Organización del Computador II

Segundo Cuatrimestre del 2011

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

Trabajo Práctico Número 2

Filtros de imagen

Lider espiritual: Angiulino "El chino" Cubrepileta

Integrante	LU	Correo electrónico
Celave, Martin	530/08	tolacelave@gmail.com
Colombo, Ricardo Gaston	156/08	ricardogcolombo@gmail.com
Lores, Fernando	718/01	ferlores@gmail.com

Índice

1. Introducción

El objetivo de este informe es describir el proceso de como la aplicacion algoritmica de filtros sobre imagenes puede ser optimizada notablemente(en comparación con ANSI **C**) si se aprovecha el modelo de instrucciones SIMD.

Para ello implementamos siete algoritmos en **C** y en *assembler*. Los mismos recorren las imágenes aplicando diferentes cálculos. Las implementaciones de alto nivel son bastantes triviales y directas gracias a la posibilidad que nos brinda el lenguaje para acceder a las distintas posiciones de memoria. Sin embargo la gran desventaja que tenemos aquí es que accedemos y procesamos los datos de a uno. En cambio, en la implementación de bajo nivel, si bien podemos acceder a una gran cantidad de datos por vez, el acceso a ellos hace que los algortimos a veces se vuelvan ingeniosos.

En la siguiente sección explicaremos como encaramos la implementación de los filtros pedidos. Primero describiremos la estrategia que elegimos para iterar las imágenes junto con sus variaciones. Luego nos abocaremos a describir la mecánica que utilizamos en cada algoritmo para procesar los datos. Como dijimos antes, las implementaciones en alto nivel son bastantes directas, por lo que no nos detendremos mucho explicándolas. El lector puede remitirse driectamente al código para un mayor detalle de los mismos.

En la sección 3 realizaremos mediciones de las distintas implementaciones sobre un conjunto de imagenes dado, comparando y analizando los resultados obtenidos. Es claro imaginar que debería haber una gran diferencia en cuanto *performance* y esperamos a lo largo de este informe verificar dicha conjetura. Sin embargo tambien veremos que existen casos menos favorables que otros donde si bien la mejora es grande, comparada con otros casos, no lo es tanto.

Por último en la sección 4 recapitularemos los resultados obtenidos, analizando la relación costo-beneficio de las distintas implementaciones. Ademas comentaremos las principales dificultades encontradas y otras experiencias que obtuvimos durante la realización de este trabajo práctico.

2. Algoritmos

A la hora de implementar los algoritmos en *assembler* decidimos utilizar la tecnología SSE version 2. Si bien versiones mas actuales proveen instrucciones que en ciertos casos hubiesen sido más efectivas para el procesamiento de bytes empaquetados, nos era difícil desarrollar en nuestras casas, donde la version SSE de los procesadores no era tan avanzada.

2.1. Iterarando imágenes

La primer decisión a tomar a la hora de implementar los algoritmos fue como recorreremos las imágenes. En **C** optamos por la manera convencional, recorriendo secuencialmente la matriz, leyendo de a píxeles y cuando llegamos al final de cada fila salteamos el padding. Es claro que hay maneras más eficientes pero nos pareció apropiado tener esta implementación como base de comparación para todas las mejoras que implementaremos.

Por otro lado, la implementación en *assembler* fué mas ingeniosa ya que al utilizar la tecnología SSE debiamos, en principio, aprovechar al máximo lectura de a 16 bytes. Los problemas se presentan cuando quiero asegurarme de que estoy procesando todos los píxeles y que estoy evitando correctamente el *padding*. En este trabajo práctico procesamos imágenes a color y en escala de grises. Aunque la estrategia fue la misma, se presentaron sutiles diferencias para estos dos casos.

2.1.1. Imágenes escala de grises

En este caso cada píxel es representado por un byte. La iteración en **C** es bastante simple (Algoritmo 2.1). En cambio en *assembler*, cada vez que accedemos a memoria con un registro SSE estamos cargando 16 píxeles. Por lo tanto la estrategia será recorrer cada fila realizando sucesivas lecturas hasta que queden menos de 16 píxeles para que termine la fila. Para procesar los últimos datos, reposicionamos el índice para cargar los últimos píxeles y volvemos a iterar. Es

claro que algunos píxeles serán recalculados, pero de esta manera podemos reutilizar el código de las iteraciones normales. Cuando inicio el proceso de la nueva fila, avanzo el índice teniendo en cuenta el *padding*. Podemos observar el pseudocódigo en el Algoritmo 2.2. Cabe aclarar que los pseudocódigos presentados en el informe son de caracter ilustrativo, con el objetivo de explicar las estrategias utilizadas. Para un pseudocódigo mas detallado remitirse a los comentarios del código *assembler*.

Algoritmo 2.1 ITERACIÓN EN C - ESCALA DE GRISES

```
1: for i = 0 to n do
2:     fila ← i × row_size
3:     for pos = 0 to m do
4:         dst[fila + pos] ← procesar(src[fila + pos])
5:     end for
6: end for
```

Algoritmo 2.2 ITERACIÓN EN ASSEMBLER - ESCALA DE GRISES

```
1: for i = 0 to n do
2:     fila ← i × row_size
3:     for pos = 0 to m do
4:         dst[fila + pos] ← procesar_16(src[fila + pos])
5:         pos ← pos + 16
6:         if pos = m then
7:             procesar_siguiente_fila
8:         else if pos > m then
9:             pos ← w - 16
10:        end if
11:    end for
12: end for
```

2.1.2. Imágenes a color

En este caso cada píxel de las imágenes de entrada esta representado por tres bytes(BGR). La única modificacion que sufre la implementacion en **C** es que a la hora de acceder a la matriz de entrada multiplicamos el iterador de la columna por tres. Además podemos acceder a los distintos colores RGB desplazandonos cero, una o dos posiciones a partir de esa posición (Algoritmo 2.3).

Algoritmo 2.3 ITERACIÓN EN C - COLOR

```
1: for i = 0 to n do
2:     fila_s ← i × src_row_size
3:     fila_d ← i × dst_row_size
4:     for pos = 0 to m do
5:         dst[fila_d + pos] ← procesar(src[fila_s + 3 ∗ pos])
6:     end for
7: end for
```

En *assembler*(Algoritmo 2.4), las lecturas de a 16 bytes nos dejan en los registros 5 píxeles completos mas el primer byte del siguiente. Aquí decidimos descartar ese último byte, procesar los datos y escribir en la imagen de salida solamente los resultados obtenidos. Es decir que mientras la imagen de entrada es recorrida de a 15 bytes(5 píxeles), la imagen de salida es recorrida de a 5 bytes. Por este motivo resulta imperioso llevar dos iteradores de posición.

También es necesario ajustar la ultima iteración de cada fila. Cuando la lectura de los últimos 16 bytes, estos no quedan cargados como queremos. Para poder procesar los últimos 5 píxeles con las mismas instrucciones que tenemos en el ciclo debemos realizar un desplazamiento a derecha de un byte (figura 1).

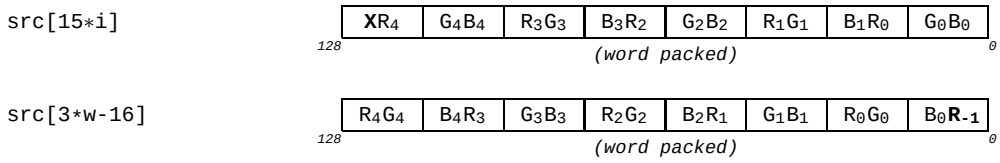

Figura 1: diferencia en la carga en la última iteración de imágenes a color

Por otro lado, una vez que realizamos la ultima iteración, debemos ajustar la condición de corte(linea 8). Como estamos avanzando de a 15 bytes, vamos a haber finalizado la columna cuando la diferencia entre el iterador y el ancho total en bytes de la imagen de entrada sea igual a uno. Es fácil de ver que esto solo sucede cuando se reajusta el índice(linea 11), ya que no existe un caso donde falte procesar un solo byte si vengo iterando de a multiplos de 3.

Algoritmo 2.4 ITERACIÓN EN ASSEMBLER - COLOR

```
1: for i = 0 to n do
2:     fila_s ← i × src_row_size
3:     fila_d ← i × dst_row_size
4:     for pos = 0 to m do
5:         dato_16 ← src[fila + pos]
6:         dst[fila + pos] ← procesar_16(dato)
7:         pos ← pos + 15
8:         if pos + 1 = m then
9:             procesar_siguiente_fila
10:        else if pos > m then
11:            pos ← w - 16
12:            dato_16 ← desplazar_der_1b(src[fila + pos])
13:            goto linea 6
14:        end if
15:    end for
16: end for
```

2.2. Separar canales

El objetivo de este filtro es generar tres imágenes distintas en escala de grises. Cada imagen debe contener en cada píxel lo correspondiente al valor de la capa que estamos separando del píxel original. La implementacón en **C** es trivial, ya que puedo direccionar a cada valor del píxel que estoy recorriendo de una manera directa[1]. por otra parte, en *assembler* debemos aislar cada valor RGB de manera que posteriormente podamos manipularlos. En general esto se hace con instrucciones de desempaquetado. Sin embargo en nuestro caso preferimos utilizar otra estrategia. La idea es realizar desplazamientos (empaquetados de words) a izquierda y derecha para desempaquetar todos los valores. En la figura 2 podemos observar como con un desplazamiento a derecha de un byte (linea b) podemos desempaquetar ciertos datos, mientras que con un desplazamiento primero a izquierda y luego a derecha, logramos separar los valores que estaban faltando (linea c).

[1] para más información de la implementación de la función remitirse al código fuente

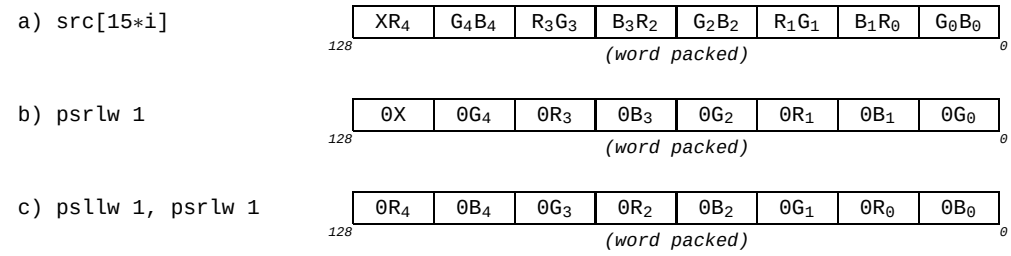

Figura 2: Estrategia de desempaquetamiento imágenes a color

Una vez desempaquetados estos datos, preparamos máscaras para colocar ceros en los words que no necesitamos. Si prestamos atención en como quedan conformados los registros desempaquetados, notamos que a veces vamos a necesitar filtrar tres words y a veces dos words. Es por eso que solo dos mascaras (desplazandolas a izquierda y derecha) fueron suficientes para separar todos los B, luego los R y por útimo los G. Las máscaras estan compuestas por words de *unos* donde estan los datos que queremos y ceros donde estan los valores que no necesitamos en este momento. En la figura 3 podemos observar como quedan los registros despues de utilizar las máscaras para filtrar los valores G (lineas d y e) y como queda combinado en un solo registro todos los valores que necesitamos (linea f).

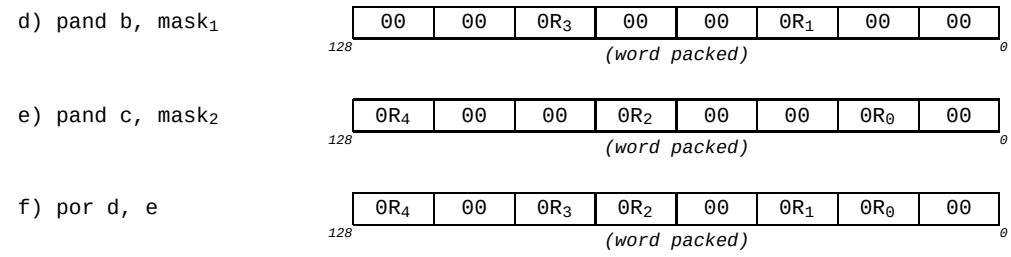

Figura 3: filtrado y combinación usando máscaras 0000FF0000FF0000 y FF0000FF0000FF00

Ahora resta reacomodar los datos para poder escribirlos en memoria. Para esto usamos las instrucciones de intercambio que el procesador nos ofrece: *pshufd, pshuflw y pshufhw*. Con estas tres instrucciones es posible realizar las tres funciones mas comunes de intercambio a nivel de words: *broadcast*, *swap*, *reverse*[2]. La figura 4 muestra como reacomodamos los words (lineas g y h) y empaquetamos para obtener en los bytes menos significativos los cuatro píxeles ya procesados (linea i). En este punto solo nos queda escribir en la imagen destino esos 4 bytes, desplazar el registro SSE de modo que el quinto valor quede en la parte menos significativa, y también escribirlo. Cabe destacar que para hacer esta última escritura ultilizamos un registro de propósito general, porque no existe una instrucción que nos permita copiar un byte determinado del registro SSE a memoria.

El proceso de los otros canales son similares. Siempre utilizamos la misma estrategia: separamos los valores, los enmascaramos, los combinamos, y por ultimo realizamos los intercambios necesarios para poder bajarlos a memoria. Para mayor detalle de como procesamos los canales B y G remitirse a los comentarios del código adjunto.

[2]Intel©64 and IA-32 Architectures Optimization Reference Manual, sección 5.4.11

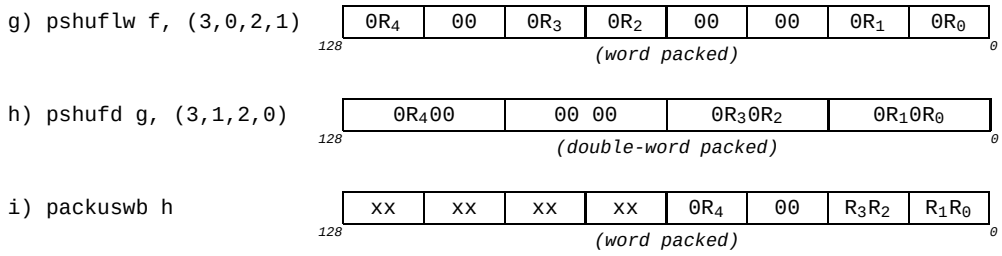

Figura 4: *swap* de words y double-words

2.3. Monocromatizar

2.3.1. Monocromatizar $\epsilon = 1$

El objetivo de la funcion monocromatizar es convertir una imagen color a escala de grises. En la implementación en **C** la imagen se recorre linealmente como una matriz, píxel a píxel, accediendo a los valores de cada componente de color para realizar el siguiente cálculo: **(R+2*G+B)/4**, siendo el resultado lo que se almacena en la imagen destino.

La idea en *assembler*es similar solo que, aprovechando la capacidad de almacenamiento de los registros de 128 bits y teniendo en cuenta que en cada uno de estos registros entran 16 bytes, podemos procesar de hasta 5 píxeles por cada lectura de memoria, en lugar de avanzar de a uno. En importante destacar que, como dijimos en la sección 2.1.2, inicialmente cuando leemos en memoria la imagen fuente, también cargamos un tercio del sexto píxel sin embargo esta información se descarta.

Lo primero que hacemos, es triplicar lo leido en memoria y aplicar desplazamientos a derecha con el fin de queden alineados los registros para una posterior suma. La figura 5 muestra como quedan los registros después de dichos desplazamientos. En este momento, por ejemplo, en el byte menos significativo de cada registro tenemos el valor de color del primero píxel. Los valores del siguiente píxel se encuentran en la cuarta posición, y asi sucesivamente se pueden identificar cuales son los bytes que contienen los valores alineados.

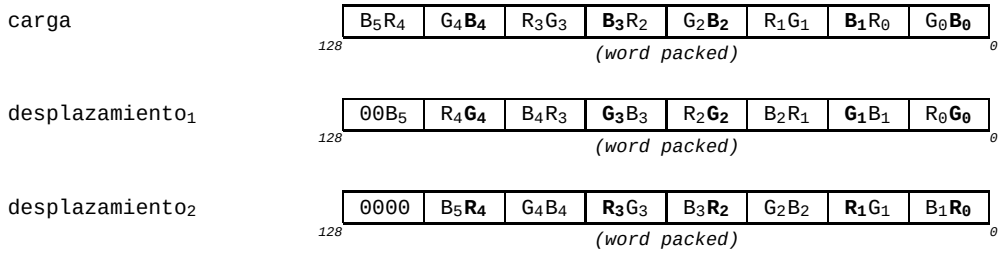

Figura 5: como se corren los datos de manera favorable

Es claro de observar que no es posible hacer el cálculo anteriormente descripto sin antes desempaquetar los datos, ya que de lo contrario se perderia podria saturar alguna suma. No obstante, también es claro que el resultado deberia poder ser expresado en byte. Es importante destacar que no utilizamos punto flotante ya que nuestra estrategia fue primero realizar las sumas y las multiplicaciones y por ultimo aplicar la división. Es por esto que podemos trabajar con enteros sin perder precisión.

Por cada uno de los registros mencionados anteriormente, desempaquetamos a words tomando como segundo operando un registro de 128 bits con ceros, y los dividimos en partes bajas y altas, como muestra la figura 6. Luego multiplicamos el segundo registro (el que contiene los valores correspondientes al verde), luego sumamos todas las partes altas entre si y las partes bajas entre

si, y por último dividimos los resultados por cuatro.

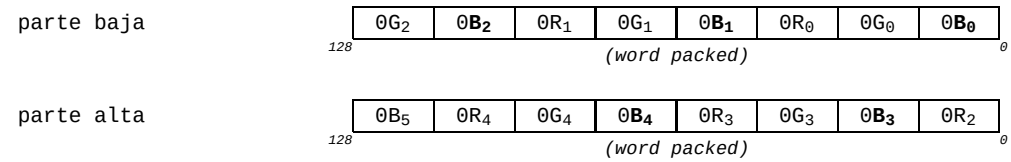

Figura 6: desempaquetado del registro base

Para multiplicar y dividir usamos las instrucciones *psrlw* y *psllw* para realizar un desplazamiento de bits a derecha e izquierda respectivamente en cada word del registro, pues el multiplicando y el divisor son potencias de dos.

Una vez realizados los cálculos necesarios notemos que tenemos los resultados en dos registros separados con lo cual debemos proceder a unirlos para luego copiarlos.

El proceso para unirlos consiste en dos partes, una es aplicar una máscara para generar ceros en las posiciones que tengo basura en los dos registros, y luego combinarlos realizando un *pand*. De esta manera me quedan los datos necesarios separados por ceros (figura7).

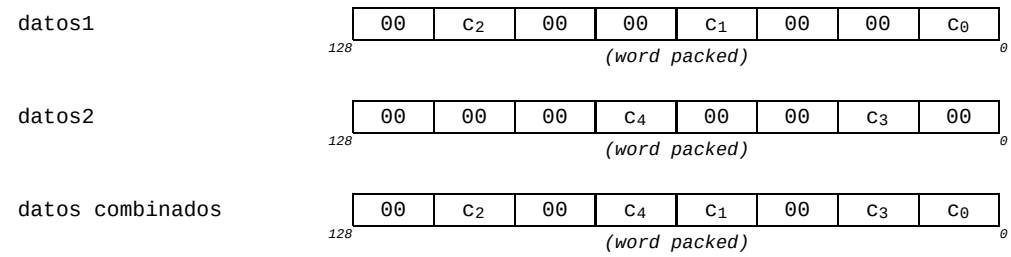

Figura 7: filtrado y combinación usando máscaras 00FF0000FF0000FF Y FF0000FF0000FF00

Una vez que los resultados quedaron combinados en un único registro utilizamos instrucciones que provee la arquitectura para intercambiar datos dentro de un registro (similar a lo realizado en **separar canales**) con el fin de colocarlos de manera contigua y luego empaquetar. Si bien estas instrucciones pueden afectar la performance son realmente útiles e indispensables para obtener la funcionalidad deseada. Una vez empaquetado nuevamente a bytes, copiamos el *double word* menos significativo a memoria y con eso cubrimos los primeros 4 píxeles procesados. Luego hacemos un corrimiento del registro a fin que el último dato procesado quede en la posción menos significativa. Aqui nuevamente nos vemos obligados a usar un registro de propósito general para copiar el último byte a memoria.

2.3.2. Monocromatizar $\epsilon = \infty$

Esta funcion es bastante similar a la anterior. La implementación en **C** es identica a la anterior solo que esta vez en lugar de combinar los valores de color, debemos quedarnos con el máximo.

El procedimiento para la implentación en *assembler* es similar cuando $\epsilon = 1$, al menos la primera parte cuando triplicamos los registros y los desplazamos adecuadamente para alinear los valores de color. Es interesante notar que contamos con una ventaja con respecto al caso anterior ya que no necesitamos desempaquetar pues el set de instrucciones cuenta con una funcion (*pmaxub*) que calcula en máximo byte a byte entre dos registros SSE. Esto no solo facilita el desarrollo sino que además incrementa la performance con respecto al otro monocromatizar, ya que nos ahorramos la conversión de empaquetados. El resultado despues de calcular el máximo entre las tres copias del registro puede observarse en la figura 8.

Una vez obtenidos los máximos de cada píxel(solo nos interesan 5 de ellos), debemos reubicarlos para que queden de manera contigua. Para esto utilizamos la misma estrategia que en separar

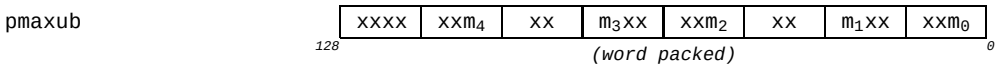

Figura 8: registro una vez calculado los valores máximos para cada píxel

canales. Duplicamos el registro y realizamos un corrimiento de un byte a izquierda en cada *word* para obtener ceros a la izquierda de cada una de las partes bajas de cada *word*. En la otra copia hacemos un corrimiento a derecha y luego a izquierda separando esta vez los valores que fueron perdidos en la operación del otro registro (figura 9).

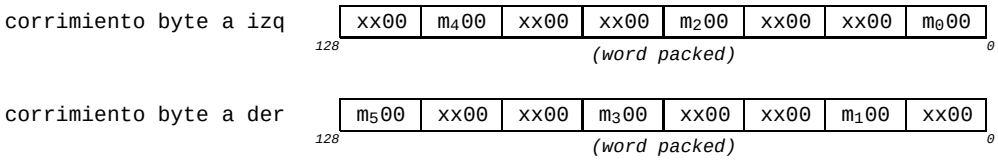

Figura 9: corrimientos para generar ceros

Una vez logrado esto debemos combinar los resultados en un único registro, para ello utilizamos máscaras como lo hicimos en $\epsilon = 1$ para colocar ceros, y luego realizamos un *por* para unirlos, como muestra la figura 10.

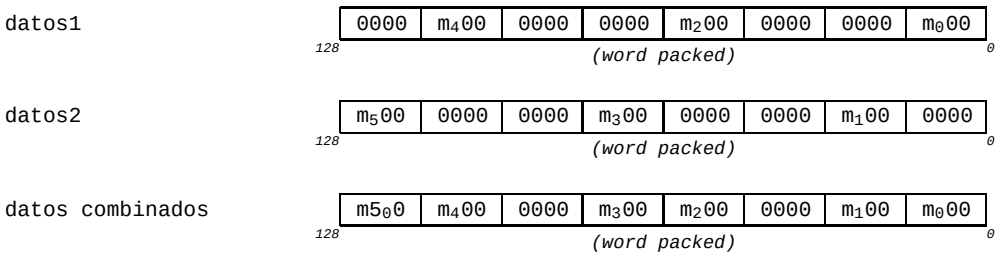

Figura 10: filtrado y combinación usando máscaras 00FF0000FF0000FF y FF0000FF0000FF00

Nuevamente ordenamos los contenidos de los registros utilizando las instrucciones de intercambio de words y double words, empaquetamos y de forma análoga a otro monocromatizar, realizamos el proceso de copiado a la imagen destino.

2.4. Umbralizar

Este filtro proproduce imágenes con solamente tres colores (blanco, gris y negro), dependiendo del valor que tenga cada píxel. El procesamiento de cada píxel esta definido por la siguiente función:

$$I_{out}(p) = \begin{cases} 0 & \text{si } p \leq umbralMin \\ 128 & \text{si } umbralMin < p \leq umbralMax \\ 255 & \text{si } p > umbralMax \end{cases}$$

La implementación en **C** es nuevamente trival. Recorremos, como siempre en estos casos, secuencialmente de a un byte mientras que el procesamiento de cada píxel se reduce a dos comparaciones para elegir el valor correcto del resultado.

En cambio, para la implementación en *assembler* procesaremos de a 16 bytes, y no tendremos que desempaquetar los valores

tenemos varios pasos. Primero vimos que para realizar las comparaciones era necesario tener los umbrales empaquetados(de a bytes) en los registros de 128 bits. Para esto cargamos los valores

que recibimos como parámetro de entrada en un registro de proposito general. Luego realizamos desplazamientos de a de 8 bits a la izquierda y sucesivas sumas del valor hasta que conseguimos un doble *word* lleno. Despues movemos a dos registros de 128 bits el registro de propósito general con el umbral mínimo y máximo y utilizamos la instruccion de intercambio *shufle* de double words para replicar el valor a lo largo de todo el registro SSE.

Por otro lado también necesitabamos un registro que contenga el número 128, que representa el gris en la imágen. Para obtenerlo realizamos el mismo procedimiento que hicimos con los valores mínimos y máximos, solo que el valor inicial lo obtenemos a aprtir de la constante 8080h[3] .

El resultado de este proceso se ve en la figura 11.

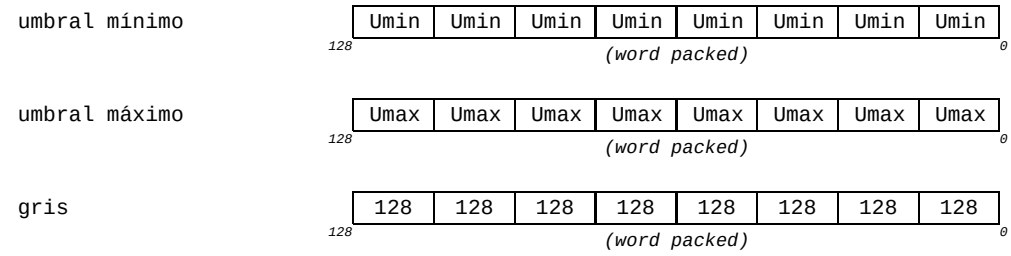

Figura 11: constantes necesarias para el proceso del algoritmo

Una vez ya definidos los umbrales en los registros comenzamos a procesar los datos. Para saber cuales son los elementos que estan por sobre el umbral hacemos una resta saturada entre el registro que contiene el umbral máximo y los valores originales de la imágen. De esta manera nos quedan ceros en los valores que eran mayores o iguales al máximo y valores distintos a cero en las otras posiciones. Entonces utilizamos la instrucción *pcmpeqb* para armar una máscara a partir de la comparación de este resultado contra un registro lleno de ceros, obteniendo por resultado Fh en las posiciones donde el valor original era mayor al umbral y ceros en las otras posiciones.

Ahora realizo un procedimiento similar. Hacemos una resta saturada es entre el valor original y el valor mínimo, dando por resultado un registro con ceros en los bytes que estan debajo del umbral mínimo (nuevamente gracias a la saturacion). Luego, utilizando la instrucción *pcmpeqb* contra un registro lleno de ceros, armamos otra máscara con Fh en los bytes que estan debajo del umbral mínimo.

Luego combinamos las dos máscaras con un *por* y al resultado le aplicamos la instrucción *pcmpeqb* con un registro con ceros. Esta última operacion es, en este caso, similar a una negacion bit a bit. Por lo que obtenemos Fh donde había antes ceros, es decir, en las posiciones donde se encuentran los valores medios de la imagen.

Por último, utilizamos esta última máscara junto con el registro que contiene la constante 128 para realizar un *pand* y obtener como resultado un registro con los grises en sus posiciones correspondientes. Por último solo queda colocar blancos en las posiciones donde la imagen es mayor al umbral. Recordamos que tenemos calculada una mascara con Fh en esas posiciones y ceros en las demás, por lo que un simple *por* combinaria este registro el con registro con grises, obteniendo los bytes necesarios para escribirlos en la nueva imagen.

2.5. Invertir

En este filtro, dada una imágen en escala de grises, debemos calcular la resta entre el valor máximo de un píxel y el valor de la imagen, es decir, calcular la resta entre 255 y el píxel dado.

La implementación en **C** es mas que trivial, ademas del ciclo ya explicado (con su correspondiente procesamiento lineal de a byte), es solo una resta lo que realiza la función.

Por otra parte en *assembler* lo primero que realizamos es cargar un registro con el valor 255 en cada byte. Para esto usando cualquier registro aplicamos la instrucción *pcmpeqb* contra si

[3]el valor 80h corresponde al valor 128 en decimal

mismo, dejando todos bits en uno, con lo cual el registro queda completo del valor deseado. Luego realizamos la resta entera saturada *psubb* (aunqe podria ser tambien la no saturada), asi de esta manera obtenemos los valores procesados.

2.6. Normalizar

El objetivo de este algoritmo es modificar cada pixel para ampliar el rango de valores de toda la imagen. Para esto utilizamos la fórmula general (ecuación 1), estableciendo los valores máximos y mínimos posibles para el caso de las imagenes en escala de grises (255 y 0 respectivamente). Luego realizamos algunas operaciones álgebraicas a fin de poder establecer una constante de multiplicación que procesará cada pixel.

Entonces el algoritmo queda divido en tres partes: la primera busca el máximo y mínimo de la imagen, la segunda calcula la constante K mientras que la segunda procesa los valores aplicando la ecuación ya descripta.

$$
\begin{aligned}
I_{out}(i,j) &= (I_{in}(i,j) - max) \times \left(\frac{a-b}{max-min}\right) + a \\
I_{out}(i,j) &= (I_{in}(i,j) - max) \times \left(\frac{255-0}{max-min}\right) + 255 \\
I_{out}(i,j) &= \frac{255 \times (I_{in}(i,j) - min)}{max-min} \quad \textit{factor comun (max - min)} \\
\text{Sea } K &\leftarrow \frac{255}{max-min} \\
I_{out}(i,j) &= K \times (I_{in}(i,j) - min)
\end{aligned}
\tag{1}
$$

Ecuación 1: Desarrollo de la fórmula de normalizado

Búsqueda de máximo y mínimo

Para encontrar los valores máximos y mínimos nos valemos de las instrucciones *pmaxub* y *pminub*. Las mismas comparan los valores byte a byte quedandose con los correspondientes a cada función. Entonces, despues de recorrer la matriz obtenemos dos registros, uno con los candidatos a máximo y otro con los candidatos a mínimo (paso 0).

Lo que queda por hacer entonces es comparar los bytes dentro de cada registro para identificar al mayor y menor. Para ello duplicamos el registro y realizamos sucesivos intercambios para comparar todos con todos. Iniciamos invirtiendo la parte alta por la parte baja de registro y volviendo a utilizar las instruciones de máximo y mínimo para comparar (paso 1). Entonces ahora tenemos ocho candidatos en la parte alta del registro. Repetimos el proceso utilizando intercambiando los dos double-words mas significativos (paso 2). Ahora tenemos cuatro candidatos, una vez mas repetimos la operacion intercambiando ahora los words mas significativos (paso 3).

En este punto tenemos dos candidatos en el word mas significativo del registro. Para compararlos, duplicamos el registro y hacemos un desplazamiento a izquierda de un byte (paso 4). De esta manera, al comparar queda en el byte mas significativo el valor buscado.

Una vez obtenidos estos dos valores, hacemos un desplazamiento a izquierda de tres bytes, y en el *double word* mas significativo nos queda el valor deseado. Ahora hacemos un *broadcast* a todos los *double word* y el registro nos queda con los valores deseados empaquetados a *double word*.

La Figura 12 muestra como se van reduciendo los candidatos aplicando las operaciones antes descriptas.

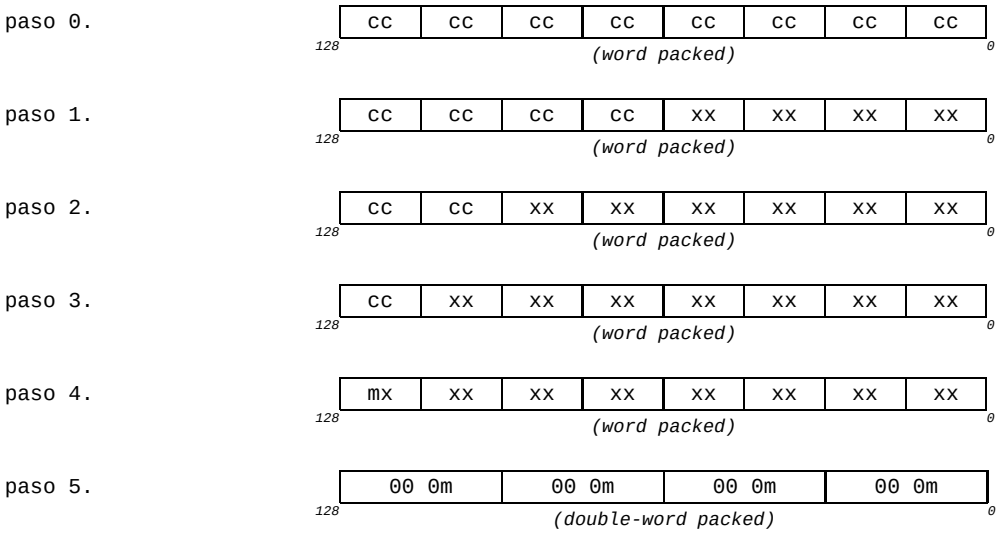

Figura 12: *busqueda de máximo y mínimo*

Cálculo de la constante K

Para calcular la constante K a partir de el valor máximo y mínimo obtenidos en el paso anterior, es importante tener en cuenta que dicho valor es un número decimal. Es por esto que para no perder realizamos las cuentas utilizando como tipo de dato empaquetado al decimal con punto flotante de precisión simple. Entonces lo primero que realizamos fue una conversion de los registros que contenian el máximo y mínimo empaquetados en *double words* a un formato empaquetado *float single presicion*, luego, realizando las operaciones descriptas en la ecuación 1 obtenemos un registro con la constante empaquetada como *float*. Lo mismo hacemos con el registro que contiene el mínimo, pues lo necesitamos en los cálculos.

Procesamiento de los píxeles

En esta etapa recorremos la matriz de a 16 píxeles, la desempaquetamos a *words* separando y parte alta y baja. Cada parte es a su vez desempaquetada a *double words* y convertida a empaquetados de punto flotante. Entonces, por cada lectura nos quedan cuatro registros con cada uno cuatro valores listos para ser procesados sin perder precisión. A los mismos se les resta el valor mínimo con la resta de punto flotante (*subps*) y se multiplica por la constante K (*mulps*).

Una vez calculado cada píxel se realiza el camino inverso al descripto empaquetando el resultado. Primero se convierte el resultado de empaquetado de punto flotante a *double word*, luego a *word* y por último a byte. Es importante resaltar que los empaquetamientos que realizamos son con saturación signada. Es decir que si el valor esta fuera del rango permitido se satura al máximo o al mínimo según corresponda.

Una vez que el resultado es empaquetado, éste se baja a memoria y se continua con la siguiente iteración.

2.7. Suavizado Gaussiano

El objetivo de este filtro es obtener una imagen resultante con reducción de ruido y difuminación. Para ello debemos procesar cada píxel utilizando la información de sus aledaños. La idea es calcluar un tipo de promedio ponderado de los valores del punto que estoy analizando con sus nueve vecinos inmediatos. En la figura 13 podemos ver los coeficientes de multiplicación de los píxeles.

1/16	2/16	1/16
2/16	4/16	2/16
1/16	2/16	1/16

Figura 13: Promedio ponderado del entorno

Con lo cual el píxel resultante sera el píxel que se encuentra en el centro del entorno, y el mismo tendra el siguiente valor:

$$\begin{aligned} I_{out}(i,j) = \quad & I_{in}(i-1,j-1)\cdot 1/16 + I_{in}(i-1,j)\cdot 2/16 + I_{in}(i-1,j+1)\cdot 1/16 + \\ & I_{in}(i+0,j-1)\cdot 2/16 + I_{in}(i+0,j)\cdot 4/16 + I_{in}(i+0,j+1)\cdot 2/16 + \\ & I_{in}(i+1,j-1)\cdot 1/16 + I_{in}(i+1,j)\cdot 2/16 + I_{in}(i+1,j+1)\cdot 1/16 \end{aligned}$$

Aca también la implementación en C es trivial teniendo la formula mencionada anteriormente. Básicamente nos valemos de que acceder a las posiciones de la matriz en **C** es inmediato y directo. Es decir que en cualquier momento podemos acceder a cualquier posicion de la matriz con los índices correspondientes.

Por el lado de la implementacion en *assembler* vuelve a ser mas compleja. Al ver que cada píxel debía ser la suma de su entorno ponderado, decidimos dividir en 3 partes el algoritmo para tratar cada una de las lineas de la matriz resultante.

Comenzamos el algoritmo realizando una lectura de 16 bytes, luego realizamos una segunda y tercer lectura sucesiva(para guarda información del entorno) desplazando el puntero base en uno y dos bytes respectivamente. La figura 14 muestra esas lecturas. El objetivo aca nuevamente es que queden alineados los datos que necesito sumar.

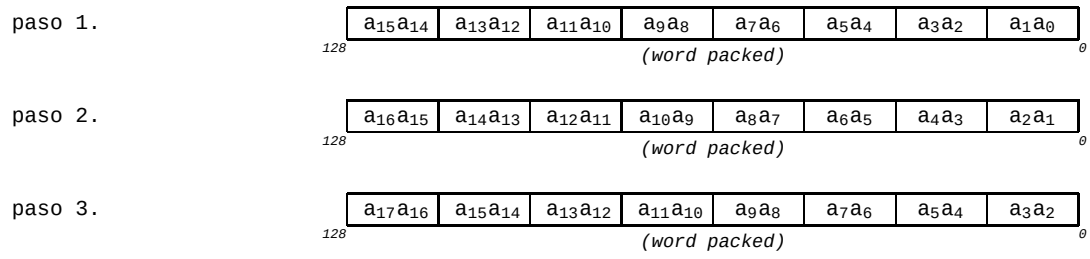

Figura 14: Levantado de datos de la primer linea

Luego desempaquetamos los datos para operar como words, obteniendo asi dos registros por cada uno de los anteriores, uno para la parte baja y otro para la alta. Es importante destacar que los desempaquetamientos siempre los hacemos con registros que tienen todo ceros, de esta manera el valor que nos interesa se ve inalterado.

Luego se utiliza la operación *psllw* para realizar un desplazamiento empaquetado de a words de un bit (que sería equivalente a multiplicar por 2) en los registros de la segunda lectura y sumamos las partes bajas guardándolas en los acumuladores.

Ya procesada la primer linea del entorno procedemos a la segunda. Sumamos a nuestro índice el ancho de la imágen (contemplando el *padding*) para pararnos en la segunda fila de la matriz, realizamos nuevamente tres lecturas desplazadas, desempaquetamos y multiplicamos los registros por sus correspondientes valores. En este caso 4 para el centro y 2 para los extremos. Aca nuevamente utilizamos instrucciones de desplazamiento para la multiplicación. Una vez procesada la parte alta y baja de la segunda linea, la acumulamos con los resultados de la primera.

Por último pasamos a la tercer linea de la misma manera que la anterior, y realizamos el mismo procedimiento: tres lecturas desfazadas, desempaquetamos, multiplicamos, y acumulamos los valores correspondientes a la parte alta y parte baja. En este punto sólo nos queda dividirlos por 16. Lo realizamos nuevamente con una operación de desplazamiento de bits.

Ahora solo nos queda empaquetar los datos con saturación y escribirlos en la imagen destino. Es importante notar que en este algoritmo utilizamos muchos registros debido a las lecturas desplazadas. En cierto punto no tuvimos mas registros libres por lo que tuvimos que utilizar una variable local de 16 bytes para guardar uno de los acumuladores. Que como trabajo a futuro ver si es posible realizar una implementación que no utilice esa memoria auxiliar, ni que realice tantos accesos a memoria. Probablemente utilizando la misma estrategia aqui descripta pero procesando de a menos datos, es decir utilizando solamente las información que obtenemos con una sola lectura y no con tres desfazadas.

Iterando la imagen

Primero tuvimos que tener en cuenta ciertas limitaciones en la el cálculo de los píxeles que nos imponía el procedimiento, ya que al utilizar un entorno de 3x3 para calcular el píxel del centro, en los bordes no es posible calcularlo. Es por esto que nuestro algoritmo recorre todo menos los bordes de la imagen de entrada.

Por otro lado, en ciclo se leen mas de 16 píxeles, a diferencia de los otros algoritmos, ya que al realizar 3 lecturas moviendose de a 1 píxel se estan levantando 18 y se procesan solo 15. Entonces debemos asegurarnos de siempre tener esos 18 píxeles delante nuestro. Si no los tenemos debemos realizar el reajuste para procesar los últimos píxeles.

3. Mediciones

Para realizar las mediciones utilizamos las herramientas provistas por la cátedra para la medición de ciclos de *clock*, utilizando una computadora con procesador Intel T2500 2.0GHz Core Duo. El objetivo será medir los distintos algoritmos, comparando las distintas versiones para distintos tipos de imagenes de entrada. Para ello creamos un conjunto de *scripts* de *bash* que realizan las distintas mediciones que necesitamos. Los mismos se encuentran en la carpeta mediciones junto con una explicación de como utilizarlo.

3.1. Estableciendo Criterios

Antes de comenzar tuvimos que tomar ciertas decisiones de cómo ibamos a realizar dichas mediciones. El primer problema que encontramos al utilizar el reloj del procesador para medir es que nuestro proceso puede llegar a ser interrumpido por otros procesos o por el sistema operativo. Hemos tomado dos medidas al respecto, la primera es ejecutar las mediciones estableciendo máxima la priodidad del proceso. Para ello Linux provee el comando `nice`, que ejecutándolo como *root*, permite decirle al sistema operativo que nuestras mediciones tienen prioridad sobre los otros programas en ejecución. Mientras que la segunda medida fue realizar mil mediciones por cada prueba, promediando el resultado. El número elegido aparece como una solución de compromiso con el tiempo de ejecución de las muestras.

Otro factor a analizar fue que muestras seleccionamos para medir. Debido a la naturaleza de los algoritmos implementados, podemos ver facilmente que la complejidad esta relacionada directamente con el tamaño de la entrada. Es decir que las dos implementaciones de cada filtro se van a comportar diferente dependiendo del tamaño de la imágen[4] y no del tipo de imagen a procesar. Por lo tanto podriamos tomar imagenes de distinto tamaño y comparar como se comportan los distintos filtros y las distintas implementaciones, observando las variaciones de *clocks* con respecto a la cantidad de píxeles procesados. Sin embargo elegimos otro camino. Decidimos utilizar un tamaño fijo (dentro de los posibles) de píxeles y analizar el comportamiento de los algoritmos al variar la relación entre la cantidad de filas y de columnas de la imagen.

Para las mediciones base utilizamos una imagen de 800x600. Es decir que estamos trabajando con imágenes de 480.000 píxeles (aprox. 0,5Mp). Luego tomamos distintas imágenes reacomodando las filas y columnas, según criterios que explicaremos en la siguiente sección. Sin embargo a veces

[4]Es importante aclarar que consideramos como tamaño de la imagen, la cantidad total de píxeles que esta tiene

no fue posible llegar a la misma cantidad exacta de píxeles. En estos casos elegimos combinaciones donde la diferencia sea mínima y podemos asegurar que en ningún caso la diferencia de tamaño supera un $0,1\,\%$ del total.

3.2. Análisis de factores

Antes de decidir las combinaciones de filas y columnas, analizamos con que tipo de imágenes nuestros algoritmos podían llegar a comportarse una una manera especial. Como conjetura inicial establecimos que no deberia haber fluctuaciones en el las funciones desarrolladas en **C** ya que el recorrido de la matriz es lineal y de a un elemento por vez, con lo cual las dimensiones no deberían ser un factor relevante.

En cambio, para el caso de las funciones en *assembler*, podemos observar que su gran ventaja es que pueden procesar de a muchos datos. Sin embargo su desventaja es que si la imagen no es múltiplo de la cantidad de píxeles que procesa por iteracion, entonces se debe realizar un reajuste (ver sección 2), con el objetivo de procesar los últimos píxeles de la fila.

Entonces, para elegir exactamente las combinaciones de filas/columnas tenemos que diferenciar dos tipos de filtros: Los que toman como parámetro de entrada imágenes a color (monocromatizar, separar canales) y los que toman imágenes en blanco y negro (umbralizar, invertir, normalizar, suavizar).

Para los que toman imagen blanco y negro, los algoritmos procesan de a 16 píxeles por vez, con lo cual si el ancho es multiplo de 16, el algoritmo realizaría una cantidad justa de lecturas en memoria (*caso favorable*). Por el contrario, cuando la anchura no es multiplo de 5 el algoritmo debe realizar una lectura extra al final de cada fila para procesar los bytes restantes(*caso desfavorable*). Es posible que esta lectura extra se haga solo para procesar un píxel. Es decir que estariamos recalculando 15 bytes para procesar solo uno. Esta penalización se vería acrecentada conforme aumenta la cantidad de filas.

En cambio, los algoritmos que procesan imágenes a color avanzan de a cinco píxeles. Por lo que podemos aplicar el mismo razonamiento que para los algoritmos en blanco y negro modificando el factor de multiplicidad a 5.

Cabe destacar que el filtro suavizar debido a su naturaleza procesa de a 15 píxeles, por lo que el caso favorable se calcula como un múltiplo de 15 sumándole dos píxeles más correspondiente a los bordes que no se procesan.

3.3. Eleccion de imágenes

Como primera medida, realizamos mediciones con una imágen de 800x600, para poder establecer un punto de comparación para las otras imágenes de entrada. Luego, teniendo en cuenta lo descripto en la sección anterior, elegimos tamaños de imágenes que provoquen casos favorables y casos desfavorables, para poder compararlos.

Para el segundo grupo de mediciones elegimos para los algoritmos en blanco y negro imágenes de 16 y 17 píxeles de ancho para los casos favorable y desfavorable respectivamente. Estos valores son los anchos más pequeños que producen los casos que buscamos. Por el contrario, para los algoritmos a color elegimos anchos de 10 y 11 píxeles. Para el filtro suavizar elegimos los anchos 17 y 18 píxeles. Es de esperar que en este conjunto de mediciones la penalización del *assembler* se vea exagerada.

Luego realizamos otro conjunto de mediciones con imagenes mas anchas. Intentando, a priori, diluir las penalizaciones cuando las haya. En este punto encontramos un imprevisto. La librería openCV genera un error al intentar abrir imágenes mas anchas que 1453 píxeles. Es por eso que elegimos un ancho lo mas cercano a ese valor y que fuera múltiplo de 16 y de 5, de esta manera nos serviría como caso favorable tanto para los algoritmos a color como los de blanco y negro. El ancho elegido fue 1440, ya que cumple las condiciones anteriores. Por otra parte elegimos como ancho para los casos desfavorables el valor 1441. Para el filtro suavizar elegimos los anchos 1442 y 1443 píxeles.

3.4. Experimentos

3.4.1. Caso base

En este experimento se miden todos los filtros en ambas implementaciones en una imagen de 800x600 píxeles. En el cuadro 1 podemos observar los siguientes índices:

- **ciclos**: ciclos de reloj consumidos por cada implementación.
- **diferencia de ciclos**: modulo de la diferencia entre ciclos de reloj en **C** y en *assembler*.
- **porcentaje de mejora**: porcentaje de eficiencia de un implementación con respecto a la otra.

En el cuadro podemos ver que en todos los filtros la implementación en *assembler* consume menos ciclos de reloj, un resultado que esta dentro de lo esperado, sin embargo llama la atención lo significativa que es esta diferencia. Por ejemplo en el filtro invertir, que es el que menos instrucciones realiza, obtuvimos un porcentaje de mejora superior al 96 %. El caso del filtro que menos porcentaje de mejora tuvo fue monocromatizar_uno, que no obstante, obtuvo una mejora superior al 50 %. Si bien suponiamos que podia haber una diferencia en la eficiencia no sabiamos que seria tan marcada, lo cual es muy positivo, ya que significa que dio réditos implementar en un lenguaje de más bajo nivel y mayor dificultad a cambio de un mejor rendimiento, siempre utilizando la tecnología *SSE*.

FUNCIÓN	# PIXELS	CICLOS C	CICLOS ASM	Δ CICLOS	% MEJORA
invertir	800x600	7.317.170	287.367	7.029.803	96,07 %
normalizar	800x600	23.331.351	1.539.564	21.791.786	93,40 %
umbralizar	800x600	13.538.943	540.008	12.998.935	96,01 %
suavizar	800x600	32.483.342	3.968.673	28.514.669	87,78 %
monocromatizar_uno	800x600	17.332.554	5.513.335	11.819.219	68,19 %
monocromatizar_inf	800x600	18.555.484	4.468.378	14.087.106	75,92 %
separar_canales	800x600	20.421.239	6.219.616	14.201.623	69,54 %

Cuadro 1: Resultados caso base

3.4.2. Imagenes altas

En este experimento comparamos todos los filtros para imágenes mas altas que anchas. Utilizando medidas especiales para cada caso, como explicamos anteriormente.

En este experimento introdujimos un nuevo índice denominado **penalización**. El mismo compara la implementación en *assembler*, para un *input* favorable y un desfavorable. Básicamente se calcula como la diferencia de ciclos de esto dos casos expresada en porcentaje, tomando como base la cantidad de ciclos del caso favorable.

En el cuadro 1 podemos observar los resultados para los filtros que toman imágenes en escala de grises. En ella podemos ver una gran mejora de *assembler* con respecto a **C** tanto en la imagen de 16x30000 como en la de 17x28234. Con esto concluimos que aún en un caso no favorable este sigue siendo más eficiente. Sin embargo, podemos ver tambien que en los casos no favorables, la implementación en *assembler* demora mucho mas que en los favorables.

Es notable también el monstruoso porcentaje de penalización (340 %, 165 % y 234 %, para invertir, normalizar y umbralizar respectivamente), confirmando nuestras conjeturas sobre los casos favorables y desfavorables. Puntualmente esperabamos una penalización de alrededor del doble, pero en la mayoría de casos fue más. Creemos que se debe, de alguna manera, a la complejidad

del filtro, es decir a la cantidad de ciclos que toma el procesamiento de una iteración. Por ejemplo, invertir que es por lejos la funcion mas simple de todas, el incremento es de casi tres veces y media, mientras que en normalizar, donde recorremos dos veces la imagen, tenemos un porcentaje mas bajo pero igualmente superior al doble.

El porcentaje de penalización para el filtro suavizar también nos resultó llamativo. Primero porque fue el único que dio exactamente lo que esperamos. pero también porque dicho porcentaje es bajo en comparación a los otros filtros. Lo cual nos hace inferir que tal vez podriamos implementarlo un poco mas eficientemente, o que tal vez se debe a la complejidad misma del algoritmo.

En este punto hemos realizado una revisión de la implementación del código de suavizar y vimos que tambien seria posible desarrollar una version que procese de a menos bytes, de esta manera disminuiriamos en un tercio los accesos a memoria[5]. Sería interesante ver si realmente procesar de menos píxeles pero con menos accesos a memoria presenta una mejora en la *performance*. Es una pena que hemos tenido el tiempo suficiente para realizar dicha comprobación. Sin embargo queda como un trabajo a futuro.

FUNCIÓN	# PIXELS	CICLOS C	CICLOS ASM	Δ CICLOS	% MEJORA	PENALIZACIÓN
invertir	16x30000	7.706.474	408.430	7.298.044	94,70 %	
	17x28234	7.642.414	1.799.789	5.842.625	76,45 %	340,66 %
normalizar	16x30000	24.321.128	1.821.235	22.499.894	92,51 %	
	17x28234	23.895.976	4.829.856	19.066.120	79,79 %	165,20 %
umbralizar	16x30000	14.287.793	680.832	13.606.961	95,23 %	
	17x28234	14.253.437	2.275.706	11.977.731	84,03 %	234,25 %
suavizar	17x28234	29.283.006	3.773.630	25.509.376	87,11 %	
	18x26666	29.320.184	7.404.941	21.915.243	74,74 %	96,23 %

Cuadro 2: Resultados Imágenes altas blanco y negro

Por otra parte el cuadro 3 muestra los resultados para los filtros que toman imágenes a color. Aqui también podemos apreciar la gran diferencia de ciclos en **C** y *assembler* con su porcentaje de mejora en todos los casos mayor que un 50 %. Sin embargo la penalización para los casos desfavorables no son tan grandes (entre 17 % y 32 %). Creemos que esto se debe a que no estamos en la misma situación de caso favorable/desfavorable de las imágenes en escala de grises. Para estarlo deberiamos haber procesado imaágenes de 5 y 6 píxeles respectivamente. De esta manera, la primera imagen necesitaría solo un ciclo para procesar la fila y y el caso desfavorable necesitaria dos. Aqui el caso favorable necesita dos iteraciones, mientras que el desfavorable necesita 3. Esta diferencia expresada en porcentaje es menor. Cabe aclarar que no elegimos las imagenes recién descriptas pues nuestro algoritmo procesa imagenes color de como mínimo 6 píxeles de ancho.

Como una observación general de estas mediciones de imágenes altas, en contraste con el caso base podemos apreciar cómo la distribución de píxeles no influencia significativamente en la cantidad de ciclos de reloj de las implementaciones en **C** mientras que en *assembler* este factor puede llegar tener mucha relevancia.

3.4.3. Imagenes anchas

En este experimento comparamos todos los filtros para imágenes mas anchas que altas, utilizando las medidas explicadas al inicio de esta sección.

La tabla 4 nos muestra los resultados de este experimento. Primero podemos observar que el porcentaje de mejora entre las implentaciones es casi tan alto como el de el caso base en todos los filtros. Sin embargo en este experimento las penalizaciones sufridas por el *assembler* al procesar

[5]esto es porque actualmente se leen 18 bytes por iteración - tres lecturas corridas de a un byte

función	# pixels	ciclos C	ciclos ASM	Δ ciclos	% mejora	penalización
mono_uno	10x48000	18.232.058	6.168.852	12.063.205	66,16 %	
	11x43636	18.059.469	8.144.680	9.914.789	54,90 %	32,03 %
mono_inf	10x48000	18.978.920	5.410.115	13.568.806	71,49 %	
	11x43636	18.967.671	6.368.449	12.599.222	66,42 %	17,71 %
separar_c	10x48000	20.874.906	7.176.690	13.698.216	65,62 %	
	11x43636	20.800.016	9.195.779	11.604.237	55,79 %	28,13 %

Cuadro 3: Resultados Imágenes altas color

casos desfavorables no son tan grandes como en el experimento anterior. Esto vuelve a confirmar nuestras conjeturas sobre los casos favorables/desfavorables.

Que la penalización sea menor que en el experimento anterior muestra que efectivamente a imágenes mas anchas el costo de tener que reprocesar algunos pocos píxeles se diluye. Podemos pensar al costo de reprocesamiento total como la cantidad de filas por el costo de una iteración. Con las imágenes altas, las matrices tenian muchas más filas con lo cual el costo era alto. Aqui, para la misma cantidad de píxeles totales, al tener menos filas, tenemos un costo menor.

También es interesante notar cómo en los filtros que procesan blanco y negro(excepto suavizar) la penalización todavia es considerable, mientras que en los otros la diferencia es casi imperceptible. Creemos que esto se debe a que en los filtros a color, como cada píxel son en realidad tres bytes, la penalización se diluye mucho mas, pues la información que se esta procesando es tres veces mas grande que en los casos blanco y negro. Asi que es de esperar la diferencia fuera como mínimo una tercera parte.

Sin embargo, no todo es color de rosa en la vida, y nos preguntamos porqué suavizar se parece mas a las funciones color que a las de su clase. Nuevamente creemos que tal vez se deba ya sea a la complejidad de la implentación que realizamos(que tiene muchos accesos a memoria) o a la complejidad propia del algoritmo. Tambien es cierto que suavizar no fue el filtro que peor mejora tuvo en el caso base con respecto a la implementación en **C**. Y aunque no podamos justificar totalmente el porqué, las evidencias muestran que este filtro es el mas estable de todos, pues a pesar de las variaciones que estamos realizando los resultados no cambian tanto como en los otros filtros y son siempre similares al caso base.

Otra observación general que notamos fue que aqui también la cantidad de ciclos consumidos por la implementación en **C** es similar al de los casos base, por el contrario la implementación en *assembler* en todos los casos los filtros consumen más ciclos que en su caso base (salvo normalizar que la cantidad de ciclos es comparable).

función	# pixels	ciclos C	ciclos ASM	Δ ciclos	% mejora	penalización
invertir	1440x333	7.277.945	313.567	6.964.378	95,69 %	
	1441x333	7.274.547	459.617	6.814.930	93,68 %	46,58 %
normalizar	1440x333	23.352.742	1.557.537	21.795.205	93,33 %	
	1441x333	23.326.450	1.793.687	21.532.763	92,31 %	15,16 %
umbralizar	1440x333	13.004.767	543.850	12.460.917	95,82 %	
	1441x333	13.013.609	715.497	12.298.113	94,50 %	31,56 %
suavizar	1442x333	32.468.271	3.905.847	28.562.424	87,97 %	
	1443x333	32.483.078	3.945.091	28.537.988	87,85 %	1,00 %
mono_uno	1440x333	17.292.896	5.499.913	11.792.983	68,20 %	
	1441x333	17.337.919	5.529.326	11.808.594	68,11 %	0,53 %
mono_inf	1440x333	18.526.267	4.450.636	14.075.631	75,98 %	
	1441x333	18.521.016	4.483.062	14.037.954	75,79 %	0,73 %
separar_c	1440x333	20.427.885	6.163.018	14.264.867	69,83 %	
	1441x333	20.457.241	6.214.148	14.243.092	69,62 %	0,83 %

Cuadro 4: Resultados Imágenes anchas

4. Conclusiones

- En todos los filtros y las imágenes que procesamos la tecnología *SSE* supera ampliamente la implementación en **C** en términos de *performance*, aún en los casos borde. Sin embargo también ese ahorro tiempo de ejecución se ve traducido en un esfuerzo mucho mayor en la etapa de desarrollo de los filtros. Mientras que cada algoritmo se resolvía en **C** en minutos, la implementación en *assembler* llevaba varias horas y tal vez dias. Es cierto que esto se debe que no estabamos acostumbrados a trabajar con dicha tecnología, sin embargo no solo el desarrollo fue lento sino tambien fue tedioso y a veces encontramos muchas dificultades a la hora de encontrar errores. Inmaginamos que en ciertos casos tal es la diferencia de *performance* es determinante y por ende es redituable el costo extra de la implementación,*debugging* y adaptación al lenguaje ensamblador.

- Nuestra conjetura inicial era que la implementación en C deberia ser alrededor de 16 veces mas lenta que la de *assembler*. Nunca inmaginamos que pudiera ser tan grande la diferencia, tal vez sobreestimamos lo que el compilador de C hace a la hora de compilar. Tal vez utilizando opciones de optimización nos hubiesemos acercado mas a esa realción teórica.

- La distribución de píxeles dentro de una imágen puede llegar a ser un factor a tener en cuenta para cuando se aplica la versión optimizada. Por el contrario, las implentaciones en **C** se comportaron de manera estable a lo largo de todos los experimentos.

- La manera de recorrer las imágenes en determinados tipos de filtro pudieron reutilizarse como es el caso de las funciones que van de color a escala de grises, análogamente lo mismo ocurre con las imágenes en blanco y negro.

- Siempre hay un menor porcentaje de penalización en los filtros que realizan más operaciones y accesos a memoria ya que no hay tanta diferencia entre reprocesar datos y las operaciones intermedias en cada iteración de un ciclo.

- Seguramente no estamos contemplando cierta informacion, pues si bien los resultados en general daban como esperabamos a veces las diferencias son mas grandes o mas pequeñas de lo que esperabamos o podemos justificar con los elementos que contamos. Creemos que un análisis mas profundo, en conjunto con otros experimentos que individualicen otros aspectos del procesador podrian echar mas luz sobre los misterios de la optimización de bajo nivel.
- El set de instrucciones de la verion 2 fue más que suficiente para realizar las tareas pedidas.
- Compartivamente, el desarrollo en *assembler* fue la actvidad que más tiempo abarcó, con respecto al desarrollo de este informe, mediciones y el desarrollo en **C**.
- Los casos de prueba seleccionados ayudaron a probar nuestras conjeturas iniciales y experimentalmente pudimos corroborar que la selección fue correcta.
- Hubieron experimentos que no pudimos realizar por limitaciones de tiempo pero que nos hubiesen gustado ver, entre ellos: alinear la memoria a 16 bytes para utilizar la instruccion de movimiento alineado y ver las diferencias de *performance*, realizar mediciones contra los filtros en **C** usando las opciones de optimización del compilador e implementar cada filtro en *assembler* sin utilizar tecnología *SSE*.

Referencias

[1] Intel©64 and IA-32 Architectures Optimization Reference Manual

[2] Intel©64 and IA-32 Architectures Software Developer's Manual,Volume 1: Basic Architecture

[3] Intel©64 and IA-32 Architectures Software Developer's Manual,Volume 2A: Instruction Set Reference, A-M

[4] Intel©64 and IA-32 Architectures Software Developer's Manual,Volume 2B: Instruction Set Reference, N-Z

[5] Enunciado del tp

[6] http://orga2.com.ar, diapositivas de las clases