Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de computación

TP Final Organización del Computador 2

Compresor JPG

Agustina Ciraco - LU 630/06 - agusciraco@gmail.com
Verónica Coy - LU 652/08 - verocoy@gmail.com
Natasha Martinelli - LU 656/08 - nmartinelli23@gmail.com

Resumen

El objetivo de este trabajo es realizar un compresor JPG para archivos de video. La implementación se realizará en C y en C con algunas funciones en assembler para comparar resultados entre ambas implementaciones. Además para compilar dicho código se utilizará el ICC que es el compilador de intel para optimizar más el código.

Keywords

JPEG, compresor, Huffman, lenguaje C, SIMD, sse, compilador

Índice

1. Introducción **2**
1.1. Tecnologías a considerar . . . 2
1.2. Algoritmo JPEG . . . 2
1.2.1. Resta 128 . . . 2
1.2.2. Transformada DCT . . . 2
1.2.3. Cuantización . . . 4
1.2.4. Codificación . . . 4
1.3. Consideraciones generales . . . 4

2. Desarrollo **6**
2.1. Desarrollo Preliminar . . . 6
2.2. Profiling . . . 6
2.3. Optimización mediante SIMD . . . 7

3. Resultados **9**
3.1. Eficiencia como compresor . . . 9
3.1.1. PSNR . . . 9
3.1.2. Entropía . . . 11
3.2. Tiempos de cómputo . . . 12

4. Conclusiones **14**

1. Introducción

El objetivo de este trabajo es realizar un estudio comparativo entre diferentes tecnologías (C,ASM,GCC,ICC) a la hora de implementar un algoritmo específico. En este caso se implementó el algoritmo JPEG de compresión de imágenes como caso de estudio particular.

1.1. Tecnologías a considerar

Con el fin de poder visualizar las ventajas de las diferentes tecnologías abordadas en la materia, se decidió implementar un algoritmo como el JPEG, que será explicado en detalle posteriormente. Dicho algoritmo posee numerosos cálculos que se conforman por sumatorias y diferentes cuentas reutilizables. Estos cálculos suelen ser costosos en el procesamiento estructurado iterativo que propone un lenguaje como C, y suele ser muy provechoso atacar los mismos con tecnologías SIMD como las que propone Assembler.

Luego, también resulta interesante barajar diferentes opciones de compiladores, dada que la traducción que estos hacen del código al lenguaje de máquina puede ser muy diferente. Muchas veces, el hecho de como un compilador aprovecha las características particulares de una arquitectura es un elemento determinante en el tiempo de cómputo de un algoritmo.

Por lo explicado anteriormente, se consideraron las siguientes opciones de lenguajes y compiladores:

- Lenguaje C con compilador GCC.
- Lenguaje C con compilador ICC.
- Lenguaje ASM con instrucciones de SSE con compilador GCC.
- Lenguaje ASM con instrucciones de SSE con compilador ICC.

Lo que se buscó en este trabajo es tratar de constratar la realidad de los tiempos de cómputo con la hipótesis de que las instrucciones SIMD son ideoneas para ciertos cálculos, sumado al hecho de que el compilador ICC puede tomar ventajas sobre la arquitectura en particular.

1.2. Algoritmo JPEG

El algoritmo JPEG es un algoritmo de compresión de imágenes con pérdida, lo que significa que parte de la compresión nace del hecho de descartar información de la imagen original. El exito histórico de este algoritmo se basa en la elección de la información a recortar de tal forma que visualmente tenga el menor impacto posible para el ojo humano. Para esto el algoritmo realiza una transformada sobre la imagen que dispone la información en un espacio donde es más simple discernir cual información es importante y cual no.

En toda la explicación siguiente, se considerará una imagen como una matriz de dimensión HxW. Cada entrada de la matriz corresponde a un pixel de la imagen y es representado con un valor numérico entre 0 y 255. Esta consideración es para imágenes en escala de grises, de contar con imágenes a color, la representación es análoga pero se poseen 3 matrices diferentes, para los diferentes canales de color.

A continuación se detallan los pasos del algoritmo:

1.2.1. Resta 128

En este simple paso lo único que se hace es restarle 128 a todos los píxeles de la imagen. Esto se hace debido a que, dada la transformada que se va a utilizar posteriormente, resulta conveniente tener los valores distribuidos alrededor del cero.

1.2.2. Transformada DCT

En este paso la idea principal es transformar la información en la matriz original a un espacio donde sea más simple saber que información descartar. A priori, si se toma la matriz de una imagen cualquiera no resulta simple poder decir automaticamente que pixel se podría descartar de la imagen

sin perder información sensible al ojo humano. Es por esto que a la matriz original se le aplica la Transformada de Coseno Discreta para transformar la información a un nuevo espacio.

La Transformada de Coseno Discreta (DCT) representa un ejemplo típico de transformación de señales en donde se pasa de un plano espacial o temporal, a un espacio de frecuencias. La idea de esta transformada es cambiar el plano de representación de una señal para que, en vez de representar cada valor punto a punto, se pueda representar la señal como una combinación lineal de cosenos de diferentes frecuencias. En el caso de señales de dos dimensiones como en el procesamiento de imágenes, se busca representar la imagen como una combinación lineal de combinaciones de cosenos en dos variables diferentes, con el fin de que se puedan representar las ondulaciones de la señal en las dos dimensiones presentes.

El cálculo de la DCT para señales de una dimensión corresponde al siguiente cálculo:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \quad k = 0, \ldots, N-1. \tag{1}$$

En donde N corresponde al tamaño de la imagen original.

En el caso de una imagen, la DCT en dos dimensiones se basa simplemente en aplicar la DCT a cada una de las filas y, una vez hecho este primer paso, a cada una de las columnas.

Lo que se puede observar en la cuenta explicitada es que la frecuencia del coseno utilizado depende de k, que es la posición en el vector. Por lo tanto, esto quiere decir que los cosenos con menor ondulación se encuentran al principio del vector, y al final se encuentran los cosenos con frecuencias muy altas. Lo interesante de esta observacion es que este hecho no depende de la señal, siempre los valores correspondientes a los cosenos de altas frecuencias se encuentran al final de la señal transformada. Lo que obviamente si varía con cada señal, es cuales son dichos coeficientes. Si se tiene una señal muy oscilante es posible que se necesiten cosenos con frecuencias muy altas para poder representarla, lo que resultaría en coeficientes altos en las últimas posiciones de la señal transformada.

En el caso de la DCT en dos dimensiones, el coseno de menor frecuencia se encuentra arriba y a la izquierda, y las frecuencias de las ondulaciones en ambas direcciones van creciendo a medida que nos movemos por la matriz hacia abajo y hacia la derecha. De esta manera, el valor correspondiente a la esquina inferior derecha, corresponde al coeficiente que acompaña a los cosenos que mayor ondulación tienen en ambas direcciones.

La idea principal de la compresión del algoritmo JPEG se basa en castigar fuertemente, hasta incluso hacer desaparecer, a los coeficientes correspondientes a los cosenos de altas frecuencias basandose en el hecho de que estos pertenecen a frecuencias correspondientes a detalles dificilmente percibidos por el ojo humano. Nuevamente, lo interensante de esto, es que no importa cuál sea la imagen de entrada, siempre es conveniente castigar los cosenos de altas frecuencias.

Si bien castigar los cosenos de altas frecuencias parece una buena idea, esto puede resultar perjudicial para la imagen para cuando la misma posee valores muy oscilantes y esto se traduce en coeficientes altos para dichos cosenos. Cuando sucede esto y se hace desaparecer dichos coeficientes, la imagen reconstruida puede perder mucho más que solamente detalles, haciendo posible la aparición de varios *artifacts* en la imagen reconstruida. Es por esto que un punto muy importante a tener en cuenta en algoritmo de JPEG es la división de la imagen en bloques de 8x8 píxeles. De esta manera, la cercanía espacial de los píxeles de cada bloque hace que la señal dentro de los mismos no sea demasiado oscilante, haciendo que en general no sea tan común la presencia de coeficientes altos en los cosenos de altas frecuencias.

Un contraejemplo de lo antedicho es cuando se encuentra un borde dentro de un bloque de 8x8. Las presencias de bordes hace que la señal resulte oscilante. Luego, al castigar a los cosenos de altas frecuencias, suele suceder que al reconstruir la imagen los bordes se difuminan notoriamente. Es por esto que el algoritmo de compresión JPEG sirve para fotografías naturales en donde las transiciones suelen ser suaves y continúas; pero suele tener un pobre desempeño para imagenes artificiales con bordes marcados, por ejemplo imagenes con presencia de texto o números con bordes bien delimitados del fondo.

Por el momento, este paso del algoritmo solo realiza la transformación a otro espacio de variables, todavía no se hizo nada para lograr compresión ya que a priori toda la información se encuentra detallada de otra manera y es posible realizar la transformada inversa, la IDCT, para volver a la imagen original. En el siguiente paso, se explicará como se castiga a los coeficientes de los cosenos de altas frecuencias.

1.2.3. Cuantización

El proceso de cuantización es una de las principales partes de compresión en el algoritmo de JPEG. En general, cuantizar una señal significa agrupar diferentes valores posibles en un solo valor representante. Esto ayuda a la compresión debido a que en vez de tener varios valores diferentes, se tiene un solo valor como representante de varios con mucha mayor frecuencia de aparición, lo que ayuda a los algoritmos típicos de compresión como Huffman, que será nombrado en una sección posterior.

En general, cuando no se tiene información sobre la señal, se suelen realizar cuantizaciones uniformes que agrupan a los valores de forma análoga independientemente del valor que sea. Por ejemplo, si una señal tiene valores entre 0 y 100, una cuantización uniforme típica podría ser dividir todos los valores por 10 y tomar su parte entera. De esta forma, todos los valores entre 0 y 9 pasarán a ser representados con un 0, los valores entre 10 y 19 con un 1, y así sucesivamente. Es claro que este paso representa una pérdida de información ya que una vez que hacemos esto no tenemos formar de reconstruir la señal original de forma exacta. Todos los valores entre 20 y 29 ahora son representados con un 2 y no se puede saber que valor eran originalmente, es posible que cuando se reconstruya la imagen se elija un valor como el 25 para representar los valores originales, pero como se ve, esto no representa exactamente la información original.

Lo interesante de JPEG es como este algoritmo puede cuantizar de una manera no uniforme gracias a que la información en este punto se encuentra transformada a un espacio de frecuencias donde se sabe cual información importa más y cual menos. Dividir ciertos píxeles por valores más grandes (y luego tomar su parte entera) significa una cuantización mayor porque se estan formando bolsas de valores más grandes, un solo representante para más valores diferentes. Dividir por números más chicos representa una cuantización menor.

Dicho lo anterior, el algoritmo de JPEG decide no cuantizar uniformente sino que divide por valores más grandes a los coeficientes correspondientes a los cosenos de altas frecuencias y divide por números pequeños a los coeficientes de los cosenos de bajas frecuencias.

Dado que, como se explico anteriormente, el algoritmo de JPEG trabaja cada bloque de 8x8 de forma separada, se presenta una matriz de cuantización del mismo tamaño que sirve para realizar la división punto a punto.

Esta matriz fue producto de estudios psicovisuales al desarrollarse el estandar JPEG, lo que quiere decir que se sometió a diferentes individuos a imágenes cuantizadas y decuantizadas con diferentes matrices y se les pedía una devolución sobre la calidad de imagen obtenida.

La matriz del algoritmo de JPEG es:

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}.$$

Se puede ver como la matriz anterior se condice con el hecho de intentar castigar más fuertemente a los coeficientes correspondientes a los cosenos de altas frecuencias.

1.2.4. Codificación

El último paso del algoritmo JPEG es la codificación de los valores obtenidos por los pasos anteriores. Para esto se utiliza la codificación de Huffman debido a su demostrada eficiencia como codificador.

Esta codificación se basa en asignar códigos de longitud menor a valores con mayor frecuencia de aparición.

Para una explicación más detallada ver [3]

1.3. Consideraciones generales

Luego de presentar el algoritmo de JPEG algo interesante a remarcar es como la cuenta correspondiente a la DCT contiene sumatorias y cálculos que siempre dependen de los mismos 8 elementos de

la fila o columna que se esta tratando. Además es el único cálculo que no es puntual si no que depende de otros valores además del que se está procesando por lo que es de esperar que este cálculo sea el que tome mayor tiempo. Por otro lado, la cuantización siempre se realiza haciendo la misma cuenta con la misma matriz.

Las características mencionadas sobre los diferentes pasos del algoritmo hacen que resulte esperanzador la implementación con instrucciones SIMD para la reducción de tiempos de cómputo.

Por otro lado, cabe destacar que en las diferentes citas bibliográficas sobre el algoritmo JPEG como en [4], suelen aparecer algunos pasos más a realizar, dependiendo la implementación. Algunos de ellos son la transformación de colores al espacio YCbCr y el umbralado de valores posterior a la cuantización. Estos pasos se suelen corresponder con fundamentos provenientes del procesamiento de imágenes y no representan cómputos extensivos como para modificar el eje de observación principal de este trabajo.

Por último, notamos que este trabajo el algoritmo de JPEG se utilizó sobre videos también. Para esto se tomó la decisión simplificada de aplicar el algoritmo de JPEG a cada frame del video. Si bien varios algoritmos lo que hacen es tratar de aprovechar la relación espacial-temporal que existe entre cada frame, en nuestro trabajo el eje central es poder comparar el tiempo de cómputo de JPEG con las diferentes tecnologías por lo que se aplicó directamente el algoritmo sobre cada uno de los frames de entrada.

2. Desarrollo

2.1. Desarrollo Preliminar

Luego del relevamiento conceptual realizado sobre el algoritmo JPEG, se dispusó realizar una implementación preliminar del mismo para detectar los puntos en donde poder insertar las optimizaciones en código assembler.

En este punto se implementó la totalidad de los pasos del algoritmo JPEG en lenguaje C. Dicha implementación se realizó sobre el sistema operativo Ubuntu 12.04 32-bits.

Si bien casi toda la implementación se realizó en lenguaje C plano con sus librerias standard, se utilizó la librería OpenCV [1] para facilitar las cuestiones ligadas al manejo de video. Dicha librería resultó de particular utilidad para poder obtener video a partir de la cámara para su posterior fragmentación en frames individuales, dado que fue la forma elegida para procesar video en este trabajo.

Una vez implementado y testeado el algoritmo JPEG sobre imágenes y videos se probó su desempeño. Como se podrá observar en la sección de resultados, los tiempos de cómputo por frame de este desarrollo preliminar pueden resultar aceptables o no dependiendo la aplicación. En particular, se observó que el tiempo de cómputo promedio por frame al desarrollar solo en lenguaje C y usar GCC como compilador no llega a la velocidad necesaria para poder hacer compresión o decomprensión *on the fly* para un video típico de unos 30 frames por segundo.

Si bien es claro que para mejorar la performance la mejor decisión probablemente sea traducir todo el algoritmo al lenguaje Assembler, es claro también que esto resulta un trabajo mayúsculo que no siempre se justifica al mirar la relación costo-beneficio, o no por lo menos para hacer la traducción total del código.

Luego, en la siguiente sección se explica como se observó donde poner el foco de la traducción a Assembler para la optimización de la performance.

2.2. Profiling

Hacer Profiling es una opción idónea para saber que funciones del algoritmo traducir al lenguaje Assembler. La técnica de Profiling es una técnica de análisis dinámico de programas que sirve para identificar dentro de que funciones se está mayor tiempo en una ejecución particular del programa, también identifica la cantidad de llamadas que se realizan a cada una de las funciones.

Si bien los cálculos resultantes del algoritmo son completamente dependientes de la imagen o del video de entrada, la cantidad de cálculos realizados por cada una de las partes del algoritmo no debería variar por los valores que toman los píxeles de la señal de entrada. Es por este motivo que se consideraron 50 frames cualesquiera tomados consecutivos de la cámara como una buena muestra para realizar el Profiling del programa.

En este trabajo se utilizó una de las herramientas más populares para Profiling en lenguaje C, *gprof*. Para utilizar esta herramienta se debe compilar el programa a analizar con un flag particular con el fin de que el mismo genere cierta información necesaria para el profiling. Una vez generada la información de Profiling, se corre el programa *gprof* con el fin de visualizar dicha información.

Existen muchas maneras de visualizar la información que *gprof* proporciona. Sin embargo, dado que estamos ante un proyecto acotado, alcanza simplemente con la información cruda que proporciona sobre los tiempos de cómputo en cada una de las funciones del programa.

A continuación se transcribe un extracto de la salida correspondiente a la corrida del *gprof*

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 53.30      2.18     2.18   240000     0.01     0.01  dct2
 42.79      3.93     1.75   240000     0.01     0.01  idct2
  1.22      3.98     0.05   240000     0.00     0.00  quantization
  0.73      4.01     0.03   240000     0.00     0.00  mult_quantization
```

En dicha salida, las funciones se encuentran ordenadas por porcentaje de tiempo de cómputo consumido de mayor a menor. Dada la notoria predominancia de las dos primeras funciones, alcanza con mostrar las 4 primeras funcionas para ilustrar esto.

Se puede observar que las funciones *dct2* y *idct2* correspondientes a la Transformada de Coseno Discreta en dos dimensiones, y a su inversa, son las que ocupan más del 90 % del tiempo de cómputo total del programa. Esto se condice con la hipótesis presentada anteriormente que indicaba que se esperaba que el mayor tiempo de cómputo se lo llevasen estas funciones dado que son las únicas que no son puntuales, sino que poseen sumatorias sobre toda la fila o columna que están procesando.

El resultado presentado también se adecua correctamente a lo mencionado sobre que probablemente no sea beneficioso convertir todo el algoritmo al lenguaje Assembler, sino solo una parte.

Dados los resultados presentados, que refuerzan la hipótesis esgrimida, se optó por realizar la optimización del código de las funciones correspondientes a la Transformada de Coseno Discreta para lograr un impacto en la performance total del algoritmo.

2.3. Optimización mediante SIMD

En esta sección se explayará la traducción realizada a la implementación de la Transformada de Coseno Discreta.

Esta transformada se realiza en cada bloque de 8x8 de la imagen y para realizarla en dos dimensiones lo que se hace es aplicarla primero a las filas y luego, sobre el resultado de esta primera aplicación, se aplica a las columnas.

Es por esto que lo se detalla en esta sección es la aplicación de la Transformada de Coseno Discreta a una señal unidimensional de 8 valores. La extensión a los demás cálculos necesarios se despreden del presentado. De la misma manera, solo se detalla un paso de la transformación, los pasos para la antitransformación son análogos a los presentados, aunque los coeficientes en los cálculos son ligeramente diferentes.

Antes de seguir, se recuerda cual es el cálculo unidimensional de la Transformada de Coseno Discreta:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right] \quad k = 0, \ldots, N-1. \tag{2}$$

Lo primero a observar en la cuenta es el hecho de que lo que se está buscando son los coeficientes de una señal en una base de cosenos. Como se vé en el cálculo, los cosenos dependen de dos variables, n y k. Esto, en señales de 8 valores, da lugar a 64 cosenos diferentes que serán reutilizados en todo bloque de 8x8 de toda imagen a la que se le aplique el algoritmo JPEG.

Es por esto que parece razonable, en vez de calcular los cosenos varias veces, tener los mismos precomputados para ir utilizadonlos cuando sea necesario.

A continuación se ve el código correspondiente a la definición de los cosenos necesarios.

```
cos_elem0_1: dd 1.0, 1.0, 1.0, 1.0
cos_elem0_2: dd 1.0, 1.0, 1.0, 1.0
cos_elem1_1: dd 0.980785, 0.831470, 0.555570, 0.195090
cos_elem1_2: dd -0.195090, -0.555570, -0.831470, -0.980785
cos_elem2_1: dd 0.923880, 0.382683, -0.382683, -0.923880
cos_elem2_2: dd -0.923880, -0.382683, 0.382683, 0.923880
cos_elem3_1: dd 0.831470, -0.195090, -0.980785, -0.555570
cos_elem3_2: dd 0.555570, 0.980785, 0.195090, -0.831470
cos_elem4_1: dd 0.707107, -0.707107, -0.707107, 0.707107
cos_elem4_2: dd 0.707107, -0.707107, -0.707107, 0.707107
cos_elem5_1: dd 0.555570, -0.980785, 0.195090, 0.831470
cos_elem5_2: dd -0.831470, -0.195090, 0.980785, -0.555570
cos_elem6_1: dd 0.382683, -0.923880, 0.923880, -0.382683
cos_elem6_2: dd -0.382683, 0.923880, -0.923880, 0.382683
cos_elem7_1: dd 0.195090, -0.555570, 0.831470, -0.980785
cos_elem7_2: dd 0.980785, -0.831470, 0.555570, -0.195090
```

Lo único a remarcar sobre el código presentado es el hecho de que, dado que se utilizará la tecnología SSE, solo se pueden almacenar cuatro cosenos en cada uno de los registros a utilizar, dando lugar a 16 vectores de cosenos a reutilizar.

A continuación lo que se muestra es el código implementado para el cálculo de un elemento, la diferencia entre los diferentes elementos solamente radica en los cosenos utilizados y en las constantes de normalización que el algoritmo requiere.

```
.caso_elem0:
xorps xmm1, xmm1
xorps xmm2, xmm2
movdqu xmm2, [eax + edi*4] ; xmm2 = [inicio+(i+k)*witdh*4+j*4]
cvtdq2ps xmm1, xmm2 ; xmm1 = [inicio+(i+k)*witdh*4+j*4] en flotante de 32b
movdqu xmm2, [cos_elem0_1]
mulps xmm1, xmm2
movdqu xmm3, xmm1 ; xmm3 = los primeros 4px con la cuenta del cos listos para sumarse
movdqu xmm2, [eax + edi*4 + 16] ;aca van los elementos de la imagen a procesar
cvtdq2ps xmm1, xmm2 ; xmm1 = [inicio+(i+k)*witdh*4+j*4+16] en flotante de 32b
movdqu xmm2, [cos_elem0_2]
mulps xmm1, xmm2 ; xmm1 = los segundos 4px con la cuenta del cos listos para sumarse
movdqu xmm2, xmm3 ; xmm2 = los primeros 4px con la cuenta del cos listos para sumarse
; xmm1 = [x1,x2,x3,x4]  xmm2 = [y1,y2,y3,y4]
haddps xmm1, xmm2 ; xmm1 = [y1+y2,y3+y4,x3+x4,x1+x2]
xorps xmm2, xmm2 ; xmm2 = [0.0,0.0,0.0,0.0]
haddps xmm1, xmm2 ; xmm1 = [0.0,0.0,y1+y2+y3+y4,x3+x4+x1+x2]
haddps xmm1, xmm2 ; xmm1 = [0.0, 0.0, 0.0, y1+y2+y3+y4+x3+x4+x1+x2]
movdqu xmm2, [sqrt_of_eight] ; xmm2 = [1.0,1.0,1.0,sqrt(8)]
divps xmm1, xmm2 ; xmm1 = primer elem calculado / sqrt(8)
movdqu xmm2, xmm1
mov edx, temp
movd [edx + 0], xmm1  ; paso a temp el calculo del 1er elemento de la fila
```

Lo importante a remarcar de este código es como se aprovechan los registros SSE y las instrucciones SIMD para realizar el cálculo de la Transformada de Coseno Discreta. En vez de realizar la sumatoria que indica la cuenta con cada multiplicación y cada suma por separado, el presente código realiza cuatro multiplicaciones a la vez, por cada uno de los cosenos precomputados. Luego, el código hace uso de las instrucciones de suma horizontal para lograr la sumatoria deseada en menos operaciones.

Esta reducción en la cantidad de sumas y multiplicaciones realizadas debería resultar en notorias mejoras en el tiempo de cómputo de la Transformada de Coseno Discreto y, por ende, en toda la ejecución del algoritmo JPEG.

3. Resultados

En esta sección presentaremos los resultados al ejecutar el algoritmo JPEG sobre diferentes señales de entrada.

Dada la naturaleza de este trabajo, dividiremos esta sección en dos partes. Por un lado presentaremos los resultados correspondientes a la performance del algoritmo JPEG como tal. Es decir, presentaremos como se desempeña en cuanto a compresión y en cuanto a la calidad de la imagen obtenida. Por otro lado se verá la eficiencia del algoritmo en cuanto a los tiempos de cómputo requeridos para el mismo.

Ambos tipos de resultados se presentaran para 3 tipos de señales de entrada diferentes:

- Imagen. Se trabajo con la imagen *Lena.bmp*
- Video capturado desde la cámara. Se tomarán 50 frames para realizar promedios de las diferentes métricas
- Video en formato .avi. El video utilizado tiene 30 frames para realizar promedios de las diferentes métricas.

3.1. Eficiencia como compresor

Si bien en el contexto de la materia la faceta que nos importa es la relacionada a la performance en cuanto a tiempos al usar lenguaje ensamblador o no, nos parece importante analizar la performance del algoritmo JPEG como compresor debido a que es su utilización natural y cotidiana.

Para poder analizar el comportamiento de JPEG como compresor analizaremos dos métricas:

- Peak Signal-to-Noise Ratio (PSNR). Es una métrica para analizar la calidad de la imagen obtenida al comprimir y descomprimir una señal.
- Entropía. Es una métrica para analizar el nivel de compresión alcanzable.

3.1.1. PSNR

El Peak Signal-to-Noise Ratio es una métrica que analiza la calidad de una señal sometida a alguna modificación, con respecto al valor original de esa señal. En nuestro caso, la modificación es la compresión y descompresión con perdida dadas por el algoritmo JPEG. Luego, lo que se toma como señal original es la imagen en el momento anterior a utilizar JPEG, y lo que se toma como señal modificada es la que se obtiene luego de realizar los diferentes pasos de compresión y descompresión.

El PSNR se define como:

$$PSNR = 10 \cdot \log_{10} \left(\frac{MAX_I^2}{MSE} \right) \tag{3}$$

En donde MAX_I se refiere al rango máximo de la señal en cuestión. Es decir que si estamos trabajando con imágenes en donde cada pixel se representa como 8 bits, el rango máximo de la señal será 255.

Por otro lado, MSE se refiere al Error Cuadrático Medio y se define como:

$$MSE(X) = \frac{1}{h \times w} \sum_{i=1}^{h} \sum_{j=1}^{w} (\hat{X_{ij}} - X_{ij})^2 \tag{4}$$

En donde X es una señal bidimensional de altura h y ancho w. $\hat{X}$ es la misma señal luego de sufrir la modificación.

Es decir que el MSE es una medida de que tan alejada esta la señal reconstruida de la señal original, y el PSNR es una medida de calidad que se encuentra en función de relación que existe entre la amplitud de la señal y la diferencia entre las señales.

A continuación se presenta la tabla de los PSNR obtenidos con las diferentes señales de entrada.

Señal	PSNR
Imagen	33.9425
Webcam	38.1875
VideoAvi	29.2095

Resultados PSNR

Los resultados que se ven en la tabla indican que todas las diferentes señales tiene valores de PSNR dentro de los esperados. En general, la literatura de compresión de imágenes coincide en que valores de PSNR superiores a 20db es una buena reconstrucción. Esto se condice con el hecho de que el algoritmo JPEG es cotidianemente usado para comprimir imágenes sin percibirse deterioros importantes de las mismas.

A continuación se observa la imagen *Lena.bmp* antes y despues de algoritmo JPEG:

Lena original

Lena reconstruida

Con las anteriores imágenes se puede ver que la reconstrucción es visualmente aceptable, presentando muy buenos resultados.

3.1.2. Entropía

La entropía de una imagen es una métrica que sirve para analizar el potencial nivel de compresión alcanzable.

Existen otras métricas para medir cuantó se comprimió una determinada señal. En particular, la métrica más obvia y correcta es la simple comparación entre el tamaño que ocupa la señal antes de ser procesada y el tamaño que ocupa luego de sufrir la modificación, en este caso por el algoritmo JPEG.

Si bien dicha métrica es la correcta para analizar una señal en particular, no suele ser la mejor medida para analizar como se comporta en líneas generales un algoritmo de compresión. Esto sucede debido a que cuando se comprime una imagen o un video, en el archivo que se transfiere hay más información que simplemente la matriz de valores correspondientes a la imagen. Además de esta matriz hay más imformación como headers, timestamps, etc.

Es por esto que una buena medida para analizar un compresor es medir la relación entre las entropías de la matriz de valores de la imagen de entrada, y la matriz de valores de la imagen comprimida. Esto es así debido a que el algoritmo JPEG comprime con el algoritmo de Huffman para el cual se encuentra muy estudiada su performance, pudiendose probar que la longitud media de código al aplicar este esquema de codificación no puede superar en más de un punto a la entropía [2].

De esta manera, si bien no se está midiendo los tamaños exactos antes y después de la compresión, se está midiendo un valor que se sabe que aproxima muy bien a la longitud media de código y que desprecia factores que no deberían ser tenidos en cuenta como los headers o los timestamps.

A continuación se presenta la tabla de resultados para la entropía:

Señal	Antes	Después
Imagen	7.4450	0.7022
Webcam	7.1828	0.4740
VideoAvi	7.2413	1.0555

Resultados Entropía

En esta tabla la columna *Antes* corresponde a la entropía de la señal en el momento previo a hacer cualquier transformación. Esto quiere decir que la longitud media de código si se aplicase Huffman sobre la señal original sería aproximadamente el valor presente en esta columna.

Por otro lado, la columna *Después*, presenta la entropía de la señal en el último punto del algoritmo de compresión antes de realizar la codificación de Huffman. Es decir que este número representa un aproximado de cuanto será la longitud de código media de la señal transmitida.

En esta tabla se puede ver que la conjunción de los procesos de la Transformada de Coseno Discreta junto al paso de cuantización logran disminuir fuertemente la entropía de la señal.

Se puede ver que si en cada tipo de señal se toma el peor caso posible, la relación de compresión es muy buena. El peor caso de cada uno de los tipos de señales correspondería a creer que Huffman lograría codificar con una longitud media de código igual a la entropía para la señal original, pero que lograría codificar con una longitud media de código de 1 más la entropía en el caso *Despues*. Aún tomando este peor caso, se puede ver que siempre se alcanzarían relaciones de compresión de 4 a 1.

3.2. Tiempos de cómputo

En esta sección nos dedicaremos a mostrar los resultados obtenidos en cuanto a los tiempos de cómputo del algoritmo en sus diferentes versiones entre lenguaje y compilador. Las opciones a analizar son:

- Algoritmo completo en lenguaje C con GCC como compilador.
- Algoritmo completo en lenguaje C con ICC como compilador.
- Algoritmo en lenguaje C, con optimización de la DCT en ASM, con GCC como compilador.
- Algoritmo en lenguaje C, con optimización de la DCT en ASM, con ICC como compilador.

Según lo planteado en la introducción de este trabajo, y lo revisado en la sección de desarrollo mediante el profiling del algoritmo, se espera que la optimización mediante ASM al algoritmo de la Transformada de Coseno Discreta traiga mejoras considerables, dado que gracias que se pudo identificar que este es el lugar donde más tiempo de cómputo se necesita.

En cuanto a la elección de compilador, a priori no se tiene una estimación de las diferencias en la performance, aunque sí se espera que el compilador ICC pueda introducir optimizaciones particulares que incurran en considerables mejoras.

A continuación se presentan los resultados obtenidos en cuanto a los tiempos de cómputo del algoritmo JPEG. En el caso de que la señal de entrada sea un video, se presenta el promedio de tiempo utilizado por frame.

Señal	C+GCC	C+ICC	ASM+GCC	ASM+ICC
Webcam	0.4327	0.1223	0.2164	0.0654

Resultados tiempo de cómputo en segundos

En esta primera instancia se presenta solamente los resultados provenientes de procesar 50 frames tomados de la cámara con una dimensión 640x480. En este caso los resultados corresponden a realizar todo el proceso de compresión y descompresión pero sin realizar la compresión de Huffman. Esto se decidió así ya que el paso de compresión de Huffman escribe para cada frame la información de la codificación a un archivo y prefirió omitirse los tiempos de entrada salida.

En la tabla presentada se pueden observa diferencias muy notorias entre las diferentes opciones de lenguaje y compilador. En cuanto a la distinta elección de compilador, bajo el mismo lenguaje, se puede ver que ambas opciones son consistentes llevando a una reducción en un cuarto de los tiempos. Por otro lado, el hecho de haber optimizado cierta parte del código en lenguaje Assembler llevó a una reducción de la mitad del tiempo.

Las dos mejoras presentadas combinadas, hacen que entre la mejor y la peor opción haya un *SpeedUp* de más de 6 veces la velocidad de cómputo.

Esto muestra una muy notoria y satisfactoria mejora en los tiempos de ejecución. De hecho las mejoras incluidas hacen que este algoritmo, que en realidad es para imágenes y está muy ingenuamente adaptado a videos, sea hasta considerable para su utilización en descompresión *on the fly* de videos comprimidos con este esquema.

A continuación se presentan los tiempos de ejecución realizando todos los pasos de compresión, incluido Huffman. Es decir que resume los tiempos que le tomaría a cada una de las partes involucradas en la utilización del standard.

Señal	C+GCC	C+ICC	ASM+GCC	ASM+ICC
Imagen	0.3940	0.2512	0.2135	0.2087
VideoAvi	0.1406	0.0988	0.0921	0.0920

Resultados tiempo de cómputo en segundos

En este caso, los resultados de la imagen corresponden a una con dimensiones 512x512 y el video tiene dimensiones 320x240. Lo que se puede ver en este resultado es nuevamente una considerable mejora entre las diferentes opciones y, condiciendose con el resultado de la tabla presentada anteriormente, se puede ver como este resultado es claramente dependiente del tamaño de la señal de entrada.

Aún teniendo en cuenta las escrituras a archivo, se puede ver como en la imagen se logra casi duplicar la velocidad de cómputo total del algoritmo y en el caso del video se ve una mejora de casi el 50 % del tiempo.

Todos estos valores muestran resultados muy interesantes sobre la optimización de código y la elección de un buen compilador. La mezcla de estas dos cuestiones mostraron reducciones de tiempo de cómputos muy remarcables, haciendo posible la utilización de la nueva implementación en contextos donde la primera implementación ingenua no parecía utilizable dados sus prohibitivos tiempos de cómputo.

4. Conclusiones

En la presente sección presentamos las principales conclusiones generales que se desprenden de la experimentación realizada. También se aprovecha para remarcar algunos puntos que se consideran interesantes sobre cuestiones que se fueron encontrando durante la realización del presente trabajo.

- La conclusión principal de este trabajo es claramente el hecho de que optimizar código a muy bajo nivel puede dar muy buenos resultados. Si bien esta conclusión es bastante obvia, parece importante remarcarla dada que la experimentación realizada arrojó evidencia muy fuerte de la misma. La optimización a bajo nivel no solo sirvió para mejorar los tiempos de corrida, sino también para poder obtener una implementación de JPEG utilizable en una situación de compresión *on the fly*, cuestión que no era posible con el código original.

  Es claro que la optimización de código mediante lenguaje ASM e instrucciones SIMD no es algo facil de llevar a proyectos en gran escala en donde las decisiones de diseño están por encima de la performance temporal. Sin embargo, parece interesante la opción de optimizar ciertos algoritmos de cómputo exhaustivo como en este caso fue la Transformada de Coseno Discreta, dentro de un marco de un algoritmo bastante más completo.

- En un segundo lugar, es importante destacar el rol del compilador en la utilización del algoritmo. La diferencia entre los compiladores utilizados no fue despreciable, viendose marcadas diferencias entre ellos. Esto resulta muy importante debido a que muchas veces se invierte mucho tiempo en pequeñas optimizaciones de código para lograr la más mínima mejora, cuando en realidad podría suceder que la variación a otro compilador traiga más beneficios, por ejemplo porque existen compiladores que hacen mejor uso de arquitecturas o de cómputos particulares.

- También nos parece importante remarcar lo interesante de haber aprendido a utilizar una herramienta como *gprof*. En cuestiones de optimización de código, el Profiling juega un papel importante. En vez de intentar ciegamente traducir todo el algoritmo a lenguaje ASM, *gprof* probó ser una herramienta realmente muy útil para saber donde poner el esfuerzo. Gracias a esta técnica se pudieron identificar fehacientemente las funciones que utilizaban más del 90 % del tiempo total, para traducir así sus implementaciones.

- Por otro lado, en cuanto a las tecnologías utilizadas para la implementación, el trabajo realizado resultó interesante para interactuar con herramientas que no habían sido utilizadas durante la cursada. En primer lugar se remarca lo interesante de haber necesitado interactuar con las librerías de opencv, ya que son librerías muy comunmente utilizadas en cualquier implementación de visión por computadora y de procesamiento de imágentes. Por otro lado, resultó novedoso haber tenido que entender los formatos internos de diferentes tipos de archivos para, por ejemplo, poder extraer la información de una imagen pudiendo descartar cualquier información adicional como headers o timestamps.

- Por último, destacamos lo interesante de haber podido aplicar las técnicas aprendidas en la materia para implementar un algoritmo como JPEG. El algoritmo JPEG es un algoritmo estandar dentro del procesamiento de imágenes, pero a la vez representa un caso de un algoritmo que se usa cotidianemente en muchisimas aplicaciones. También resultó interesante el hecho de haber atacado un algoritmo de un area distinta, lo cual nos obligó a estudiar conceptos alejados a la materia como la Transformada de Coseno Discreta, la noción de entropía y la métrica PSNR.

Referencias

[1] *http://opencv.org/.*

[2] N. Abramson. *Information theory and coding.* Electronics Series. McGraw-Hill, 1963.

[3] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition, 2001.

[4] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.