introducen los estándares de compresión JPEG y MPEG, y se explican los métodos que suelen emplear los algoritmos para seguir el estándar.

El tercer capítulo describe la tarjeta gráfica, su estructura y la forma en que se programa.

En el capítulo cuatro se explican las implementaciones de los algoritmos JPEG y MPEG, y se citan algunas posibles optimizaciones que pueden ser investigadas.

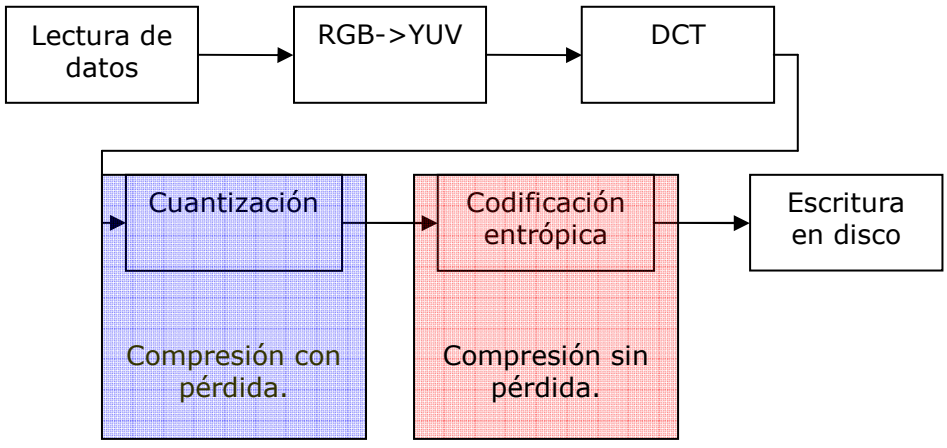
En el quinto capítulo se hace un estudio con los resultados obtenidos, se analiza las ventajas e inconvenientes de cada implementación y se sacan conclusiones sobre el uso del nuevo hardware en los compresores JPEG y MPEG.

2- Algoritmos tratamiento de imágenes

El concepto de compresión se define como el proceso de conseguir un volumen de datos inferior al original para representar una determinada información. Fundamentalmente se distinguen dos tipos de compresiones: las *compresiones sin pérdida* y las *compresiones con pérdida*. Las primeras permiten recuperar la representación original de la información de forma exacta a partir de la versión comprimida. Las compresiones con perdida permiten solo una reconstrucción aproximada de la representación original. A lo largo de este capítulo introduciremos los estándares de compresión JPEG y MPEG, así como un breve esquema que debe cumplir cualquier algoritmo para ajustarse a dichos estándares. Las descripciones completas de los estándares, así como una implementación básica y nuestras propuestas de optimización se desarrollan en detalle en el capítulo 4.

2.1.-Estándar JPEG (Joint Photographic Experts Group)

JPEG es un método de compresión para imágenes muy extendido en nuestra sociedad. Se trata de un formato grafico con un avanzado esquema de compresión con pérdida destinado a imágenes-estáticas o de video. Esta característica de pérdida de calidad es la diferencia radical frente a cualquier otro tipo de formato. Durante el proceso de compresión se pierden algunos datos. La calidad de la imagen es menor, pero el tamaño del fichero disminuye vertiginosamente (se alcanzan proporciones del 50:1 en imágenes en color **[1]**). En la figura Gráfica 2. 1, se muestra el flujo de datos minimizado del algoritmo JPEG, se explicará más en detalle en el capítulo 4.



Gráfica 2. 1 Flujo de datos minimizado algoritmo JPEG

Esta perdida de detalle, sin embargo, resulta casi imperceptible ya que la tecnología JPEG se basa en la percepción humana no lineal del color y la capacidad del cerebro para reconstruir la imagen a partir de una cantidad menor de información. El proceso por el que se reduce el tamaño del archivo está fundado en el uso de los espacios de color YUV (que delimitan una coordenada de luminosidad Y, y dos de cromatismo, U y V) en vez del RGB (que utiliza tres canales para indicar la cantidad de Rojo, Verde y Azul de cada píxel). En este último caso, los tres valores son absolutamente imprescindibles para definir el color de cada píxel. Sin embargo, en espacios YUV el detalle de la imagen es independiente del color. Esto permite comprimir más información en los dos canales de color que en el canal de brillo, sin perder casi nada del detalle visible. Por ello, traducir una imagen a YUV es el primer paso en cualquier algoritmo de compresión antes de iniciar el autentico trabajo. En el caso concreto del JPEG, la función matemática DCT (Discrete Cosine Transform) es la base del estándar. Se encarga de transformar los datos de una imagen a un formato más adecuado para la compresión con pérdida. Pueden descartarse, mediante cuantización, más o menos datos de color dependiendo de las preferencias del usuario. Esto significa mayor o menor gama de colores, pero con menor o mayor ratio de compresión, respectivamente.

Estos dos factores son la esencia del JPEG, pero aún queda un paso para que el proceso termine: la aplicación interna a la imagen de métodos de compresión sin perdida, que reducen todavía más el tamaño del archivo original. Tiene algunos problemas. El primero de ellos es el denominado bloqueo, que supone la visibilidad no deseada de los límites entre distintos bloques de píxeles. Otro son los desplazamientos de color cuando las gamas de cuantización escogidas son pocas. Además, los saltos bruscos entre colores tienden a suavizarse, perdiendo algo de definición. Y por último –y quizás el mas grave-, que una imagen una vez comprimida, perderá calidad si se modifica y vuelve a comprimir.

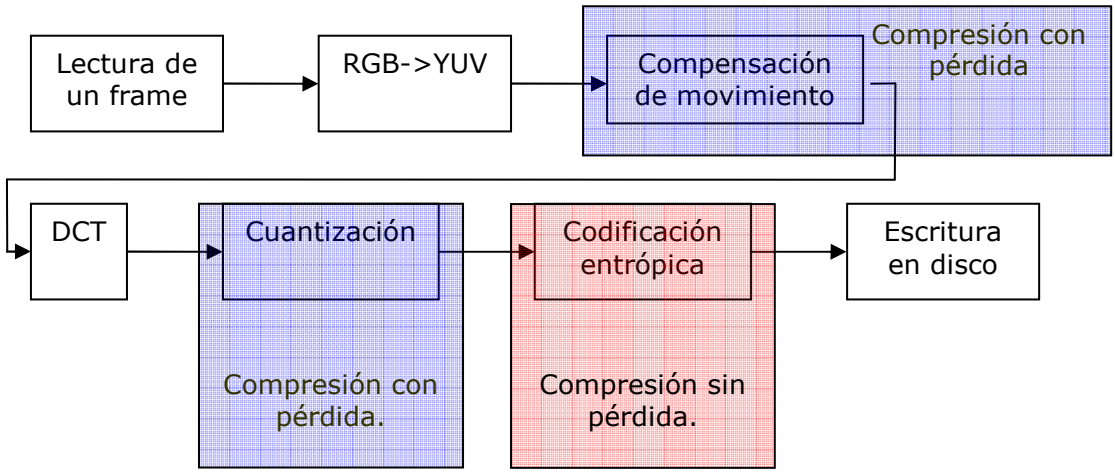
Es aplicable también para comprimir imágenes de video. Sin embargo, existen otros algoritmos más específicos para esta tarea como el MPEG, que utiliza métodos de compensación de movimiento, compresión intrafotograma o interfotograma e interpolación. A continuación explicamos un poco en profundidad el esquema del MPEG.

2.2.-MPEG:

Codificar video es el proceso de comprimir y descomprimir una señal de video digital. Hay que entender los conceptos de los formatos de muestreo y la calidad de las medidas. El video digital es una representación de una escena natural, tomada en un instante de tiempo y un lugar determinado.

Los pasos básicos que sigue el estándar MPEG son los siguientes, Gráfica 2. 2:

1. **Cambio de escala de colores a YUV** (explicado en JPEG).
2. **DCT**[1]: Se busca una codificación con pocos coeficientes de las frecuencias.
3. **Codificación Entrópica**.
4. **Compensación de movimiento**



Gráfica 2. 2 Diagrama básico del algoritmo de compresión MPEG

2.-DCT

La frecuencia espacial en los videos es bastante baja, y su variación en el tiempo es lenta. Se busca una transformada que

[1] **Discrete Cosine Transform**

pueda concretar la energía en muy pocos coeficientes. Para la aplicación de la transformada, lo primero que se busca es bajar la complejidad. Ello lo conseguimos dividiendo la imagen en bloques. El tamaño óptimo es de 8x8, que son transformados de acuerdo a la DCT. La DCT asocia a cada coeficiente una función de frecuencia específica ya sea horizontal o vertical, y su valor (después de la transformada).La DCT no ha reducido el número de bits que se requieren para representar ese bloque. La compresión se realiza al comprobar que la distribución de los coeficientes generados no sea uniforme. La compresión se consigue saltándose todos los coeficientes que están cerca de cero y cuantificando los restantes, produciendo pérdidas. **[2]**

3.- Codificación Entrópica

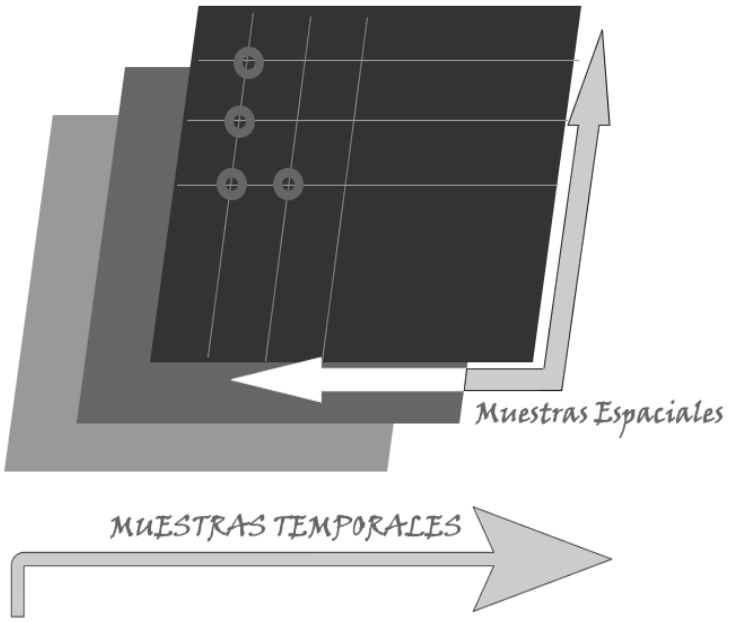
Los codificadores entrópicos son aquellos que tienen en cuenta la frecuencia de aparición de los signos a la hora de asignarles un código binario de representación, es decir, al asignar los códigos de longitud variable.

En MPEG, los codificadores utilizados implementan el algoritmo de asignación de Huffman que se basa en asignar códigos de longitud más grande a los símbolos que aparecen menos y más cortos a los símbolos más probables, disminuyendo de esta forma la tasa binaria. **[3]**

4.-Compensación de movimiento

Esta técnica tiene como objetivo principal eliminar la redundancia temporal entre las imágenes que componen una secuencia con el fin de aumentar la compresión. Para eliminar dicha redundancia, transmitiremos la diferencia entre un píxel en una posición de un fotograma y el píxel situado en la misma posición pero en el fotograma siguiente.

Esto sirve cuando las imágenes son estáticas. Pero lo normal es tener imágenes dinámicas y por tanto no podemos implementar lo anterior tal cual, sino que previamente tendremos que estimar el movimiento que ha sufrido un píxel de un objeto de un fotograma al siguiente. (Utilización de vectores de movimiento asociado) Gráfica 2. 3.



Gráfica 2. 3 Representación de localidad espacial y temporal

MPEG1 y MPEG2 son los estándares de codificación de video más extendidos en este momento. La técnica básica de MPEG 1 – 2 es la compensación completa del movimiento (DCT) y un algoritmo de código híbrido (DPCM[2]) **[1] [4]**. La diferencia principal del MPEG2 es que introduce soporte para video entrelazado., además se amplían el ancho de banda y la resolución máxima de los videos (ambas características se explican más adelante). Nosotros tomaremos el MPEG2 por ser el de uso más extendido.

El formato de video MPEG2

Una secuencia de vídeo tiene tres tipos de redundancia que un esquema de codificación necesita explotar en orden de conseguir una muy buena compresión:

- **Espacial**
- **Temporal**
- **Psicovisual**

El sistema de compresión MPEG-2 (al igual que MPEG-1) utiliza la Transformada Discreta del Coseno(DCT) y codificación de

[2] Differential Pulse Code Modulation

entropía para transformar un bloque de píxeles en códigos de longitud variable (VLC). Los bloques son la mínima unidad de codificación en el algoritmo MPEG. Se componen de píxeles de 8x8 y pueden ser de tres tipos: luminancia (Y), componente rojo de la crominancia Cr y el componente azul de la crominancia Cb. Mediante la DCT los bloques adquieren la forma de VLC, que no son más que la representación de de los coeficientes cuantificados de la DCT.

Los codificadores MPEG-2 producen tres tipos de de imágenes: *intra-frame* (o imágenes I), imágenes *interframe* causales (o imágenes P) e imágenes *interframe* bidireccionales (o imágenes B). **[1]**

Las **imágenes I**: Se codifican como si fuesen imágenes fijas utilizando la norma JPEG, por tanto, para decodificar una imagen de este tipo no hacen falta otras imágenes de la secuencia, sino sólo ella misma. No se considera la redundancia temporal. Se consigue una moderada compresión explotando únicamente la redundancia espacial. Una imagen I siempre es un punto de acceso en el flujo de bits de vídeo. Son las imágenes más grandes. **[3]**

Las **imágenes P**: Están codificadas como predicción de de la imagen I ó P anterior usando un mecanismo de compensación de movimiento. Para decodificar una imagen de este tipo se necesita, además de ella misma, la I ó P anterior. El proceso de codificación aquí explota tanto la redundancia espacial como la temporal. **[3]**

Las **imágenes B**: Se codifican utilizando la I ó P anterior y la I ó P siguiente como referencia para la compensación y estimación de movimiento. Para decodificarlas hacen falta, además de ellas mismas, la I ó P anterior y la I ó P siguiente. Estas imágenes consiguen los niveles de compresión más elevados y por tanto son las más pequeñas. **[3]**

Existen otro tipo de imágenes llamadas imágenes *intraframe* de baja resolución (o **imágenes D**) que son de las mismas características que las I pero con menos resolución. Se usan en aplicaciones que no necesitan gran calidad, como el avance rápido. **[3]**

Resumen de las características de MPEG

MPEG-1: Establecido en 1991, se diseñó para introducir video en un CD-ROM. Por aquel entonces eran lentos, por lo que la velocidad de transferencia quedaba limitada a 1.5 Mbps y la resolución a 352 x 240 píxeles. La calidad es similar al VHS y se usa para videoconferencias, el formato CD-i, etc. Es capaz de aportar mayor calidad si se le proporciona mayor velocidad.

MPEG-2: Establecido en 1994 para ofrecer mayor calidad con mayor ancho de banda (entre 3 y 10 Mbps). En esa banda, proporciona 720 x 486 píxeles de resolución, es decir, calidad TV. Ofrece compatibilidad con MPEG-1.

MPEG-3: Fue una propuesta de estándar para la TV de alta resolución, pero la versión fue interrumpida debido a que el MPEG2, con mayor ancho de banda, podía cumplir la misma función. **[4]**

MPEG-4: Se trata de un formato de muy bajo ancho de banda y resolución de 176 x 144 píxeles, pensado para videoconferencias sobre Internet. Realmente está evolucionando mucho y hay fantásticos codificadores software que dan una calidad semejante al MPEG-2 pero con mucho menor ancho de banda. **[3]** **[1]**

3. La tarjeta gráfica

La tarjeta gráfica es un componente hardware presente en los computadores actuales, que se encarga de procesar y obtener las imágenes que el usuario ve en el monitor. Para conseguir una imagen visible y real para los humanos, se necesitan realizar dos operaciones: procesar los gráficos tridimensionales que se van a dibujar, y enviar la señal de imagen adaptada al monitor. Esta última función es la más sencilla, y no repercute significativamente en el rendimiento[5]. En los últimos años, y debido especialmente a la industria de los videojuegos, este hardware ha evolucionado hasta el punto de introducir en la tarjeta un procesador especifico (Graphics Processing Unit) para el tratamiento de gráficos tridimensionales. En la actualidad este hardware se encuentra en continúa evolución, añadiendo más memoria, ampliando el ancho de banda y utilizando procesadores más generales.

A continuación, citamos los elementos principales de una tarjeta gráfica con una pequeña descripción (Fig. /*la que sea*/):

- **GPU(Graphics Processing Unit)** :Unidad encargada de ejecutar todas las operaciones e instrucciones sobre los datos. Haciendo un símil, se podría comparar con la CPU, la cual centraliza todos los cálculos en un computador.

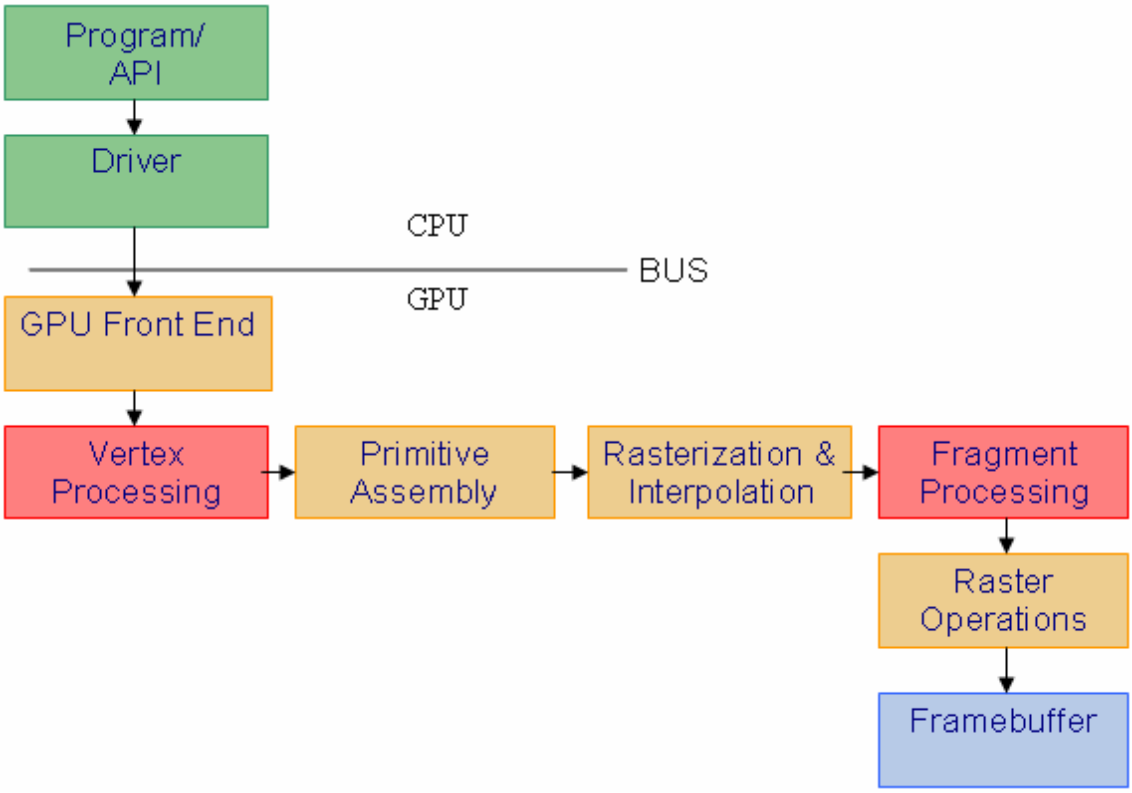
- **Memoria** Almacena los datos con los que trabaja la GPU. Estos pueden ser principalmente vértices, texturas o información de profundidad.

- **RAMDAC** Se encarga de convertir la señal de digital a analógico para que pueda ser reproducida por el monitor. Tiene en cuenta el modo de visualización y la frecuencia de refresco que tendrá la imagen.

- **Bus de conexión** Es el componente que se inserta en la ranura de la placa base, y conecta la tarjeta gráfica con el resto del computador. A través de este se realiza todo el intercambio de información con la CPU. Actualmente son de dos tipos AGP o PCI-Express, siendo este último el más nuevo y el que ofrece mayor ancho de banda.

3.1 Estructura GPU

La GPU es un procesador especializado para trabajar con gráficos, sobre los que aplica una serie de operaciones llamadas primitivas. Debido a que trabaja sobre un conjunto de datos muy específico, son procesadores muy especializados, concretamente diseñados para realizar cálculos en coma flotante y de forma paralela. Su estructura sigue un pipeline con un número variable de etapas, por las que pasa y se trata el flujo de información que produce la imagen final. Su estructura está representada en la figura **[6]**:



Gráfica 3. 1 Flujo de datos de la GPU

A continuación explicamos brevemente cada una de las fases por las que pasan los datos en la GPU.

Program/API : Programa que ejecuta el usuario, escrito en una API como OpenGl o DirectX

Driver: Comunicación entre el software y el hardware, su implementación permanece oculta al usuario.

GPU Front End La primera etapa, es la que recibe los datos directamente de la CPU a través del bus. Es muy importante la velocidad del bus, pues marca el flujo de datos CPU-GPU.

Vertex Processing Realiza transformaciones sobre los vértices de los objetos 3d.

Primitive Assembly Convierte los vértices en puntos, líneas y/o polígonos en el espacio. También se encarga de enlazar estos elementos entre sí formando fragmentos, que son la unidad básica de la etapa siguiente.

Rasterization & Interpolation Para cada elemento de la etapa anterior, llamado fragmento; calcula su área y coordenadas baricéntricas.

Fragment Processing En esta etapa se pueden programar diferentes operaciones sobre los fragmentos.

Raster Operations Calcula que elementos se tienes que dibujar, según la profundidad a la que se encuentran, y si hay elementos delante. También aplica las transparencias.

3.2 Modelo de programación

3.2.1 Diferencias con la CPU

El modelo de programación de la GPU difiere en gran medida del que tiene la CPU, fundamentalmente por la naturaleza de los datos con los que trabaja. La GPU está diseñada para trabajar exclusivamente con gráficos tridimensionales, sobre los que es necesario aplicar una serie reducida de operaciones. Sin embargo la CPU puede trabajar con un conjunto mucho más amplio de problemas, y tiene que ser lo suficientemente general para soportarlos todos.

Una ejecución sobre la GPU empieza cuando el programa envía un conjunto de polígonos a la tarjeta gráfica. Sobre los vértices de estos polígonos se aplican transformaciones, para después aplicar la rasterización, en la que los vértices son transformados en píxeles de pantalla o fragmentos. La última etapa selecciona cuales de estos fragmentos serán dibujados en la pantalla, atendiendo a los efectos y transparencia que tengan, y los envía al framebuffer que no es más que una porción de memoria reservada, que almacena los píxeles que se están visualizando.

Como vemos las operaciones siguen un orden similar en todas las ejecuciones, además las unidades básicas de procesamiento, los vértices, son independientes entre si. Esto permite que el cálculo pueda ser abordado de una forma mucho más rápida, de forma paralela. Por otro lado, los gráficos están representados por números en coma flotante. Siendo estas dos, las características más relevantes, nos encontramos con procesador paralelo de coma flotante, capaz de ejecutar cierto tipo de operaciones de forma mucho más rápida que la CPU, y muchas veces inutilizado por las aplicaciones no gráficas.

3.2.2 Programación de la GPU

Inicialmente las GPU´s estaban configuradas con una funcionalidad fija[7], establecida en el momento de fabricación y sin posibilidad de cambiarla. Tenían implementados un conjunto de algoritmos que permitían realizar las operaciones más comunes, pero no había forma de hacer algo diferente. Para solventar esta limitación aparecieron las partes programables, permitiendo al usuario implementar sus propios algoritmos y ejecutarlos sobre la GPU.

La GPU dispone de dos etapas programables ubicadas en: el procesador de vértices (Vertex Processing) y el procesador de fragmentos (Fragment Processing). Estas etapas pueden ser programadas por los usuarios, pero también pueden mantener la funcionalidad prefijada.

Vertex Shaders, se ubican en la parte programable del procesador de vértices, recibiendo como entrada vértices y texturas, sobre los que realizan operaciones de transformaciones e iluminación. Los datos recibidos por un vertex shader son de dos tipos:

- **Uniform variables:** Son variables cuyo valor permanece constante durante la ejecución del shader, y su valor no se puede cambiar hasta que este finalice.

- **Vertex attributes:** Son atributos particulares para cada vértice, y de solo lectura. Los más importantes son:

  - *Posición:* Las 3 coordenadas espaciales del punto
  - *Normal:* Vector normal al vértice para una cara
  - *Coordenadas de textura*
  - *Tangente*

Debido a que esta etapa se encuentra en mitad del pipeline, necesita algún modo de enviar los resultados a la etapa siguiente. Esto se consigue con las Varying variables que se explican a continuación.

- **Varying variables:** Son variables cuyo valor se escribe en los vertex shaders, y se lee en los fragment shaders.

Fragment Shaders: después de las etapas de rasterización e interpolación ,en las que se calculan los píxeles que representarán la imagen; los fragment shaders pueden modificar sus atributos, concretamente el color, y su visibilidad, terminando el procesamiento del pipeline.

3.2.3 Lenguaje GLSL

El GLSL (OpenGL Shader Language) es un lenguaje de alto nivel, específico para escribir vertex y fragment shaders. Concretamente se trata de una ampliación de la API OpenGL.

Para correr un shader en GLSL han de seguirse los siguientes pasos:

1. Escribir el código fuente del programa, en un archivo de texto.
2. Compilar el programa, con el compilador de GLSL.
3. Cargar el programa en la GPU.

A partir de este momento el shader forma parte del pipeline, y se aplica sobre el flujo de datos de la GPU explicado en el apartado anterior.

1. Escribir el código fuente

La sintaxis de GLSL es muy parecida a la de C, pero contiene algunas peculiaridades, como son los tipos vectores y matrices Al igual que en C el código ejecutable va dentro de funciones que siguen la estructura:

```
returnType function funcName(type0 arg0, … typen argn){

    //Function code

    ....

    return returnValue;

}
```

En la especificación de los argumentos, también es posible describir como son tratados los parámetros **[in|out|inout]** y/o **[const]**.

GLSL soporta los tipos representados en la tabla Tabla 3. 1

void	Para funciones que no devuelven valor
bool	Tipo condicional, □rae o false
int	Entero con signo
float	Punto flotante
vec2	Vector de dos float
vec3	Vector de tres float
vec4	Vector de cuatro float
bvec2	Vector de dos Boolean
bvec3	Vector de tres Boolean
bvec4	Vector de cuatro Boolean
ivec2	Vector de dos enteros
ivec3	Vector de tres enteros
ivec4	Vector de cuatro enteros
mat2	Matriz 2x2 floats
mat3	Matriz de 3x3 floats
mat4	Matriz de 4x4 floats
sampler1D	Un manejador para acceder a texturas 1D
sampler2D	Un manejador para acceder a texturas 2D
sampler3D	Un manejador para acceder a texturas 3D
samplerCube	Un manejador para acceder a una textura mapeada en un cubo

Tabla 3. 1 Tipos de GLSL

2. Compilar el programa

La llamada al compilador de shaders se hace a través de OpenGL, en el programa principal que va a usar los shaders. La secuencia que debe seguirse es la siguiente:

Crear el shader	***GLuint glCreateShader(GLenum shaderType);*** ***Parámetro:*** ***shaderType:*** ▪ GL_VERTEX_SHADER ▪ GL_FRAGMENT_SHADER

	Devuelve el identificador único que tendrá el shader.
Cargar el código fuente	***void glShaderSource (GLuint shader, int numOfStrings, const char **strings, int *lenOfStrings);*** ***Parámetro:*** ***shader:*** el identificador del shader ***numOfStrings:*** el número de cadenas de texto que tiene el array ***strings:*** las cadenas de texto con el código fuente del shader. ***lenOfStrings:*** array con la longitud de cada cadena
Compilarlo	***void glCompileShader(GLuint shader);*** ***Parámetro:*** ***shader***: el id. del shader

Tabla 3. 2 Tabla comandos GLSL para compilación de programa

Cargar el shader	
Crear un programa contenedor para el shader	***GLuint glCreateProgram(void);***
Adjuntar el shader al programa	***void glAttachShader(GLuint program, GLuint shader);***

	Parámetro: ***Program***: el id. del programa contenedor ***Shader***: el id. del shader que vamos a adjuntar
Lincar el programa	***void glLinkProgram(GLuint program);*** ***Parámetro:*** ***Program***: el id. del programa
Cargar el programa	***void glUseProgram(GLuint prog);*** ***Parámetro:*** ***Prog***: el id. del programa que vamos a usar, en caso de que sea cero, se vuelve a la funcionalidad fija

Tabla 3. 3 Tabla comandos GLSL cargar Shader

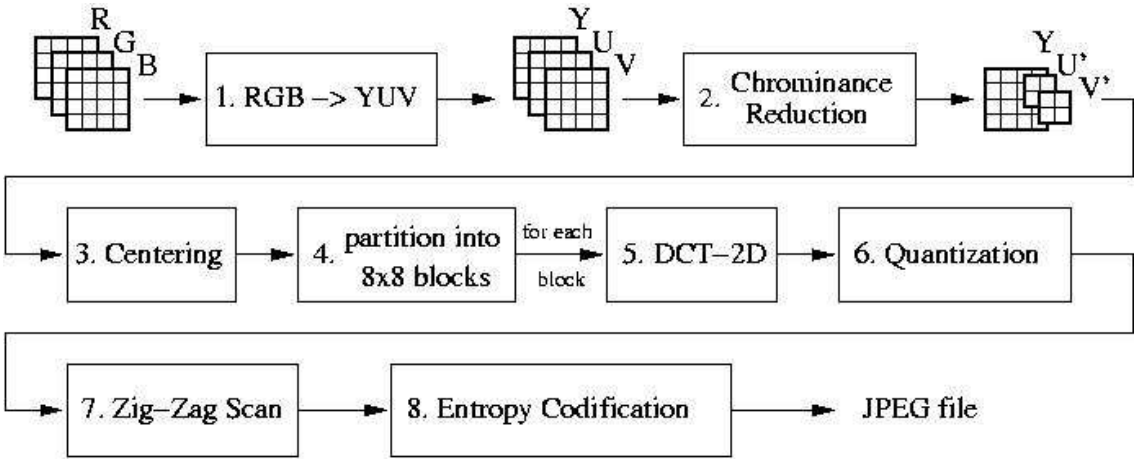
4- Propuestas de optimización y paralelización de las implementaciones del algoritmo JPEG y MPEG

A lo largo de este capítulo explicaremos en profundidad los dos estándares en los que se centra nuestro trabajo, JPEG y MPEG. De cada uno de ellos mostraremos una implementación básica o habitual y plantearemos ciertas modificaciones, que llevaremos a la práctica en el siguiente capítulo, para optimizar el rendimiento global y paralelizar su ejecución. En ambos casos planteamos optimizaciones futuras que han ido surgiendo en vista a los resultados obtenidos. Comenzaremos explicando el algoritmo JPEG ya que es parte de la base del algoritmo MPEG.

4.1 JPEG

1.1 Análisis del algoritmo general

En este apartado describimos el esquema general del algoritmo JPEG, y analizamos las distintas etapas con el fin de decidir que optimizaciones vamos a aplicar.



Gráfica 4. 1 Esquema general del algoritmo JPEG

El esquema de la figura muestra con todo detalle las fases presentes en el algoritmo JPEG que se explicó brevemente en el capítulo 2. No todas son igual de relevantes, concretamente estamos interesados en las que tengan un mayor peso en el coste temporal del algoritmo, esas son las siguientes:

1. **Transformación del espacio de color** (RGB -> YUV)
2. **DCT**[3]: Transformación de la información al espacio de la frecuencia.
3. **Cuantización**: Reducción de la información, basada en la percepción del ojo.
4. **Codificación Entrópica**: Codificación de los bits, para obtener una mayor compresión.

Trasformación del espacio de color.

Como ya se dijo en la introducción al JPEG, lo primero que ha de realizar cualquier algoritmo de este tipo es cambiar la forma de representar la información. A este proceso se le conoce como cambio del espacio de color RGB al YUV. Este modelo representa en tres matrices las componentes de luminancia (**Y**), crominancia azul (**U**) y crominancia roja (**V**) de una imagen, en lugar de la representación del color mediante las componentes básicas rojo (Red), verde (Green) y azul (Blue) propias del espacio RGB. El objetivo de este cambio es aprovechar dos defectos del ojo humano: el hecho de que el ojo humano es mucho más sensible al cambio en la luminancia que en la crominancia, es decir, notamos más claramente los cambios de brillo que de color. El otro es que notamos con más facilidad pequeños cambios de brillo en zonas homogéneas que en zonas donde la variación es grande.

Sean R, G y B las matrices que contienen la cantidad de rojo, verde y azul para cada píxel de la imagen respectivamente, las matrices Y, U, V se calculan como sigue: **[9]**

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$U = -0.299 \cdot R - 0.587 \cdot G + 0.886 \cdot B$$

$$V = 0.701 \cdot R - 0.587 \cdot G - 0.114 \cdot B$$

La transformación de color se aplica mediante operaciones aritméticas de punto flotante sobre cada píxel de la imagen, su coste temporal tiene cierta importancia en el tiempo total, y estará presente en nuestra búsqueda de optimizaciones.

[3] **Discrete Cosine Transform**

Reducción de crominancia y centrado.

Una opción que se puede escoger al guardar la imagen, es reducir la información del color respecto a la de brillo (debido al defecto en el ojo humano comentado anteriormente). Hay varios métodos: si este paso no se aplica, la imagen sigue en su espacio de color YUV, (esta reducción se entiende como 4:4:4), con lo que la imagen no sufre pérdidas. Puede reducirse la información cromática a la mitad, 4:2:2 (reducir en un factor de 2 en dirección horizontal), con lo que el color tiene la mitad de resolución (en horizontal), y el brillo sigue intacto. Otro método, muy usado, es reducir el color a la cuarta parte, 4:2:0, en el que el color se reduce en un factor de 2 en ambas direcciones, horizontal y vertical. Si la imagen de partida estaba en escala de grises (blanco y negro), puede eliminarse por completo la información de color, quedando como 4:0:0.

En nuestro estudio obviaremos esta etapa, por ser poco costosa y sin posibilidad de optimización, dejando la imagen con su configuración original 4:4:4.

Descomposición en bloques de 8x8

Llegados a este punto tenemos tres matrices, Y, U y V, y tenemos que comprimir la información que contienen. Por ello, es lógico pensar que a partir de ahora utilizaremos el mismo método para las tres matrices.

Se divide cada matriz en bloques de 8x8, de forma que los bloques que no sean múltiplos de 8 se rellenan copiando la última fila y columna (si fuese necesario). El motivo por el cual se dividen en bloques de este tamaño es puramente empírico y esta comprobado que es el tamaño que mejor rendimiento brinda **[9]**.

Cada submatriz 8x8 es independiente del resto y se convierte en la unidad mínima con significado para la compresión. Debido a esto, cada submatriz 8x8 se trata de forma independiente al resto siguiendo el mismo procedimiento que se explica a continuación.

Transformada Discreta del Coseno 2D(DCT-2D)

La transformada en coseno discreto es un caso particular, aplicada a las señales discretas (muestreos) de la transformada de Fourier, que descompone una señal periódica en una serie de funciones de seno y coseno armónicos.

La expresión de la DCT es

$$C_b(u,v)=\left(\frac{2}{M}\right)^{\frac{1}{2}}\cdot\left(\frac{2}{N}\right)^{\frac{1}{2}}\cdot\gamma(u)\cdot\gamma(v)\cdot\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\cos\left[\frac{\pi\cdot u}{2\cdot M}(2\cdot i+1)\right]\cdot\cos\left[\frac{\pi\cdot v}{2\cdot N}(2\cdot j+1)\right]\cdot X_b^c(i,j)$$

Dónde:

$$\gamma(n)=\begin{cases}\frac{1}{\sqrt{2}}\leftarrow n=0\\ 1\leftarrow n\neq 0\end{cases}$$

El coste computacional de la implementación directa de este cálculo es muy elevado (del orden de n^4); y no suele utilizarse en los algoritmos por ser excesivamente lenta e innecesaria. En su lugar se realiza la implementación rápida de la DCT **[10]**.

DCT rápida

Una propiedad de la DCT es que es separable, es decir, la DCT de dos dimensiones se puede hallar a partir de las DCT de una dimensión. En esta propiedad se basa el algoritmo rápido.

La idea es calcular la DCT de una dimensión sobre las filas de cada bloque, y sobre el resultado, aplicar de nuevo la DCT de una dimensión, esta vez sobre las columnas.

$$F(n)=C(n)\cdot\sum_{i=0}^{N-1}f(i)\cdot\cos\left[(2\cdot i+1)\cdot n\cdot\frac{\pi}{2\cdot N}\right]$$

¿Qué hemos conseguido con esta transformación? Esta transformación aplicada a cada uno de los bloques, genera para cada uno de ellos una nueva matriz de 8x8 compuesta por los coeficientes de las componentes de frecuencias espaciales cada vez más elevadas a medida que nos alejamos del origen (arriba a la izquierda), que representa la componente continua (luminancia media) del bloque.

El valor de estos coeficientes disminuye rápidamente cuando nos alejamos del origen de la matriz, terminando generalmente en una serie de 0.

De esta forma, si un bloque es de luminancia y color uniformes, únicamente el primer coeficiente no será nulo, y así sólo habrá que codificar un único coeficiente en lugar de 64.

Con este método rápido para calcular la DCT-2D conseguimos que el coste computacional se reduzca a la mitad (en la implementación CPU).

Esta etapa es una de las principales dentro del algoritmo JPEG, y también la más costosa en tiempo con más de un 50% del total. Por esto la tendremos en cuenta en nuestro estudio, enfocando las posibles optimizaciones sobre ella.

Cuantización

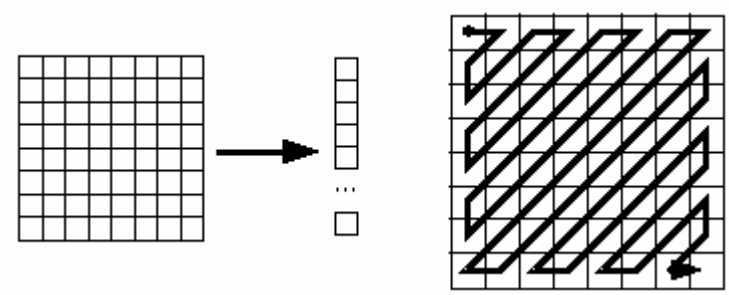
Como ya habíamos comentado, el ojo humano es muy bueno detectando pequeños cambios de brillo en áreas relativamente grandes, pero no cuando el brillo cambia rápidamente en pequeñas áreas (variación de alta frecuencia), esto permite eliminar las altas frecuencias, sin perder excesiva calidad visual. Esto se realiza dividiendo cada componente (Y, U, V) en el dominio de la frecuencia (tras aplicar DCT) por una constante para ese componente, y redondeándolo a su número entero más cercano. Este es el proceso en el que se pierde la mayor parte de la información (y calidad) cuando una imagen es procesada por este algoritmo. El resultado de esto es que los componentes de las altas frecuencias, tienden a igualarse a cero, mientras que muchos de los demás, se convierten en números positivos y negativos pequeños.

En los algoritmos JPEG la cuantización se realiza justo después de la DCT, y aunque su coste no es alto, simplemente multiplica cada píxel por un factor, se puede optimizar junto con la DCT sin dificultad. Es por lo que la incluiremos en nuestras optimizaciones, aun sabiendo que no reportará ganancias significativas.

La última parte del algoritmo consiste en reordenar los 64 elementos de los bloques 8x8 según un recorrido en sic-Saa, y utilizar un método de compresión sin pérdida para optimizar el resultado. Generalmente, el método utilizado es la codificación de Huffman debido a su sencillez.

Esta última etapa es puramente secuencial, porque trabaja a nivel de píxel comprimiendo las cadenas de bits. Es por tanto poco apta para ser abordada por métodos paralelos, y no la tendremos en cuenta en nuestras optimizaciones.

Recorrido SIG-SAG y codificación entrópica



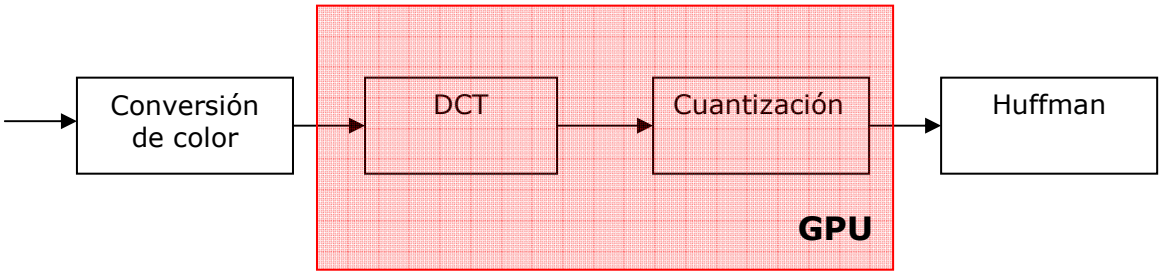
Gráfica 4. 2 Recorrido zigzag de una matriz

Una vez explicado el algoritmo, y visto que toda la carga de cómputo se encuentra en el cálculo de la DCT, planteamos nuestras optimizaciones para acelerar este cálculo.

4.1.2-Optimización 1: Implementación sobre la GPU (DCT lenta)

En esta primera optimización del algoritmo JPEG utilizaremos la GPU como procesador paralelo de coma flotante, para ver las posibilidades que puede ofrecer en algoritmos de compresión de imágenes, y las diferencias que guarda con la CPU.

Teniendo en cuenta el conocimiento acerca del algoritmo JPEG y su comportamiento, obtenido durante la etapa de análisis, decidimos empezar esta primera optimización llevando a la GPU las etapas: descomposición en bloques , DCT y cuantización.



Gráfica 4. 3 Esquema de optimización 1 JPEG

Descomposición en bloques de 8x8 y Transformada Discreta del Coseno 2D(DCT-2D)

En nuestra implementación sobre la GPU toda la información de la imagen se transmite en forma de textura, lo que nos permite pasar directamente, cada una en un canal, toda la información de las tres matrices Y, U, V; y así, trabajar con toda la información a la vez. Debido a la naturaleza del procesador vectorial GPU la descomposición en bloques no es necesaria físicamente, pero el algoritmo requiere que la información se trate de dicha forma. Para solucionar esto, hemos realizado, en la GPU, una descomposición virtual (a través de variables de control) de la textura en bloques 8x8.

En esta primera implementación hemos optado por codificar el algoritmo de la DCT que sigue la definición, también conocida como DCT lenta. Nuestra decisión se debe a que es la más sencilla de implementar, y queríamos ver si la GPU era capaz de conseguir un tiempo bueno para esta implementación, ya que la CPU es incapaz de hacerlo.

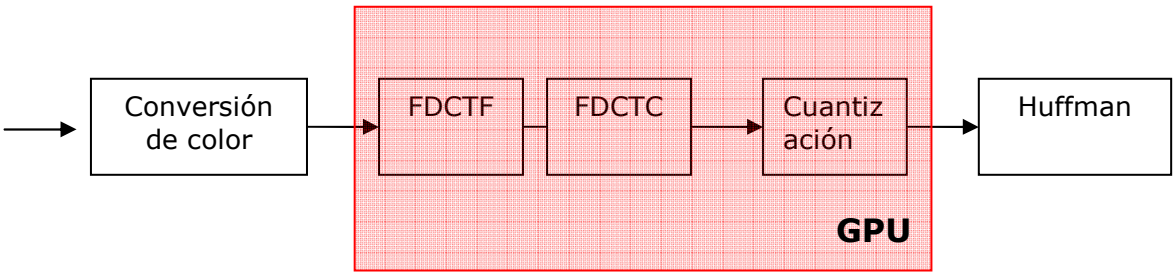
Cabe destacar que en la implementación CPU los cálculos se hacen de forma secuencial para cada una de las tres matrices, mientras que la GPU hace los tres cálculos a la vez ya que en una sola textura se pasa toda la información.

Cuantización

La implementación de este módulo es la misma que en la versión CPU pero aplicada directamente al final del Shader DCT.

4.1.3-Optimización 2: Implementación sobre la GPU(DCT rápida)

Visto los resultados obtenidos con la anterior optimización, y con objetivo de comparar la GPU y la CPU en igualdad de condiciones, realizamos la segunda optimización, en la que implementamos la DCT rápida sobre la GPU. En este caso el cambio principal con respecto a la anterior, está en el shader de la DCT que sigue el esquema rápido.



Gráfica 4. 4 Esquema de optimización 2 JPEG

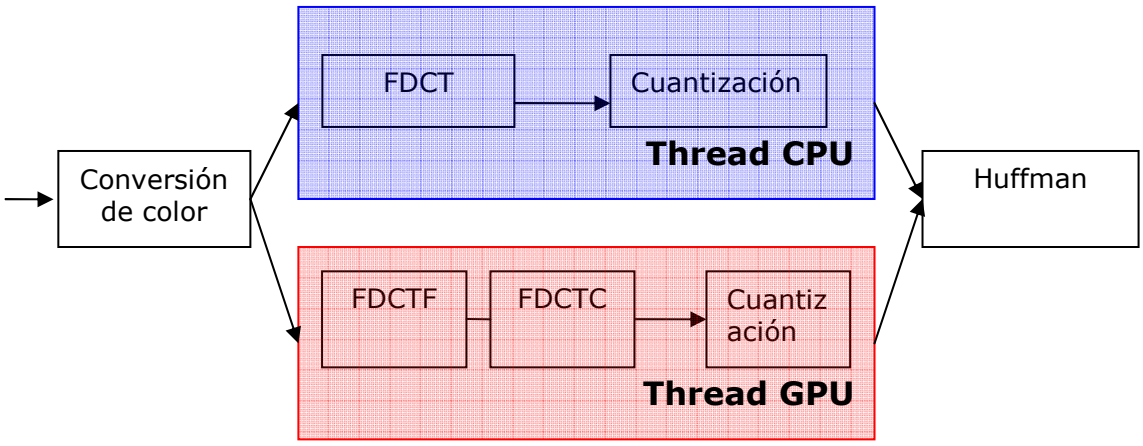
Descomposición en bloques de 8x8 y Transformada Discreta del Coseno 1D(DCT-1D)

La diferencia con respecto a la anterior DCT reside en que aquí los cálculos para cada píxel se realizan teniendo en cuenta solo 16 de los 64 vecinos que hay en un bloque 8x8. De este modo se reduce notablemente la cantidad de cálculos necesarios. Sin embargo para realizar la DCT completa se necesitan utilizar dos shaders, uno que la realiza por filas, y el otro por columnas. Esto implica que el algoritmo hará dos pasadas, es decir dos ejecuciones sobre la GPU. Aun así, como podemos ver en los resultados finales, esta implementación es mucho más rápida que la optimización 1.

4.1.4-Optimización 3: Implementación con threads(CPU-GPU)

Otro estudio interesante de optimizaciones se puede hacer aprovechando que la GPU y CPU son dos procesadores independientes, y por lo tanto pueden realizar operaciones simultáneamente.

Teniendo en cuenta el estudio realizado sobre el algoritmo JPEG, vemos que las optimizaciones tienen que centrarse en la etapa de la DCT. Esto sumado al hecho de que la etapa final del algoritmo es puramente secuencial nos lleva a pensar en una implementación paralela que no viole las dependencias de datos, y que procure mantener a los dos procesadores con carga de trabajo. En esta línea optamos por partir la imagen, y realizar la mitad del cálculo de la DCT y cuantización en cada procesador. Al terminar cada procesador de realizar el trabajo, dejará los datos en la etapa siguiente, codificación Huffman. Para evitar que un procesador se quede parado, estudiamos cual de los dos era más rápido para introducir en él la primera mitad de la imagen, la cual debe pasar antes por Huffman.



Gráfica 4. 5 Esquema optimización 3

4.1.5-Otras optimizaciones

En este apartado presentamos otras propuestas de optimización al algoritmo que no hemos llegado a probar, pero que podrían aportar resultados interesantes.

Conversión espacio de color

Podría ser interesante optimizar esta etapa, ya que trabaja sobre todos los píxeles de la imagen, y mientras en la CPU se hace de forma secuencial, en la GPU se podría explotar todo el paralelismo disponible.

Una de las razones por la que no decidimos implementarla, es porque para cargar la imagen en una textura, teníamos que recorrerla en la CPU antes, y aprovechamos esto para aplicarle la conversión de color directamente. Si no hubiésemos tenido que realizar este recorrido, sí hubiésemos pasado la etapa a la GPU, pero en nuestro contexto no compensaba realizar una pasada más sobre el pipeline del procesador gráfico.

Una posible implementación de este modulo sería diferente a la utilizada en la CPU. Recordemos que para transformar el espacio de RGB a YUV utilizábamos tablas precalculadas, para minimizar el número de operaciones aritméticas. En este caso al disponer de un procesador muy especializado en operaciones de este tipo varias veces más rápido (Pentium 4: 12GFLOPS ,ATI Raleón X1800XT:

120GFLOPS **[11]**), sería preferible ahorrar transferencias de memoria entre la CPU y la GPU.

Implementación sobre la CPU:

Y = ((tablaYR(R)+tablaYG(G)+tablaYB(B))>>16)-128;

Cr = (tablaCrR(R)+tablaCrG(G)+tablaCrB(B))>>16;

Cb = (tablaCbR(R)+tablaCbG(G)+tablaCbB(B))>>16;

Total: (10 operaciones ,9 accesos a memoria) x píxel

Además hay que contar el gasto requerido para cargar las tres tablas de 256 números en coma flotante, en la memoria de la CPU.

Implementación sobre la GPU:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

$$U = -0.299 \cdot R - 0.587 \cdot G + 0.886 \cdot B$$

$$V = 0.701 \cdot R - 0.587 \cdot G - 0.114 \cdot B$$

Total: (15 operaciones,1 acceso a memoria)x píxel

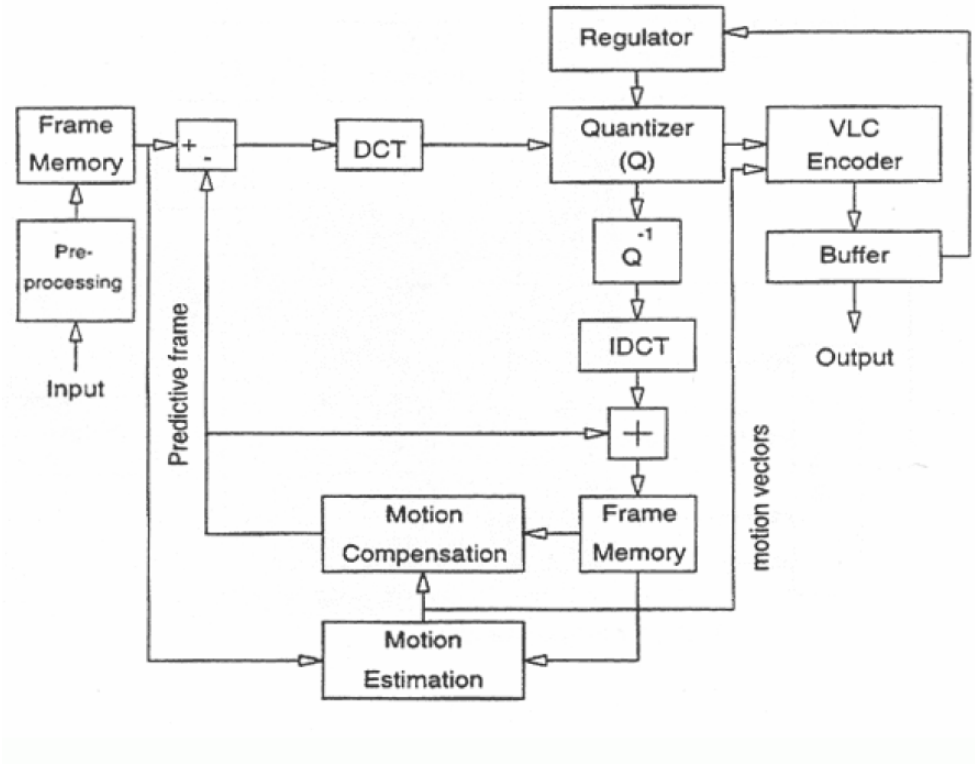
Vemos que esta etapa podría implementarse de varias formas, sin embargo nosotros no vamos a ahondar más en ella por no ser el objetivo final de este proyecto. Aun así dejamos abierta una posible implementación en la que las tablas fueran precalculadas en la GPU, y almacenadas en una textura. Esta alternativa podría sacar ventaja ahorrando la transferencia entre CPU y GPU, y varias operaciones aritméticas.

4.2 MPEG2

4.2.1 Análisis del algoritmo general.

El estándar MPEG2 no especifica una arquitectura concreta, solo impone una serie de normas y restricciones que deben cumplir los videos en este sistema**[12].** Sin embargo para satisfacer estos requisitos los codificadores/decodificadores suelen tener una organización muy similar.

En nuestro caso hemos realizado la implementación de un codificador MPEG2, con una arquitectura por etapas(Fig. 4.3.1)



Gráfica 4. 6 Arquitectura codificador MPEG-2

El codificador trabaja sobre una secuencia de imágenes(frames), que se van obteniendo de archivos bitmap sin comprimir. Estos frames se cargan en memoria y posteriormente se tratan para comprimir la secuencia completa. Los pasos que sigue un frame dentro del codificador son:

- Se analiza la información del frame, sus píxeles, junto con la situación de este dentro de la secuencia. De este modo se decide el tipo de frame que es :

    - Frame I: Se codifica sin tener en cuenta la redundancia temporal, como una imagen separada. Este tipo de codificación es similar a la que se realiza para las imágenes JPEG. El motivo de que existan frames I es para introducir nueva información(píxeles) que no podía ser predicha y evitar la degradación que se produce en las estimaciones
    - Frame P: Este tipo de frames utiliza las imágenes anteriores para estimar sus píxeles. Para ello es necesario un proceso costoso en el que se recorre la imagen precedente (en el tiempo) buscando bloques que coincidan con los de la imagen que se quiere estimar.
    - Frame B: Aquí la estimación se realiza a partir de las imágenes posterior y anterior, basándose en el hecho de que los píxeles de las imágenes cercanas en el tiempo tienden a ser parecidos en ambas direcciones.

- **Motion estimation:** Esta etapa es la encargada de explotar la redundancia temporal del vídeo, y se aplica sobre los frames de tipo P y B, los de tipo I no pasan por esta etapa.

  Cuando llega a un frame se divide en bloques de mxn píxeles, llamados *macrobloques*, que conforman la unidad de trabajo del motion estimation. Las dimensiones de estos macrobloques no vienen especificadas en el estándar dejándose al diseñador la tarea de elegir el tamaño que mejor se ajuste a su situación, sin embargo lo más común es que sean de 16x16.

  La codificación temporal busca remplazar macrobloques del frame que se está procesando, por vectores de desplazamiento con respecto a los macrobloques de frames posteriores o anteriores. Para ello se van recorriendo los macrobloques y por cada uno, se realiza una búsqueda en una región de la imagen de referencia. Esta región se conoce como *ventana de búsqueda.* Para estimar si un bloque es una buena aproximación, se ha de computar una

función sobre los píxeles de ambos macrobloques. En nuestro caso utilizamos la *suma absoluta de las diferencias* (SAD).

- **Motion compensation:** Una vez se han calculado los vectores de movimiento, es necesario escoger el que mejor ajuste al macrobloque El método de búsqueda del *macrobloque* que optimiza el SAD también depende de la implementación; están los que realizan búsqueda ehuxastiva *(fullsearch)* y los que solo realizan unas comprobaciones determinadas *(logarítmico).*

    - ***El fullsearch*** aplica el SAD a todos los macrobloques que están dentro de la ventana de búsqueda.

    - ***El método logaritmo*** divide la ventana de búsqueda en cuadrantes, y escoge el cuadrante en el que mejor resultados haya obtenido, de esta forma evita realizar búsquedas en otras zonas que tienen una menos probabilidad de aproximar al macrobloque.

    Al finalizar la búsqueda se obtiene un vector relativo al macrobloque actual, que apunta al que mejor lo aproxima, menor valor del SAD; junto con el valor de SAD obtenido. Según este valor este dentro de un umbral o no, se decide si predecir el macrobloque a partir del vector, o mantenerlo y desechar el vector*(macrobloque I).*

- **DCT:** Esta etapa, se procesa de forma idéntica a como se realiza en el algoritmo JPEG; con la distinción de que ahora la unidad de básica son bloques de 16x16 que deben ser divididos en cuatro bloques de 8x8 para aplicar la transformada.

4.2.2 Optimización 1: Implementación del MPEG sobre la GPU (DCT rápida)

Uno de los principales objetivos de este trabajo era estudiar implementaciones alternativas del algoritmo de codificación de video MPEG que hicieran uso de la GPU. Por la naturaleza del procesador gráfico se hacía impensable migrar todo el código del MPEG para ser ejecutado en la GPU, sin embargo esto no es un problema ni siquiera algo necesario, pues basta con centrarse en las etapas más costosas e intentar reducirlas en la GPU.

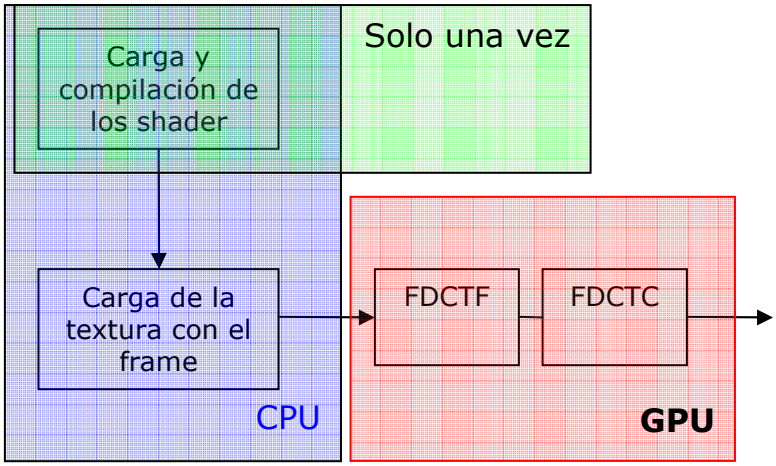
Después de realizar un análisis sobre la versión del algoritmo en CPU, llegamos a la conclusión de que el 70% del tiempo se encontraba repartido entre dos módulos. La DCT con aproximadamente un 20% y la estimación de movimiento con el 50% **[Datos en pruebas MPEG]** , luego quedaba fijado el futuro esfuerzo de la investigación a intentar reducir estos tiempos con apoyo de la GPU.

MPEG con DCT sobre la GPU

En esta línea de trabajo nos centramos en pasar el módulo de la DCT ,que se aplica sobre todos los bloques de un frame, a la GPU. El tipo de transformada utilizada por el MPEG es FDCT, similar a la implementación del algoritmo JPEG, con algunas particularidades. Entre estas particularidades la más importante es que en mpeg de los 3 canales de color, los canales crominancia rojo y crominancia azul son de tamaño la cuarta parte que el canal de luminancia (debido a la fase de reducción que no aplicamos en el JPEG). Esto permite que el mpeg ahorre espacio al guardar los frames a costa de sacrificar calidad. En cuanto al tiempo de GPU se refiere no supone ninguna ganancia, pues el procesador trabaja sobre texturas de 1,3 o 4 canales, siendo cada canal del mismo tamaño en bytes.

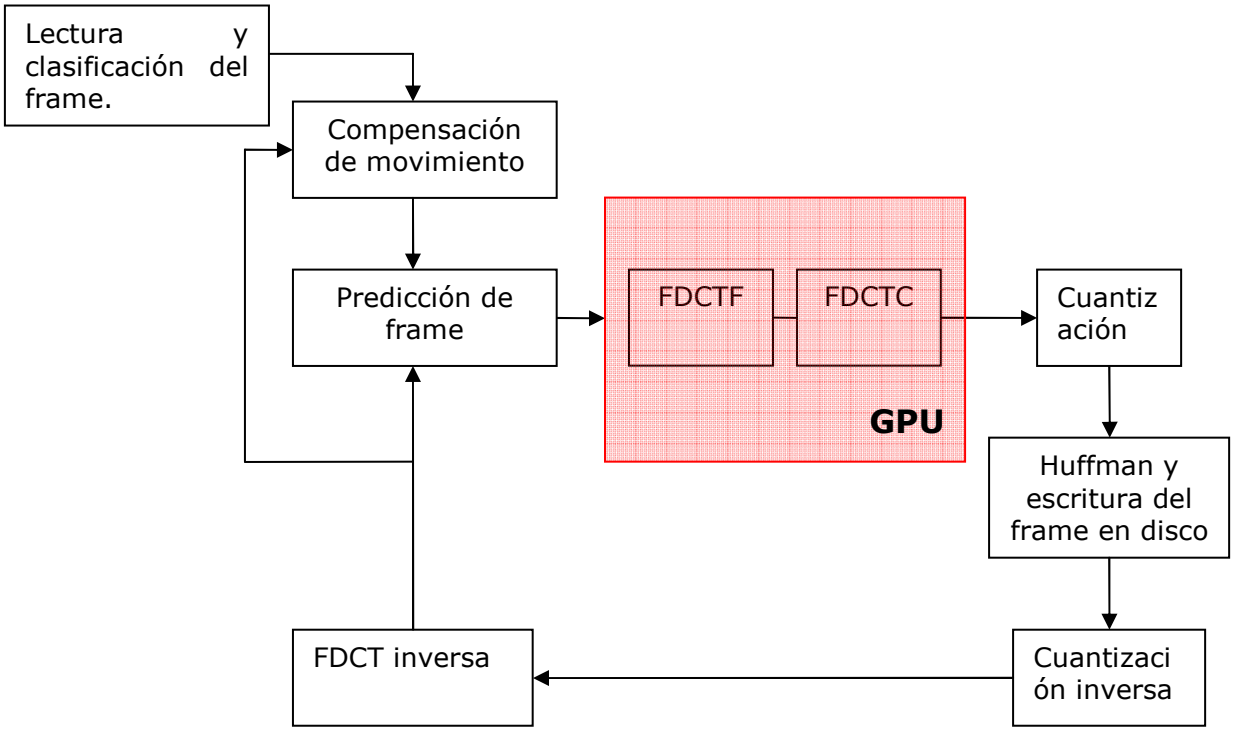
El nuevo módulo de la DCT sobre la GPU, que suplanta al de la CPU, sigue el esquema (Fig. 4.3.2):

1. Carga y compilación de los shaders. Esta etapa en realidad solo se ejecuta una vez, y se encarga de inicializar los programas que ejecutará la GPU.
2. Carga de la textura con los píxeles del frame actual. Inicialización de los buffers y texturas intermedias.
3. Se ejecuta la primera pasada del shader que realiza la FDCT por columnas. El resultado se escribe en la textura intermedia.
4. Segunda pasada del shader con la FDCT por filas. El resultado se guarda en la memoria de la CPU.



Gráfica 4. 7 Optimización 1 : DCT en la GPU

Este algoritmo se ejecuta sobre todos los frames de la secuencia de video. Con él hemos pasado una parte del pipeline a la tarjeta gráfica. En la siguiente imagen se muestra el procesado completo de un frame teniendo en cuenta la última optimización (Fig. 4.3.3):



Gráfica 4. 8 Tratamiento de un frame.

Como puede verse en este esquema, el tratamiento de un frame no termina cuando su codificación se escribe en el disco; sino que hay

todo un proceso de vuelva atrás para recuperar el frame original. Esto se debe a que el procesado de los próximos frames utilizará este frame para calcular la estimación de movimiento y la predicción. Por lo tanto existe una dependencia lineal entre el frame que se procesa y los siguientes a él (esta dependencia nos obligará a desechar una cuarta optimización) **[4.3.4 Posibles optimizaciones]**.

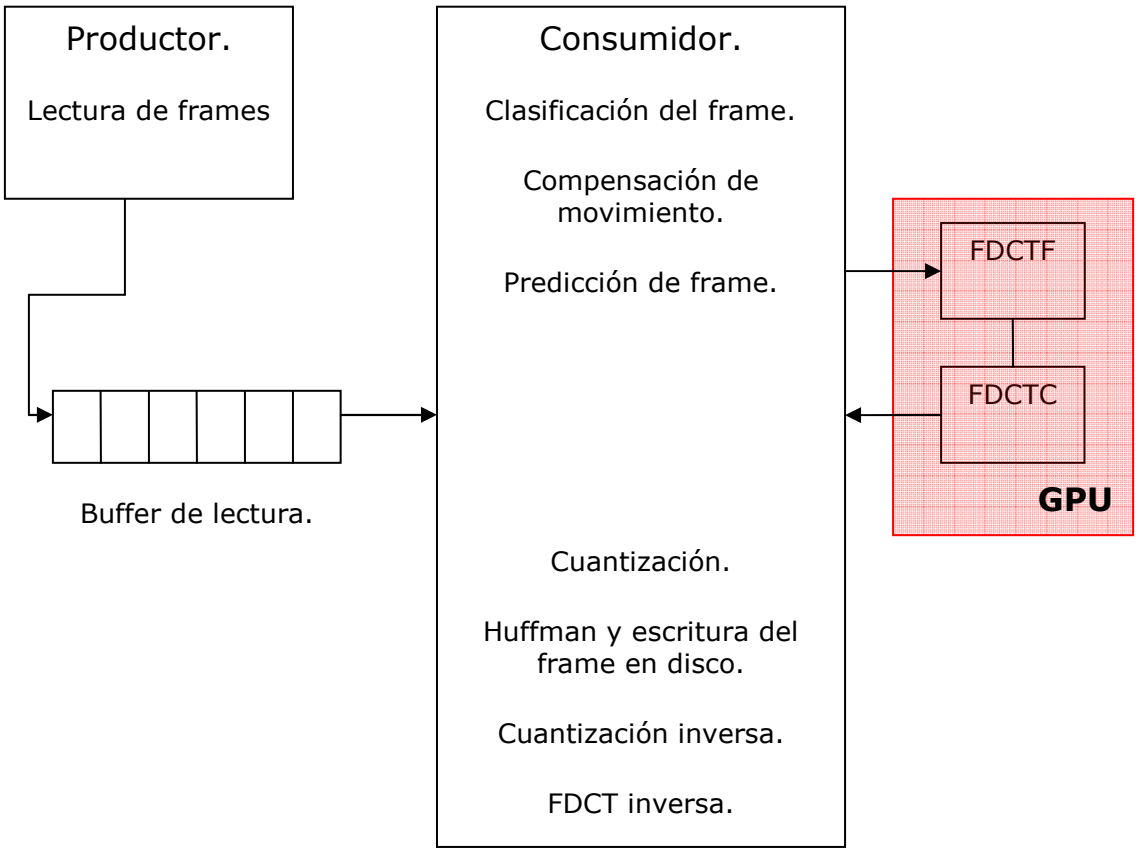
4.2.3 Optimización 2: Implementación del MPEG con threads (CPU-GPU)

Con la optimización 1, ya hemos conseguido ganancia y por tanto hemos cumplido con uno de los objetivos de este trabajo. Aún así, y viendo que la introducción de hilos en el JPEG mejora el tiempo total de cómputo. Nos planteamos una segunda optimización que incluya un sistema de hilos.

Hasta ahora, todas las operaciones sobre un frame se realizan de forma secuencial en una única etapa (desaprovechando potencia, por ejemplo, cuando la GPU esta haciendo cálculos). La nueva mejora que proponemos es realizar un pipe simple de dos segmentos dentro del tratamiento de cada frame de forma que se aproveche al máximo el uso de los dos procesadores (CPU-GPU).

Mantendremos un hilo productor cuyo objetivo es leer de disco las imágenes a procesar y guardarlas en un buffer de lectura; un segundo hilo consumidor, que gestione el procesado de los frames (incluyendo el cálculo de la DCT en la GPU). Así, mientras el hilo consumidor está calculando la DCT en la GPU, el hilo productor puede aprovechar e ir guardando en el buffer nuevos frames (Fig. 4.3.4). Con esta mejora pretendemos ahorrar tiempo de espera a la CPU en los accesos a disco, que se producían en la implementación anterior, debidos a la lectura de frames por demanda.

Hay que destacar que esta mejora de paralelización no aporta nada nuevo al esquema que utiliza la GPU, y bien podría aplicarse directamente al algoritmo básico que solo utiliza la CPU.



Gráfica 4. 9 Optimización 2: algoritmo MPEG con threads.

4.2.4 Optimización 3: Implementación del MPEG con threads (CPU-CPU)

Siguiendo el esquema de la propuesta anterior, planteamos otra optimización en la que los dos hilos se ejecuten completamente sobre CPU sin utilizar la GPU. Con esta optimización esperamos sacar el máximo rendimiento a los procesadores de doble núcleo actuales o equipos antiguos con tarjetas gráficas obsoletas.

4.2.5 Otras optimizaciones.

En esta sección trataremos optimizaciones que podrían aplicarse al algoritmo MPEG para mejorar su rendimiento. De cada una de ellas daremos las motivaciones que nos llevan a plantearlas así como el tipo de mejora que supondría. Todas estas, son mejoras que proponemos como continuación de nuestro trabajo y que han surgido del análisis de los resultados.

3. Implementación de la estimación de movimiento en la GPU.

4. Implementación de un pipe segmentado procesando 4 o más frames al mismo tiempo.
5. Implementación de un pipe segmentado procesando 4 o más frames de GOP's distintos al mismo tiempo.

4.2.5.1 Implementación de la estimación de movimiento en GPU.

La estimación de movimiento, como se verá en los resultados, es la etapa más costosa de todo el algoritmo. Se basa en calcular la suma de diferencias absolutas (SAD) entre el bloque de píxeles 16x16 que se quiere estimar con todos los bloques de píxeles 16x16 contenidos en lo que se conoce como la ventana de búsqueda (fracción del frame donde es posible encontrar el bloque estimado).

La implementación de esta etapa es paralelizable, especialmente el cálculo del SAD. En la CPU consiste en calcular de forma secuencial la diferencia píxel a píxel; esto, en el procesador gráfico, es completamente paralelo. Esta optimización debería mejorar el tiempo de cálculo de forma drástica (recordemos que esta etapa consume prácticamente el 40% del tiempo total) **[5.2 Pruebas y resultados]**. Los problemas que presenta llevar la compensación de movimiento a la GPU son, principalmente:

1. El uso de al menos tres texturas para tener los datos en la GPU. Eleva el tiempo debido a transferencia CPU-GPU.
2. La realización de múltiples pasadas a través del procesador gráfico.
3. La poca uniformidad a la hora de tratar los datos. Dependiendo del tipo de frame que se procesa la estimación de movimiento necesita usar uno o dos frames como referencia; además de que la búsqueda que se aplica puede ser también distinta (fullsearch o logarítmica independientemente de tener que calcular el SAD).

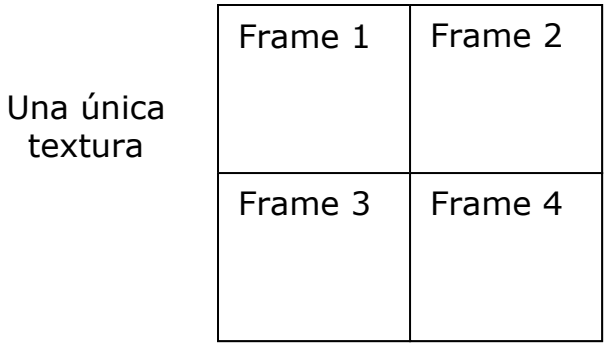
Aún con estos inconvenientes pensamos que esta optimización puede producir buenos resultados.

4.2.5.2 Algoritmo paralelo por frames.

El rendimiento del hardware gráfico frente a la CPU se ve aumentado a medida que el tamaño de los datos a procesar se vuelve más grande **[5 Pruebas y resultados]**. Este hecho, combinado con que el estándar MPEG-2 está pensado para videos cuyos frames tengan una dimensión máxima de 720x576, y que las tarjetas

gráficas de media soportan texturas de 2048x2048; nos ha inspirado para plantear una optimización que aproveche estas ventajas.

La idea es por qué no procesar los frames de 4 en 4 (u otras cantidades) de forma que el cálculo de la DCT se haga con una textura que contenga los cuatro frames en una disposición 2x2 (Fig. 4.3.5)



Gráfica 4. 10 Optimización 5: distribución de los frames

Esto presenta como ventaja que se realizan cuatro veces menos pasadas por la tarjeta gráfica y que aprovecha mejor el uso del hardware gráfico al utilizar texturas muy grandes (para texturas pequeñas no merece la pena la transferencia de datos con la GPU tal y como queda reflejado en los resultados).

Por desgracia, esta implementación, que no altera demasiado la arquitectura básica, tiene el inconveniente que se comentó con anterioridad en este trabajo. Los frames están sujetos a una dependencia lineal, de forma que no se puede procesar uno hasta que no se termine el anterior. Debido a esto es inviable la realización de esta mejora.

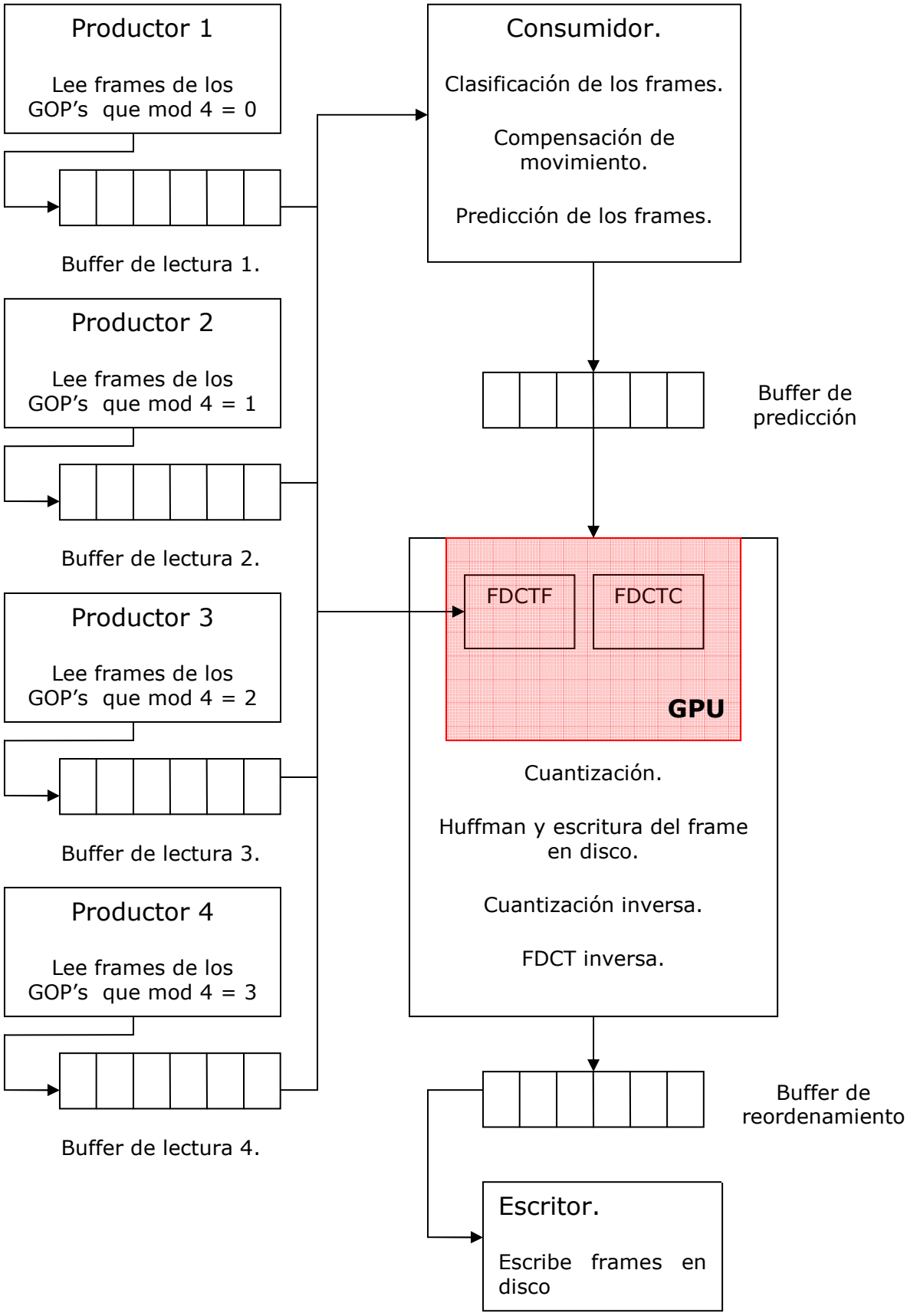
4.3.5.3 Algoritmo paralelo por frames de diferentes GOP's.

Los videos MPEG están organizados mediante secuencias de frames que a su vez se agrupan en GOP's (Group Of Picture). Los GOP's son unidades independientes entre sí que mantienen toda la información necesaria para codificar sus frames. ¿Por qué no aplicar la idea de la optimización anterior pero con frames de distintos GOP's?

Es viable, pero complica bastante la estructura del algoritmo, ya que hay que duplicar todas las estructuras para cada GOP adicional que se quiera procesar.

Con esta mejora, se procesan frames de varios GOP's al mismo tiempo, lo que produce un problema de sincronización a la hora de escribir en disco. No se pueden escribir todos los frames procesados hasta que todos los frames anteriores hayan sido escritos en disco.

Como una aproximación planteamos la siguiente arquitectura de un trabajo futuro para mejorar el algoritmo MPEG (Fig. 4.3.6).



Gráfica 4. 11 Optimización 5 : Arquitectura del algoritmo paralelo

5.-PRUEBAS Y RESULTADOS

Con el fin de analizar las diferentes propuestas del capítulo anterior, hemos llevado a cabo pruebas intensivas sobre todas las implementaciones realizadas. Para que los datos obtenidos fueran lo más fiable posibles utilizamos material de dos fuentes distintas: por un lado las imágenes y videos estándar, que se utilizan en todas las pruebas y benchmarks; y por otro, material obtenido por nosotros, que pensamos puede ser representativo para estas implementaciones en particular.

Debido a que se trata de un proyecto de investigación, que pretende estudiar distintas optimizaciones temporales, la variable que mediremos será el tiempo de ejecución. Como es bien sabido, el hardware y los equipos utilizados para las pruebas juegan un papel fundamental en la duración de las ejecuciones. Por este motivo hemos realizado un amplio estudio de todas las implementaciones sobre diferentes plataformas.

Las tres plataformas utilizadas son:

Equipo doméstico (César)

```
Sistema operativo: Linux, Ubuntu 7.04 sist. mínimo

Compilador utilizado: gcc

CPU: Pentium 4 Prescott con Hiper Thread 3.4 GHz 1024MB RAM-DDR2

Tarjeta Gráfica: NVidia GeForce 7300 GT
```

Como podemos ver en la tabla de especificaciones, este es un equipo de ámbito doméstico que perfectamente puede estar presente en la mayoría de los hogares. No consta con ningún hardware caro, y dispone de una tarjeta gráfica, montada para jugar y disfrutar de contenido multimedia. Lo incluimos en nuestro estudio para ver lo que las optimizaciones pueden ofrecer sobre un hardware *de andar por casa*.

Equipo diseño gráfico (Yelmo)

```
Sistema operativo: Linux, Debian sist. mínimo

Compilador utilizado: gcc

CPU: Pentium 4 3.4 GHz 2048MB RAM-DDR2

Tarjeta Gráfica: NVidia GeForce 7800GTX
```

Este equipo ya se distingue del anterior, sobre todo por la tarjeta gráfica que posee, una NVidia GeForce 7800GTX. Esta tarjeta ya está considerada de gama media-alta, y nos ofrece un procesador bastante más potente que el anterior.

Equipo avanzado (K2)

```
Sistema operativo: Linux, Debian sist. mínimo

Compilador utilizado: gcc

CPU: Pentium Core 2 Duo E6600 2048MB RAM

Tarjeta Gráfica: NVidia GeForce 8800GTX
```

Finalmente disponemos de un equipo con una configuración avanzada, que definitivamente queda fuera del ámbito doméstico. En CPU, tiene un Core 2 Duo, compuesto por dos procesadores con 4mb de memoria cache, en este aspecto aventaja notablemente a los dos predecesores, que son de anterior generación. El otro plato fuerte es su tarjeta gráfica, la NVidia GeForce 8800GTX, que actualmente está entre las mejores del mercado.

El esquema general de las pruebas se compone de dos bloques, el algoritmo JPEG y el algoritmo MPEG. Para cada bloque hacemos un estudio detallado del algoritmo, identificando las zonas de código candidatas a ser optimizadas mediante paralelización, y las comparativas de las distintas estrategias, utilizando como caso base la ejecución en CPU. Todas las mediciones se han realizado 10 veces, descartando el caso mejor y el caso peor, y haciendo la media de las restantes.

5.1 Pruebas y resultados algoritmo JPEG

En esta sección plasmaremos los resultados obtenidos al ejecutar las cuatro implementaciones del JPEG realizadas. El algoritmo recibe como entrada un imagen sin comprimir, formato BMP, y devuelve una imagen comprimida JPEG. Cada ejecución por lo tanto requiere de una imagen de entrada; serán las características de estas imágenes las que determinen el tiempo empleado en la compresión.

Las características más relevantes de una imagen son:

- **Dimensiones:** número de píxeles, normalmente se expresa: alto x ancho. También determina el espacio que ocupa el archivo en disco.
- **Entropía de la imagen:** indica lo diferente que son sus píxeles entre sí. Por ejemplo la entropía en una imagen toda del mismo color es mínima.

De estas dos, la entropía determina el grado de compresión que se consigue después de aplicar el JPEG, mientras que las dimensiones marcan el tiempo que empleará el algoritmo.

En nuestro caso particular, estamos interesados en optimizaciones temporales, luego la característica más importante para el estudio será el tamaño de la imagen, quedando la entropía en un segundo plano. Dicho esto pasemos a introducir las imágenes que utilizaremos para medir el JPEG.

En la tabla Tabla 5.1 1 se muestran la relación de dimensiones de las 16 imágenes distintas utilizadas para las pruebas, dentro de estas hay 13 estándares y 3 tomadas por nosotros, estas ultimas las hemos introducido porque dentro del estándar había falta de imágenes grandes(2048x2048), que en nuestra investigación juegan un papel importante.

Nombre de la imagen	Dimensiones
Clock	256x256
House	256x256
Jelly beans	512x512
Imagen4	512x512
Water(D38H.E.)	512x512
San Diego	512x512
Imagen7	512x512
Lena	512x512
Couple	512x512
Car and APCs	1024x1024
Stockton	1024x1024
Airplane (U-2)	1024x1024
21 level step wedge	1024x1024
Píxel ruler	2048x2048
256 level test pattern	2048x2048
General test pattern	2048x2048

Tabla 5.1 1 Batería de pruebas JPEG

A continuación hacemos el análisis temporal del algoritmo base con vistas a su paralelización.

5.1.1 Análisis temporal algoritmo general en CPU

El algoritmo general como su nombre indica, es la base de todas las implementaciones JPEG, y se utiliza como punto de partida para planear las optimizaciones. Las medidas obtenidas servirán para conocer cuantitativamente el comportamiento del algoritmo y disponer de datos suficientes para estimar las etapas a optimizar.

Para este análisis hemos optado por medir dos variables:

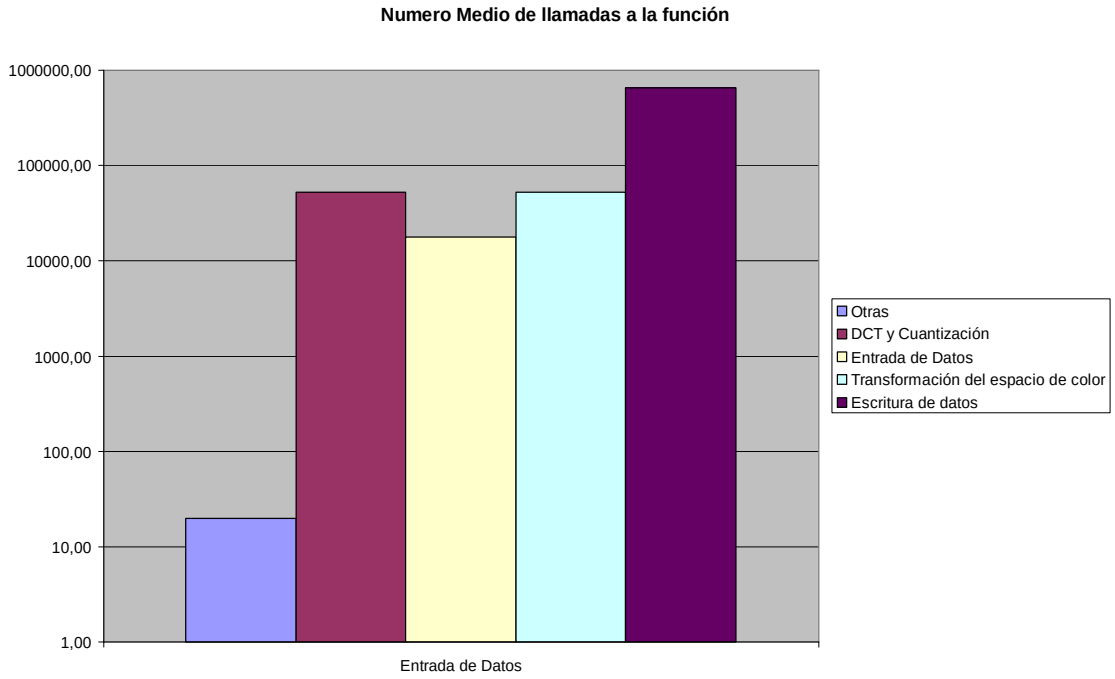
- **Número de llamadas a funciones:** Al tomar estos datos, podemos hacernos una idea más precisa de cómo se ejecuta el algoritmo, además de ser útil para ver lo paralelizable que puede ser una etapa. Por ejemplo si una función se llama muchas veces, una por cada píxel concretamente, esto sugiere que es posible que se pueda ejecutar en paralelo para varios píxeles. Sin embargo una función que solo se llama un vez, es posible que pueda ser puramente secuencial. De todas el número de llamadas no tiene porque ser siempre representativo, y es necesario un análisis mas calmado de cada caso particular.

- **Tiempo total de cómputo para cada función:** Dado el objetivo del proyecto resulta obvio realizar esta medida, pues queremos saber cuales funciones demoran más.

Número de llamadas a funciones.

Sistema operativo: Linux, Ubuntu 7.04 sist. mínimo
Compilador utilizado: gcc profiler
CPU: Pentium 4 Prescott con Hiper Threading 3.4 GHz, 1024 MB DDR-2
Tarjeta Gráfica: NVidia GeForce 7300 GT

Teniendo el algoritmo de JPEG implementado en la CPU, hacemos un estudio de las funciones que aparecen, Gráfica 5.1 1. Las características que estamos buscando son funciones que sean llamadas muchas veces dentro de la ejecución del algoritmo, y que sean altamente paralelizable en GPU. Entendemos como altamente paralelizable en GPU todo procedimiento, con cálculos uniformes, que contenga algún bucle cuyos datos calculados no dependan de datos obtenidos en el mismo bucle, o dando una definición un poco más informal, que invirtiendo los índices del bucle el resultado no cambia.

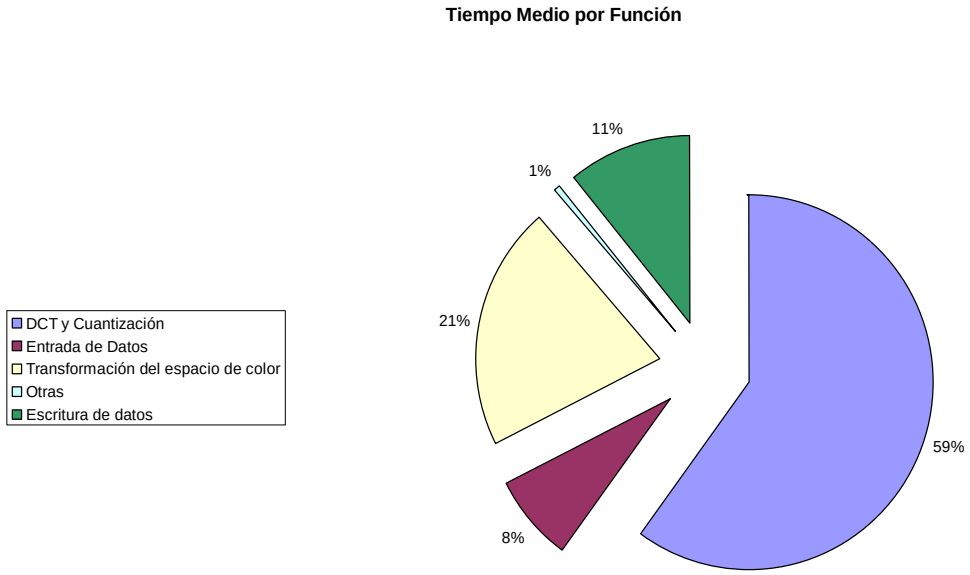


Gráfica 5.1 1 Número medio de llamadas a funciones JPEG

Número medio de llamadas que se hace a cada función en el algoritmo JPEG. (escala logarítmica).

En la Gráfica 5.1 1 las llamadas a las funciones están en el eje de ordenadas en escala logarítmica. Observamos que las llamadas de entrada salida del programa se deben hacer de forma secuencial, la DCT y la transformación de color se llama el mismo número de veces. El siguiente estudio que hacemos es el porcentaje de tiempo que utiliza cada función. No siempre una función que se llame muchas veces será la más costosa. Estos dos factores nos ayudarán a decidir la parte a paralelizar.

Tiempo total de cada función



Gráfica 5.1 2 Tiempo Medio por Función JPEG
Porcentaje del tiempo total empleado por cada función en el algoritmo JPEG

En el estudio del porcentaje de tiempo que utiliza cada función Gráfica 5.1 2, ,nos decidimos mejorar la función cuyo tiempo fuese mayor, la función DCT. La función DCT es la encargada de aplicar la transformada discreta del coseno, variante de la transformada de Fourier, y luego multiplicar el resultado por una matriz de cuantización.

Obtenemos un caso base, la implementación del algoritmo JPEG en CPU, que nos sirva como referente para sacar conclusiones con las optimizaciones que propondremos.

Tiempos de JPEG en CPU (caso base) en las distintas plataformas

Tenemos 3 plataformas diferentes, una de gama baja, gama media, gama alta.

Equipo César (Gama media)

Equipo Yelmo(Gama media)

Equipo K2 (Gama alta)

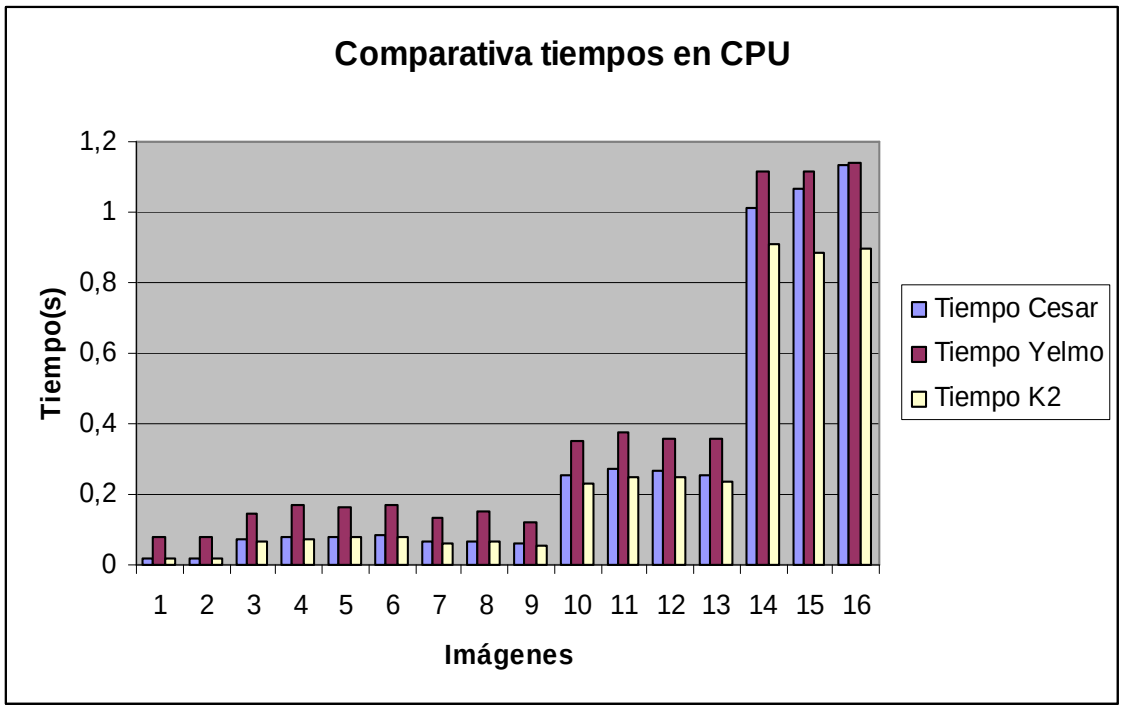
Observamos en la Tabla 5.1 2 que unos de los factores que más influyen en el tiempo es la dimensión de la imagen, por ejemplo las dos primeras imágenes tienen dimensión 256*256, frente a las 7 siguientes de 512.512. Al aumentar 4 veces el número de píxeles, se ha aumentado de media 7 veces el tiempo de ejecución.

	César	Yelmo	K2
Clock	0,0172	0,0769	0,0181
House	0,0197	0,0796	0,0208
Jelly beans	0,0704	0,1459	0,0666
Imagen4	0,0794	0,1711	0,0750
Water	0,0798	0,1641	0,0761
San Diego	0,0824	0,1668	0,0794
Imagen7	0,0645	0,1355	0,0609
Lena	0,0681	0,1509	0,0653
Couple	0,0590	0,1235	0,0539
Car and APCs	0,2532	0,3497	0,2280
Stockton	0,2711	0,3755	0,2478
Airplane	0,2695	0,3590	0,2485
LSW	0,2569	0,3545	0,2333
Pixel ruler	1,0146	1,1141	0,9071
LTP	1,0656	1,1169	0,8873
GTP	1,1339	1,13768	0,8970

Tabla 5.1 2 Tiempos (en segundos) del algoritmo JPEG en CPU

Comparativa tiempos en las distintas CPU

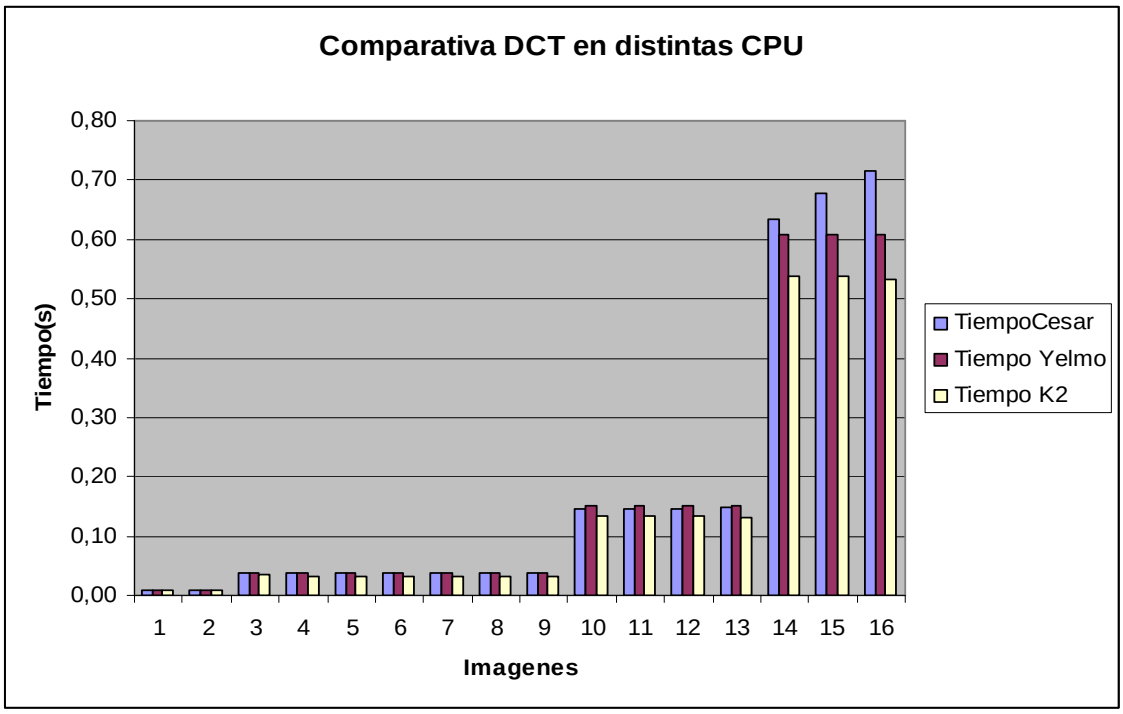
Con los resultados obtenidos podemos hacer un pequeño análisis, Gráfica 5.1 3, de las diferencias entre los distintos procesadores en el algoritmo JPEG.



Gráfica 5.1 3 Comparativa de Tiempos JPEG CPU en las 3 plataformas

Como podemos apreciar en la gráfica, los resultados obtenidos son los esperados, pues el equipo con el procesador más potente (el K2), aventaja a los otros dos en aproximadamente en un 20% del tiempo total. Los 2 megabytes de caché por procesador, con los que cuenta el Core 2 Duo del K2, sumados a la nueva tecnología de ejecución de instrucciones en desorden, representan el factor diferenciador en la comparativa.

A continuación observamos los tiempos de la función DCT , Gráfica 5.1 4, observamos que frente al tiempo total en cada una de las plataformas, Gráfica 5.1 3, es aproximadamente el 60 % que obtuvimos al hacer el estudio de la aplicación, Gráfica 5.1 2.



Gráfica 5.1 4 Comparativa de tiempos en las 3 CPUs

No es casualidad que la mayor parte de la ganancia que se consigue con el procesador del K2 se produzca en esta etapa, la DCT, pues es la más costosa de todo el JPEG, (ver Gráfica 5.1 2). Esta etapa trabaja sobre los píxeles de la imagen en bloques de 8x8, y para cada cálculo del coeficiente utiliza los 16 píxeles vecinos. Es aquí donde se nota la importancia de una caché más grande, pues los píxeles no tienen que ir a buscarse a memoria principal, ahorrando mucho tiempo.

A continuación expondremos diferentes optimizaciones para intentar mejorar los tiempos de ejecución finales. El estudio de cada una de ellas se compone de: una explicación de por qué decidimos esa optimización, los resultados, y análisis de los mismos.

5.1.2 Estudio temporal optimización 1: Implementación de la DCT(lenta) en GPU.

Como ya hemos visto en el apartado anterior, la DCT es una de las funciones con mayor porcentaje del tiempo total del algoritmo. Para esta primera implementación utilizamos el procesador de la tarjeta gráfica como coprocesador matemático para poder hacer la DCT de forma paralela. Esta primera implementación de la DCT en GPU ejecuta el algoritmo en todos los píxeles al mismo tiempo, lo que nos hace prever unos buenos resultados frente al algoritmo base, que se ejecutaba secuencialmente.

Antes de comenzar el análisis del tiempo total debemos hacer algunas consideraciones:

- El algoritmo JPEG se ejecuta sobre una sola imagen. Como consecuencia de esto, el flujo de entrada no es grande (una matriz de datos de ancho * alto). Esto implica que el tiempo de cálculo real del algoritmo es bastante pequeño sobre todo si consideramos el acceso a disco y memoria principal que utiliza la mayor parte del tiempo total.
- El algoritmo que hace uso de la GPU necesita inicializar el contexto OpenGL, compilar y cargar los shaders, y enviar la imagen a la tarjeta gráfica. Todo esto supone un tiempo de setup elevado.

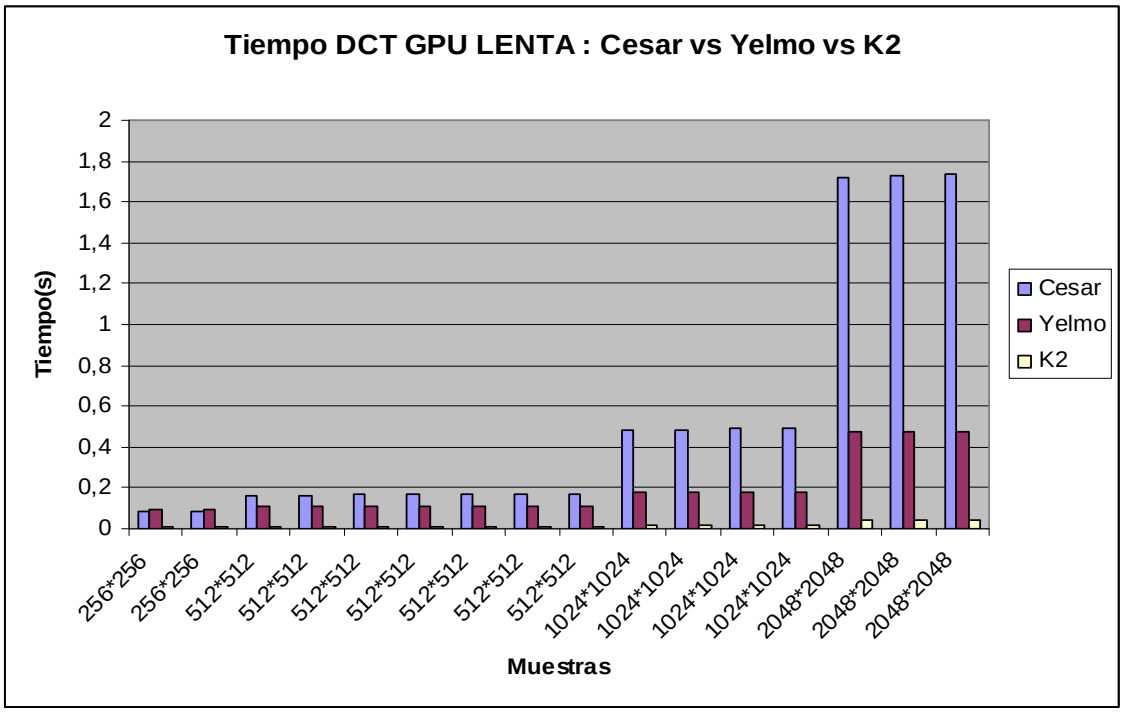
Dicho esto no esperamos que los tiempos totales sean mejores que los conseguidos en la CPU, aun así el estudio completo incluye los tiempos de la DCT, en los que ambos procesadores estarán en igualdad de condiciones.

Denotaremos en las tablas a esta optimización como GPU + CPU. Las plataformas donde mediremos estas pruebas son César ,Yelmo y K2.

	César	Yelmo	K2
Clock	0,0866	0,0891	0,0066
House	0,0864	0,0894	0,0066
Jelly beans	0,1633	0,1067	0,0084
Imagen4	0,1638	0,1071	0,0084
Water	0,1654	0,1071	0,0084
San Diego	0,1668	0,1072	0,0084
Imagen7	0,1680	0,1073	0,0084
Lena	0,1701	0,1073	0,0084
Couple	0,1726	0,1068	0,0085
Car and APCs	0,4803	0,1809	0,0157
Stockton	0,4841	0,1809	0,0158
Airplane	0,4875	0,1811	0,0158
LSW	0,4912	0,1806	0,0158
Pixel ruler	1,7207	0,4769	0,0451
LTP	1,7300	0,4765	0,0451
GTP	1,7382	0,4763	0,0451

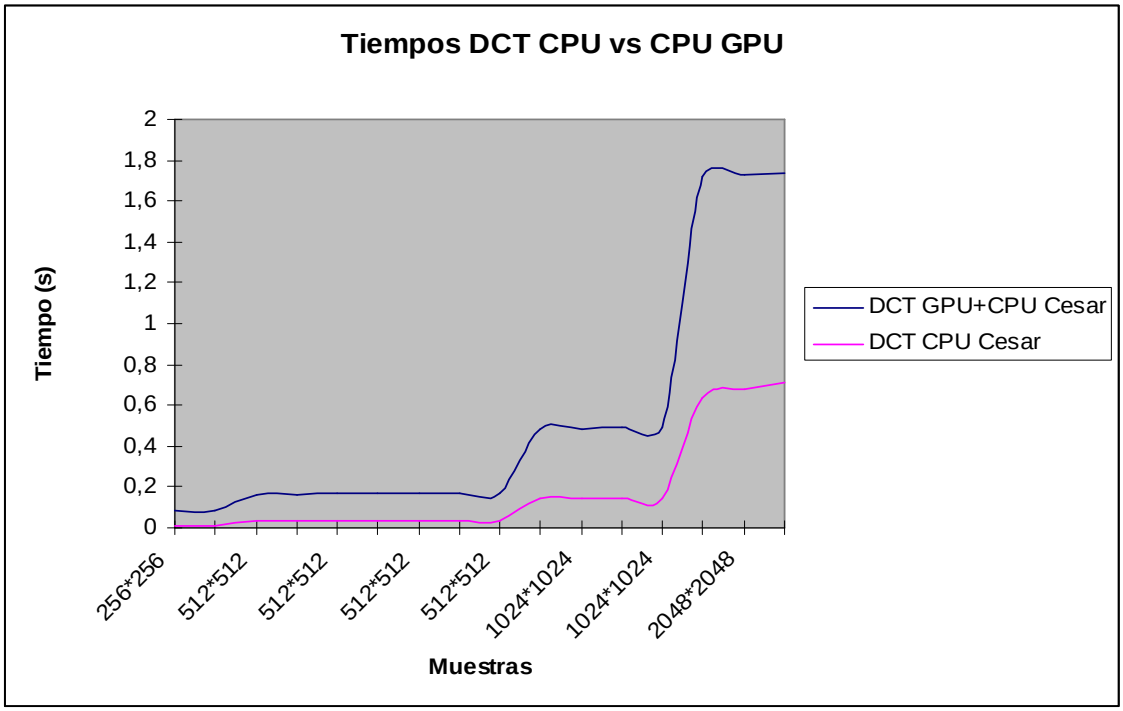
Tabla 5.1 3 Tiempos (segundos) función DCT Lenta en GPU sobre las 3 plataformas

Observamos el tiempo de la llamada a la función DCT respecto a las 3 plataformas, Gráfica 5.1 4. Los tiempos van decrementándose en k2 respecto a César y a Yelmo, ya que su tarjeta gráfica es más potente



Gráfica 5.1 5 Tiempo DCT lenta en JPEG en GPU

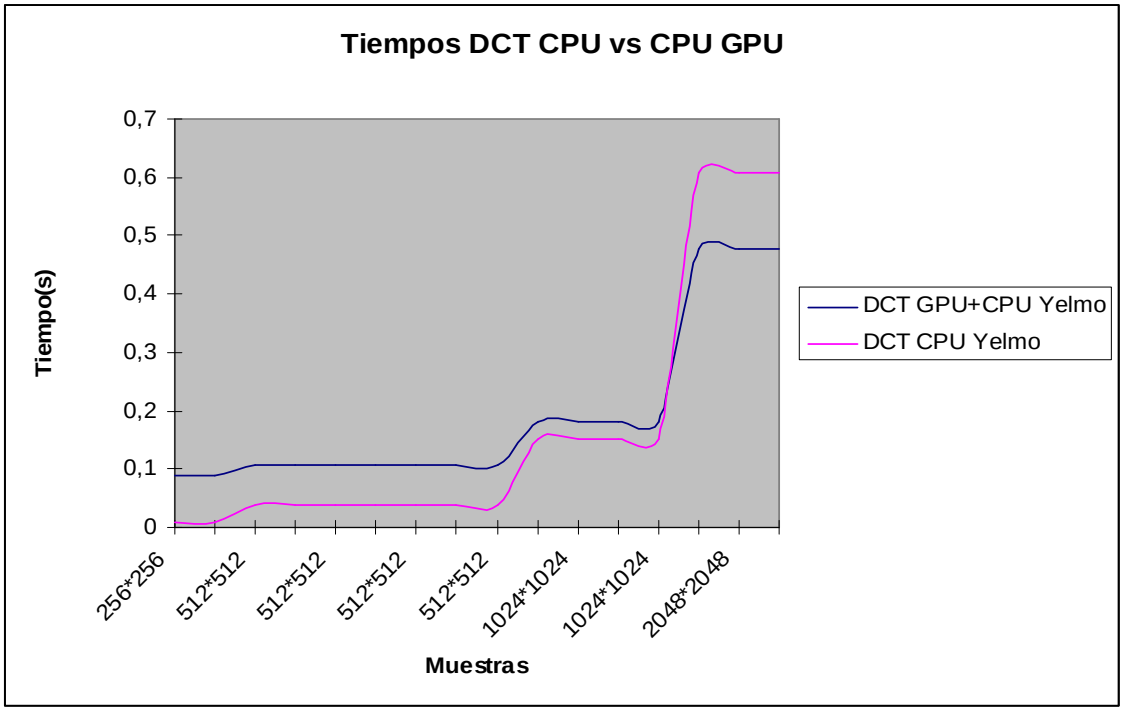
La comparación entre el tiempo de DCT con respecto al caso base :



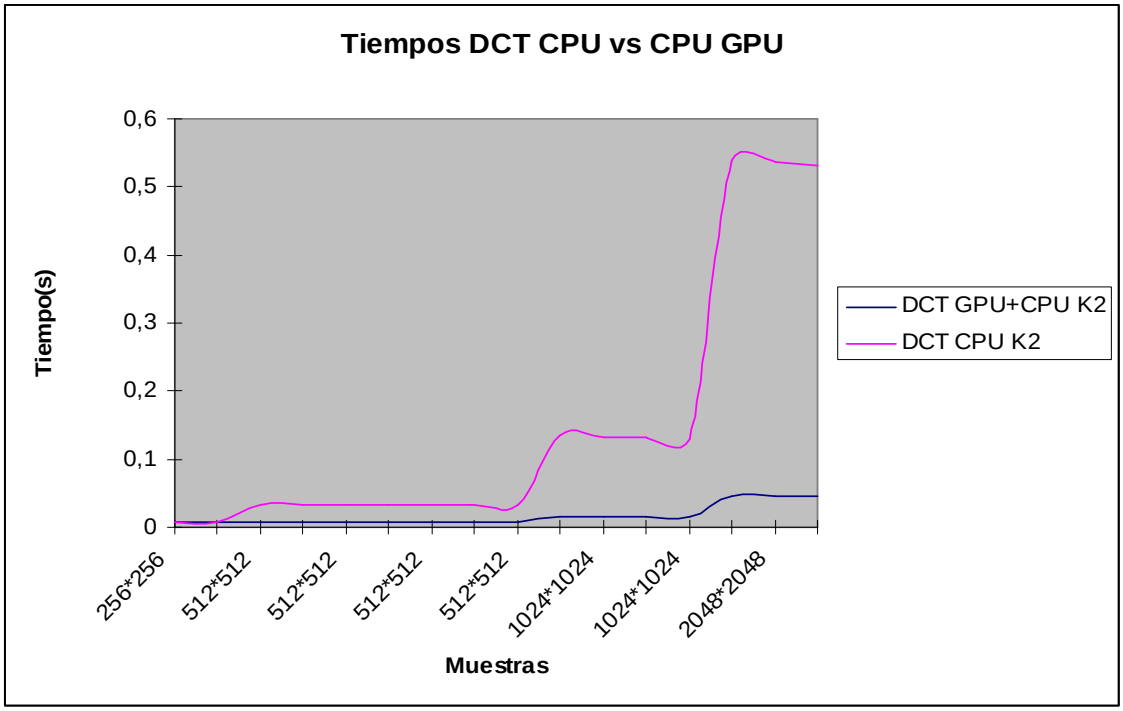
Gráfica 5.1 6 Tiempos DCT en algoritmo JPEG CPU César vs CPU+GPU César

En la Gráfica 5.1 6 observamos que los tiempos de DCT lenta en GPU son muchos mayores ya que la tarjeta gráfica no está optimizada para hacer muchos accesos a textura, ahora mismo en cada se hacen 64 accesos por píxel. Como veremos en la siguiente gráfica, Gráfica 5.1 7 , vemos que la nueva generación de tarjeta , la 7800 GTX da un mejor soporte a ese déficit de accesos, permitiéndonos tener ganancia para imágenes de dimensiones 2048*2048. Para imágenes pequeñas todavía tenemos un déficit de ganancia.

Los resultados obtenidos en el equipo K2, equipo más potente tanto en CPU como en GPU, Gráfica 5.1 8, observamos que el déficit observado en las texturas pequeñas de Yelmo, Gráfica 5.1 7, queda más que superado , y el tiempo de cómputo para imágenes grandes ha bajado considerablemente en varios órdenes de magnitud.



Gráfica 5.1 7 Tiempos DCT en algoritmo JPEG :CPU Yelmo vs CPU + GPU Yelmo



Gráfica 5.1 8 Tiempos DCT en algoritmo JPEG : CPU K2 vs CPU + GPU K2

Conclusión de esta primera optimización: Para hardware doméstico como el que tiene el equipo César, la DCT emplea tiempos enormes, que la convierten en una implementación inviable y poco

interesante, teniendo la posibilidad de implementar la DCT rápida. Sin embargo con GPU más potentes cambia totalmente el panorama, siendo el algoritmo de la DCT lenta el óptimo para utilizar.

La siguiente optimización que proponemos es la implementación del algoritmo de forma que haga menos accesos a la textura. El número de accesos a las texturas vienen condicionado por el hardware, y es una operación muy costosa. Con esta nueva implementación de la DCT que llamaremos DCT rápida intentamos mejorar los tiempos que hemos obtenido antes.

5.1.3 Estudio temporal optimización 2: Implementación sobre la GPU (DCT rápida)

Tras haber implementado la DCT en el apartado anterior de una forma que tiene 64 accesos por textura por píxel, pensamos que adaptando el algoritmo a que tenga menos accesos, conseguiremos un incremento de las prestaciones del programa. A lo largo de

En este apartado analizamos los tiempos obtenidos con la segunda optimización, en la que implementamos la DCT rápida en GPU. Siguiendo la línea de los estudios anteriores se plasman los tiempos totales del algoritmo, y los tiempos de la DCT rápida en GPU. Para poder analizar bien los resultados utilizamos los obtenidos en la CPU como referencia.

Todas las tablas se seguirá la nomenclatura anterior : CPU para denotar el algoritmo base, CPU + GPU para la implementación de DCT lenta en GPU , y añadimos CPU + GPU_R para la implementación de la DCT rápida en GPU.

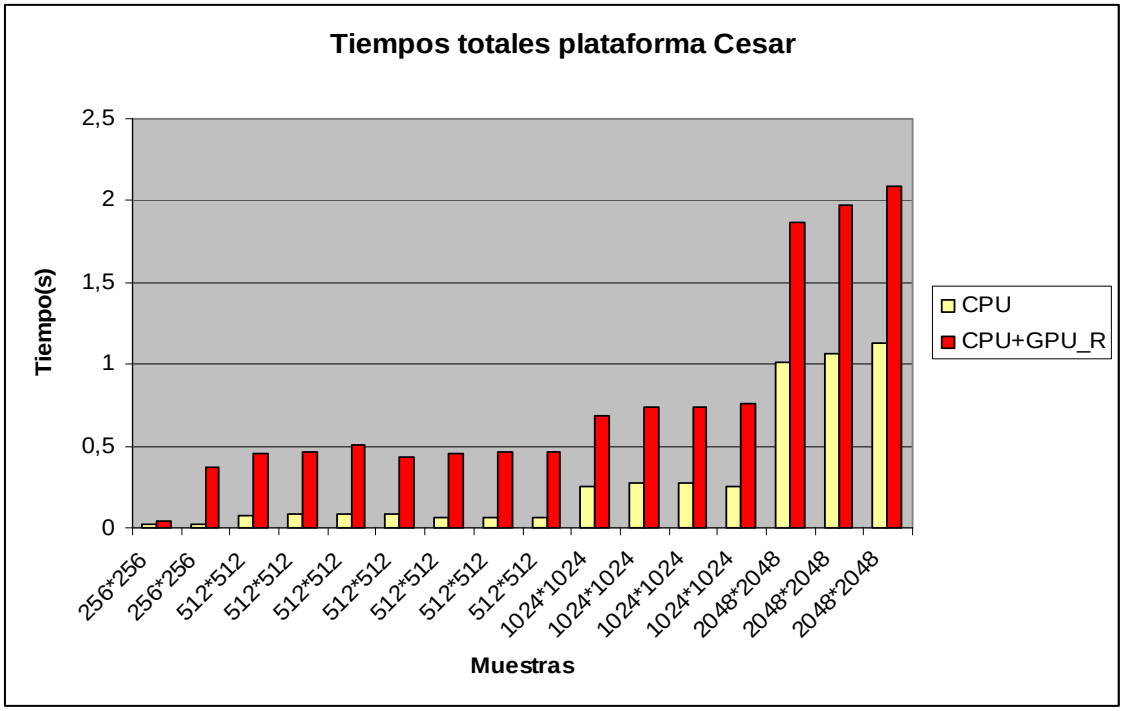
Tiempos totales

Todas las consideraciones del apartado anterior, con respecto a los tiempos totales, son igualmente válidas para este.

	César	Yelmo	K2
	CPU+GPU_R	CPU+GPU_R	CPU+GPU_R
Clock	0,0424	0,5499	0,0199
House	0,3686	0,5571	0,3152
Jelly beans	0,4546	0,6178	0,3432
Imagen4	0,4676	0,6354	0,3291
Water	0,5088	0,6331	0,3201
San Diego	0,4292	0,6506	0,3471
Imagen7	0,4488	0,6128	0,3587
Lena	0,4598	0,6111	0,3430
Couple	0,4696	0,6067	0,3085
Car and APCs	0,6852	0,8264	0,4462
Stockton	0,7419	0,8456	0,4875
Airplane	0,7411	0,8438	0,5225
LSW	0,7637	0,8392	0,4928
Pixel ruler	1,8645	1,6400	1,0281
LTP	1,9765	1,6759	1,0451
GTP	2,0922	1,6858	1,1204

Tabla 5.1 4 Tiempos Totales (segundos)JPEG: de la CPU GPU_R en las 3 plataformas

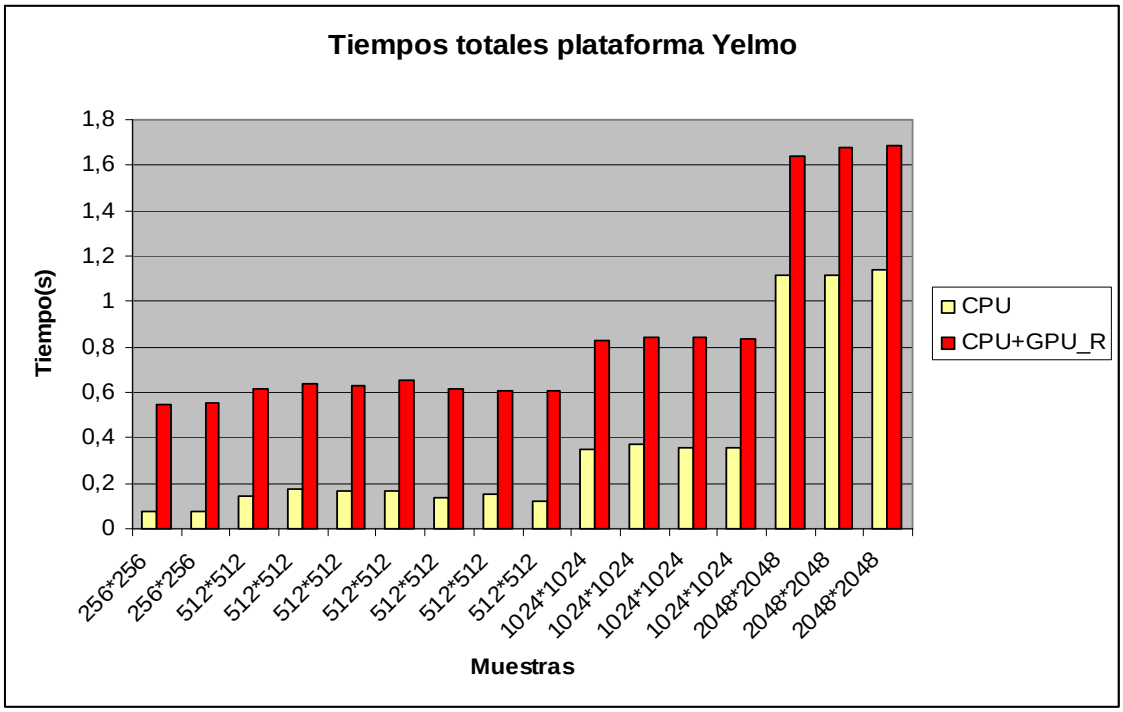
Observamos en la tabla de tiempos totales, Tabla 5.1 4, con el uso de microprocesador más rápido, y tarjeta gráfica más rápida conseguimos una mayor ganancia en K2 respecto a los dos anteriores. En este punto del estudio ya se empieza a apreciar la diferencia entre las dos tarjetas. Si bien el tiempo total en CPU es similar en ambos equipos (*César, GTP: 1.133s, Yelmo, GTP: 1.137s*), se puede ver como el tiempo total en GPU ha bajado significativamente (*César, GTP: 2.092s, Yelmo, GTP: 1.685s*). Y por lo tanto ya se empieza a vislumbrar el peso de las GPU en los tiempos totales.



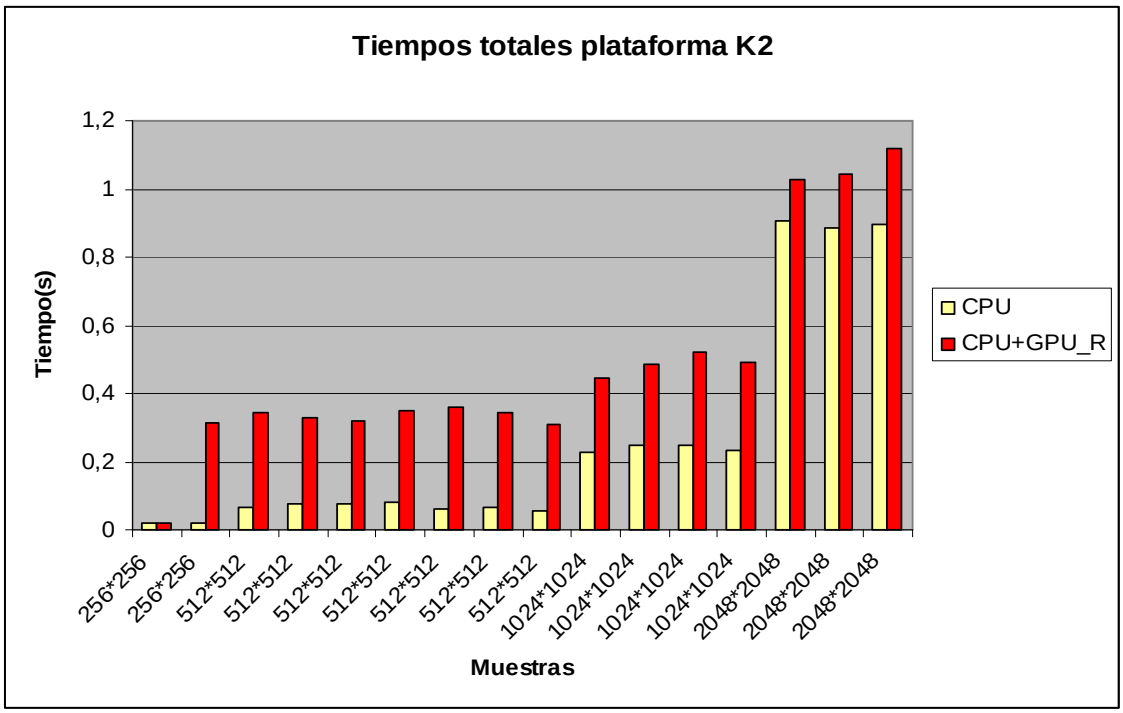
Gráfica 5.1 9 Comparativa de ejecución de algoritmo JPEG en CPU vs CPU+GPU_R en César

Observamos que aunque hemos obtenido ganancia respecto al algoritmo original en DCT, el tiempo de creación de contexto de OpenGL es muy caro.

Observamos que en Yelmo, Gráfica 5.1 10, se va consiguiendo reducir el tiempo entre la implementación en CPU, y el algoritmo Rápido. La ejecución en K2 , Gráfica 5.1 11, también muestra lamisca tendencia que en Yelmo.



Gráfica 5.1 10 Comparativa de ejecución de algoritmo JPEG en CPU vs CPU+GPU_R en Yelmo



Gráfica 5.1 11 Comparativa de ejecución de algoritmo JPEG en CPU vs CPU+GPU_R en K2

Tiempos DCT

La medida de tiempos de la etapa DCT, es donde la CPU y la GPU se compararán más directamente. Ahora no se tiene en cuenta más que el tiempo empleado para el cálculo de los coeficientes DCT en ambos procesadores.

Equipo César

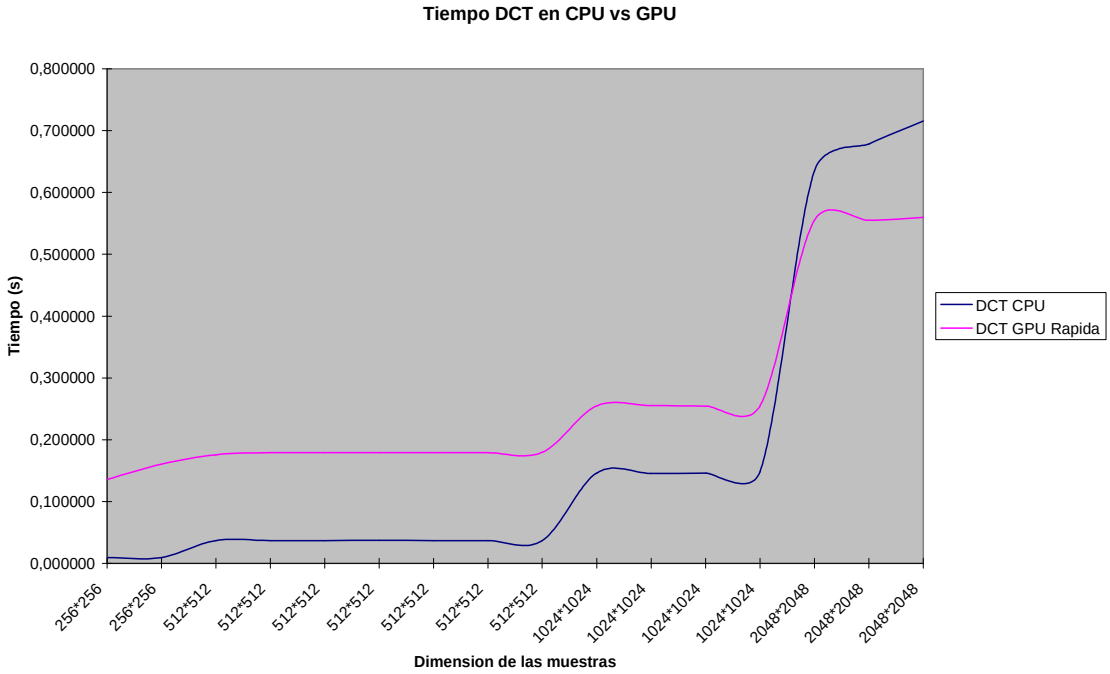
	César	Yelmo	K2
Clock	0,13570	0,2556	0,0405
House	0,16026	0,2557	0,0405
Jelly beans	0,17560	0,2583	0,0430
Imagen4	0,17888	0,2599	0,0429
Water	0,17885	0,2602	0,0415
San Diego	0,17918	0,2605	0,0415
Imagen7	0,17912	0,2604	0,0416
Lena	0,17882	0,2602	0,0416
Couple	0,17946	0,2600	0,0415
Car and APCs	0,25458	0,2752	0,0455
Stockton	0,25505	0,2780	0,0459
Airplane	0,25441	0,2763	0,0460
LSW	0,25458	0,2769	0,0455
Pixel ruler	0,55489	0,3451	0,0617
LTP	0,55488	0,3470	0,0613
GTP	0,55947	0,3452	0,0617

Tabla 5.1 5 Tiempos (segundos) DCT CPU y GPU_R en las 3 plataformas

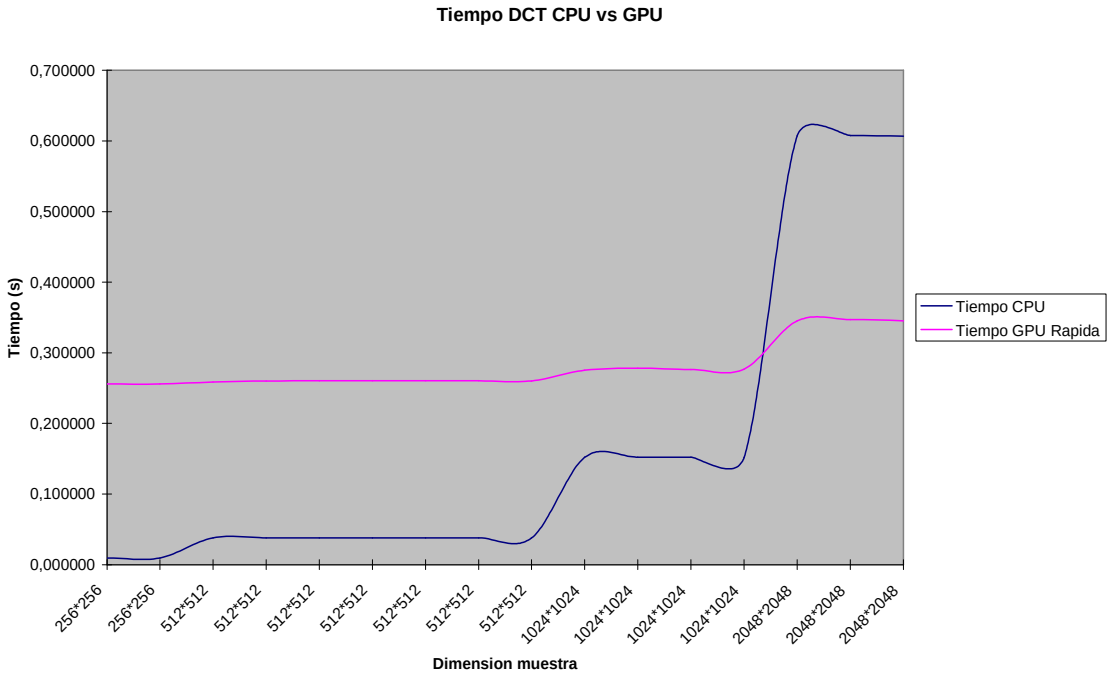
La etapa DCT ha disminuido su tiempo considerablemente,Tabla 5.1 5, llama la atención la potencia de las nueva generación de GPU, la NVidia 8800 GTX , que está dentro de la plataforma K2, que es capaz de reducir en varios órdenes de magnitud el resultado de la gama de tarjetas anterior (7800 GTX Yelmo).

Observamos en el primer equipo ,Gráfica 5.1 12 , dos cosas interesantes, por una lado se ve que la versión en CPU es la más rápida para la mayoría de las dimensiones de imagen. Pero por el otro vemos como la GPU para imágenes grandes (2048x2048) supera a la CPU. El análisis que hacemos de la situación es que para imágenes pequeñas la CPU puede aprovechar mejor la memoria caché, pero cuando se trata de imágenes grandes con varios megabytes de píxeles, aflora el paralelismo implícito en la GPU.

Hemos de recordar que estamos trabajando con una tarjeta de uso doméstico, luego no eran de esperar resultados muy espectaculares.

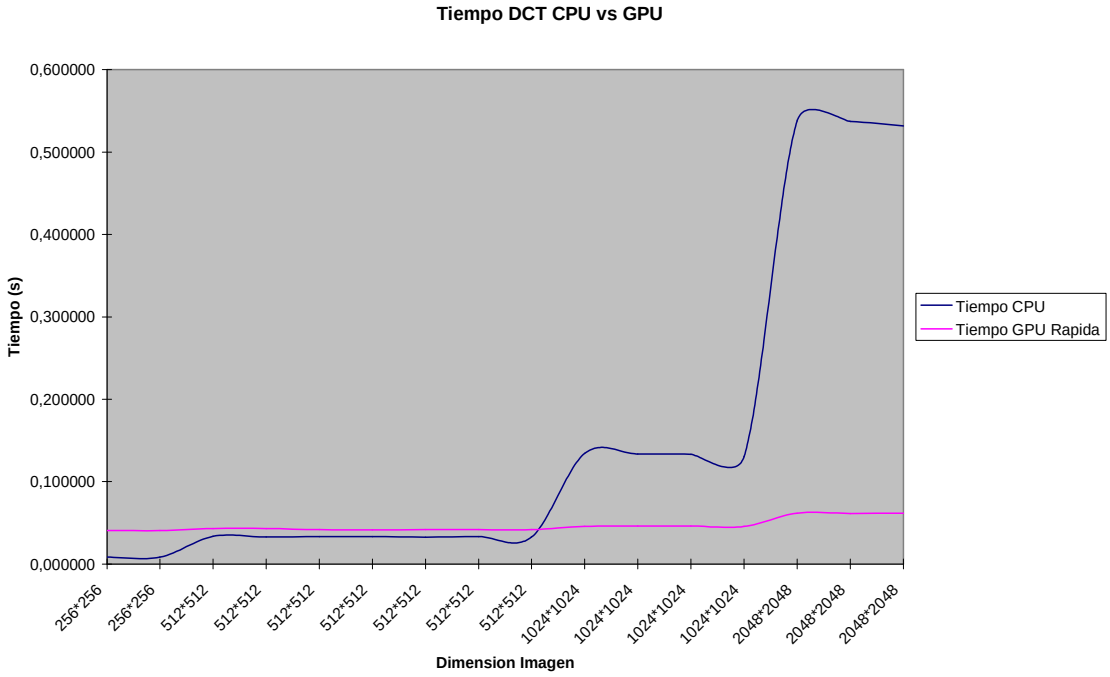


Gráfica 5.1 12 Comparativa tiempos DCT : CPU vs CPU+GPU_R en César



Gráfica 5.1 13 Comparativa tiempos DCT : CPU vs CPU+ GPU_R en Yelmo

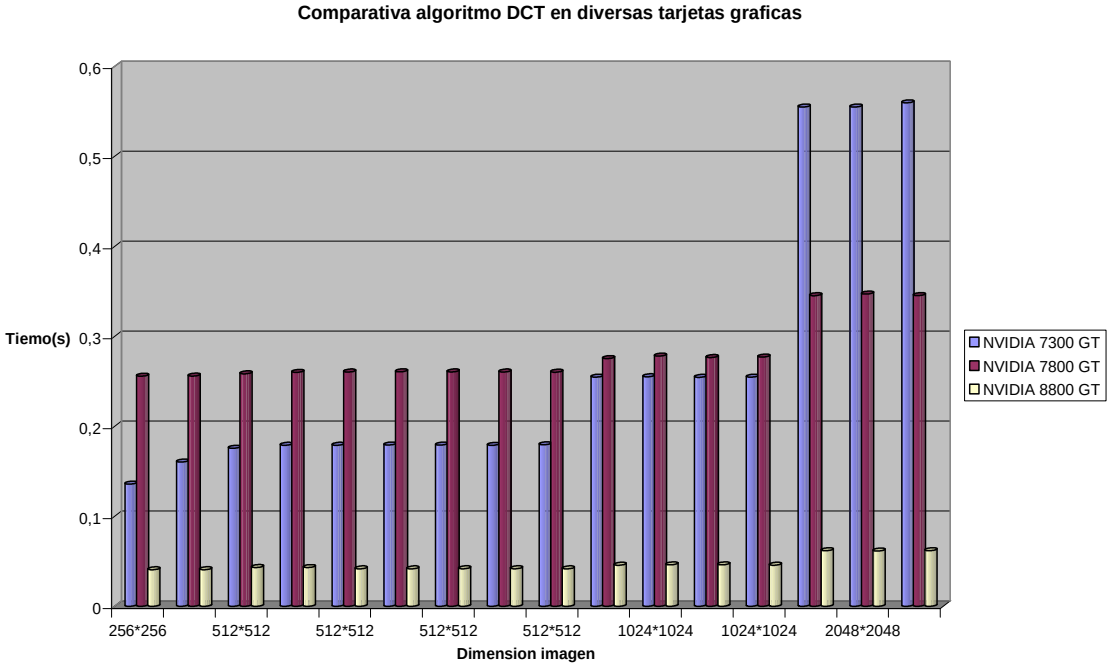
Tras esta simulación, Gráfica 5.1 13, observamos que el tiempo de la GPU es casi constante para todos los tamaños, solo aumenta un poco para texturas de 2048 x 2048. La interpretación que hacemos de esto es que esta tarjeta tiene muchas unidades paralelas, que no llegan a utilizarse todas hasta que se carga una imagen lo suficientemente grande.



Gráfica 5.1 14: Comparativa tiempos DCT: CPU vs CPU+ GPU_R en K2

Las pruebas en la ultima tarjeta, Gráfica 5.1 14, la más potente de las tres, sirven para terminar de corroborar lo que ya íbamos viendo a lo largo del estudio, y es que la GPU es mucho más rápida que la CPU en el cálculo de la DCT. Se observa con claridad que este algoritmo exprime toda la potencia paralela de los procesadores gráficos, dejando atrás a la CPU como procesador secuencial.

Como ya hemos ido comentando a lo largo de estas pruebas, el modelo de tarjeta es determinante en el tiempo empleado para la DCT, esta gráfica sirve termina de ilustrar este hecho , Gráfica 5.1 15.



Gráfica 5.1 15 Comparativa DCT en CPU+GPU_R en las 3 plataformas

En este momento hemos conseguido una ganancia alta con la función DCT, la siguiente optimización que nos planteamos en el algoritmo es el consumo de CPU. Planteamos una optimización con dos hilos diferentes, repartiendo en cada uno de ellos la mitad de la carga de la imagen, uno de los hilos se ejecutará en CPU frente al otro que lo hará en GPU.

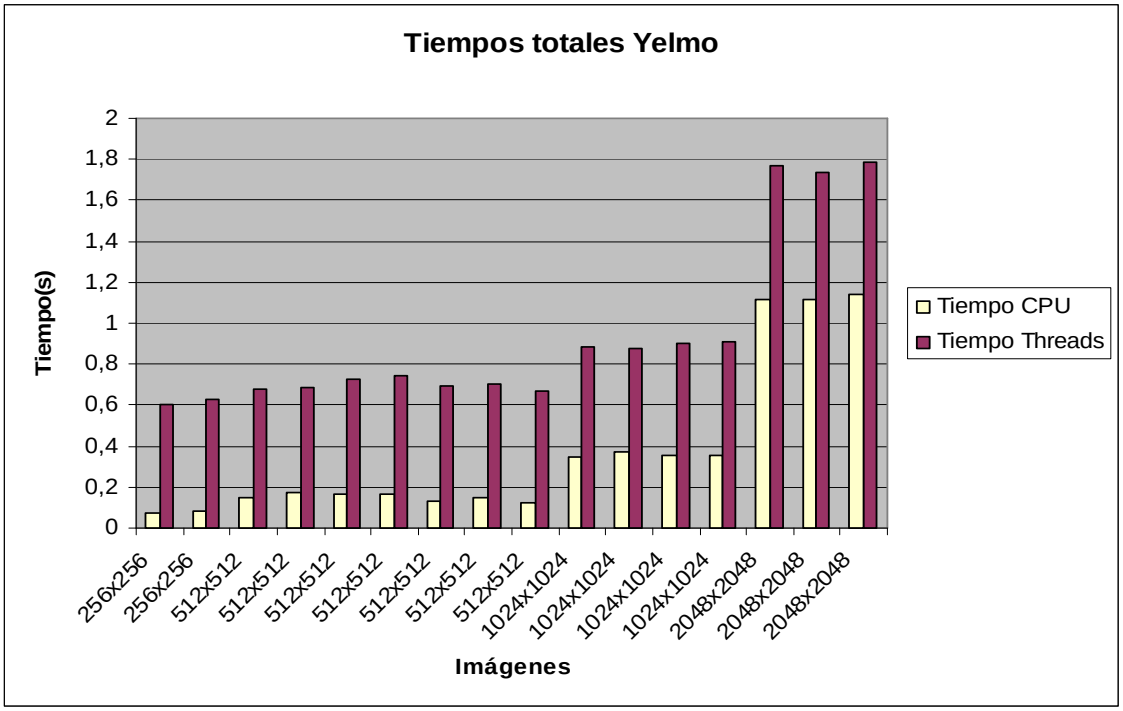
5.1.4 Estudio temporal optimización 3: Implementación con threads (CPU-GPU)

Esta tercera implementación del JPEG, hace uso de los dos procesadores, la CPU y la GPU en paralelo. Queremos ver si el tiempo total conseguido mejora, y si el paralelismo existente en la CPU se hace visible. Esta optimización solo lo probamos sobre Yelmo y K2 por sus arquitecturas.

	Yelmo	K2
Clock	0,6058	0,2530
House	0,6247	0,2901
Jelly beans	0,6808	0,3462
Muestra4	0,6891	0,3378
Water	0,7247	0,3149
San Diego	0,7455	0,3001
Muestra7	0,6965	0,3317
Lena	0,7037	0,3303
Couple	0,6672	0,2805
Car and APCs	0,8804	0,4249
Stockton	0,8780	0,4604
Airplane	0,8985	0,4762
LSW	0,9094	0,4513
Pixel ruler	1,7722	0,9471
LTP	1,7314	0,9580
GTP	1,7810	0,9414

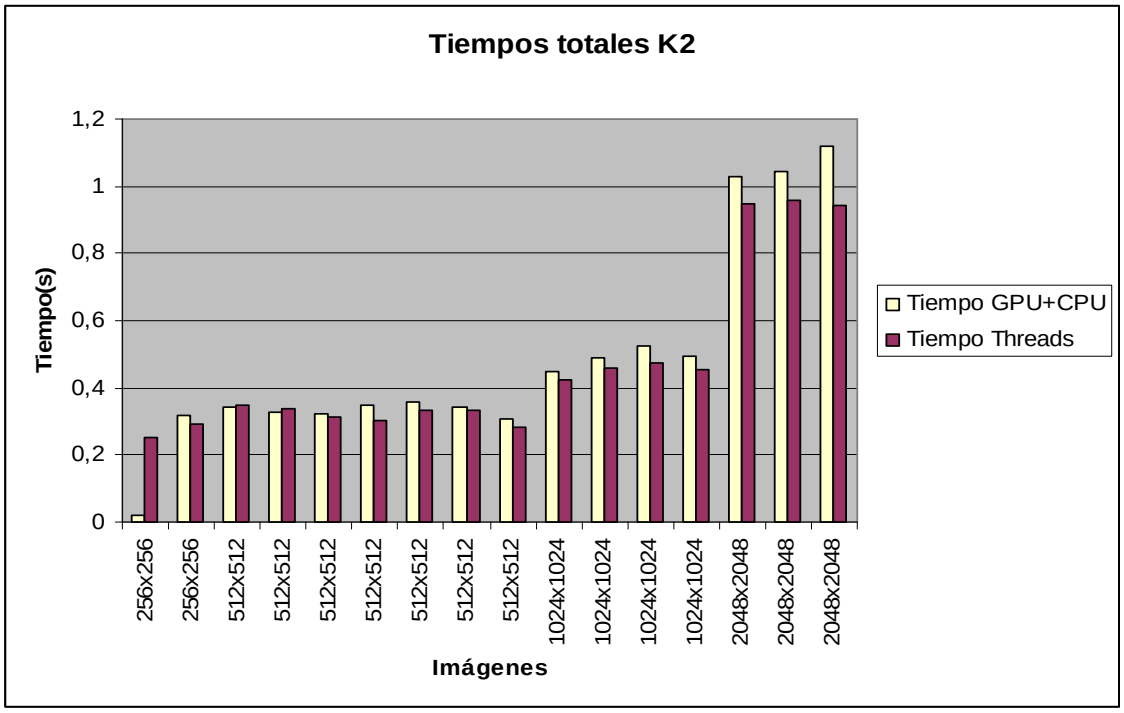
Tabla 5.1 6 Tiempos totales (segundos) JPEG: CPU GPU Thread en las 2 plataformas

Como se puede apreciar en la tabla los datos de la optimación 3,Tabla 5.1 6, sobre el equipo Yelmo, no mejoran a los obtenidos en la implementación en la que se ejecutaba la DCT rápida sobre la GPU con un solo thread, Gráfica 5.1 16. El motivo de esto es que Yelmo no cuenta con un procesador paralelo, lo que impide que los dos threads se ejecuten simultáneamente. Esto no quiere decir que la parte que se ejecuta sobre la GPU no lo haga en paralelo con la CPU. Pero no debemos olvidar que el thread de GPU consume cierto tiempo de procesador en la carga y preparación de los programas y texturas.



Gráfica 5.1 16 Tiempos totales JPEG : CPU vs CPU+GPU Threads en Yelmo

Observamos como los tiempos totales de esta tercera implementación se acerca mucho a los obtenidos por el algoritmo de referencia sobre la CPU. Luego podemos concluir que esta es la mejor de las tres optimizaciones si el equipo del que se dispone tiene un procesador paralelo.



Gráfica 5.1 17 Comparativa tiempos totales optimitzacio2 (CPU+GPU_R) con optimización 3 (CPU+GPU THreads)

En el caso del K2 la situación cambia totalmente con respecto a la obtenida en Yelmo, Gráfica 5.1 17 , esto se debe principalmente al procesador que usa el K2, que dispone de un paralelismo implícito mucho mayor. Así se consigue que ambos threads se ejecuten simultáneamente, y por tanto el tiempo se reduzca en gran medida.

5.1.5 Conclusiones estudio JPEG

Una vez concluido el estudio de las distintas implementaciones y optimizaciones del algoritmo JPEG, llegamos a las siguientes conclusiones:

Algoritmo general JPEG: De las tres optimizaciones experimentadas ninguna ha conseguido mejorar el tiempo total, esto no nos sorprende, pues como ya dijimos el tamaño de los datos de entrada de este algoritmo es relativamente pequeño, haciendo que el tiempo total este muy influenciado por operaciones ajenas al propio algoritmo como son la lectura de disco y memoria principal.

Debido a que las tres optimizaciones hacen uso de la GPU, necesita un tiempo de inicialización y carga del contexto significativo, el tiempo total que emplean se ve abultado. Otra cosa que influye negativamente en el tiempo total es la transferencia de los datos a la GPU a través del bus, que aunque actualmente ha mejorado mucho todavía sigue siendo significativamente costosa. Aun suponiendo que el tiempo de transferencia fuera igual al que emplea la CPU con la memoria principal, la GPU todavía tendría que esperar a que fueran leídos en la memoria principal desde disco, mientras la CPU ya podría comenzar a procesar estos datos.

Cálculo de la DCT: Las conclusiones que sacamos del estudio de este algoritmo difieren bastante de las anteriores, pues partimos de una situación en la que los datos ya se encuentran cargados en los procesadores y listos para ser procesados. En este contexto la potencia del procesador se vuelve decisiva como hemos podido comprobar. Además las distintas implementaciones de la DCT de las que disponemos: DCT directa o lenta, y FDCT o DCT rápida, difieren mucho en su comportamiento según el procesador en el que se ejecuten.

La DCT rápida es la que mejor se comporta en CPU, pues trabaja solo sobre 16 píxeles, mientras que la DCT lenta lo hace sobre los 64 de un bloque.

Para la GPU el comportamiento temporal de la DCT rápida mejora notablemente según utilizamos una tarjeta más potente. Además hay que decir que el tiempo en GPU supera varias veces al empleado por la CPU. La máxima mejora conseguida

$$speedup_{DCTrapida} = \frac{T_{CPU}}{T_{GPU}} = \frac{0{,}531811}{0{,}061732} = 8{,}7$$

Este valor termina de demostrar la superioridad de la GPU en la ejecución del algoritmo de la DCT.

Para la DCT lenta el hardware gráfico empleado, es todavía más decisivo, pues hemos visto que en César y Yelmo el tiempo es bastante peor que la implementación rápida. Sin embargo, en K2 sorprende la velocidad con la que se ejecuta la DCT lenta, superando a la implementación de la DCT rápida.

$$speedup_{\frac{DCTrapida}{DCTlenta}} = \frac{T_{DCTrapida}}{T_{DCTlenta}} = \frac{0{,}061732}{0{,}045133} = 1{,}3$$

Estos resultados sobre la DCT lenta, describen muy bien la situación que puede darse cuando estamos programando sobre la GPU, en la que algoritmos desechados para ser ejecutados sobre la CPU resultan ser más rápidos cuando se ejecutan sobre procesadores paralelos como el de la tarjeta gráfica.

$$speedup_{DCTlenta} = \frac{T_{CPU}}{T_{GPU}} = \frac{0{,}531811}{0{,}045133} = 11{,}78$$

5.2 MPEG

5.2.1 Estudio del algoritmo MPEG en CPU.

Sistema operativo: Linux, Ubuntu 7.04 sist. completo para el codec

Compilador utilizado: gcc

CPU: Pentium 4 Prescott con Hiper Thread 3.4 GHz 1024 MB DDR-2

Tarjeta Gráfica : NVidia GeForce 7300 GT

Con objeto de centrar nuestra línea de trabajo en el MPEG realizamos un estudio detallado de la implementación del algoritmo sobre la CPU. Estructuraremos este apartado en las siguientes pruebas:

- Estudio del algoritmo MPEG en CPU.
- Estudio de implementación MPEG con fdct en GPU
- Estudio de implementación MPEG paralelizado

Antes de nada, vamos a comentar los videos y las características, empleados para las pruebas. Hemos usado ocho videos distintos,Tabla 5.2 1, cinco son videos caseros con los que hemos tratado de cubrir toda la gama de posibilidades que ofrece el MPEG2, y los tres restantes se corresponden a videos tipo (Benchmark).

	Dimensiones del frame	# frames
ppm1	320*240	69
ppm2	320*240	3000
ppm3	640x480	150
ppm4	640x480	150
ppm5	352x240	50
ppm6	352x240	34
ppm7	352x240	34
ppm8	352x240	34

Tabla 5.2 1 Muestras de Video para las pruebas del algoritmo MPEG

Los cinco primeros corresponden a los videos caseros que hemos utilizado como pruebas. Vamos a analizarlos un poco más en detalle:

1. ppm1: Video en blanco y negro. Cámara fija. Con cambios de plano radicales. Con movimiento por todo el frame.
2. ppm2: Video en blanco y negro. Cámara fija. Con muchos cambios de plano suaves. Movimiento por todo el frame. El motivo de incluir esta muestra es ver el comportamiento de los algoritmos a largo plazo. Porque como ya vimos en el JPEG, inicializar el contexto openGL y preparar la GPU es bastante costoso y no queremos que los resultados para los videos "cortos" se vean ensombrecidos por este acarreo de coste que no es más que un espejismo si consideramos un video real.
3. ppm3: Video en color. Cámara fija. Sin cambios de plano. Movimiento suave por todo el frame.
4. ppm4: Video en color. Cámara en movimiento. Sin cambios de plano. Movimiento constante y brusco.
5. ppm5: Video en color. Cámara fija. Sin cambios de plano. Movimiento escaso y bien localizado en el centro del video.

Mientras que los tres restantes corresponden a los videos Benchmark, de los cuales se adjuntas las referencias

6. ppm6:"Claire"

   Evaluation of robust interframe MPEG video watermarking
   Moradi, S.; Gazor, S. Electrical and Computer Engineering, 2005. Canadian Conference on Volume , Issue , 1-4 May 2005 Page(s): 1923 - 1926
   Digital Object Identifier 10.1109/CCECE.2005.1557358
   Summary: Frame type changing is a very common attack on watermarked video signal. Re-synchronization and making watermark more ro

7. ppm7:"football"

   Motion adaptive error resilient encoding for MPEG-4
   Worrall, S.T.; Sadka, A.H.; Sweeney, P.; Kondoz, A.M. Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP apos;01). 2001 IEEE International Conference on Volume 3, Issue , 2001 Page(s):1389 - 1392 vol.3
   Digital Object Identifier 10.1109/ICASSP.2001.941188

   Summaryptimising delivery of video codecs such as MPEGall.

8. ppm8:"Foreman"

   Linear rate-distortion models for MPEG-4 shape coding
   Zhenzhong Chen; King Ngi Ngan
   Circuits and Systems for Video Technology, IEEE Transactions on Volume 14, Issue 6, June 2004 Page(s): 869 - 873
   Digital Object Identifier 10.1109/TCSVT.2004.828331

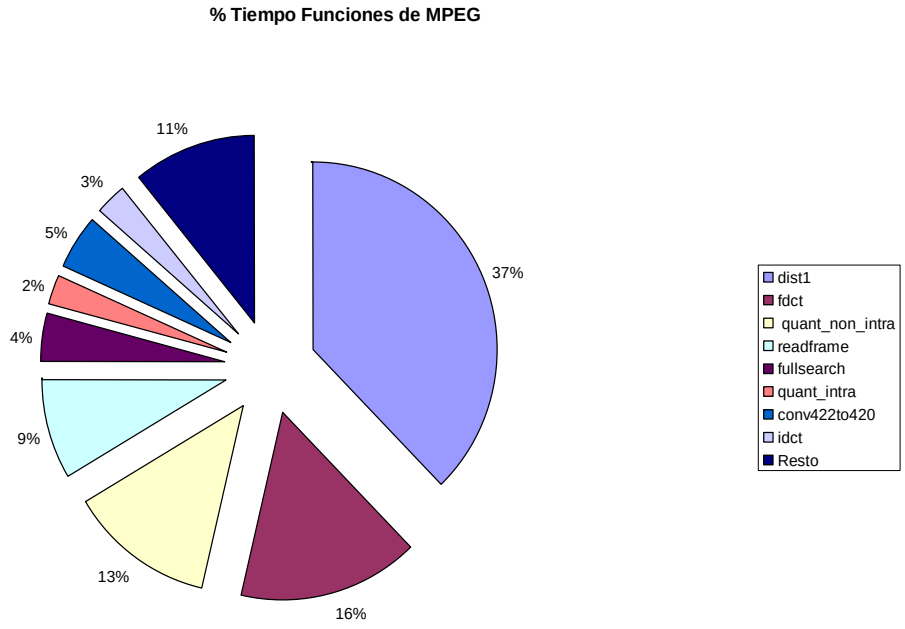
   Summary: In this letter, we explore the rate-distortion (R-D) characteristics of binary shape information in the MPEG-4 standard. The shape coding sc

La secuencia de tipos de frame que utilizaremos para cada GOP será: I B B P B B. El motivo de elegir esta configuración es que, en la vida cotidiana, todos los videos que se codifican usando MPEG2 siguen esta configuración. Quizá en otros ámbitos donde se desee mantener una calidad muy alta, la secuencia contenga mas frames tipo P y I; pero este caso no lo contemplamos por alejarse del propósito de este trabajo.

A continuación hacemos el estudio detallado del algoritmo MPEG, viendo el número de llamadas que se hace a cada función, y determinando el tiempo medio de cada función.

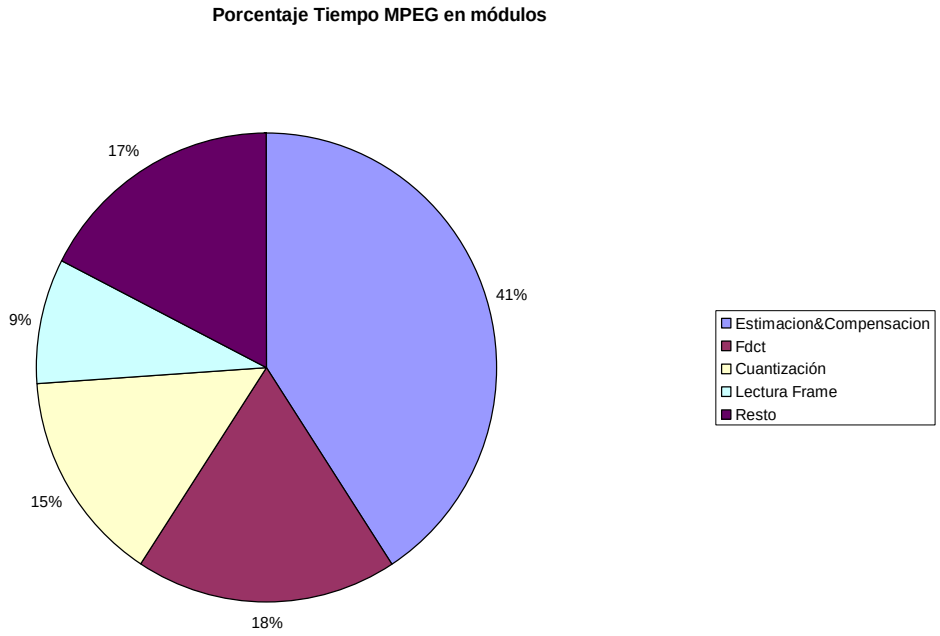
Estudio del algoritmo MPEG en CPU.

Hacemos un estudio detallado del tiempo que tarda cada función, para contar con datos que nos guíen en la búsqueda de optimizaciones en la GPU, Gráfica 5.2 1.



Gráfica 5.2 1 Porcentaje de tiempo que tarda cada función en el algoritmo MPEG

Englobamos las funciones anteriores en los 5 grupos que hemos separado y nos queda el siguiente gráfico, Gráfica 5.2 2. Podemos comprobar que el bloque de Motion es el que más tarda , seguido de la FDCT y de la cuantización. En esta primera lectura de los datos vamos a intentar paralelizar los bloques de Fdct y Cuantización mediante el uso de la GPU como coprocesador matemático en coma flotante.



Gráfica 5.2 2 Porcentaje de tiempo del algoritmo MPEG dividido en módulos

La medida de tiempos del algoritmo MPEG marca los módulos que hay que intentar optimizar con ayuda de la GPU, a continuación describimos en orden decreciente de costes la viabilidad de pasarlos a la GPU:

- **Estimación y compensación de movimiento:** Este módulo está compuesto por las dos funciones dist1 y fullsearch, es paralelizable en la GPU y estudiaremos la posibilidad de implementarlo.
- **Fdct:** Encargado de hacer la transformada discreta del coseno, es otro módulo paralelizable en GPU. Englobamos las dos funciones fdct, y ifdct. La ifdct cabe destacar que es mucho menos costosa porque solo se aplica a los frames I.
- **Cuantización:** Engloba los métodos cuant_non_intra, y cuant_intra. Por el momento esos métodos quedarán en la CPU.
- **Lectura Frame:** Engloba la lectura de los frames sin comprimir, guardados en disco como imágenes bmp. Debido a que ahora disponemos de dos procesadores, es posible repartir el algoritmo en distintos hilos de ejecución , para asegurar que ambos procesadores no estén desaprovechados. Una posible línea de investigación puede ser hacer que el hilo de la CPU vaya leyendo frames de disco, mientras en la GPU se realizan las otras etapas. De

esta forma los frames se iría colocando en un buffer del que se alimentarían los módulos de la GPU.

El módulo de compensación de movimiento y la DCT tardan, entre los dos, el 60% del tiempo total. Debido a esto, son los módulos que debemos tratar de optimizar.

En primer lugar, vamos a obtener un análisis temporal inicial referente a nuestra batería de prueba en los distintos equipos. Con esto pretendemos hacernos una idea del rendimiento que puede ofrecernos cada equipo, ya que ejecutamos sobre ellos la misma implementación del algoritmo. Indicamos mediante color azul el algoritmo que mejor tiempo da para cierta muestra, de color rojo el algoritmo que peor tiempo da y de color negro aquel que se mantiene entre medias. Mediante estos colores pretendemos hacer ver de forma fácil, cual es el algoritmo que ofrece mejores resultados para cada máquina.

Tomaremos como datos de interés los tiempos totales y los porcentajes del motion, predict FDCT y putpic, que corresponden con los módulos de estimación de movimiento, predicción de frames (que aunque, en comparación con el resto, tarde poco, es una etapa muy importante a tener en cuenta), la FDCT con la peculiaridad que se calcula en GPU en toda las implementaciones salvo en la básica, y el putpic, que es la función encargada de escribir en disco.

Los datos que mostramos a continuación son las medidas del algoritmo de MPEG en CPU. Estas medidas, todas en segundos se componen de 2 partes diferenciadas, la primera tabla de cada configuración son los porcentajes de tiempo que esa función ha utilizado, Tabla 5.2 2,Tabla 5.2 4,Tabla 5.2 6 , mientras que la segunda tabla que aparece a continuación es el tiempo total de cada muestra, Tabla 5.2 3,Tabla 5.2 5,Tabla 5.2 7.

Recordamos que las muestras han sido tomadas 10 veces, descartando el caso mejor y peor, y haciendo la media de resto.

Este estudio nos servirá para tomar unos valores base con los que comparar el resto de optimizaciones.

Equipo César

	motion	predict	FDCT	putpic
ppm1	53,0521	1,3220	24,4431	15,8524
ppm2	55,5108	0,8983	16,9531	8,9136
ppm3	54,6900	1,2764	24,9409	13,8328
ppm4	66,6489	0,9896	18,6109	10,0546
ppm5	46,0138	1,3794	29,0860	16,4210
Claire	49,3846	1,2887	27,5526	15,6069
Football	61,3443	1,0539	20,1728	13,1745
Foreman	65,5739	0,9047	18,4515	10,9067

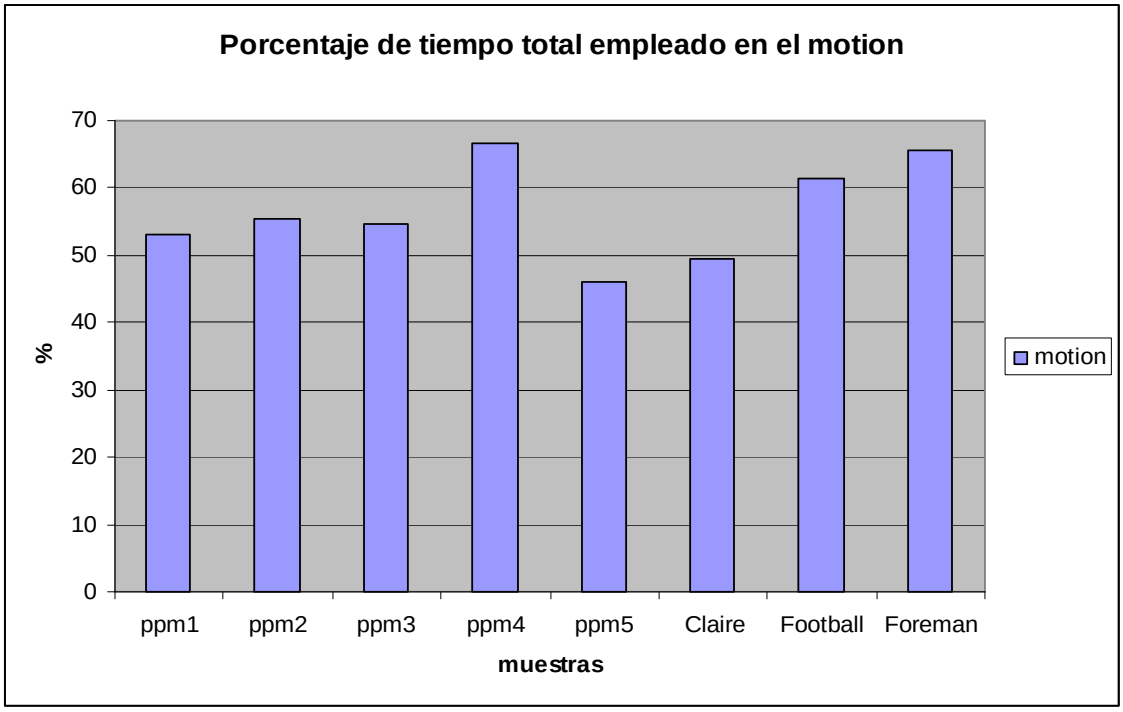
Tabla 5.2 2 Porcentaje de tiempo total consumido en cada muestra /función.

	Tiempo total (segundos)
ppm1	6,9776
ppm2	616,2385
ppm3	128,8615
ppm4	130,0541
ppm5	13,5933
Claire	4,8618
football	4,5061
Foreman	5,5380

Tabla 5.2 3 Tiempos medios totales(segundos), algoritmo CPU en configuración César

De estos datos, Tabla 5.2 2, Tabla 5.2 3, cabe destacar que el porcentaje del motion se mantiene en torno al 52%-57% salvo para las muestras ppm4 y "Foreman" que asciendo hasta el 66%; y muestra ppm5, en la que desciende hasta el 46% (Gráfica 5.2 3).

Esto es interesante, ya que como se comentó en las descripciones de los videos, ppm4 y "Foreman" concretamente tienen un índice de movimiento de cámara muy elevado; mientras que ppm5 es un video a cámara fija y con poco movimiento. Con esto nos hemos dado cuenta de que el coste del motion depende del tipo de video que se procese.



Gráfica 5.2 3 Porcentaje de tiempo del algoritmo MPEG empleado en hacer el motion sobre César.

Podemos conjeturar que, ya que nuestras optimizaciones mejoran el cálculo de la DCT. Para videos que tengan alto índice de motion, la ganancia que obtendremos con nuestras mejoras será menor.

Equipo Yelmo

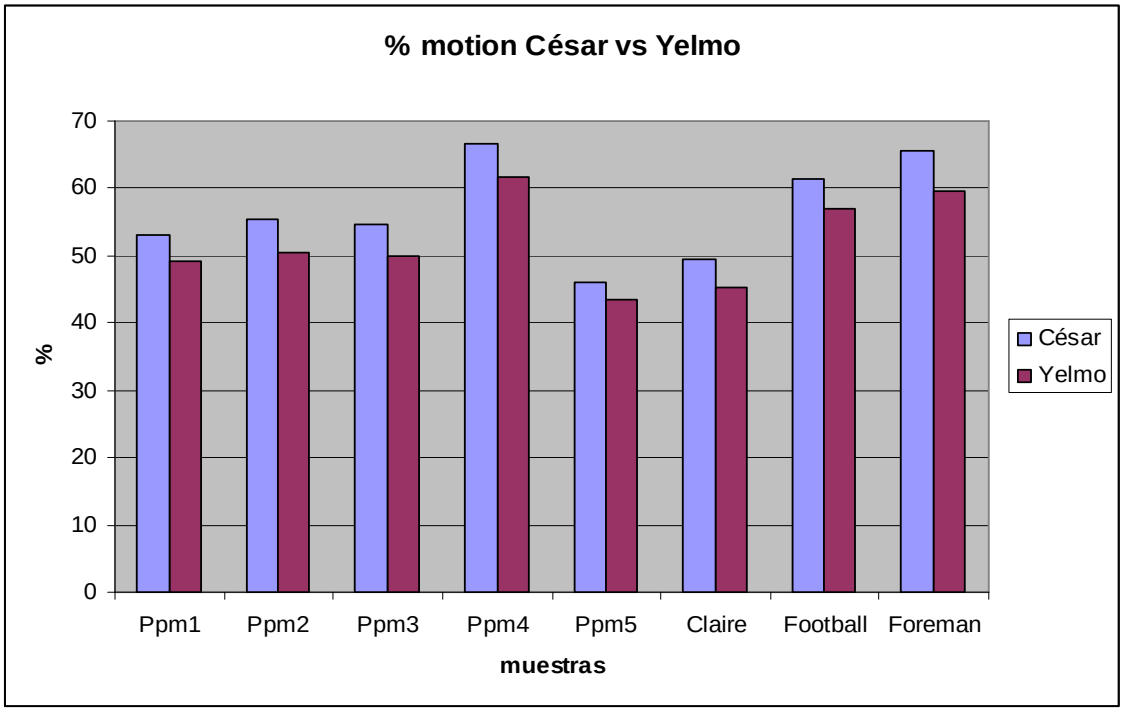
	motion	predict	FDCT	putpic
Ppm1	49,0621	1,2672	24,8825	18,0504
Ppm2	50,4827	1,3101	25,6145	16,7049
Ppm3	49,8767	1,3534	25,3071	17,0206
Ppm4	61,7442	1,0762	19,2520	13,3067
Ppm5	43,5526	1,3614	28,7657	18,6275
Claire	45,2047	1,2437	27,5533	18,7741
Football	57,1126	1,0814	20,9691	15,3491
Foreman	59,7199	1,0082	19,6537	14,4802

Tabla 5.2 4 Porcentaje de tiempo total consumido en cada muestra /función.

	Tiempo total (segundos)
ppm1	7,4931
ppm2	267,8013
ppm3	54,9043
ppm4	58,3803
ppm5	9,1513
Claire	3,6501
football	3,7810
Foreman	3,8873

Tabla 5.2 5 Tiempos medios totales (segundos), algoritmo CPU en configuración César

En primer lugar, queremos destacar la diferencia de tiempos que se obtiene frente a César ,Gráfica 5.2 1,para unos volúmenes de datos elevados. Por otra parte el porcentaje que se dedica al motion es considerablemente menor (del orden de un 5%).



Gráfica 5.2 4 Porcentaje de tiempo del algoritmo MPEG empleado en hacer el motion sobre César y Yelmo

Si tenemos en cuenta que el motion es la etapa que más accesos a memoria realiza, podemos explicar la diferencia de tiempos totales entre César y Yelmo como la influencia de la caché.

Equipo K2

	motion	predict	FDCT	putpic
Ppm1	56,9474	1,4505	24,5209	10,7212
Ppm2	56,4823	1,4671	25,0190	10,5455
Ppm3	57,1698	1,8645	24,6093	9,93786
Ppm4	66,8331	1,2774	19,0573	7,8417
Ppm5	52,1017	1,5483	28,2272	10,8615
Claire	52,2745	1,4701	28,3218	10,5933
football	65,4673	1,2090	19,6648	8,5073
Foreman	68,1981	1,0982	18,0664	7,9858

Tabla 5.2 6 Porcentaje de tiempo total consumido en cada muestra /función.

Tabla 5.2 7 Tiempos medios totales (segundos), algoritmo CPU en configuración César

	Tiempo total (segundos)
ppm1	5,9221
ppm2	232,6096
ppm3	48,3018
ppm4	48,5486
ppm5	7,6535
Claire	3,1370
football	3,3398
Foreman	3,4703

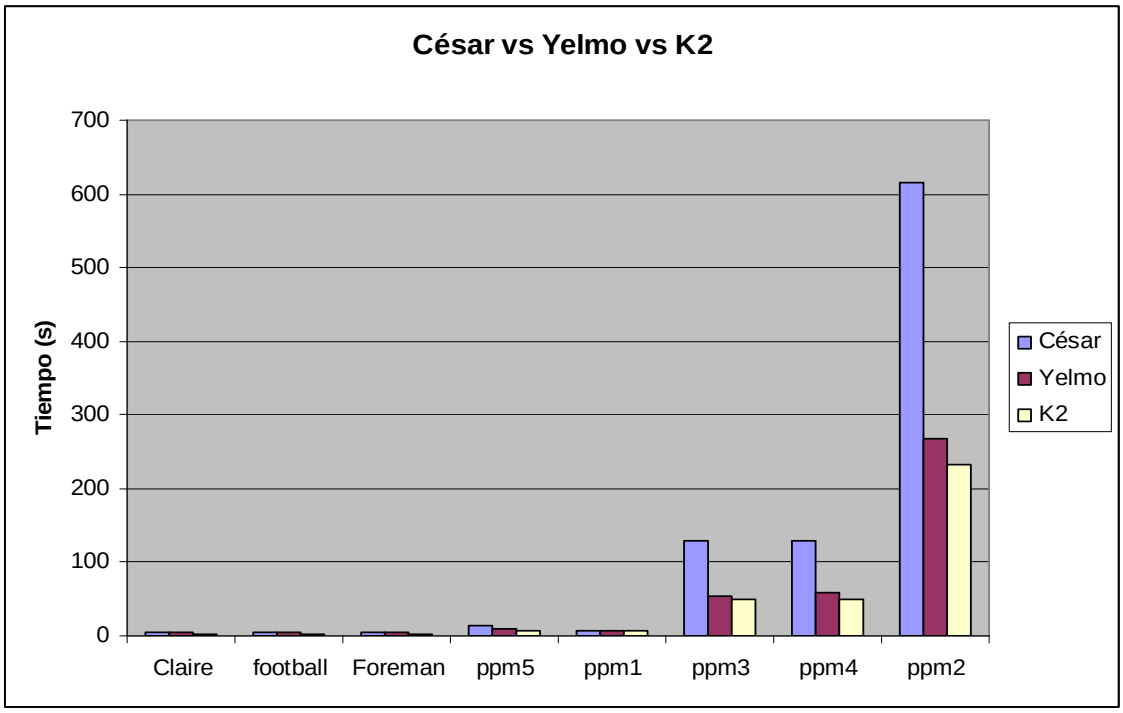
Tomamos estos datos sobre las tres máquinas como datos básicos que resultan de aplicar el algoritmo sin optimizaciones. Usando estos valores como referencia pretendemos medir los SpeedUp provocados por las optimizaciones.

Sin embargo, también puede ser interesante hacer un estudio del mismo algoritmo sobre los distintos equipos. En cuanto a esto, no tenemos ninguna duda que los resultados serán favorables al K2 frente a Yelmo y a Yelmo frente a César, pero creemos interesante analizar la situación para ver cuan mejor se comportan los equipos.

Para hacer las comparaciones de tiempos totales, ordenamos la batería de pruebas de forma creciente respecto el volumen de datos,Tabla 5.2 8, Gráfica 5.2 5,.

	César	Yelmo	K2
Claire	4,8618	3,6501	3,1370
football	4,5061	3,7810	3,3398
Foreman	5,5380	3,8873	3,4703
ppm5	13,5933	9,1513	7,6535
ppm1	6,9776	7,4931	5,9221
ppm3	128,8615	54,9043	48,3018
ppm4	130,0541	58,3803	48,5486
ppm2	616,2385	267,8013	232,6096

Tabla 5.2 8 Tabla con los tiempos totales (segundos) del algoritmo MPEG en CPU en las 3 configuraciones ordenadas por número de frames.



Gráfica 5.2 5 Gráfica de tiempos totales en CPU de las tres configuraciones . Algoritmo MPEG

Aclaramos que en esta gráfica, Gráfica 5.2 5, en realidad la comparación es únicamente entre procesadores, ya que no se utiliza ni GPU ni otras optimizaciones. Nos sorprende ver los resultados obtenidos sobre César en comparación con Yelmo, sobretodo teniendo en cuenta que los procesadores son prácticamente iguales. Para las muestras que tienen un volumen de datos pequeño, los resultados son muy parecidos, siendo Yelmo siempre un poco más rápido. Sin embargo cuando el volumen de datos aumenta, los resultados de César se degradan obteniendo unos tiempos totales que duplican los de Yelmo. Esta situación se debe al efecto de la caché. César cuenta con una caché de 1Mb mientras que Yelmo con 2Mb, siendo la velocidad de los procesadores la misma. Esta

diferencia de caché provoca que cuando el volumen de datos es grande, César tenga un mayor tiempo de acceso a memoria, retrasando así el tiempo total.

Tras este estudio realizamos la evaluación de la primera mejora, el uso de la GPU como coprocesador matemático para realizar cálculos en paralelo a la CPU.

5.1.2 Estudio temporal Optimización 1: Implementación sobre la GPU(DCT rápida)

Como ya se explicó en la parte correspondiente de este trabajo, la primera optimización que hemos realizado es incluir el cálculo de la DCT en el procesador gráfico, para comprobar si tenemos ganancia vamos a ejecutar las mismas pruebas que hicimos con el algoritmo básico sobre los tres equipos. Los resultados que aquí se expresan están tomados como en el apartado anterior, incluyendo dos tablas por equipo, una que corresponde al porcentaje de tiempo que ha utilizado esa función , y la segunda al tiempo de ejecución para esa muestra.

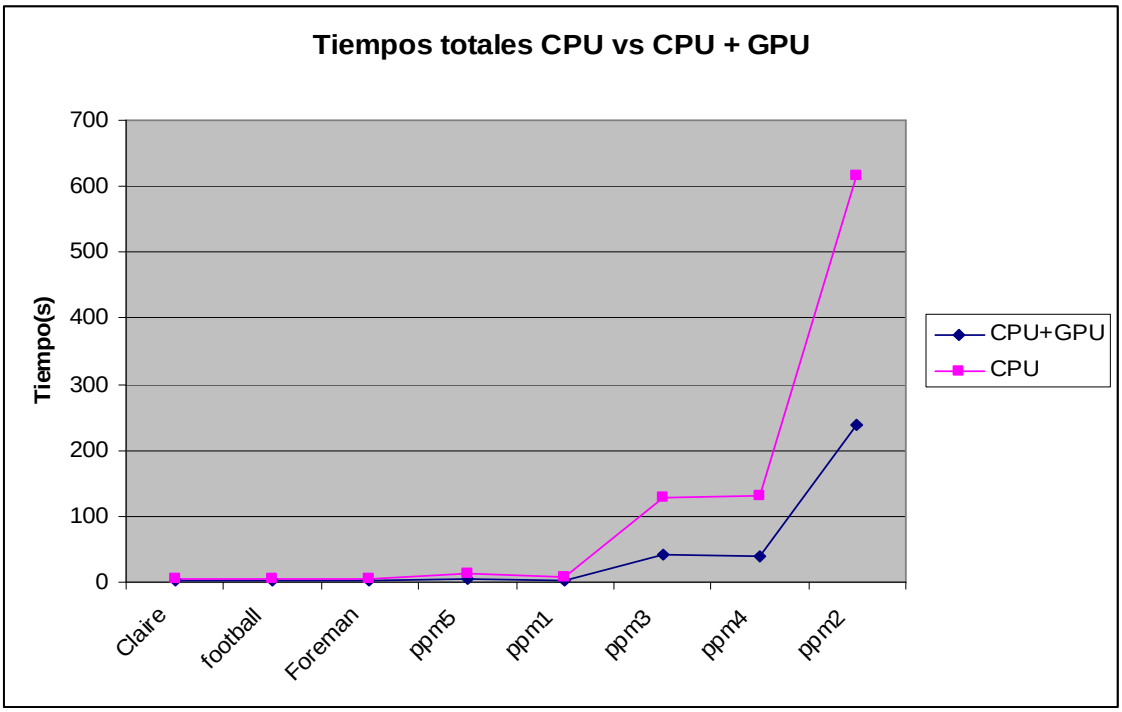
Equipo César

	motion	predict	FDCT	putpic
ppm1	35,0987	0,9889	45,6025	14,3148
ppm2	35,3617	1,3400	45,5861	12,6262
ppm3	43,4765	1,0409	38,5943	11,8242
ppm4	52,6475	0,9111	32,4329	10,1910
ppm5	38,3617	1,3400	42,5861	12,6262
Claire	29,3835	1,0341	49,9655	15,2080
Football	53,8421	0,8768	29,1269	11,3968
Foreman	47,6240	0,8144	36,9424	11,3424

Tabla 5.2 9 Porcentaje de tiempo utilizado en el MPEG primera optimización, plataforma César.

	Tiempo total
ppm1	3,1686
ppm2	238,7836
ppm3	40,9446
ppm4	39,8911
ppm5	5,4490
Claire	1,9153
football	2,2106
Foreman	2,3271

Tabla 5.2 10 Tiempo total (segundos) de ejecución de la primera optimización, en la plataforma César.

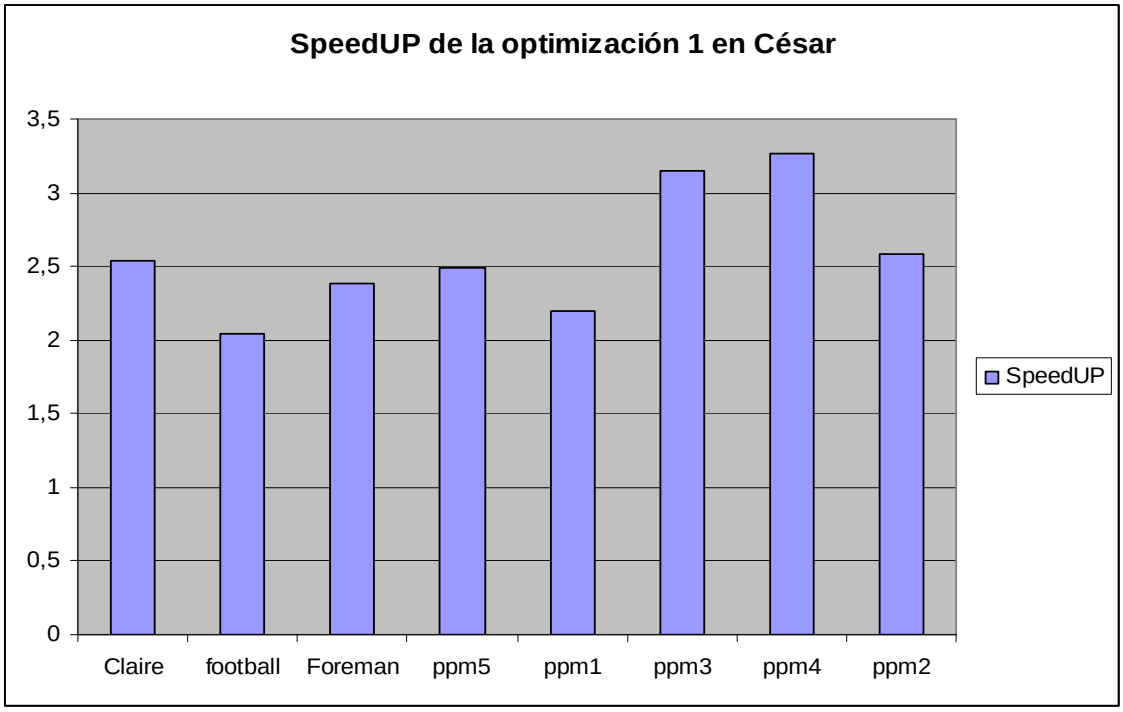


Gráfica 5.2 6 Comparativa totales algoritmo básico vs optimización1 (DCT en GPU) en César

En esta gráfica ,Gráfica 5.2 6, comprobamos el efecto de la ejecución en paralelo de la DCT en GPU. Tanto para videos pequeños como Claire o football, como para videos grandes tales como ppm4 y ppm2 conseguimos una speedUp medio de 2,5, Gráfica 5.2 7.

Equipo Yelmo

A continuación se muestran los resultados obtenidos sobre la máquina yelmo. Como primera impresión, podemos observar que el tiempo de ejecución es notablemente menor en todas las muestras. Esto era de esperar ya que el procesador gráfico de Yelmo esta catalogado como medio-alto y ofrece un rendimiento mejor que la CPU.



Gráfica 5.2 7 Representación del SpeedUp obtenido en César con la primera optimización.

	motion	predict	FDCT	putpic
ppm1	24,1390	0,7264	58,6217	12,8595
ppm2	14,9231	0,6133	70,1842	11,0969
ppm3	25,2822	0,8228	56,8246	13,2031
ppm4	34,1918	0,7604	49,9680	11,6716
ppm5	17,2049	0,7543	64,5794	13,4555
Claire	18,8312	0,6363	64,4228	12,3412
Football	30,1866	0,6540	54,8667	11,1034
Foreman	33,2840	0,6170	52,3637	10,6669

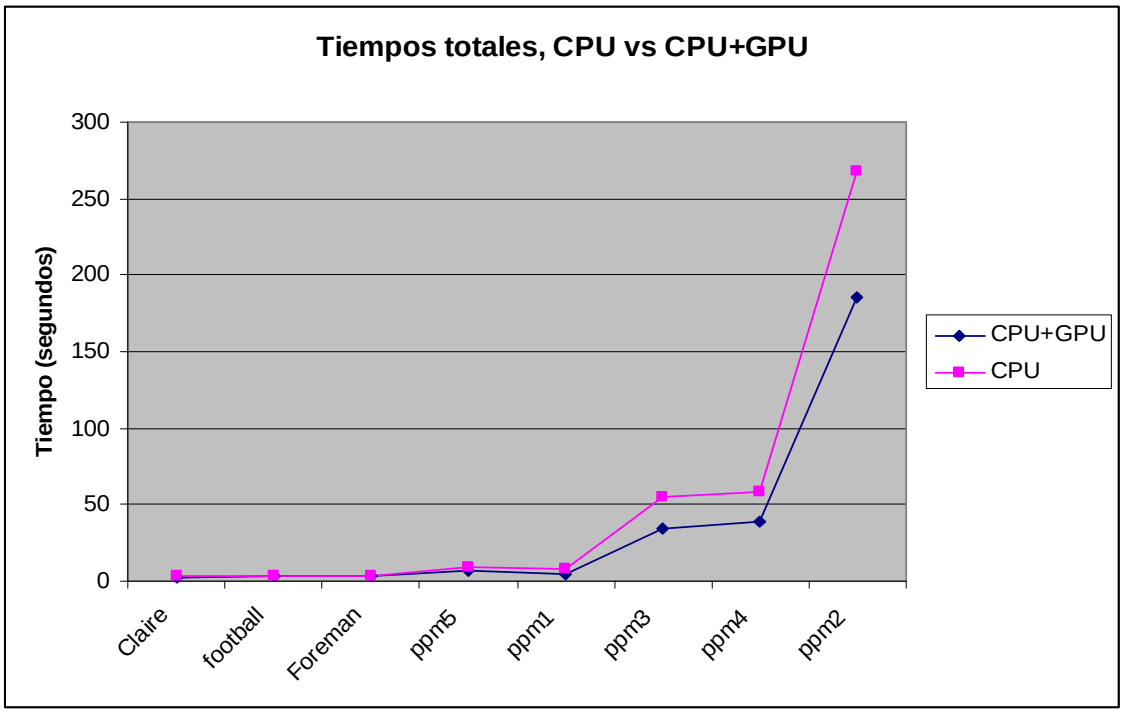
Tabla 5.2 11 Porcentaje de tiempo utilizado en el MPEG primera optimización, plataforma Yelmo

	Tiempo total (segundos)
ppm1	4,6628
ppm2	185,3001
ppm3	34,4886
ppm4	38,6093
ppm5	6,6346
Claire	2,6525
football	3,1710
Foreman	3,5006

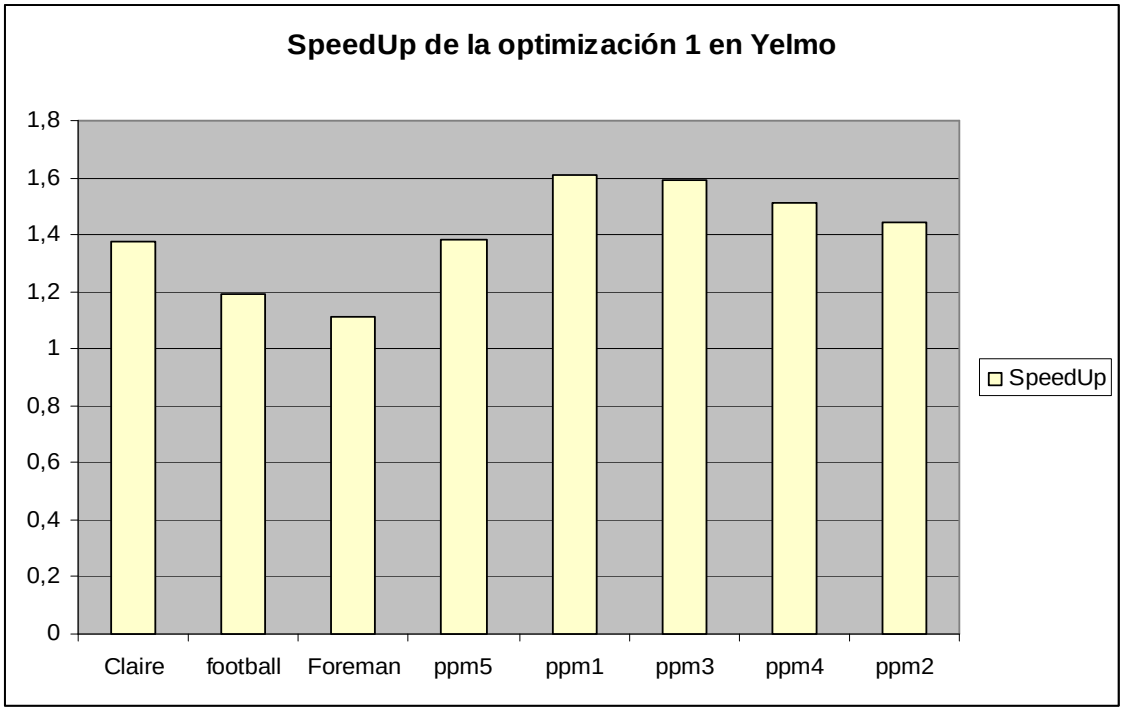
Tabla 5.2 12 Tiempo total (segundos) de ejecución de la primera optimización, en la plataforma Yelmo.

Llama la atención el porcentaje tan elevado del tiempo total que utiliza la FDCT ,Tabla 5.2 11 ,incluyendo esta optimización, ya que a nivel teórico, el tiempo de la FDCT debería disminuir; y de hecho así esta demostrado en este trabajo **[5.1 Pruebas y Resultados JPEG]**. La explicación a este fenómeno reside en que en realidad no se mide solo la FDCT. Sino que para cada frame, hay un proceso de transformación para adaptarlo al formato de textura, hay que mandar la textura a la GPU y recuperarla. Aún así, obtenemos ganancia. Dejamos abierta aquí, una posible optimización en la que toda esa preparación no fuese necesaria.

Comparemos los tiempos totales obtenidos en Yelmo hasta ahora en la siguiente gráfica, Gráfica 5.2 8. Para apreciar el crecimiento a medida que aumentan los datos procesados vamos a ordenar las muestras según el volumen de datos que manejen.



Gráfica 5.2 8 Comparativa Tiempos totales Algoritmo básico vs Optimización 1 (DCT en GPU) en Yelmo



Gráfica 5.2 9 Representación del speedup obtenido en Yelmo, con la primera optimización

Hemos obtenido un SpeedUp superior, Gráfica 5.2 9 , a 1 en todas las muestras, que varía desde 1.11 ("Foreman") hasta 1.6 (ppm1). Salvo en casos puntuales, si que se ve un aumento del SpeedUp a medida que aumenta el volumen de datos. Esto corrobora los resultados que obtuvimos en el JPEG, en los que veíamos que a medida que aumentaba el tamaño de la textura (dimensiones del frame en este caso) la GPU ganaba en rendimiento, y nos incita a plantear optimizaciones que maximicen la textura que se carga en el procesador gráfico **[4.2.5 Otras optimizaciones]**. Por otra parte, la muestra ppm2 (la que tiene mayor volumen de datos) tiene una dimensión de frame pequeña (320x240) y una gran cantidad de frames. Esto nos lleva a pensar que si tratásemos de codificar un video real, en el que el número de frames supera con creces las muestras que tenemos, el algoritmo de GPU se comportaría mucho mejor aún teniendo una dimensión de frames muy pequeña.

Como vimos en resultados anteriores, dependiendo del tipo de video que se procese, el motion tarda más o menos. Desde este punto de vista podemos añadir que al tardar el motion más (i.e. para "Foreman") se deja menos tiempo para la FDCT (recordemos que en esta optimización, es la única etapa paralelizada), y por eso en estos casos el SpeedUp es menor.

Equipo K2

Realizamos las mismas pruebas en el K2 y obtenemos lo siguientes resultados.

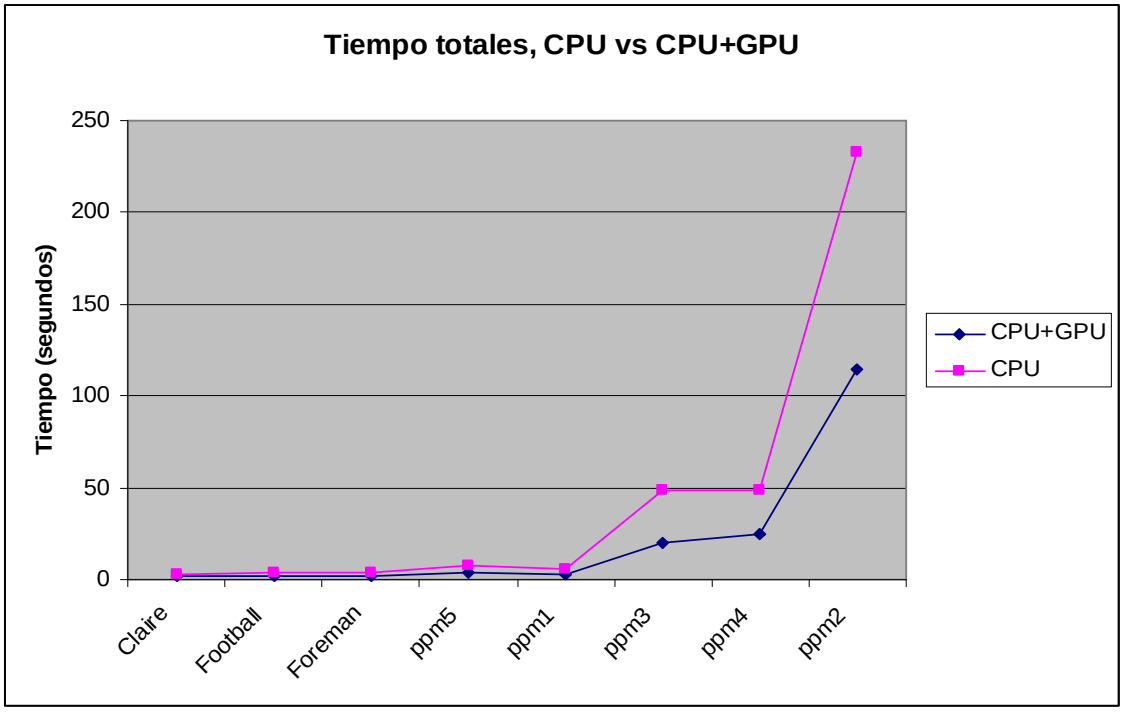
	motion	predict	FDCT	putpic
Ppm1	41,4850	1,1223	42,9282	8,0805
Ppm2	25,0557	0,9152	61,6691	6,8160
Ppm3	44,4851	1,6141	38,4975	8,4068
Ppm4	55,7869	1,0908	30,8876	6,6459
Ppm5	43,4903	1,3222	45,7500	9,2052
Claire	34,9200	1,0888	48,9254	7,8691
Football	50,8146	0,9433	36,1649	6,7569
Foreman	54,3475	0,8566	33,5291	6,3752

Tabla 5.2 13 Porcentaje de tiempo utilizado en el MPEG primera optimización, plataforma K2

	Tiempo total (segundos)
ppm1	2,9430
ppm2	114,9385
ppm3	20,0388
ppm4	24,5306
ppm5	3,8267
Claire	1,5136
Football	1,8538
Foreman	1,9775

Tabla 5.2 14 Tiempo total (segundos) de ejecución de la primera optimización, en la plataforma K2.

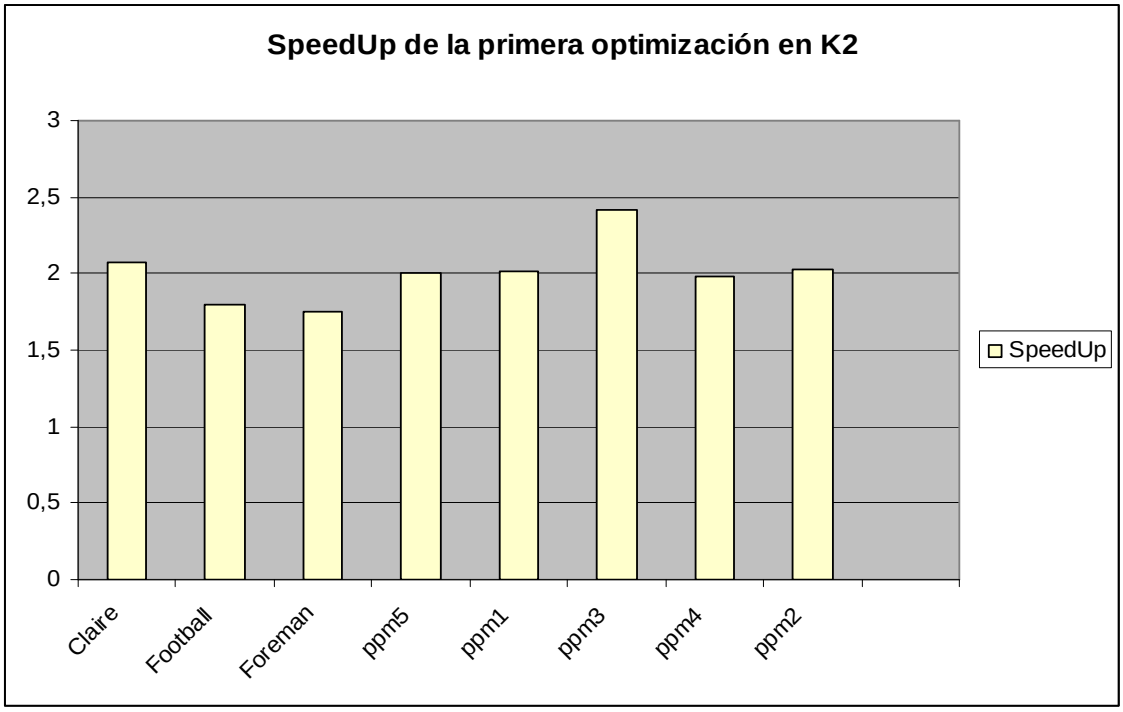
Los resultados obtenidos, Tabla 5.2 13, Tabla 5.2 14, superan con creces las expectativas. Después de ver los resultados en Yelmo con esta optimización, suponíamos que los resultados en K2 también mejorarían. Pero no imaginábamos una ganancia tan grande. Tal y como se representa en la **Gráfica 5.2.6**, llegamos a tener un SpeedUp de 2.4 para ppm3. Sin duda esto se debe a que el procesador gráfico es muchísimo más potente que el de Yelmo, y a que está optimizado para realizar tareas de propósito general.



Gráfica 5.2 10 Comparativa Tiempos totales Algoritmo básico vs optimización 1 (DCT en GPU) en K2

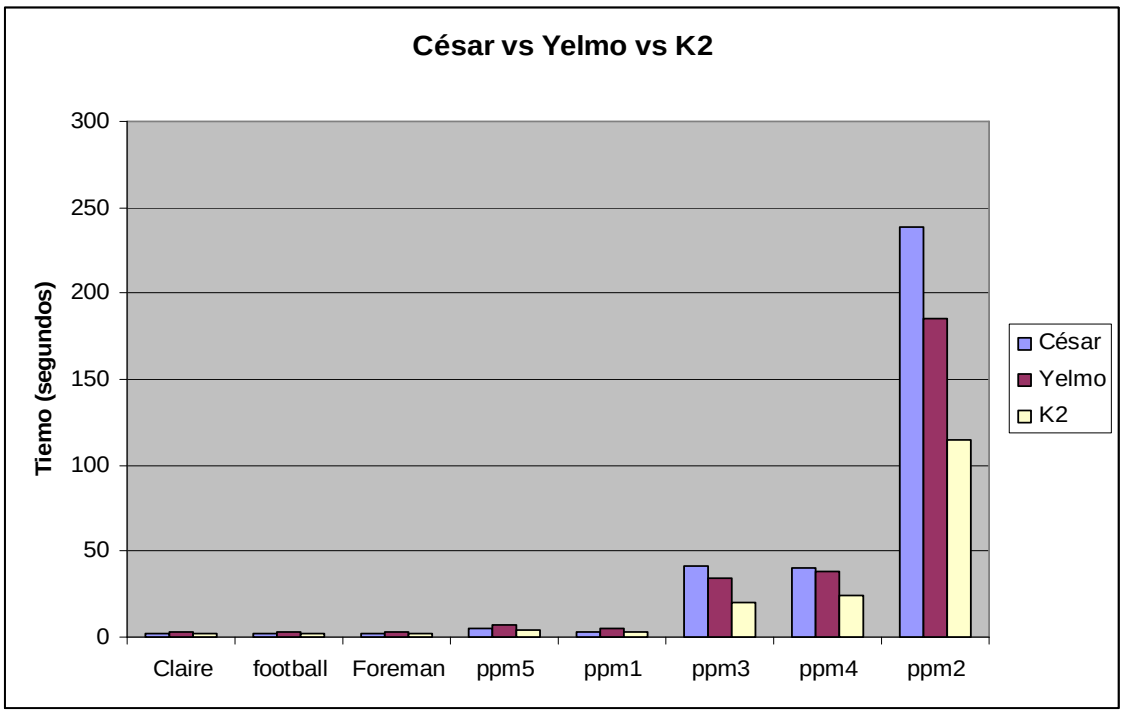
En esta gráfica, Gráfica 5.2 10 , podemos ver que aunque el K2 tenga un procesador muy potente (Core 2 Duo 4Mb caché), en lo que a términos de cálculo paralelo se refiere la GPU (en este caso la 8800) sigue siendo mas rápida. Mientras el tamaño de los frames es pequeño (del orden de 352x240) la CPU es capaz de mantenerse a la altura de la GPU, obviamente debido a la caché. Pero cuando el tamaño del los frames aumenta (ppm3 y ppm4), o el número de frames se eleva el comportamiento de la GPU es notablemente mejor.

Vemos que el SpeedUp , Gráfica 5.2 11 ,es superior a 1.5 en todos los casos, esto es algo que llama mucho la atención ya que el mismo algoritmo en Yelmo no obtenía ni de lejos unos resultados semejantes, por supuesto esta diferencia se debe a la 8800. El caso particular de ppm3 pone de manifiesto todo lo que hemos dicho sobre las dimensiones del frame ya que utiliza frames más grandes (640x480). Además el procesado del ppm3 dedica menos tiempo al motion ya que es un video a cámara fija y con movimiento leve, esto beneficia a esta mejora en el cálculo de la FDCT.



Gráfica 5.2 11 Representación del speedup obtenido en K2 con la primera optimización.

Es el momento de hacer una comparativa de los equipos usando el mismo algoritmo. Vistos los resultados hasta ahora, es obvio que el K2 es muy superior.



Gráfica 5.2 12 Comparativa Tiempos totales del algoritmo MPEG optimización 1 en las 3 máquinas.

Dada la diferencia entre los procesadores de César y Yelmo con el K2, esta gráfica no compara solo el comportamiento de las tarjetas gráficas. Sino ambas, CPU-GPU con CPU-GPU.

5.2.3 Estudio temporal Optimización 2: Implementación con threads(CPU-GPU)

Esta optimización añade un sistema de productor-consumidor a la optimización anterior. Con esto pretendemos que no se produzca espera cada vez que el algoritmo solicite un frame. En principio esperamos tener ganancia en todos los equipos.

Equipo César

	motion	predict	FDCT	Putpic
ppm1	46,8018	1,0242	33,6791	13,0049
ppm2	46,9278	1,4730	33,2008	11,2289
ppm3	53,7360	1,0014	27,6199	11,4248
ppm4	61,8458	0,8302	22,5971	9,3332
Claire	43,8260	1,0874	35,4672	13,8050
Football	53,8421	0,8768	29,1269	11,3968
Foreman	56,3411	0,8090	27,5588	10,7788

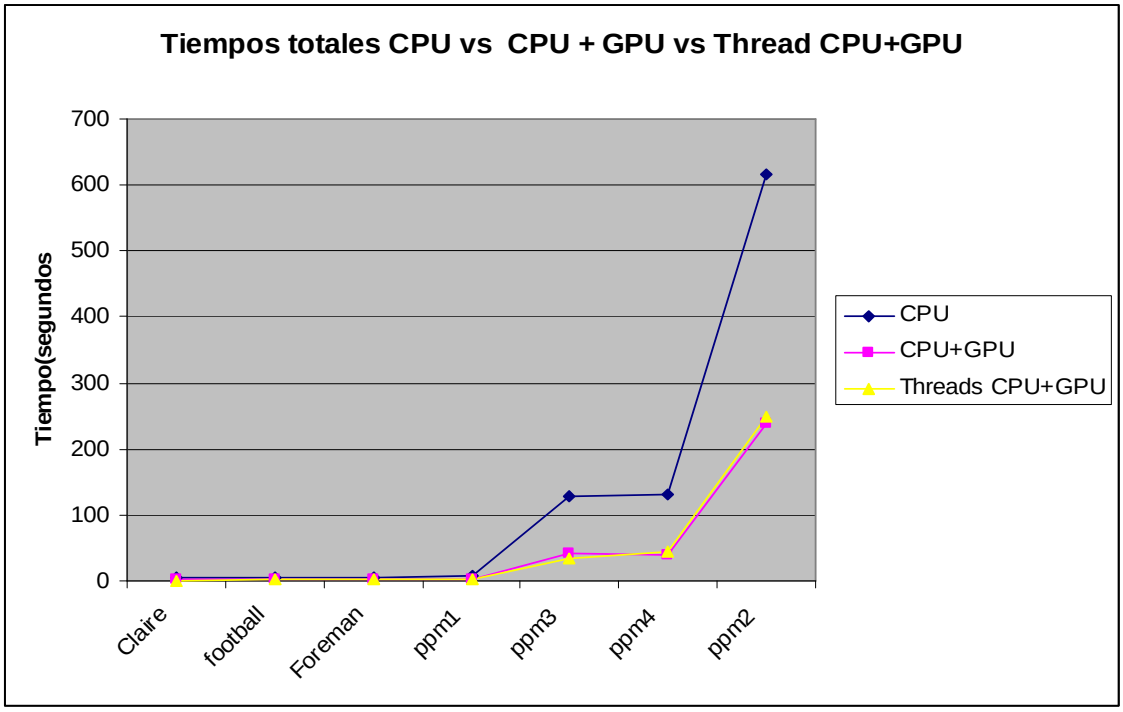
Tabla 5.2 15 Porcentaje de tiempo de la optimización 2 en César

	Tiempo total (segundos)
ppm1	3,2083
ppm2	249,7540
ppm3	35,1628
ppm4	43,9251
Claire	1,9933
Football	2,2813
Foreman	2,2876

Tabla 5.2 16 Tiempos totales algoritmo Thread en César(segundos)

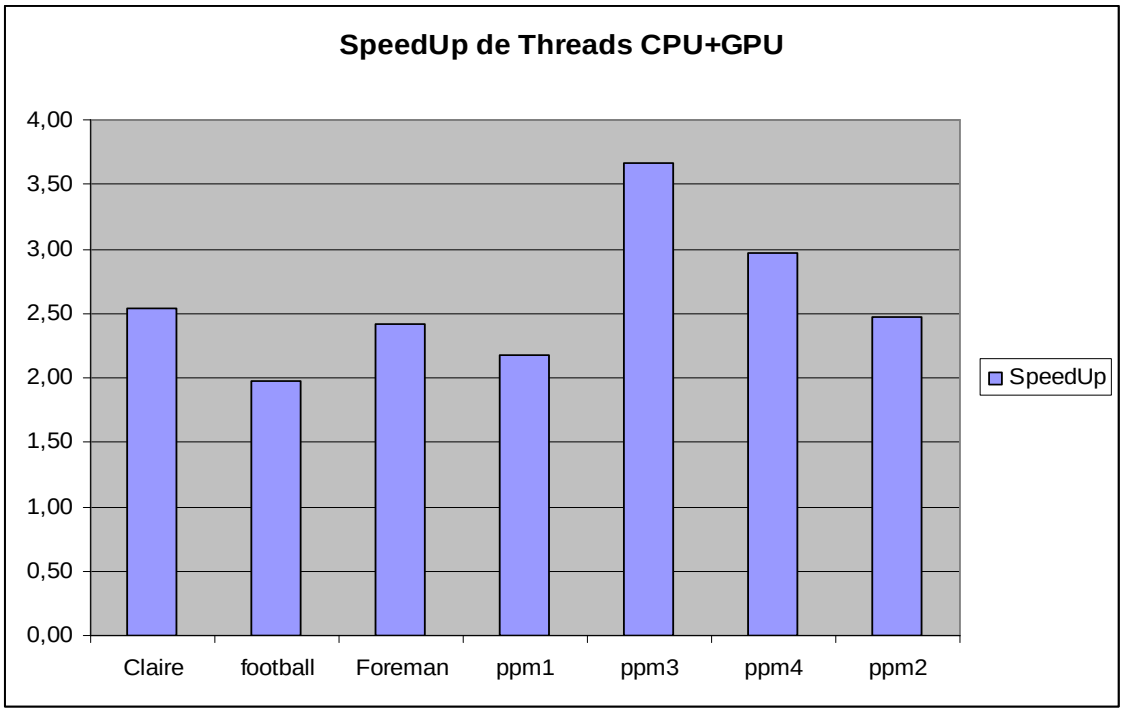
	CPU	CPU+GPU	Threads CPU+GPU
Claire	4,8618	1,9153	1,9933
football	4,5061	2,2106	2,2813
Foreman	5,5380	2,3271	2,2876
ppm1	6,9776	3,1686	3,2083
ppm3	128,8615	40,9446	35,1628
ppm4	130,0541	39,8911	43,9251
ppm2	616,2385	238,7836	249,75400

Tabla 5.2 17 Comparativa de los 2 algoritmos anteriores junto al algoritmo base en César, tiempo en segundos.



Gráfica 5.2 13 Comparativa tiempos totales César

Como podemos observar en la Gráfica 5.2 13 las dos optimizaciones del algoritmo MPEG llevan un Speedup medio de 2.5.



Gráfica 5.2 14 SpeedUp de la optimización 2 en César

Equipo Yelmo

	motion	Predict	FDCT	Putpic
ppm1	34,9041	0,8160	46,0121	13,2011
ppm2	29,5305	0,9844	50,0653	13,8575
ppm3	38,5763	0,9003	41,8174	13,3158
ppm4	45,9795	0,7984	36,6915	11,8046
Claire	30,0251	0,7149	51,2631	12,7023
Football	39,6786	0,7107	43,8235	11,34794

Tabla 5.2 18 Porcentaje tiempos totales algoritmo Threads en Yelmo

	Tiempo total (segundos)
ppm1	4,5358
ppm2	168,0805
ppm3	31,6988
ppm4	35,7205
Claire	2,6103
football	2,9268

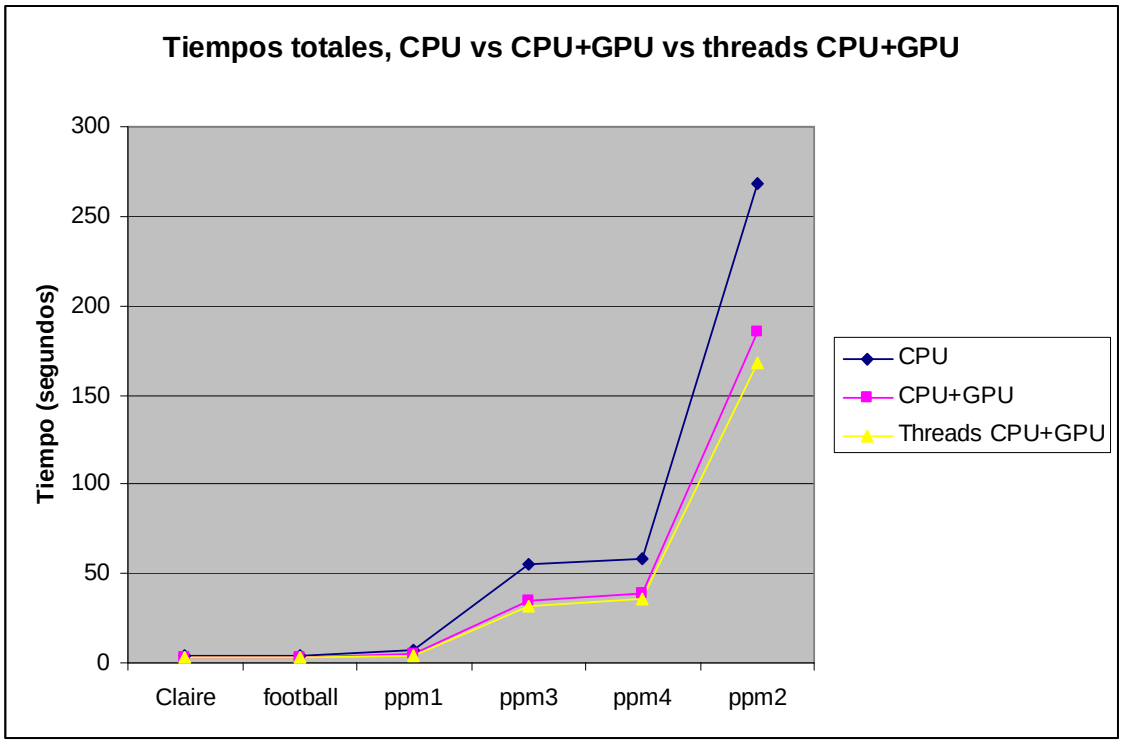
Tabla 5.2 19 Comparativa de los 2 algoritmos anteriores junto al algoritmo base en Yelmo (segundos)

Con esta tabla, Tabla 5.2 19, obtenemos los mejores resultados para cada muestra sobre Yelmo, es decir, una vez más hemos obtenido ganancia.

En el gráfico siguiente representamos la comparativa de tiempos totales de ejecución para todas las muestras del algoritmo MPEG, con las 2 mejoras y la básica, Gráfica 5.2 13. Como comprobamos para todas las muestras conseguimos un speedUp medio de 1.5, Gráfica 5.2 15. No conseguimos un speedup tan pronunciado como en la plataforma anterior (césar Gráfica 5.2 14)

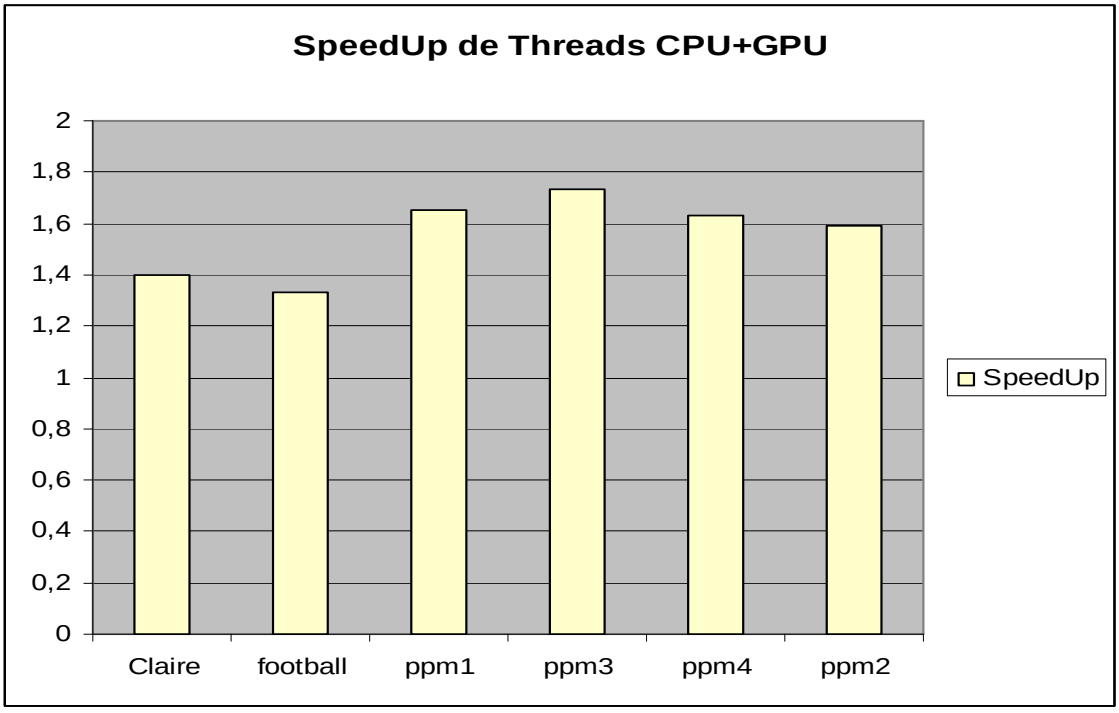
	CPU	CPU+GPU	Threads CPU+GPU
Claire	3,6501	2,6525	2,6103
football	3,7810	3,1710	2,9268
ppm1	7,4931	4,6628	4,5358
ppm3	54,9043	34,4886	31,6988
ppm4	58,3803	38,6093	35,7205
ppm2	267,8013	185,3001	168,0805

Tabla 5.2 20 Comparativa de los 2 algoritmos anteriores junto al algoritmo base en Yelmo, (segundos)



Gráfica 5.2 15 tiempos totales Yelmo

En esta otra gráfica vemos el SpeedUp respecto de la implementación básica, en la que obtenemos un SpeedUp medio de 1.5, Gráfica 5.2 16 .



Gráfica 5.2 16 Speedup de la optimización 2 sobre el algoritmo básico en Yelmo

La ganancia que se obtiene frente a la optimización 1 es muy pequeña y se debe a que Yelmo posee hiperthreading, esto es, que a pesar de tener un único procesador, es capaz de ejecutar dos hilos en paralelo sin necesidad de compartir el procesador por rodajas de tiempo. El hecho de que la ganancia sea pequeña no es definitivo, ya que a medida que el número de frames de las muestras aumenta también lo hace el SpeedUp.

Ahora vamos a mostrar los resultados sobre el equipo tecnológicamente más potente, el K2, que incluye un microprocesador Core 2 Duo, con una tarjeta gráfica de última generación NVidia 8800 GTX.

Equipo K2

Después de ver el comportamiento de esta optimización en Yelmo vamos a ver que resultados nos ofrece el hardware de alto rendimiento.

	motion	predict	FDCT	Putpic
ppm1	41,6822	1,1073	42,6933	8,1335
ppm2	29,8386	1,0899	54,3737	8,1042
ppm3	44,3827	1,6157	38,8965	8,1279
ppm4	55,9628	1,0900	30,7906	6,5986
Claire	35,0781	1,0493	48,7953	7,8763
football	50,4801	0,9366	36,5969	6,7288

Tabla 5.2 21 Porcentaje tiempos totales algoritmo Thread en K2

	Tiempo total (segundos)
ppm1	2,9100
ppm2	110,0063
ppm3	20,0858
ppm4	24,6796
Claire	1,6503
football	1,8633

Tabla 5.2 22 Tiempos totales de ejecución (segundos) en K2

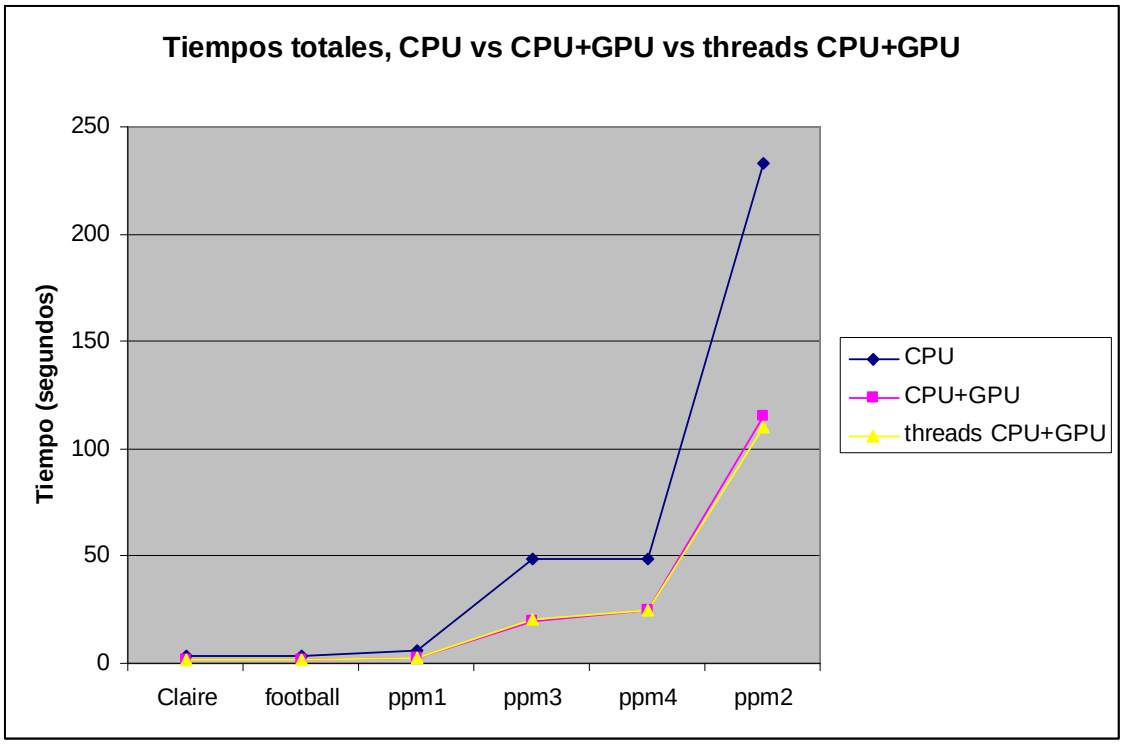
En contra de todo pronóstico, los tiempos obtenidos en el K2,Tabla 5.2 22 , son prácticamente idénticos a los obtenidos con la optimización 1, de hecho para "Claire" los tiempos son mejores solo con la optimización 1. La explicación que le damos a esto es que, la primera optimización ya reduce de forma drástica el tiempo total y esto hace que la petición de frames sea muy rápida, poniendo en apuros al hilo productor. Además, la implementación de nuestro buffer de lectura esta hecha de forma que cada vez que se lee un frame, se reserva memoria dinámica para albergarlo y cuando ya no se necesita más, se libera la memoria; en contraposición, las implementaciones que no utilizan un buffer reservan memoria una única vez y la usan la misma para todos los frames. El problema que esto conlleva es que reservar memoria y liberarla son operaciones costosas y elevan el tiempo total.

Las pruebas realizadas en Yelmo, Tabla 5.2 21 , no muestran este problema, ya que el hilo consumidor no es tan rápido como en el K2,

ya hemos comentado a lo largo de todo este trabajo la diferencia en potencia de cálculo entre la 7800 (Yelmo) frente a la 8800 (K2).

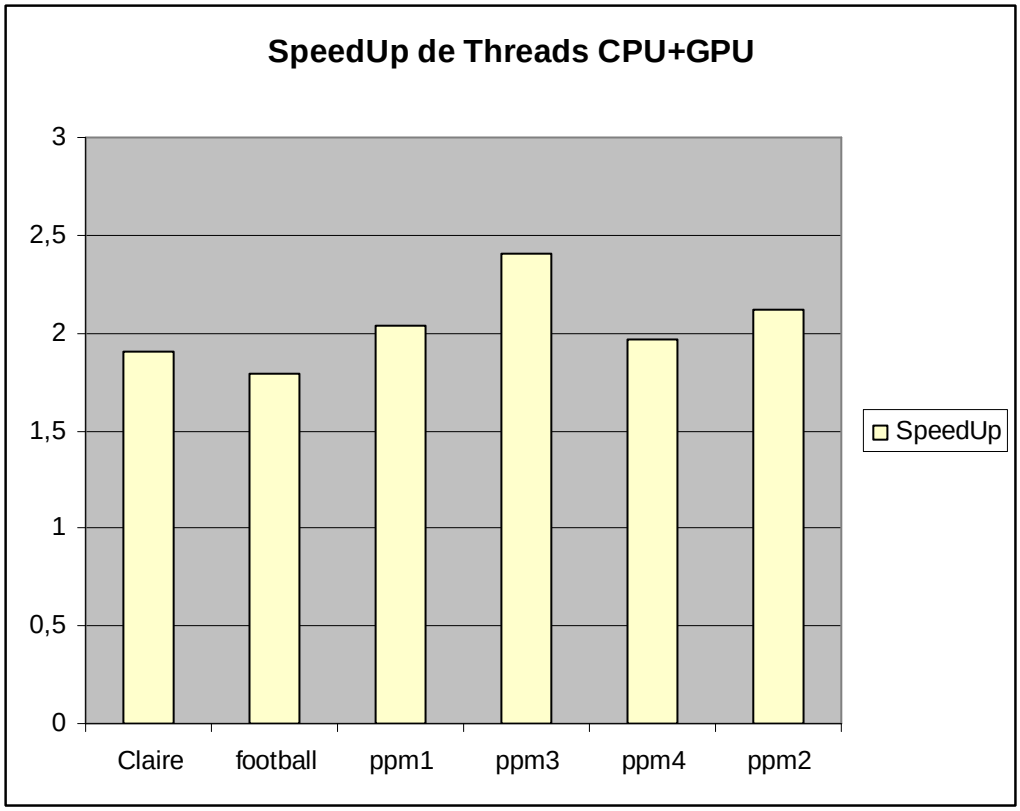
	CPU	CPU+GPU	threads CPU+GPU
Claire	3,137	1,5136	1,6503
football	3,3398	1,8538	1,8633
ppm1	5,9221	2,9143	2,9100
ppm3	48,3018	20,0388	20,0858
ppm4	48,5486	24,5306	24,6796
ppm2	232,6096	114,9385	110,0063

Tabla 5.2 23 Tabla de tiempos totales (segundos) de los 3 algoritmos K2



Gráfica 5.2 17 Gráfica tiempos totales en K2

Por supuesto, si esta optimización se realizase con un buffer que reservase memoria una única vez se vería algo de mejora, pero con la implementación actual, los dos efectos se anulan entre sí (mejora por los hilos, pérdida por las reservas sucesivas de memoria).



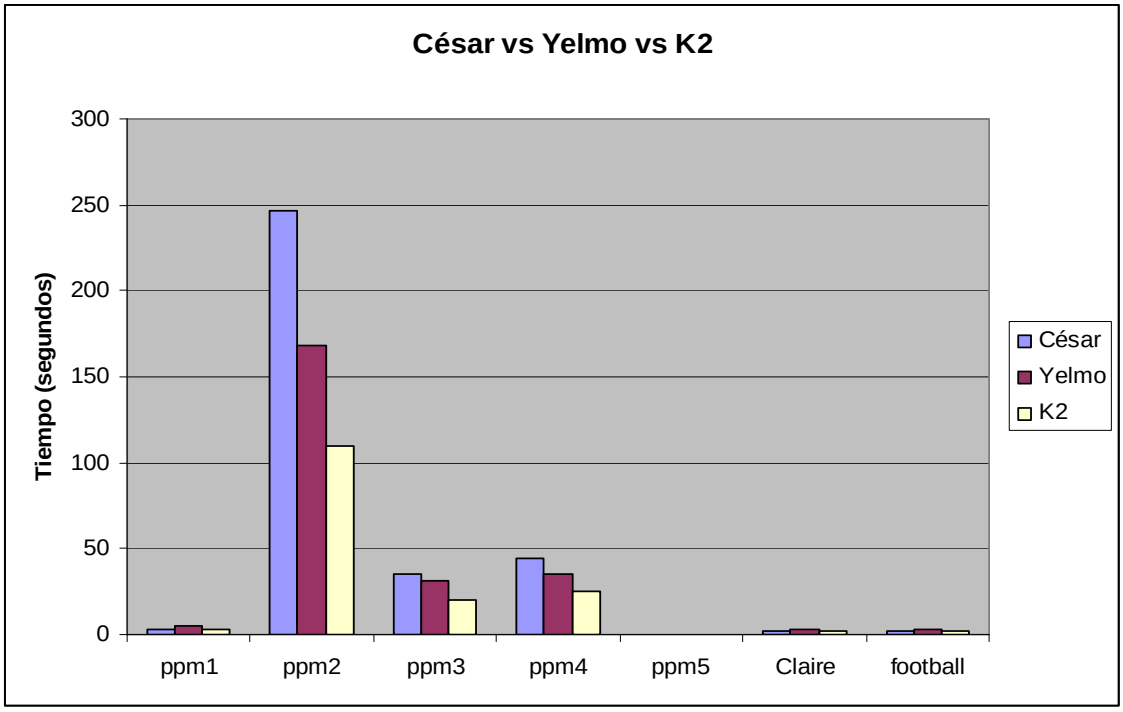
Gráfica 5.2 18 SpeedUp de la optimización 2 sobre algoritmo básico en K2

Visto los resultados obtenidos, la gráfica que representa el SpeedUp de esta optimización con el algoritmo básico no es interesante. Simplemente comentar que ahora, viendo las gráficas de los SpeedUp, todas tienen la misma forma (aunque den distintos valores), esto corrobora que el rendimiento está muy ligado a las características de los videos. Por esto, no podemos sacar unas conclusiones que engloben el comportamiento del MPEG para cualquier video.

Por último vamos comparar el comportamiento del mismo algoritmo (Threads CPU+GPU) en los tres equipos, Gráfica 5.2 19. Siempre que hemos hecho esta comparativa a lo largo del trabajo, hemos mantenido que Yelmo obtendría mejores resultados que César y que K2 los obtendría sobre Yelmo. En este caso esperamos, y viendo los resultados que hemos obtenido, que la diferencia entre los tiempo totales sea menor, esto es debido a que el algoritmo en K2 es más lento con esta optimización que solo con la primera, mientras que en César y Yelmo esta optimización si que mejora el rendimiento global.

	César	Yelmo	K2
Ppm1	3,2054	4,5358	2,91
Ppm2	246,2310	168,0805	110,0063
Ppm3	35,1628	31,6988	20,0858
Ppm4	43,9251	35,7205	24,6796
Claire	1,9933	2,6103	1,6503
football	2,2813	2,9268	1,8633

Tabla 5.2 24 Tabla de tiempos (segundos) totales de ejecución en segundos en las 3 plataformas



Gráfica 5.2 19 Comparativa de la optimización 2 en los 3 equipos.

La diferencia entre los tiempos se ha reducido con respecto a la optimización 1, en los tres equipos, pero aun sigue siendo muy elevada. Aquí queda reflejada la diferencia de potencia de cálculo de la 8800 (en K2) frente al resto de tarjetas gráficas. También observamos que esta optimización obtiene resultados muy similares a la optimización 1.

Hemos visto, que la implementación con hilos utilizando la GPU obtiene la misma ganancia que la optimización 1. Sin embargo, tenemos un equipo con un procesador de doble núcleo muy potente capaz de procesar varios hilos a la vez de forma independiente. Esto nos motiva a realizar la misma optimización de hilos pero utilizando solamente la CPU que es lo que veremos a continuación.

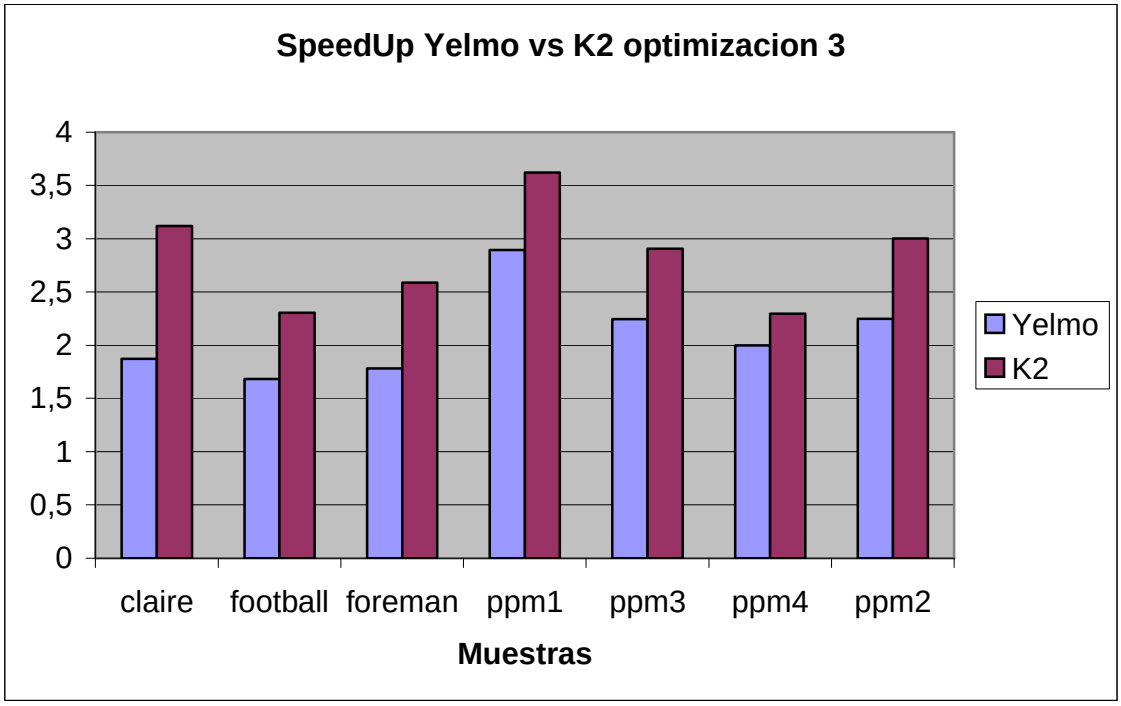
5.2.4 Estudio temporal Optimización 2: Implementación con threads(CPU-CPU)

La siguiente propuesta de optimización es la utilización de hilos en CPU. Hacemos esta propuesta por las altas prestaciones que nos ofrecen las plataformas de estudio.

	yelmo	K2
claire	1,9498	1,0053
football	2,2467	1,4495
foreman	2,1805	1,3408
ppm1	3,1617	2,1142
ppm3	24,4505	16,6232
ppm4	29,1973	21,1555
ppm2	119,1118	77,4712

Tabla 5.2 25 Tabla de tiempos (segundos) totales de ejecución en segundos en yelmo y k2 en la optimización 2.

En la tabla de tiempos, Tabla 5.2 25, podemos observar el potencial de los procesadores actuales. Conseguimos ganancia en Yelmo gracias al HiperThread, y ganancia en K2 por se de doble núcleo. De hecho esta ganancia es superior a la que obtenemos con las optimizaciones de GPU, esto se debe a que la comunicación entre los procesadores dentro del chip (en el caso de K2) o en el mismo procesador (Yelmo) es más rápida que la comunicación con la GPU. A continuación mostramos los SpeedUp conseguidos frente a la implementación básica.



Gráfica 5.2 20 SpeedUp tercera optimización Yelmo

Podemos ver que la estimación de movimiento sigue siendo determinante para no conseguir uniformidad en el SpeedUp. Esto se ve en las muestras ppm3 y ppm4 que teniendo el mismo número de frames y la misma resolución, se obtiene una diferencia de casi un punto en el SpeedUp. También se ve que los videos de menor resolución salen beneficiados frente a los de mayor resolución, como es el caso de ppm1 contra ppm3 y ppm4.

5.1.5 Conclusiones estudio MPEG.

Una vez concluido el estudio de las distintas implementaciones y optimizaciones del algoritmo MPEG, llegamos a las siguientes conclusiones:

- El uso de la GPU para acelerar el cálculo de la DCT es altamente recomendable ya que reduce el tiempo total de ejecución incluso con tarjetas gráficas que no sean de última generación.
- No podemos establecer una ganancia fija, ya que el tiempo de ejecución de cada parte del algoritmo esta íntimamente ligado a las características del video que se quiere comprimir. Entre estas características podemos destacar:
  - Tipo de movimiento.
  - Cambios de plano.
  - Color o Blanco y negro.
- Para videos de resolución pequeña (hasta 352x240), el algoritmo en CPU se comporta de forma muy similar al que utiliza la GPU; mientras que para videos de resolución mayor el algoritmo en GPU es superior.
- A medida que el número de frames aumenta, la diferencia entre los algoritmos de CPU y GPU se incrementa. Esto significa que para videos grandes (películas, videos caseros, etc…) el algoritmo de GPU ofrece un tiempo de cómputo menor, que mejora en comparación con el de CPU a medida que el tamaño del video aumenta.
- La utilización de hilos como esquema de paralelización de la entrada del algoritmo MPEG no aporta demasiada ganancia, y si el hardware gráfico es de última generación los resultados son incluso peores.

En la actualidad ha surgido un nuevo formato de video/TV denominado Alta Definición que utiliza una resolución de 1080 líneas. Cuando este nuevo formato se estandarice y sea de uso cotidiano, algoritmos que utilicen la GPU como forma de paralelización obtendrán mejores resultados frente a los algoritmos secuenciales.

6 – Conclusiones y trabajo futuro

En el capítulo uno definimos como objetivos:

- Adquirir un conocimiento claro y profundo acerca del funcionamiento de los algoritmos de compresión de imágenes y video, JPEG y MPEG.
- Estudiar las posibilidades de los procesadores gráficos como hardware de propósito general.
- Estudiar y experimentar con técnicas de paralelización de algoritmos en JPEG y MPEG
- Diseñar e implementar distintas optimizaciones sobre estos algoritmos, haciendo uso de las técnicas y elementos antes citados.
- Obtener conclusiones claras acerca de la utilidad de todas estas tecnologías en el campo de la compresión de imágenes.

Tras la realización del trabajo, hemos alcanzado todos los objetivos, llegando a las siguientes conclusiones:

- Los algoritmo de compresión MPEG y JPEG pueden ser mejorados si se reescriben teniendo en cuenta las nuevas mejoras hardware.
- La GPU es mucho más rápida que la CPU ejecutando algoritmos paralelos, como comprobamos en el caso de la DCT.
- Para obtener el máximo beneficio de la GPU, esta debe usarse con un flujo de datos relativamente grande, para compensar el tiempo de inicialización y carga.
- Los algoritmos más rápidos en la CPU no tienen por que serlo en GPU. Hemos comprobado que en el hardware de ultima generación, la DCT-2D,o DCT lenta es más rápida que la FDCT (DCT rápida).
- El uso de threads resulta interesante cuando la CPU y GPU ejecutan cálculos en paralelo, ejemplo de esto lo hemos visto en JPEG.
- Los equipos con CPUs más lentas son los que mayor ganancia consiguen, cuando utilizamos la GPU.
- El rendimiento de los algoritmos que utilizan la GPU es menos sensible a la carga de la CPU. Paralelismo a nivel de programas.
- Las nuevas CPUs son significativamente más rápidas en la ejecución de algoritmos de compresión. Hasta 3 veces más (*P4 HT*, equipo César, vs *Core 2 Duo*, equipo K2, gráfica 5.2.5).

En este trabajo hemos estudiado los beneficios que las nuevas mejoras hardware pueden aportar a los algoritmos existentes. Para aprovechar tales mejoras es necesario estudiar el problema concreto buscando puntos potencialmente paralelizables. Los resultados han demostrado que con una reestructuración del código, es posible conseguir aumentos en el rendimiento muy significativos. También hemos visto como hay algoritmos y etapas más aptos para ser llevados a la GPU; ejemplo claro de ello ha sido la DCT que, por ser altamente paralelizable ha incrementado su rendimiento sobre la CPU en más de once.

El estudio también ha servido para experimentar con el potencial presente en los nuevos microprocesadores. Las comparativas con la anterior generación demuestran que ha habido un aumento importante en rendimiento, lo que al final se nota en la ejecución de los programas. Las dos características más importantes que han propiciado esta mejora, han sido por un lado el gran aumento del tamaño de las memorias caché, y por otro el paralelismo introducido en los procesadores. También decir que los modelos actuales que disponen de dos microprocesadores en un mismo chip, cuentan con la ventaja de tener una menor sobrecarga, pues el trabajo se reparte entre ambos, y todo esto influye positivamente sobre el coste temporal de los programas.

Aunque el estudio se ha centrado en algoritmos de compresión de imágenes y video, no es difícil extenderlo a otros campos. Pues la DCT es un método ampliamente utilizado en todo tipo de algoritmos de compresión y codificación de información. Por ejemplo, los métodos de compresión de audio: MP3, ACC, WMA,… están basados en una modificación de la DCT(MDCT), y por lo tanto se podrían aplicar los resultados obtenidos para conseguir mejorar el rendimiento de estos también.

Además, a lo largo de todo el trabajo se han ido proponiendo líneas de continuación a este proyecto. A continuación nombramos las más destacadas:

- Incluir el cambio de color en los cómputos de la GPU.
- Incluir la fase de estimación de movimiento en los cómputos de la GPU.
- Algoritmo paralelo procesando varios frames de distintos GOP’s al mismo tiempo.

Bibliografía:

[1] Yun Q. Shi , Huifang Sun"Image and video compression for multimedia engineering fundamental algorithms, and standars" (publicado 2000 CRC Press)

[2] Kamisetty R Rao, Vladimir Britanak, Patrick C. Yip "*Discrete Cosine and Sine Transforms: General Properties, Fast Algorithms and Integer"* (publicado 2006 Elsevier)

[3] John Watkinson "The Mpeg Handbook: MPEG-1, MPEG-2, MPEG-4"

(publicado 2001 Focal Press)

[4] Iain E. G., Richardson , "H.264 and MPEG-4 video compression: Video Coding for Next-Generation Multimedia " (John Wiley and Sons)

[5] Peter Norton. *"Peter Norton's New Inside the PC".* Scott H. A. Clark – 2002

[6] Vaidy S. Sunderam. "*Computational Science -- ICCS 2005: ICCS 2005 : 5th International Conference, Atlanta, GA, USA,..".* Springer. 2005

[7] Bruaset, Aslak Tveito. "*Numerical Solution of Partial Differential Equations on Parallel Computers".* Springer. 2005

[8] Luis M. Corbalán, Carlos B. Amat. *"Vocabulario de información y documentación automatizada".* Publ. Universitat de Valencia. 2003

[9] Peter D. Symes. *"Digital Video Compression".* McGraw-Hill Professional*. 2003*

[10] Hervé Benoit. *"Television Digital".* Thomson Learning Ibero. 1998

[11] Nasser Kehtarnavaz, Mark Gamadia. *"Real-Time Image and Video Processing: From Research to Reality".* Morgan & Claypool Publishers. 2006

[12] MPEG-2 system level standard (ISO 13818-1) 2nd Ed