UAB
Universitat Autònoma de Barcelona

Dipòsit digital de documents de la UAB

This is the **published version** of the bachelor thesis:

Cases Domínguez, Josep M. *Compresión en CUDA : análisis y validación del estado del arte de las técnicas de predicción y transformadas*. Treball de Final de Grau (Universitat Autònoma de Barcelona), 2026 (Enginyeria Informàtica)

This version is available at https://ddd.uab.cat/record/326562

under the terms of the  license.

Comprensión en CUDA: análisis y validación del estado del arte de las técnicas de predicción y transformadas

Josep M. Cases-Domínguez

9 de febrer de 2026

Resumen– El presente Trabajo de Fin de Grado, evalúa sistemáticamente la implementación paralela en CUDA de algoritmos fundamentales de compresión de imágenes sin pérdidas. El objetivo del mismo es determinar la presencia de CUDA en el estado del arte de esas técnicas a día de hoy. Los resultados obtenidos demuestran el potencial de esta tecnología en el ámbito y también muestra algunas de sus principales complicaciones. Validando así el estado del arte de CUDA 2026 y establece futuros desarrollos para explotar el hardware moderno.

Palabras clave– Compresión de imágenes, compresión sin pérdida, decorrelación de imágenes, técnicas de transformada, técnicas de predicción, OpenMP, CUDA, benchmarking

Abstract– This Bhacerlors Degree Project systematically evaluates the parallel implementation in CUDA of fundamental lossless image compression algorithms. The main objective is to determine the how presence is CUDA in the state of the art of these techniques today. The results obtained demonstrate the potential of this technology in the field and also the main complications to consider. Therefore validating the state of the art of CUDA 2026 and tracing lines future developers to improve this methods on modern hardware.

Keywords– Image compression, lossless compresion, image decorrelation, tranform techniques, prediction techniques, OpenMP, CUDA, benchmarking

1 Introducción - Contexto del trabajo

La compresión es un elemento clave a la hora de tratar con información. Se refiere al proceso de reducir un conjunto inicial de símbolos de un alfabeto y tamaño concreto, a un conjunto final de menor tamaño. Si el conjunto final permite reconstruir exactamente el conjunto original, se dice que es una compresión sin perdida. Por el contrario, si no podemos obtener el conjunto original a partir del final, decimos que ha habido perdida.

Se pueden comprimir todo tipo de fuentes: audio, video, imagen, etc...

• E-mail de contacte: casesjosepm@gmail.com
• Menció: Tecnologies de la Informació
• Treball tutoritzat per: Xavier Fernández Mellado (DEIC)
• Curs 2025/26

Un sistema de compresión clásico consta de 3 fases principales: decorrelación, cuantización (cuando se trabaja con pérdida o near-lossless) y un codificador por entropía. En la decorrelación, se busca explotar la proximidad espacial. Por lo general los símbolos dentro de una imagen suelen ser más parecidos cuanto más cerca están uno del otro. Esta proximidad espacial genera patrones y características que pueden explotarse para reducir al máximo la cantidad de símbolos distintos que deben procesarse, disminuyendo así la entropía.

La entropía, en teoría de la información, refiere a la medida de incertidumbre de los símbolos de una fuente. Este valor mide el promedio de bits necesarios para codificar cada símbolo del conjunto. Imaginemos que existe un conjunto $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$, cada símbolo tiene una posibilidad concreta de aparecer $p(s_1), p(s_2), \ldots, p(s_n)$, esta es su incertidumbre. A menos probable sea que aparezca un símbolo más información contiene. Esta cantidad de información se expresara como: $-\log_a p(s_n)$ donde en funció de la base del logaritmo cambia la unidad. Si es base 2,

bits, si es base e, nats.

Y con la suma ponderada de la información obtenemos la entropía.

$$H(S) = -\sum_{x \in \mathcal{S}} p(x) \log_2 p(x)$$

El objetivo de un sistema de compresión, es concentrar la mayor cantidad de probabilidad en la menor cantidad de símbolos. El resultado ideal será una distribución de Laplace. Para ello contamos con dos tipos de técnicas; las transformadas y los predictores.

1. Transformada: Este término se usa en matemáticas para describir una función bijectiva que busca simplificar un problema. Una multiplicación de números romanos como XV · VII es difícil. Por lo tanto, se aplica una transformada una transformada a 15 · 7 = 105 y una vez obtenido el resultado, como es bijectiva, invertimos la transformada de vuelta a números romanos XV · VI = CV.

2. Predictores: Son técnicas que, en lugar de trabajar con los valores de toda la imagen, usan un contexto. No intentan trabajar con el global sino que intentan predecir cuál será el siguiente valor en base a su conjunto de vecinos. Un predictor muy simple es considerar que el siguiente valor será igual que el último valor leído:

   $$f(x_n) = x_{n-1}$$

   Estas funciones trabajan sobre el error entre el valor real y la predicción, siguiendo esta estructura:

   $$E = P_{real} - P_{pred}$$

   conocida como DPCM (Differential Pulse Code Modulation). Como los píxeles vecinos, por lo general, tienen valores similares, este error oscila en valores cercanos a 0 y encontramos otra distribución de Laplace. En este caso, sobre el error.

Todas estas técnicas ya han sido profundamente estudiadas y comparadas anteriormente. Pero siendo técnicas desarolladas entre las décadas de 1990-2000 no han sido probadas con tanto detalle en CUDA (Compute Unified Device Architecture).

1.1 Cuda

CUDA es una API de computación heterogénea en paralelo oficialmente en lanzada por Nvidia en 2007 [1] que permite enviar conjuntos de instrucciones a ejecutar en la GPU (denominada *device*), mientras la CPU actúa como *host*.

El que estos dos elementos puedan interactuar fluidamente es muy valioso. Mientras la CPU esta diseñada para ejecutar un número finito de operaciones complejas, la GPU esta diseñada para ejecutar masivamente instrucciones sencillas.

1.2 Arquitectura básica

Cada GPU de Nvidia está formada por múltiples *Streaming Multiprocessors* (SM), que son las unidades mínimas independientes de ejecución. Cada SM, Figura 1,consta entre otros, de:

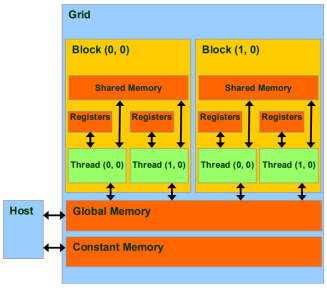
- Múltiples puertos de operación
- Memorias caché (compartida)
- *Tensor cores* para multiplicación de matrices

Fig. 1: arquitectura de un SM

La ventaja principal de la GPU son sus vector lanes. Los vectores lanes en CUDA son los 32 hilos individuales que forman un WARP. Es decir elementos individuales de calculo. Hasta un total de 1024 elementos (Threads) por SM. Los vecotres lanes són clave porque ejecutan la misma instrucción los 32 en simultaneo.

Però tambien encontramos limitaciones. El principal de ellos es la traslación de memoria

Estos arquitectura tiene una estructura de memoria que es critica. Estos son el equivalente a las caches de una CPU y se tienen que considerar en todo momento al trabajar en GPU.



Jerarquía de memoria CUDA

Cada vez que queremos ejecutar una función en la gráfica (Kernel) debemos usar la memoria global. Es importante tener en mente que acceder a la memoria global es extremadamente costoso. Tanto que cada kernel launch incurre en overhead. Por este motivo, se minimiza el número de kernels instanciados, fusionando operaciones wavelet (forward/inverse, niveles múltiples) en un único kernel cuando es posible.

1.3 Motivación y Objetivos

Este es el principal motivo del trabajo. El adaptar técnicas que nacieron en el contexto de las CPU y modernizar-las. Con esto buscamos determinar los puntos críticos como: con que tamaños de imagen la GPU tiene un impacto real, que tipos de técnica se adaptan mejor? las que atacan

por filas o por bloques? entre otros múltiples factores. Otra gran motivación para este trabajo es la metodología. Este trabajado tiene un gran componente de investigación y testing. Dos aspectos que por lo general a lo largo de la carrera no se suelen ver tanto. Asi que es una buena manera de trabajar con otras metodologías y profundizar en tecnologías del ámbito científico-tecnológico.

Es trabajo tiene como objetivo final realizar un benchmark que refleje el estado del arte de las técnicas de predictores y transformadas incorporando el factor de la programación en CUDA. Para ello buscaremos los algoritmos teóricos de las diferentes técnicas y los adaptaremos, generando implementaciones propias. Estas implementaciones conllevan una mayor carga de trabajo pero se busca aprender a leer y comprender artículos científicos complejos con más soltura. Obviamente intentaremos que las técnicas sean lo mas variadas posible. Desde predictores con diferentes niveles de contexto (Med, Ged, Gap) a transformadas tanto por filas como las Wavelets como por bloques como la DCT-tipo 2, etc... para poder apreciar diferentes variantes que afectan a la comparativa de CPU vs GPU.

2 METODOLOGÍA

Para que un benchmark sea valido se tiene que trabajar de forma coherente, introduciendo el mínimo ruido en los resultados. Y para ello necesitamos un entorno constante y los valores que mediremos.

2.1 Entorno

Este benchmark se llevará a cabo en un equipo con un procesador 7th Gen Intel(R) Core(TM) i7-8700, 6 cores con 2 threads por core, Arquitectura x86_64. Gráfica 1050 TI con el toolkit CUDA 11.5. Las pruebas se han hecho en un sistema operativo Linux-Ubuntu 22.04 TLS usando Visual Studio Code. Se han usado imágenes RGB con formato .png, con 24 de bits de profundidad es decir 8 bits por canal. Todo el código de la decorelación se ha escrito o bien en C++ o en CUDA. Mientras que los resultados y sus gráficas han sido analizadas con Python 3.10, en especial usando pandas.

Pero como queremos que la compresión sea sin pérdida, en muchas técnicas tenemos necesitamos almacenar el signo. Asi pues tendremos que usar datos de tipos signed short de 16 bits. Todo este tratmiento de lectura y cambio de tipos se ha hecho con ayuda de las librerias png++/png.hpp [2]. y OpenCV[3].

2.2 Pruebas y métricas

Para las pruebas utilizaremos principalmente los datasets Clic2020 y el KODAK. entre los dos se han utilizado unas 100 imágenes de múltiples tamaños. Adenás de imagenes puntuales de mayor tamaño

Se harán pruebas en 3 contextos principales: C++ base, OpenMP y Cuda. OpenMP es una API que nos permite explotar el parelelismo dentro de la CPU. Esto a partir del multithreading, que consiste en manejar múltiples threads manera simultanea lo que también genera condiciones de carrera que se deben tener en cuenta.

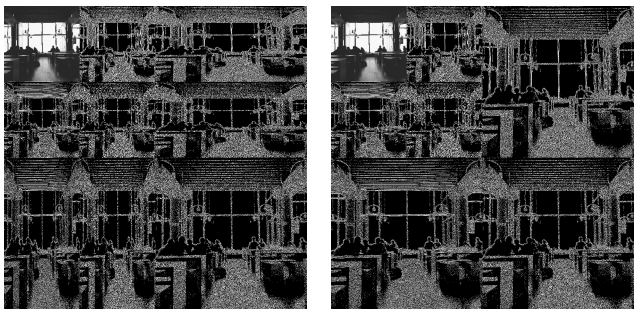
Una de las variables criticas que más sensibles al ruido que vamos a medir es el tiempo. Tanto el de compresión como el de descompresión. Tendremos que buscar el reloj más exacto para cada contexto. Para las pruebas en CPU usaremos el método omp_get_wtime(). Este nos da el wall clock time es decir el tiempo real de ejecución en lugar del tiempo. Otros relojes trabajan en base a los ciclos en CPU pero al usar OpenMP, si parelelizamos un código en 4 threads y tarda 1 segundo, un reloj de CPU diría que ha tardado 4 segundos en vez de unos porque cuenta ciegamente como si fuera secuencial. Para medir el tiempo del kernel de CUDA usaremos el CudaEvents.

También es importante decir que todas las pruebas se realizaran 3 veces y de estas se escogerá la que halla dado mejor rendimiento ya que es la que menos ruido del sistema a sufrido.

3 TRANSFORMADAS

3.1 Wavelet

Esta familia de transformadas nos permite hacer un análisis tanto de la frecuencia como ubicación de la información. Descomponemos la imagen en filas y columnas. Cada fila y columna sera un array. En cada nivel implica generar dos bandas dentro del array. Los valores Low (L) y High (H). En función del orden el que procesemos estos arrays veremos que cambia el display. Si primero aplicamos 5 niveles solo en forma de filas y luego los 5 niveles a las columnas, estamos usando un display estándar. Por otro lado, si a cada nivel intercalamos entre filas y columnas a cada nivel estamos usando una pirámide de Mallat entre otros.



(a) Estándar (b) Pirámide

Fig. 2: comparativa pirámide vs estándar

Esto genera diferentes accesos y puede ser una variable en el rendimiento. En este trabajo se han implementado los displays basándonos en el Handbook of data compresion [4] Así pues, los algoritmos de transformadas wavelet son recursivas, la salida del nivel actual se convierte en la entrada del nivel siguiente. Las wavelets necesitan que las filas y columnas tengan un tamaño que sea potencia de 2 (32, 64, 128, 256, 512, 1024...). Pero las imágenes reales tienen dimensiones como 1024×768 o 1920×1080, que no son potencias de 2 perfectas. Por ende, en vez de usar excelsos paddings de 0s, simplemente ignoraremos el ultimo elemento del array en caso de que la cantidad de elementos sea impar. Y en cada iteracions haremos el reajuste necesario.

3.1.1 Haar

La transformada de Haar es la más primitiva y sencilla de las wavelets. Se trabaja por pares. De cada pareja obtendremos un promedio (L) y una diferencia.

$$s = \frac{a+b}{2}$$

$$d = a - b$$

Obviamente cuando se divide entre dos con datos de tipo short, el valor se redondea y por tanto se podría producir una perdida. Por tanto, para poder hacerlo sin perdida, debemos de tener en cuenta un bit de paridad. Sabemos que $a - b$ era impar, $a + b$ sera impar. De esta forma podemos hacer que la versión más primitiva posible que sea sin perdida.

Pero esta transformada a evolucionado. Las versiones más recientes incorporan un lifting scheme. El lifting scheme es una técnica que nos permite descomponer las operaciones de la transformadas wavelet en pasos mas sencillos llamados, pasos de vacacional. De esta forma se reduce casi a la mitad el coste de operaciones aritméticas. Esto lo hace en tres pasos que se muestran en la Figura 3

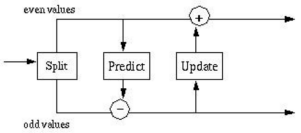


Fig. 3: Lifting scheme general

Además de este esquema en pasos, el lifting scheme también tiene la cualidad de ser in-place. Es decir que todos los cambios se pueden realizar sobre le mismo array sin memoria auxiliar. Eso si, al hacerlo in-place perdemos esta colocación por subbandas [L, L, L, L, H, H, H, H] y pasamos a un modelo [L, H, L, H, L, H, L, H]. Por lo que ademas del lifting scheme perse tendríamos que aplicar un paso de re ordenación. Para que este paso no tenga una complejidad excesiva tendremos que usar igualmente un array auxiliar.

El lifting scheme utilizado en este trabajo es una adaptación del libro Building Your Own Wavelets at Home [5] por lo que el código resultante se puede ejecutar en dos simples bucles for.

```
for(size_t i = 1; i < j; i+=2) {
    //predict
     array[i] -= array[i-1];
}

for(size_t i = 1; i < j; i+=2){
    //update
     array[i-1] += (array[i] >> 1);
}
```

Listing 1: código lifting scheme c++

Este metodo es mucho más sencillo de adaptar en CUDA. En esta implementación cada array sera un bloque y cada thread sera un elemento. De esta forma cargaremos toda la fila o columna a la memoria compartida y aplicar todos los niveles.

Al intentar llevarlo a CUDA evidentemente intentaremos procesar todas las filas y columnas que podamos simultáneamente. Podremos aprovechar que el array original esta en la memoria compartida y por tanto no existen dependencias.

3.1.2 Cohen–Daubechies–Feauveau 5/3 (CDF 5/3)

Esta es una version evolucionada de la transformada de Haar. la CDF 5/3 se usaen la rama lossless de JPEG 2000. Como es bastante posterior, las implementaciones ya empezaron con lifting schemes. En este caso, no se trabaja parejas sino que hay un reparto irregular. 5 valores L y 3 de H.

En vez de trabajar en pareja, el predict se hace de $2n + 1$ en base a $2n$ y $2n + 2$. Y el update se hace con estos tres valores y expandes a dos más.

$$y[2n+1] = x[2n+1] - \left[\frac{x[2n]+x[2n+2]}{2}\right]$$
$$y[2n] = x[2n] + [y[2n-1] + y[2n+1]]$$

Fig. 4: CDF 5/3 fifting scheme

La implementación usada en este trabajo es una adaptación hecha en base al trabajo de Gregoire Pau [6].

3.2 Discrete cosine transform (DCT)

3.2.1 Discrete cosine transform Tipo II

La DCT tipo II, usada en JPEG, es un cambio de paradigma. Desaparecen los niveles y el tratamiento por array. En el caso de las DCT se trabaja con bloques 8x8. En caso de que un bloque este incompleto, al ser solo de 1-8 columnas, es asequible hacer un padding de replicación en el que se dupliquen los valores de las ultimas n filas necesarias para completar el bloque. Estos bloques son matrices que contienen múltiples funciones de coseno. Estas funciones forman la matriz C y su transpuesta. Luego la matriz original se multiplica por estas nuevas matrices.

$$\begin{aligned} C_{ij} &= \frac{1}{\sqrt{m}} \quad i = 0 \quad j = 0 \ldots m-1 \\ &= \sqrt{\frac{2}{m}} \cos\frac{(2i+1)j\pi}{2m} \quad i = 1 \ldots m-1 \quad j = 0 \ldots m-1 \end{aligned}$$

Fig. 5: Calculo de la matriz C

Se puede deducir que esta técnica tiene perdida. Pero siendo que el origen y estructura de CUDA fueron ideas en un primer momento orientados a matrices es una prueba que merece la pena. Existen versiones reversibles de la DCT pero su uso es tan escaso que es preferible estudiar la versión estandarizada. Por lo tanto, en este caso los kernels los lanzaremos como bloques que contienen la C en lugar de filas independientes.

Para que el resultado sea valido como mínimo comprobaremos que sea virtual lossless. El PSNR (Peak signal-to-noise ratio) es una métrica que nos indica la relación entre la energía de una imagen y el ruido que haya acumulado. Este se mide en decibelios y a mayor sea su valor menos alteración existe. Cualquier valor de PSNR por encima de 40dB se considera virtual lossless ya que el ojo humano no

es capaz de detectar la perdida. Así pues si estamos por encima del umbral de los 40dB lo consideraremos valido para este trabajo.

3.3 Transformada inter-channel

Todas la transformadas hasta ahora han seguido los mismo pasos. Leer la imagen, descomponerla en los tres canales RGB. Y ya a cada canal aplicarle si transformada. Estas tecnificas que solo trabajan con uno canal a la vez se les denomina intracanal. Pero en una imagen existen técnicas complementarias al a valor que trabajan el espacio-color intercanal.

3.3.1 Reversible color transform (RCT)

Actualmente los sensores digitales cuentan con el doble photodiodos para el verde que para el rojo y el azul. Esto implica que el color verde suele ser el más limpio. La RCT explota que el verde es el canal con más información y coneirten nuestra imagen del espacio de colores RGB al YUV (Luminance, Blue-difference, Red-difference). El espacio YUV trabaja con la luminancia (brillo de luz en la imagen) de la imagen para la componente Y. La segunda parte es la crominancia. Este componente es menos sensible al ojo huma y contiene la saturación y tonalidad. Tal como hemos explicado, el verde tiene más luminancia y en la crominancia se obtiene directamente en función de este verde.

$$\begin{pmatrix} Y_r \\ U_r \\ V_r \end{pmatrix} = \begin{pmatrix} \left\lfloor \frac{R+2G+B}{4} \right\rfloor \\ R-G \\ B-G \end{pmatrix}$$

Fig. 6: Cambio de espacio de la transformada RCT

Como las dependencias son de canal a canal se denomina problema súper paralelizable CUDA. Solo tenemos los limites de cantidad de elementos por thread pero de por si, todos los elementos se podrían procesar en simultaneo tanto en el forward como en el reverse.

3.3.2 YCoCg-R

Esta técnica es la evolución de la RCT. Ahora estamos pasando del espacio de color RGB al YCoCg (luminance + offset orange + offset green). La luminancia es exactamente igual que en el caso anterior. La evolución viene en la crominancia. En este caso ya no estamos pivotando en función del verde. El offset orange es la diferencia entre el red y el blue. Mientras el offset green si que es la diferencia del orange con el verde. De esta forma se gana independencia del verde y se suaviza la predicción manteniendo un código súper paralelizable.

4 PREDICTORES

Como hemos comentado los predictores generar una predicción y un error. Este error es el que enviaremos y queremos que tenga los valores mas bajos posibles. Por lo general, las imágenes tiene cambios de color bastante suaves. Así pues, cuanto más cerca se encuentran dos pixeles más seguramente tendrán valores parecidos. Por lo tanto, es lógico querer hacer una predición en base a los vecinos inmediatos. Pero tenemos que tener en cuenta que a veces el cambio de fila a fila es muy grande mientras que el de columna a columna es muy suave o viceversa. Esto implica que cuando tratamos con múltiples vecinos tenemos que considerar cual de ellos no favorece más. Esto se define como predictores adaptativos y son en lo que se centrara este trabajo.

4.1 Median edge detection (MED)

El más sencillo de todos. Este forma parte del predictor LOCO-I usado por JPEG-LS. El MED se basa en el vecino west (izquierdo), north(superior) y north-west (esquina de los anteriores):

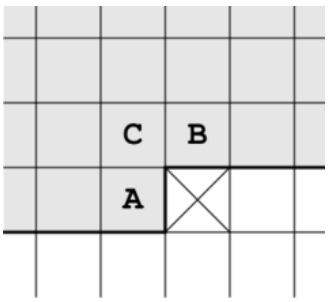


Fig. 7: Vecinos causales del MED

De estos tendremos que elegir el de menor diferencia:

$$\hat{x}_{i+1} \triangleq \begin{cases} \min(a,b) & \text{if } c \geq \max(a,b) \\ \max(a,b) & \text{if } c \leq \min(a,b) \\ a+b-c & \text{otherwise.} \end{cases}$$

Fig. 8: Condiciones de predición MED

Esto implica que la primera fila y columna tiene que estar ya procesadas antes de aplicar el predictor en si. Como el rango de valores de la imagen es de 255, usaremos el 128 como predición inicial para todas estas filas y columnas "artificiales".

Este tipo de predictores son un problema en OpenMP. Como queremos mantener el método original tenemos un impedimento con los predictores. Existe una dependencia de lectura. Para obtener $I(x)$ necesitamos $I(x-1)$. Por lo que no podemos paralelizar con openMP la descompresión de este tipo de predictores si queremos mantener el modelo.

Por otro lado en CUDA, si que podemos realizar un reverse paralelo. Esto lo haremos usando una estrategia por bloques (Tiling) combinado con un patrón stensil.

4.2 Gradient-adaptative predictor (GAP)

Este es bastante más complejo que el MED. Usado en el predictor CALIC, el GAP pasa de tener 3 vecinos causales a 7.

En este caso ya no solo miramos vecinos individuales sino que agrupamos por gradientes horizontales (g_h) y gradientes verticales (g_v).

Como se puede deducir ahora necesitamos preparar dos filas y dos columnas antes de poder hacer predicción real. Ademas, como NE y NNE están ya en la siguiente columna, tenemos que tratar la ultima columna de forma especial. Haremos la predicción solo con los otros 5 vecinos.

Este predictor es especialmente difícil en CUDA como se puede ver en anexo A2 el algoritmo de predicción del GAP tiene bastantes condicionales. CUDA no gestiona bien los condicionales.

Si nuestro condicional es para dividir números entre pares e impares, CUDA los separara y ejecutara primero solo los de un tipo aunque no existiera ninguna dependencia entre ellos. Lo cual significa que a más pequeños sean los grupos que generamos más tiempo malgastaremos en la GPU.

Una posible mejora seria usar algún barrido previo de la imagen. De esta forma podríamos lanzar kernels más acotados y con menos condicionales.

4.3 Gradient edge detection (GED)

El GED es el más reciente de los 3 por general se utiliza solo dentro de la literatura académica. Este método busca un punto medio entre rendimiento y complejidad. Este usa 5 vecinos. Se puede ver que preparamos dos filas y dos columnas como en el GAP pero no tenemos que darle ningún trato especial a la ultima columna. En este caso la predicción incluye un threshold artificial (T) que en el caso de este trabajo a sido 40.

5 IMPLEMENTACIONES EN CUDA

5.1 Wavelets

Para la implementación en GPU se ha mantenido el mismo patrón de paralelismo tanto para la Haar como para la CDF 5/3. El enfoque consiste en lanzar un bloque CUDA por cada fila o columna y asignar un thread por cada elemento dentro de esa fila o columna. Con este esquema la rejilla (grid) representa el conjunto de filas o columnas, mientras que cada bloque procesa una de ellas de forma independiente.

```
dim3 grid(rows);
int threads =(((j+31)/32 )*32);
```

Listing 2: Código del esquema de assignación de threads Haar

El número de threads se ha ajustado al múltiplo de 32 (tamaño de warp en CUDA) para maximizar la opcupación y evitar hilos inactivos. En CUDA un warp es un grupo de 32 threads que se ejecutan simultáneamente dentro de un Streaming Multiprocessor (SM) bajo el modelo SIMT (Single Instruction Multiple Threads) donde cada thread ejecuta la misma instrucción sobre distintos datos.

5.1.1 Transformada Haar

Uso de la memoria compartida. Cada bloque reserva memoria compartida para para acelerar operaciones y reducir los costosos accesos a la memoria global. En concreto emplearemos dos buffers. El primero contendrá los datos originales y el segundo almacena los resultados parciales. El uso de la memoria compartida permite reutilizar los datos dentro del bloque evitando accesos costosos a la memoria global.

Sincronización entre threads. Tras cada operación elemental del lifting scheme se usa __syncthreads(); para asegurar que todos los elementos del bloque han completado su parte antes de continuar. Esto es fundamental para evitar las condiciones de carrera dado que los coeficientes low-pass y high-pass dependen de elementos de elementos vecinos accesibles dentro del mismo bloque, por lo que es necesaria a pesar de empeorar el rendimiento.

Problemas encontrados: Durante la implementación se identificaron varios diversos problemas que introducían ruido en el resultado.

Bloques incompletos. Algunas imágenes no tiene dimensiones múltiples de 32 por lo que se generaban accesos fuera de rango. Esto se soluciono incorporando condiciones de control if($idxElement < j$).

Edge cases. Operaciones como sh[idxElement] -= sh[idxElement-1]; requerían de condiciones especificas, $idxElement > 0$, para evitar desbordamientos que en la versión CPU no se daban.

Sincronización insuficiente. La omisión de la instrucción __syncthreads(); en algunos de los pasos intermedios producía resultados inconsistentes.

5.1.2 Transformada CDF 5/3

Las diferencias específicas de la CDF 5/3 que han requerido atención especial son:

Condiciones más estrictas. La CDF requiere de accesos tanto al vecino anterior como posterior por lo que tenemos bordes mas exigentes. Esta exigencia adicional incluye también nuevas condiciones de control como idx == 0, idx == j-1.

Uso de la memoria compartida. Aunque las operaciones básicas solo requieren dos elementos extras, se reservan 4 elementos de padding para manejar casos especiales del borde derecho (j % 2 == 0) y garantizar alineación de memoria, resultando en +8 shorts totales (2 buffers × 4 padding). Por lo que la memoria compartida la de Haar +8.

5.2 Transformada DCT

La DCT-II sigue un esquema de paralelismo 2D. Divide la imagen en bloques fijos de 8x8 asignando un thread por elemento dentro del bloque 8x8 y trabajando de manera matricial.

```
dim3 threads(8,8);
dim3 blocks(columns/8, rows/8);
```

Listing 3: Código del esquema de assignación de threads, DCT

Uso de la memoria compartida. Esta técnica hace un uso especialmente intensivo de la memoria compartida ya que se declaran 3 matrices 8x8 en la memoria compartida del bloque. Debido a esto, vuelven a ser criticas las instrucciones __syncthreads(); después de cada multiplicación para mantener la integridad del bloque.

Problemas encontrados: El principal problema fue la gestión de los indices. Se debe de ser extremadamente cuidadoso con los indices de la matriz, threadIdx.x y threadIdx.y además de sus respectivas combinaciones con blockIdx.x y

blockIdx.y supuso un reto inicial debido a la inexperiencia con indexación 2D en CUDA.

También es una limitación el tipo de dato. Como ha expuesto previamente, esta es la única técnica con la que usamos datos de tipo double. Esto implica que son operaciones de punto flotante y por ende aumentan notoriamente carga de trabajo, tanto que el cuello de botella es la propia aritmética.

5.3 Predictores

Para los predictores se ha adoptado un esquema de paralelismo completamente diferente al de las wavelets o DCT. En lugar de bloques por filas, o paralelismo 2D, se asigna un thread por pixel en una rejilla 1D que recorre toda la imagen linealmente.

```
int pixels = rows * cols;
dim3 threads(256);
dim3 blocks((pixels + 255)/256);
```

Listing 4: Código del esquema de assignación de threads, predictores

Uso global de la memoria. A diferencia de las wavelets, no se usa memoria compartida.Cada thread lee directamente de los arrays globales array[], prediction[] y error[]. Siendo la reconstrucción trivial: array[idx] = error[idx] + prediction[idx];

Función especial. Todos los predictores usan __device__ __forceinline__, forzando la expansión inline del código en el kernel. Esto elimina el overhead de la llamada a la función y permite optimizaciones del compilador nvcc, permitiendo ,entre otra optimizaciones, una mejor assignación de registros.

La principal diferencia entre implementaciones es la complejidad computacional. GAP requiere 7 vecinos y condiciones de control más estrictas, frente a los 3 vecinos del MED.

Problemas encontrados.
Coalescencia de memoria. Los accesos a vecinos dispersos (izquierda-2, diagonal+1, arriba+1) generan accesos no coalescentes. En CUDA, un warp de 32 threads consecutivos que acceden a direcciones no contiguas tiene un coste significativamente major.

Divergencia de warp. Los condicionales anidados crean divergencia de ejecución. En el modelo SIMT de CUDA, cuando un warp encuentra un if(), ejecuta las ramas en serie: primero todos los threads que cumplen la condición, luego todos los demás. GAP, por ejemplo, (8 condiciones) duplica la divergencia.

5.4 Transformadas intercanal

Las transformaciones intercanal YCoCg y RCT utilizan el esquema de paralelismo 1D más simple: un thread por píxel en rejilla lineal, similar a predictores pero sin dependencias espaciales.

```
int threads = 1024;
int blocks = ((imageSize + threads
    -1)/threads);
```

Listing 5: Código del esquema de assignación de threads, inter

Estas transformadas presentan un paralelismo trivial. Cada thread trata el RGB de un pixel concreto independientemente. No necesita ni sincronización, ni memoria compartida, ni presenta divergencia, es el caso más sencillo.

6 RESULTADOS

6.1 Transformadas:

6.1.1 Haar:

La transformada Wavelet de Haar que presenta dependencia por filas a confirmado lo esperado. Todos las versiones base han presentado un rendimiento casi igual. Por tanto los datos indican que es indiferente si seguimos un display de pirámide de Mallat o bien un display estándar en cuanto a la compresión. De todos modos cabe decir que si queremos aplicar más de dos niveles, el método en pirámide empeora más rápido que el el estándar.

La transformada Wavelet de Haar, que presenta dependencia estricta por filas, confirma las expectativas teóricas. Todas las versiones base exhiben un rendimiento de compresión prácticamente idéntico (BPP_w L1 = 0.157 bits/píxel constante, Tabla 1), demostrando que es indiferente elegir entre display de pirámide de Mallat o estándar para la compresión final.

TAULA 1: RESULTADOS BASE HAAR WAVELET CON DELTAS VS L1

Config	Versión	Nivel	BPP_w	+BPP vs L1	t_{comp}	+Tiempo vs L1	Throughput
Std_Sch_base_haar	Std+Sch	L1	0.157	–	37.78	–	0.094
Std_Sch_base_haar	Std+Sch	L2	0.269	+71.3%	41.58	+10.0%	0.084
Std_Sch_base_haar	Std+Sch	L3	0.305	+94.3%	43.03	+13.9%	0.081
Std_Sch_base_haar	Std+Sch	L4	0.314	+100.0%	44.37	+17.5%	0.078
Std_Sch_base_haar	Std+Sch	L5	0.317	+101.9%	44.97	+19.1%	0.076
Std_nSch_base_haar	Std+nSch	L1	0.157	–	36.67	–	0.106
Std_nSch_base_haar	Std+nSch	L2	0.269	+71.3%	39.55	+7.9%	0.100
Std_nSch_base_haar	Std+nSch	L3	0.305	+94.3%	41.62	+13.5%	0.096
Std_nSch_base_haar	Std+nSch	L4	0.314	+100.0%	42.91	+17.0%	0.093
Std_nSch_base_haar	Std+nSch	L5	0.316	+101.3%	43.68	+19.2%	0.092
Pyr_Sch_base_haar	Pyr+Sch	L1	0.157	–	37.86	–	0.094
Pyr_Sch_base_haar	Pyr+Sch	L2	0.269	+71.3%	46.33	+22.4%	0.077
Pyr_Sch_base_haar	Pyr+Sch	L3	0.307	+95.5%	48.55	+28.3%	0.074
Pyr_Sch_base_haar	Pyr+Sch	L4	0.318	+102.5%	49.25	+30.1%	0.073
Pyr_Sch_base_haar	Pyr+Sch	L5	0.321	+104.5%	49.59	+31.0%	0.072
Pyr_nSch_base_haar	Pyr+nSch	L1	0.157	–	37.46	–	0.103
Pyr_nSch_base_haar	Pyr+nSch	L2	0.269	+71.3%	46.85	+25.1%	0.083
Pyr_nSch_base_haar	Pyr+nSch	L3	0.306	+94.9%	49.00	+30.8%	0.079
Pyr_nSch_base_haar	Pyr+nSch	L4	0.317	+101.9%	49.95	+33.4%	0.078
Pyr_nSch_base_haar	Pyr+nSch	L5	0.320	+103.8%	49.48	+32.1%	0.079

t_{comp} en ms. +Tiempo: aumento tiempo comp. vs L1. +BPP: aumento BPP_w vs L1.

No obstante, para más de dos niveles de descomposición, el método piramidal empeora más rápidamente que el estándar debido a penalizaciones de localidad de caché. La diferencia entre lifting scheme y esquema estándar resulta marginal con la implementación actual, siendo la escalabilidad temporal entre display estándar y pirámide la principal discriminante.

Finalmente, se observa un incremento considerable del rendimiento hasta el nivel 3 (L1→L3: +13-28%), con ganancias reducidas notablemente a partir del nivel 4 (saturación wavelet). Por tanto, tres niveles de transformada equilibran carga computacional y rendimiento tanto en compresión como en paralelización.

Los resultados OpenMP confirman la que el principal problema proviene del overhead. Hasta el nivel L3 sigue siendo beneficioso pero en los niveles superiores el overhead scheduling domina y se presenta un estancamiento similar al de las versiones base.

TAULA 2: RESULTADOS OPENMP STD+SCH HAAR WAVELET CON DELTAS VS L1

Config	Nivel	BPP_w	+BPP vs L1	t_{comp}	+Tiempo vs L1	t_{decomp}	Throughput
Std_Sch_openMP_haar	L1	0.157	–	15.76	–	15.75	0.233
Std_Sch_openMP_haar	L2	0.269	+71.3%	21.38	+35.6%	21.30	0.172
Std_Sch_openMP_haar	L3	0.305	+94.3%	26.94	+70.9%	26.10	0.138
Std_Sch_openMP_haar	L4	0.314	+100.0%	33.30	+111.4%	31.91	0.113
Std_Sch_openMP_haar	L5	0.317	+101.9%	38.75	+145.8%	36.05	0.098

$t_{comp/decomp}$ en ms. +Tiempo: aumento tiempo comp. vs L1. +BPP: aumento BPP_w vs L1.

Al implementar la versión en CUDA, se ha cometido el error de adaptar el demasiado fielmente la versión de CPU. Esto ya que se han tratado los canales RGB de manera individual y iterativa. Esto a generado la necesidad del uso de múltiples kernels. Estos kernels, relativamente pequeños, provocan un desfalco del rendimiento.

TAULA 3: RESULTADOS CUDA STD+SCH HAAR WAVELET CON DELTAS VS L1

Config	Nivel	BPP_w	+BPP vs L1	t_{comp}	+Tiempo vs L1	t_{decomp}	Throughput
Std_Sch_cuda_haar	L1	0.157	–	0.739	–	0.647	5.294
Std_Sch_cuda_haar	L2	0.269	+71.3%	18.97	+2466%	18.27	0.197
Std_Sch_cuda_haar	L3	0.305	+94.3%	23.04	+3019%	22.44	0.161
Std_Sch_cuda_haar	L4	0.314	+100.0%	26.48	+3483%	25.52	0.141
Std_Sch_cuda_haar	L5	0.317	+101.9%	28.36	+3736%	27.40	0.132

$t_{comp/decomp}$ en ms. +Tiempo: aumento tiempo comp. vs L1. BPP_w=0 por error numérico.

Aun y así cabe destacar que la aplicación de un solo nivel demuestra el potencial de la técnica con un destacable speed up de x51.1, Tabla 1), y a partir del decaimiento hace que el rendimiento de OpenMP y CUDA sean parecidos.

TAULA 4: SPEEDUP NORMALIZADO (BASE=1X) STD+SCH HAAR

Nivel	Base	OpenMP	CUDA
L1	x1.00	x2.40	x51.1
L2	x1.00	x1.95	x2.19
L3	x1.00	x1.60	x1.87
L4	x1.00	x1.33	x1.68
L5	x1.00	x1.16	x1.59

Speedup = t_{base}/t_{impl}.

6.2 CDF 5/3

A diferencia de Haar, CDF 5/3 predictiva requiere accesos estrictamente contiguos para predicción $p(n) = f(p(n-1), p(n-2))$. Los accesos no continuos del modelo piramidal perjudican fuertemente la transformada (Tabla 5):

TAULA 5: RESULTADOS BASE CDF 5/3 CON DELTAS VS L1

Config	Versión	Nivel	BPP_w	+BPP vs L1	t_{comp}	+Tiempo vs L1	Throughput
Std_nSum_base_cdf	Std+nSum	L1	0.200	–	48.03	–	0.077
Std_nSum_base_cdf	Std+nSum	L2	0.304	+52.1%	57.16	+19.0%	0.064
Std_nSum_base_cdf	Std+nSum	L3	0.334	+67.1%	61.11	+27.3%	0.060
Std_nSum_base_cdf	Std+nSum	L4	0.341	+70.1%	63.11	+31.4%	0.058
Std_nSum_base_cdf	Std+nSum	L5	0.342	+71.0%	63.79	+32.8%	0.058
Pyr_Sum_base_cdf	Pyr+Sum	L1	0.185	–	47.48	–	0.071
Pyr_Sum_base_cdf	Pyr+Sum	L2	0.281	+52.0%	59.03	+24.3%	0.057
Pyr_Sum_base_cdf	Pyr+Sum	L3	0.309	+67.4%	61.79	+30.1%	0.055
Pyr_Sum_base_cdf	Pyr+Sum	L4	0.316	+71.2%	62.14	+30.9%	0.055
Pyr_Sum_base_cdf	Pyr+Sum	L5	0.318	+72.2%	62.59	+31.8%	0.054
Std_Sum_base_cdf	Std+Sum	L1	0.185	–	48.19	–	0.070
Std_Sum_base_cdf	Std+Sum	L2	0.264	+42.9%	56.54	+17.3%	0.058
Std_Sum_base_cdf	Std+Sum	L3	0.281	+51.8%	60.77	+26.1%	0.053
Std_Sum_base_cdf	Std+Sum	L4	0.279	+50.7%	62.97	+30.7%	0.051
Std_Sum_base_cdf	Std+Sum	L5	0.274	+48.3%	64.97	+34.8%	0.050
Pyr_nSum_base_cdf	Pyr+nSum	L1	0.200	–	55.39	–	0.065
Pyr_nSum_base_cdf	Pyr+nSum	L2	0.309	+54.5%	68.57	+23.8%	0.053
Pyr_nSum_base_cdf	Pyr+nSum	L3	0.344	+72.1%	71.85	+29.7%	0.050
Pyr_nSum_base_cdf	Pyr+nSum	L4	0.355	+77.3%	72.54	+31.0%	0.050
Pyr_nSum_base_cdf	Pyr+nSum	L5	0.357	+78.6%	72.80	+31.5%	0.050

t_{comp} en ms. +Tiempo: aumento tiempo comp. vs L1. +BPP: aumento BPP_w vs L1.

Los resultados evidencian que el uso de OpenMP en esta implementación no ha sido satisfactorio, debido a una acumulación de factores. Entre las posibles causas se encuentra la aparición de false sharing, donde algunos threads acceden y escriben en posiciones de memoria indebidas, afectando negativamente el rendimiento de la caché.

TAULA 6: RESULTADOS OPENMP STD+NSUM CDF 5/3 CON DELTAS VS L1

Config	Versión	Nivel	BPP_w	+BPP vs L1	t_{comp}	+Tiempo vs L1	Throughput
Std_nSum_OpenMP_cdf	Std+nSum+OpenMP	L1	0.200	–	105.68	–	0.035
Std_nSum_OpenMP_cdf	Std+nSum+OpenMP	L2	0.304	+52.1%	140.17	+32.6%	0.027
Std_nSum_OpenMP_cdf	Std+nSum+OpenMP	L3	0.334	+67.1%	176.74	+67.3%	0.021
Std_nSum_OpenMP_cdf	Std+nSum+OpenMP	L4	0.341	+70.1%	231.86	+119.4%	0.016
Std_nSum_OpenMP_cdf	Std+nSum+OpenMP	L5	0.342	+71.0%	261.56	+147.5%	0.014

t_{comp} en ms. +Tiempo: aumento tiempo comp. vs L1. +BPP: aumento BPP_w vs L1.

Los resultados obtenidos para la versión en CUDA presentan un comportamiento similar al observado con la wavelet de Haar. Es muy probable que este rendimiento se deba, de igual forma, a un uso inadecuado de los kernels.

TAULA 7: RESULTADOS CUDA STD+NSUM CDF 5/3 CON DELTAS VS L1

Config	Versión	Nivel	BPP_w	+BPP vs L1	t_{comp}	+Tiempo vs L1	Throughput
Std_nSum_cuda_cdf	Std+nSum+CUDA	L1	0.200	—	1.33	–	2.761
Std_nSum_cuda_cdf	Std+nSum+CUDA	L2	0.304	+52.1%	26.67	+1905%	0.137
Std_nSum_cuda_cdf	Std+nSum+CUDA	L3	0.334	+67.1%	38.25	+2775%	0.098
Std_nSum_cuda_cdf	Std+nSum+CUDA	L4	0.341	+70.1%	44.28	+3229%	0.084
Std_nSum_cuda_cdf	Std+nSum+CUDA	L5	0.342	+71.0%	48.49	+3456%	0.078

A pesar del decaimiento en el rendimiento, se observa que la mejora en la memoria L1 es considerablemente menor que la obtenida con la wavelet de Haar. Esto resulta comprensible si se considera que el esquema CDF 5/3 es significativamente más complejo, mientras que CUDA está diseñado para optimizar la ejecución masiva de operaciones simples, lo que explica el menor aprovechamiento de la memoria L1 en este caso.

TAULA 8: SPEEDUP OPENMP Y CUDA VS BASE (STD+NSUM CDF 5/3, L1-L5)

Nivel	Base t_{comp}	OpenMP t_{comp}	Speedup OpenMP	CUDA t_{comp}	Speedup CUDA	Mejor Speedup
L1	48.03	105.68	x0.45	1.33	x36.1	CUDA
L2	57.16	140.17	x0.41	26.67	x2.14	CUDA
L3	61.11	176.74	x0.35	38.25	x1.60	CUDA
L4	63.11	231.86	x0.27	44.28	x1.43	CUDA
L5	63.79	261.56	x0.24	48.49	x1.32	CUDA
Media	**58.64**	**183.20**	**0.33×**	**31.80**	****1.84×****	**CUDA**

Speedup = $t_{base}/t_{paralelo}$. OpenMP empeora rendimiento.

6.2.1 DCT:

En el caso de la DCT, los resultados obtenidos son coherentes con lo esperado. OpenMP presenta un peor rendimiento debido a causas similares a las observadas en CDF 5/3, con un impacto aún mayor del false sharing, ya que todos los threads acceden a la misma matriz de cosenos C.

Por otro lado, la implementación en CUDA logra una mejora modesta al reutilizar esta misma matriz C. Una optimización potencial para explotar mejor los warps consistiría en replicar la matriz C (8×8) cuatro veces, formando una matriz expandida de 32×32 que facilite el coalescing de memoria y la ejecución con warps.

TAULA 9: MEJORES RESULTADOS DCT

Predictor	Config	Impl.	n_{reps}	BPP_w	BPP	t_{comp}	t_{decomp}	t_{total}	Throughput
	Base-best	base	1	0.357	2.446	14.46	14.51	28.97	0.253
DCT	OpenMP-best	OpenMP	1	0.357	2.446	18.30	15.82	34.12	0.215
	CUDA-best	CUDA	1	–	–	8.91	8.37	17.28	0.425

$t_{comp/decomp/total}$: tiempos compresión/descompresión/total (ms). BPP_w: bits por píxel ponderado. Throughput en MB/s.

TAULA 10: SPEEDUP (BASE=1X VS OPENMP/CUDA)

Predictor	Base	OpenMP	CUDA
SpeedUP	1.00	x0.84	x1.67

Speedup = $t_{\text{base}}/t_{\text{impl}}$. Base normalizado a 1x.

6.2.2 Transformadas YCO y RCT

En estas transformadas colorspace (RCT e YCoCg), CUDA muestra su máximo potencial con aceleraciones de 2.46x y 3.45x respectivamente, mientras OpenMP empeora consistentemente debido al overhead de sincronización en operaciones punto-a-punto simples.

TAULA 11: MEJORES RESULTADOS RCT Y YCO

Predictor	Config	Impl.	n_{reps}	BPP_w	BPP	t_{comp}	t_{decomp}	t_{total}	Throughput
RCT	Base-best	base	1	0.149	1.051	0.00	1.80	1.80	4.08
	OpenMP-best	OpenMP	1	0.149	1.051	0.02	2.16	2.18	3.37
	CUDA-best	CUDA	1	–	–	0.35	0.38	0.73	10.08
YCO	Base-best	base	1	0.151	1.064	1.09	1.19	2.28	3.22
	OpenMP-best	OpenMP	1	0.151	1.064	2.00	2.35	4.35	1.69
	CUDA-best	CUDA	1	–	–	0.33	0.33	0.66	11.09

$t_{\text{comp/decomp/total}}$: tiempos compresión/descompresión/total (ms). BPP_w: bits por píxel ponderado. Throughput en MB/s.

Ambas transformadas son operaciones intercanal sin dependencias de datos, lo que las hace perfectamente paralelizables y extremadamente rápidas. De hecho, el forward de RCT, que incluye una multiplicación adicional respecto al reverse, es tan eficiente en CPU que resulta más rápido que la instanciación del kernel CUDA.

Por el contrario, el reverse —con operaciones más simples (sumas y restas)— permite a la GPU explotar su paralelismo masivo, logrando una clara ventaja competitiva.

TAULA 12: MEJORAS (BASE=1X VS OPENMP/CUDA)

Transformada	Base	OpenMP	CUDA
RCT	1.00	x0.82	x2.46
YCO	1.00	x0.52	x3.45

Speedup = $t_{\text{base}}/t_{\text{impl}}$. Base normalizado a 1x.

6.3 Predictores:

En los resultados de los predictores MED, GAP y GED observamos unos grandes resultado de entre x22-x51 veces de speed up, mientras que OpenMP muestra resultados mixtos, potencialmente debido a una race condition.

TAULA 13: MEJORES RESULTADOS PREDICTORES

Predictor	Config	Impl.	n_{reps}	BPP_w	BPP	t_{comp}	t_{decomp}	t_{total}	Throughput
MED	Base-best	base	1	0.352	2.418	12.04	59.45	71.49	0.103
	OpenMP-best	OpenMP	1	0.352	2.418	16.91	59.37	76.28	0.096
	CUDA-best	CUDA	1	–	–	0.94	0.88	1.82	3.99
GAP	Base-best	base	1	0.359	2.467	73.56	59.75	133.31	0.055
	OpenMP-best	OpenMP	1	0.359	2.467	43.74	73.79	117.53	0.062
	CUDA-best	CUDA	1	–	–	1.59	0.93	2.51	2.89
GED	Base-best	base	1	0.329	2.256	14.75	36.58	51.33	0.143
	OpenMP-best	OpenMP	1	0.329	2.256	30.49	37.02	67.51	0.109
	CUDA-best	CUDA	1	–	–	1.27	0.97	2.24	3.25

$t_{\text{comp/decomp/total}}$: tiempos compresión/descompresión/total (ms). BPP_w: bits por píxel ponderado. Throughput en MB/s.

Contrariamente a la intuición inicial, el mayor speedup lo presenta GAP —precisamente el predictor con mayor complejidad computacional—. Esto se explica porque CUDA está optimizado para procesar grandes volúmenes de trabajo: al partir de un tiempo base tan elevado (133 ms), incluso una implementación imperfecta logra la mayor mejora relativa (x53.1), superando a MED y GED que parten de bases más rápidas.

TAULA 14: SPEEDUPS TIEMPO TOTAL (BASE=1X VS OPENMP/CUDA)

Predictor	Base	OpenMP	CUDA
MED	1.00	x0.94	x39.3
GAP	1.00	x1.13	x53.1
GED	1.00	x0.76	x22.9

Speedup = $t_{\text{base}}/t_{\text{impl}}$. Base normalizado a 1x.

7 CONCLUSIONES

Este TFG evaluó exhaustivamente la paralelización de algoritmos de compresión sin pérdidas mediante OpenMP y CUDA, analizando wavelets (Haar, CDF 5/3), DCT, colorspace (RCT, YCoCg) y predictores (MED, GAP, GED).

El trabajo demuestra que CUDA tiene grandes resultados en predictores y transformadas colorspace fácilmente, premiando especialmente la granularidad. También se muestra que CUDA tiene un potencial similar para aplicar-sé en Wavelets y DCT pero con complicaciones adicionales. Asimismo se ha probado que CUDA tiene un buen rendimiento con imágenes RGB de apenas entre 2-10 MBs. Por lo tanto, el tamaño necesario para que CUDA sea significativo es menor del esperado.

Se concluye que OpenMP es contraproducente. Con un Speed Up general de alrededor de x0.8 principalmente debido al overhead y la sincronización demostrando que es inadecuado en algoritmos con dependencias locales. Esto le da más valor a CUDA.

Para finalizar, quería destacar que este trabajo constituye una base para futuros proyectos como el pipeline de diversos kernels. Otros proyectos podrían incluir adaptar el proyecto a datasets industriales o puramente médicos. Quedan pendientes la fusión de los kernels de las wavelets, la ampliación de la matriz de cosenos en la DCT y otras líneas de desarrollo.

8 USO DE LA IA

En este trabajo, el uso de la IA se limita principalmente a un rol de apoyo. Considerando que a día de hoy todos los navegadores integran IA, solo con realizar una búsqueda ya obtenemos respuestas generadas por IA por lo que su uso es inevitable. Pero, aparte de esto uso indirecto, se ha usado principalmente Perplexity. Este modelo se ha usado para dos funciones principalmente. La de debug, por ejemplo un simple signo en el código que pasa desapercibido a la vista y se propaga a todo el código.
La segunda, la investigación. Perplexity esta específicamente disenyado para buscar documentación en sitios como Google Schoolar por lo que puede ser útil a la hora de buscar fuentes que luego se estudian y se valora si son útiles para el benchmark. Por último, se han usado modelos para funciones secundarias. Por ejemplo, ayudando a construir la tabla en Latex, mejorando el redactado o para comprobar posibles dependencias de la gráfica y otras.

AGRADECIMIENTOS

Quiero expresar mi profunda gratitud a todas las personas que han contribuido a la realización de este trabajo.

- A mi tutor, Xavier Fernández Mellado, por su valiosa dirección, sus comentarios constructivos y su paciencia durante todo el proceso a pesar de todos los inconvenientes que le haya causado.
- A mis padres y hermana, por su ánimo y apoyo en los momentos más intensos del trabajo.
- A los profesores de la Universidad Autónoma de Barcelona por su enriquecedora enseñanza y disponibilidad a este proyecto.

REFERÈNCIES

[1] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2025.

[2] G. E. Schalnat, A. Dilger, J. Bowler, G. Randers-Pehrson, and C. Truta, "libpng." `https://www.libpng.org/pub/png/libpng.html`, 2025. Accessed: Nov. 8, 2025.

[3] G. Bradski, "The opencv library." `https://opencv.org/`, 2000. Accessed: Nov. 8, 2025.

[4] D. Salomon and G. Motta, *Handbook of Data Compression*. Springer, 4 ed., 2007.

[5] W. Sweldens and P. Schroder, "Building your own wavelets at home," in *Wavekets in computer graphics*, ACM SIGGRAPH, 1996.

[6] V. Kirilchuk, "Cdf 5 3 discrete wavelet transform." Github, 2025. Accessed: Dec. 13, 2025.

[7] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*. University of Illnois Press, 4 ed., 1948.

[8] I. Kaplan, "Lossless wavelet compression." `http://bearcave.com/misl/misl_tech/wavelets/compression/index.html`, 2002. Accessed: Nov. 8, 2025.

[9] C. of Longitudinal Integrated Clerkships, "Clic2021." `https://data.vision.ee.ethz.ch/cvl/clic/professional_train_2020.zip`, 2025. Accessed: Nov. 8, 2025.

[10] S. Hegde and S. Ramachandran, "Implementation of cdf 5/3 wavelet transform," *International Journal of Electrical, Electronics and Data Communication*, 2014.

[11] A. Cohen, I. Daubechies, and J.-C. Feauveau, "Biorthogonal bases of compactly supported wavelets," *Communications on Pure and Applied Mathematics*, 1992.

[12] A. Bovik, ed., *Handbook of Image and Video Processing*. Academic Press, 1 ed., 2000.

[13] University College London, "An introduction to openmp." Research Computing with C++, 2016. Accessed: Dec. 13, 2025.

[14] Princeton Research Computing, "Learning resources: Openmp." Princeton University, 2016. Accessed: Dec. 13, 2025.

[15] G. S. Marcelo J. Weinberger and G. Sapiro, "Loco-i: A low complexity, context-based, lossless image compression algorithm," *IEEE Trans. Image Process.*, 1996.

[16] N. A. Giridhar Mandyam and N. j Magotra, "A dct-based scheme for lossless image compression," *IST/SPIE Symposium on Electronic Imaging*, 1995.

[17] X. Wu and N. Memon, "Calic-a context based adaptive lossless image codec," *Institute of Electrical and Electronics Engineers*, 1996.

[18] A. Avramović and A. Avramović, "Gradient edge detection predictor for image lossless compression," *52nd International Symposium ELMAR*, 2010.

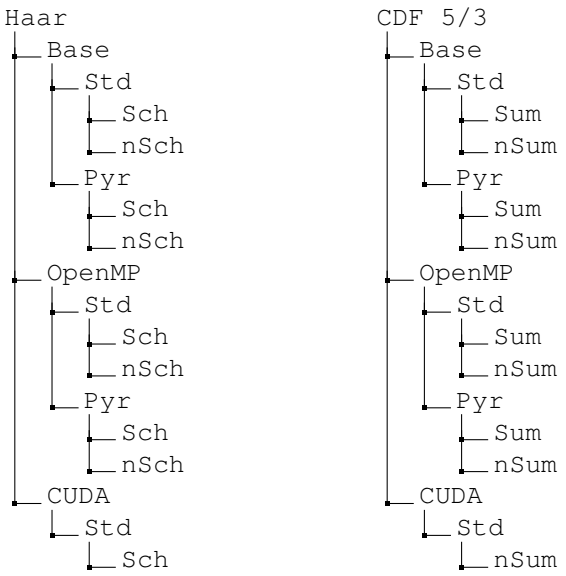
[19] V. S. Van, "Image compression using burrows-wheeler transform," Master's thesis, Helsinki University of Technology, 2009.

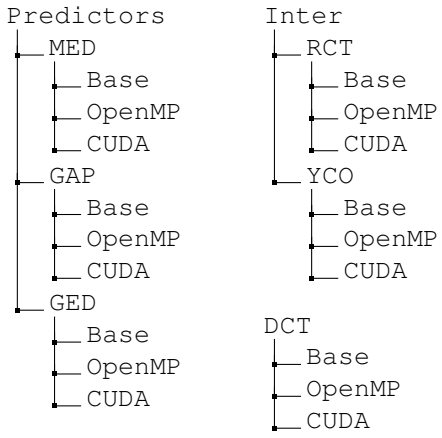
[20] A. S. Charilaos Christopoulos and T. Ebrahimi, "The jpeg2000 still image coding system: An overview," *IEEE Transactions on Consumer Electronics*, 2000.

[21] A. Avramović and A. Avramović, "Ycocg-r: A color space with rgb reversibility and low dynamic range," *52nd International Symposium ELMAR*, 2010.

APÉNDICE

A.1 Arbol de pruebas realizadas





A.2 Algoritmos

$$
\begin{aligned}
&\textit{if } g_v - g_h > 80, P = W \\
&\textit{elseif } g_v - g_h < -80, P = N \\
&\textit{else} \\
&\quad P = (W+N)/2 + (NE - NW)/4 \\
&\quad \textit{if } g_v - g_h > 32, P = (P+W)/2 \\
&\quad \textit{elseif } g_v - g_h > 8, P = (3P+W)/4 \\
&\quad \textit{elseif } g_v - g_h < -32, P = (P+N)/2 \\
&\quad \textit{elseif } g_v - g_h < -8, P = (3P+N)/4 .
\end{aligned}
$$

Fig. 9: Predicción GAP.

$$
\begin{aligned}
g_v &= |W - WW| + |N - NW| + |N - NE| \\
g_h &= |W - NW| + |N - NN| + |NE - NNE| .
\end{aligned}
$$

Fig. 10: Cálculo de gradientes.

$$
\begin{aligned}
Co &= R - B \\
t &= B + (Co >> 1) \\
Cg &= G - t \\
Y &= t + (Cg >> 1)
\end{aligned}
$$

Fig. 11: Cambio de espacio de la transformada YCoCg-r

TAULA 15: RESULTADOS EXPERIMENTALES DE COMPRESIÓN PREDICTORES HAAR (3 REPETICIONES, PROMEDIO POR CONFIGURACIÓN)

Config	Impl.	Versión	Nivel	BPP_w	BPP	t_{comp}	t_{decomp}	Throughput
resultsPNG_Pyr_Sch_base_haar_L1_base	base	Simple	L1	0.157	1.066	37.86	39.85	0.094
resultsPNG_Pyr_Sch_base_haar_L2_base	base	Simple	L2	0.269	1.842	46.33	48.79	0.077
resultsPNG_Pyr_Sch_base_haar_L3_base	base	Simple	L3	0.307	2.103	48.55	51.02	0.074
resultsPNG_Pyr_Sch_base_haar_L4_base	base	Simple	L4	0.318	2.179	49.25	51.61	0.073
resultsPNG_Pyr_Sch_base_haar_L5_base	base	Simple	L5	0.321	2.200	49.59	52.06	0.072
resultsPNG_Std_Sch_base_haar_L1_base	base	Simple	L1	0.157	1.066	37.78	40.17	0.094
resultsPNG_Std_Sch_base_haar_L2_base	base	Simple	L2	0.269	1.841	41.58	46.09	0.084
resultsPNG_Std_Sch_base_haar_L3_base	base	Simple	L3	0.305	2.089	43.03	48.13	0.081
resultsPNG_Std_Sch_base_haar_L4_base	base	Simple	L4	0.314	2.155	44.37	50.18	0.078
resultsPNG_Std_Sch_base_haar_L5_base	base	Simple	L5	0.317	2.171	44.97	51.08	0.076
resultsPNG_Std_nSch_base_haar_L1_base	base	Simple	L1	0.157	1.066	36.67	32.76	0.106
resultsPNG_Std_nSch_base_haar_L2_base	base	Simple	L2	0.269	1.840	39.55	34.12	0.100
resultsPNG_Std_nSch_base_haar_L3_base	base	Simple	L3	0.305	2.087	41.62	35.22	0.096
resultsPNG_Std_nSch_base_haar_L4_base	base	Simple	L4	0.314	2.154	42.91	35.66	0.093
resultsPNG_Std_nSch_base_haar_L5_base	base	Simple	L5	0.316	2.170	43.68	36.32	0.092
resultsPNG_Pyr_nSch_base_haar_L1_base	base	Simple	L1	0.157	1.066	37.46	33.96	0.103
resultsPNG_Pyr_nSch_base_haar_L2_base	base	Simple	L2	0.269	1.841	46.85	41.74	0.083
resultsPNG_Pyr_nSch_base_haar_L3_base	base	Simple	L3	0.306	2.100	49.00	43.54	0.079
resultsPNG_Pyr_nSch_base_haar_L4_base	base	Simple	L4	0.317	2.177	49.95	44.13	0.078
resultsPNG_Pyr_nSch_base_haar_L5_base	base	Simple	L5	0.320	2.197	49.48	43.94	0.079
resultsPNG_Std_Sch_openMP_haar_L1_openmp	openmp	Simple	L1	0.157	1.066	15.76	15.75	0.233
resultsPNG_Std_Sch_openMP_haar_L2_openmp	openmp	Simple	L2	0.269	1.841	21.38	21.30	0.172
resultsPNG_Std_Sch_openMP_haar_L3_openmp	openmp	Simple	L3	0.305	2.089	26.94	26.10	0.138
resultsPNG_Std_Sch_openMP_haar_L4_openmp	openmp	Simple	L4	0.314	2.155	33.30	31.91	0.113
resultsPNG_Std_Sch_openMP_haar_L5_openmp	openmp	Simple	L5	0.317	2.171	38.75	36.05	0.098
resultsPNG_Std_nSch_openMP_haar_L1_openmp	openmp	Simple	L1	0.157	1.066	14.68	13.39	0.262
resultsPNG_Std_nSch_openMP_haar_L2_openmp	openmp	Simple	L2	0.269	1.840	19.51	17.71	0.197
resultsPNG_Std_nSch_openMP_haar_L3_openmp	openmp	Simple	L3	0.305	2.087	23.90	22.14	0.159
resultsPNG_Std_nSch_openMP_haar_L4_openmp	openmp	Simple	L4	0.314	2.154	26.34	23.91	0.146
resultsPNG_Std_nSch_openMP_haar_L5_openmp	openmp	Simple	L5	0.316	2.170	30.05	26.92	0.129
resultsPNG_Pyr_Sch_openMP_haar_L1_openmp	openmp	Simple	L1	0.157	1.066	13.93	23.16	0.198
resultsPNG_Pyr_Sch_openMP_haar_L2_openmp	openmp	Simple	L2	0.269	1.842	16.07	28.63	0.164
resultsPNG_Pyr_Sch_openMP_haar_L3_openmp	openmp	Simple	L3	0.307	2.103	16.34	26.55	0.171
resultsPNG_Pyr_Sch_openMP_haar_L4_openmp	openmp	Simple	L4	0.318	2.179	16.39	24.16	0.181
resultsPNG_Pyr_Sch_openMP_haar_L5_openmp	openmp	Simple	L5	0.321	2.200	16.44	24.36	0.180
resultsPNG_Pyr_nSch_openMP_haar_L1_openmp	openmp	Simple	L1	0.157	1.066	13.69	12.75	0.278
resultsPNG_Pyr_nSch_openMP_haar_L2_openmp	openmp	Simple	L2	0.269	1.841	15.86	15.42	0.235
resultsPNG_Pyr_nSch_openMP_haar_L3_openmp	openmp	Simple	L3	0.306	2.100	16.24	16.60	0.224
resultsPNG_Pyr_nSch_openMP_haar_L4_openmp	openmp	Simple	L4	0.317	2.177	16.46	18.01	0.213
resultsPNG_Pyr_nSch_openMP_haar_L5_openmp	openmp	Simple	L5	0.320	2.197	16.43	19.35	0.205
resultsPNG_Std_Sch_cuda_haar_L1_cuda	cuda	Simple	L1	-	-	0.739	0.647	5.294
resultsPNG_Std_Sch_cuda_haar_L2_cuda	cuda	Simple	L2	-	-	18.97	18.27	0.197
resultsPNG_Std_Sch_cuda_haar_L3_cuda	cuda	Simple	L3	-	-	23.04	22.44	0.161
resultsPNG_Std_Sch_cuda_haar_L4_cuda	cuda	Simple	L4	-	-	26.48	25.52	0.141
resultsPNG_Std_Sch_cuda_haar_L5_cuda	cuda	Simple	L5	-	-	28.36	27.40	0.132

$t_{comp/decomp}$ en ms. BPP_w: bits por píxel ponderado. Throughput en MB/s.

Tablas completas de resultados

TAULA 16: RESULTADOS EXPERIMENTALES DE COMPRESIÓN CDF (3 REPETICIONES, PROMEDIO POR CONFIGURACIÓN)

Config	Impl.	Versión	Nivel	Reps	BPP_w	BPP	t_{comp}	t_{decomp}	Throughput
resultsPNG_Std_nSum_base_cdf_L1_base	base	Simple	L1	3	0.200	1.370	48.03	47.46	0.0769
resultsPNG_Std_nSum_base_cdf_L2_base	base	Simple	L2	3	0.304	2.090	57.16	56.96	0.0643
resultsPNG_Std_nSum_base_cdf_L3_base	base	Simple	L3	3	0.334	2.296	61.11	61.26	0.0600
resultsPNG_Std_nSum_base_cdf_L4_base	base	Simple	L4	3	0.341	2.343	63.11	62.76	0.0583
resultsPNG_Std_nSum_base_cdf_L5_base	base	Simple	L5	3	0.342	2.350	63.79	63.38	0.0577
resultsPNG_Pyr_Sum_base_cdf_L1_base	base	Simple	L1	3	0.185	1.263	47.48	55.58	0.0712
resultsPNG_Pyr_Sum_base_cdf_L2_base	base	Simple	L2	3	0.281	1.925	59.03	68.65	0.0575
resultsPNG_Pyr_Sum_base_cdf_L3_base	base	Simple	L3	3	0.309	2.121	61.79	71.75	0.0550
resultsPNG_Pyr_Sum_base_cdf_L4_base	base	Simple	L4	3	0.316	2.168	62.14	72.38	0.0546
resultsPNG_Pyr_Sum_base_cdf_L5_base	base	Simple	L5	3	0.318	2.179	62.59	72.80	0.0542
resultsPNG_Std_Sum_base_cdf_L1_base	base	Simple	L1	3	0.185	1.263	48.19	56.64	0.0700
resultsPNG_Std_Sum_base_cdf_L2_base	base	Simple	L2	3	0.264	1.811	56.54	70.22	0.0579
resultsPNG_Std_Sum_base_cdf_L3_base	base	Simple	L3	3	0.281	1.924	60.77	76.60	0.0534
resultsPNG_Std_Sum_base_cdf_L4_base	base	Simple	L4	3	0.279	1.906	62.97	80.19	0.0513
resultsPNG_Std_Sum_base_cdf_L5_base	base	Simple	L5	3	0.274	1.876	64.97	82.82	0.0497
resultsPNG_Pyr_nSum_base_cdf_L1_base	base	Simple	L1	3	0.200	1.370	55.39	57.32	0.0651
resultsPNG_Pyr_nSum_base_cdf_L2_base	base	Simple	L2	3	0.309	2.122	68.57	70.99	0.0526
resultsPNG_Pyr_nSum_base_cdf_L3_base	base	Simple	L3	3	0.344	2.366	71.85	74.24	0.0503
resultsPNG_Pyr_nSum_base_cdf_L4_base	base	Simple	L4	3	0.355	2.437	72.54	74.58	0.0499
resultsPNG_Pyr_nSum_base_cdf_L5_base	base	Simple	L5	3	0.357	2.456	72.80	75.37	0.0495
resultsPNG_Std_nSum_OpenMP_cdf_L1_openmp	openmp	Simple	L1	3	0.200	1.370	105.68	104.68	0.0349
resultsPNG_Std_nSum_OpenMP_cdf_L2_openmp	openmp	Simple	L2	3	0.304	2.090	140.17	134.08	0.0268
resultsPNG_Std_nSum_OpenMP_cdf_L3_openmp	openmp	Simple	L3	3	0.334	2.296	176.74	171.01	0.0211
resultsPNG_Std_nSum_OpenMP_cdf_L4_openmp	openmp	Simple	L4	3	0.341	2.343	231.86	230.37	0.0159
resultsPNG_Std_nSum_OpenMP_cdf_L5_openmp	openmp	Simple	L5	3	0.342	2.350	261.56	263.39	0.0140
resultsPNG_Std_Sum_OpenMP_cdf_L1_openmp	openmp	Simple	L1	3	0.185	1.263	97.15	91.34	0.0389
resultsPNG_Std_Sum_OpenMP_cdf_L2_openmp	openmp	Simple	L2	3	0.264	1.811	137.31	136.94	0.0268
resultsPNG_Std_Sum_OpenMP_cdf_L3_openmp	openmp	Simple	L3	3	0.281	1.924	168.99	170.19	0.0216
resultsPNG_Std_Sum_OpenMP_cdf_L4_openmp	openmp	Simple	L4	3	0.279	1.906	200.97	203.55	0.0181
resultsPNG_Std_Sum_OpenMP_cdf_L5_openmp	openmp	Simple	L5	3	0.274	1.876	255.94	260.51	0.0142
resultsPNG_Pyr_Sum_OpenMP_cdf_L1_openmp	openmp	Simple	L1	3	0.185	1.263	16.05	16.10	0.2283
resultsPNG_Pyr_Sum_OpenMP_cdf_L2_openmp	openmp	Simple	L2	3	0.281	1.925	18.18	18.41	0.2006
resultsPNG_Pyr_Sum_OpenMP_cdf_L3_openmp	openmp	Simple	L3	3	0.309	2.121	18.92	19.14	0.1929
resultsPNG_Pyr_Sum_OpenMP_cdf_L4_openmp	openmp	Simple	L4	3	0.316	2.168	19.09	19.26	0.1914
resultsPNG_Pyr_Sum_OpenMP_cdf_L5_openmp	openmp	Simple	L5	3	0.318	2.179	19.09	19.31	0.1912
resultsPNG_Pyr_nSum_OpenMP_cdf_L1_openmp	openmp	Simple	L1	3	0.200	1.370	15.65	15.83	0.2332
resultsPNG_Pyr_nSum_OpenMP_cdf_L2_openmp	openmp	Simple	L2	3	0.309	2.122	18.10	18.15	0.2025
resultsPNG_Pyr_nSum_OpenMP_cdf_L3_openmp	openmp	Simple	L3	3	0.344	2.366	18.81	18.89	0.1947
resultsPNG_Pyr_nSum_OpenMP_cdf_L4_openmp	openmp	Simple	L4	3	0.355	2.437	18.97	18.99	0.1934
resultsPNG_Pyr_nSum_OpenMP_cdf_L5_openmp	openmp	Simple	L5	3	0.357	2.456	19.01	19.01	0.1931
resultsPNG_Std_nSum_cuda_cdf_L1_cuda	cuda	Simple	L1	3	-	-	1.33	1.33	2.761
resultsPNG_Std_nSum_cuda_cdf_L2_cuda	cuda	Simple	L2	3	-	-	26.67	26.80	0.137
resultsPNG_Std_nSum_cuda_cdf_L3_cuda	cuda	Simple	L3	3	-	-	38.25	36.94	0.098
resultsPNG_Std_nSum_cuda_cdf_L4_cuda	cuda	Simple	L4	3	-	-	44.28	42.72	0.084
resultsPNG_Std_nSum_cuda_cdf_L5_cuda	cuda	Simple	L5	3	-	-	48.49	46.11	0.078

$t_{comp/decomp}$ en ms. BPP_w: bits por píxel ponderado. Throughput en MB/s.

TAULA 17: RESULTADOS EXPERIMENTALES DE COMPRESIÓN PREDICTORES (3 REPETICIONES)

Predictor	Config	Impl.	n_{reps}	BPP_w	BPP	t_{comp}	t_{decomp}	t_{total}	Throughput
MED	Base-rep1	base	1	0.352	2.418	12.04	59.45	71.49	0.103
	Base-rep2	base	1	0.352	2.418	12.10	59.29	71.39	0.103
	Base-rep3	base	1	0.352	2.418	12.23	59.43	71.66	0.102
	OpenMP-rep1	OpenMP	1	0.352	2.418	18.76	59.60	78.36	0.094
	OpenMP-rep2	OpenMP	1	0.352	2.418	16.91	59.37	76.28	0.096
	OpenMP-rep3	OpenMP	1	0.352	2.418	17.29	59.61	76.90	0.095
	CUDA-rep1	CUDA	1	–	–	1.04	0.87	1.90	3.82
	CUDA-rep2	CUDA	1	–	–	0.94	0.88	1.82	3.99
	CUDA-rep3	CUDA	1	–	–	0.95	0.88	1.83	3.97
GAP	Base-rep1	base	1	0.359	2.467	73.56	59.75	133.31	0.055
	Base-rep2	base	1	0.359	2.467	73.44	59.60	133.04	0.055
	Base-rep3	base	1	0.359	2.467	73.96	59.34	133.30	0.055
	OpenMP-rep1	OpenMP	1	0.359	2.467	46.10	75.09	121.19	0.061
	OpenMP-rep2	OpenMP	1	0.359	2.467	58.58	76.55	135.13	0.054
	OpenMP-rep3	OpenMP	1	0.359	2.467	43.74	73.79	117.53	0.062
	CUDA-rep1	CUDA	1	–	–	1.62	1.00	2.63	2.76
	CUDA-rep2	CUDA	1	–	–	1.65	0.95	2.61	2.79
	CUDA-rep3	CUDA	1	–	–	1.59	0.93	2.51	2.89
GED	Base-rep1	base	1	0.329	2.256	14.75	36.58	51.33	0.143
	Base-rep2	base	1	0.329	2.256	14.80	36.46	51.26	0.143
	Base-rep3	base	1	0.329	2.256	14.61	37.10	51.71	0.142
	OpenMP-rep1	OpenMP	1	0.329	2.256	30.49	37.02	67.51	0.109
	OpenMP-rep2	OpenMP	1	0.329	2.256	41.18	37.31	78.49	0.094
	OpenMP-rep3	OpenMP	1	0.329	2.256	59.14	37.75	96.89	0.076
	CUDA-rep1	CUDA	1	–	–	1.27	0.97	2.24	3.25
	CUDA-rep2	CUDA	1	–	–	1.31	0.98	2.29	3.18
	CUDA-rep3	CUDA	1	–	–	1.29	0.95	2.24	3.24

$t_{comp/decomp/total}$: tiempos compresión/descompresión/total (ms). BPP_w: bits por píxel ponderado.
Throughput en MB/s.

TAULA 18: RESULTADOS EXPERIMENTALES DCT (3 REPETICIONES)

Predictor	Config	Impl.	n_{reps}	BPP_w	BPP	t_{comp}	t_{decomp}	t_{total}	Throughput
DCT	Base-rep1	base	1	0.357	2.446	14.46	14.51	28.97	0.253
	Base-rep2	base	1	0.357	2.446	14.76	14.55	29.31	0.250
	Base-rep3	base	1	0.357	2.446	14.56	14.56	29.12	0.252
	OpenMP-rep1	OpenMP	1	0.357	2.446	18.30	15.82	34.12	0.215
	OpenMP-rep2	OpenMP	1	0.357	2.446	20.38	15.68	36.06	0.204
	OpenMP-rep3	OpenMP	1	0.357	2.446	19.19	16.19	35.38	0.207
	CUDA-rep1	CUDA	1	–	–	11.70	10.99	22.69	0.324
	CUDA-rep2	CUDA	1	–	–	11.35	10.69	22.04	0.333
	CUDA-rep3	CUDA	1	–	–	8.91	8.37	17.28	0.425

$t_{comp/decomp/total}$: tiempos compresión/descompresión/total (ms). BPP_w: bits por píxel ponderado.
Throughput en MB/s.

TAULA 19: RESULTADOS EXPERIMENTALES PREDICTORES RCT Y YCO (3 REPETICIONES)

Predictor	Config	Impl.	n_{reps}	BPP_w	BPP	t_{comp}	t_{decomp}	t_{total}	Throughput
RCT	Base-rep1	base	1	0.149	1.051	0.00	1.80	1.81	4.06
	Base-rep2	base	1	0.149	1.051	0.00	1.80	1.80	4.08
	Base-rep3	base	1	0.149	1.051	0.00	1.87	1.87	3.93
	OpenMP-rep1	OpenMP	1	0.149	1.051	0.02	2.16	2.18	3.37
	OpenMP-rep2	OpenMP	1	0.149	1.051	0.00	2.76	2.76	2.66
	OpenMP-rep3	OpenMP	1	0.149	1.051	0.00	2.19	2.19	3.35
	CUDA-rep1	CUDA	1	–	–	0.35	0.38	0.73	10.08
	CUDA-rep2	CUDA	1	–	–	0.36	0.38	0.74	9.93
	CUDA-rep3	CUDA	1	–	–	0.36	0.38	0.74	9.98
YCO	Base-rep1	base	1	0.151	1.064	1.26	1.17	2.43	3.02
	Base-rep2	base	1	0.151	1.064	1.09	1.19	2.28	3.22
	Base-rep3	base	1	0.151	1.064	1.20	1.16	2.36	3.11
	OpenMP-rep1	OpenMP	1	0.151	1.064	2.52	2.80	5.32	1.38
	OpenMP-rep2	OpenMP	1	0.151	1.064	2.00	2.35	4.35	1.69
	OpenMP-rep3	OpenMP	1	0.151	1.064	2.07	3.23	5.30	1.39
	CUDA-rep1	CUDA	1	–	–	0.33	0.33	0.66	11.09
	CUDA-rep2	CUDA	1	–	–	0.35	0.33	0.68	10.76
	CUDA-rep3	CUDA	1	–	–	0.37	0.33	0.70	10.50

$t_{comp/decomp/total}$: tiempos compresión/descompresión/total (ms). BPP_w: bits por píxel ponderado.
Throughput en MB/s. RCT rep1: error numérico detectado.