DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Filtros de imágenes

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Bogetti Gianfranco	693/15	gianbogetti7@hotmail.com
Feliu Santiago	644/15	santiagofeliu96@gmail.com
Ingaramo Pablo	544/15	pablo2martin@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Objetivos generales

En este trabajo práctico utilizaremos las instrucciones de $SIMD$ para poder trabajar con paralelización de datos. También haremos la misma versión de cada filtro una en ASM y la otra en C para luego poder compararlas. Además intentaremos comprobar algunos hechos teóricos sobre la microarquitectura.

2. Desarrollo

2.1. Blit

El propósito de este algoritmo es superponer la esquina superior del lado derecho de una imagen con una imagen de Perón. Dado que la imagen de Perón es de 89 píxeles de ancho y 128 de alto nos resulto intuitivo separar el algoritmos en diferentes segmentos.
El primer segmento (lineas 29-45 de blit_asm.asm) se ocupa de calcular el resto de dividir el ancho de la imagen de Perón por 4.
El segundo segmento, que empieza en la linea 47 (blit_asm.asm), es donde empieza el ciclo principal denotado por la etiqueta çiclo1".
En cada iteración de este ciclo se que voy a estar en el comienzo de una fila de la imagen que estoy recorriendo, por lo cual copio los primeros 4 píxeles de la imagen a la imagen destino ya que en este momento la imagen de Perón y la imagen no se superponen.
Luego le sumo al puntero que utilizo para iterar sobre la imagen el número calculado en el primer segmento. Esto se hace por que en cada iteración voy a procesar de a 4 píxeles y al sumar este número me adelanto la cantidad de píxeles necesarios para que el número de píxeles que tengo que recorrer hasta llegar a la imagen de Perón sea múltiplo de 4.
El tercer segmento empieza con la etiqueta çiclo2". Al comienzo de este ciclo copio a xmm1 los cuatros píxeles que me apunta el puntero a en RDI y luego me fijo si estoy parado en la columna justo antes de donde comienza la imagen de Perón. En caso de que esto se cumple me fijo si estoy en la fila donde comienza la imagen de Perón.
Si cualquiera de estos dos casos no se cumplen, salto al segmento del código que comienza con la etiqueta 'continuar'. En caso contrario continuo a el segmento que empieza con la etiqueta 'blit'.
El cuarto segmento,'blit', se encarga de decidir si los 4 píxeles que tengo en el registro xmm0 son magenta o no.
Para hacer esto cargo una constante al registro xmm3 de tal manera que tenga 4 píxeles magentas. Luego hago una mascara con la instrucción 'pcmpeqd xmm3, xmm0;' la cual me deja en el registro xmm3 4 doble palabras que son 0xFFFFFFFF si xmm0 era magenta y 0x00000000 de lo contrario.
Luego uso la instrucción blendvps que une los píxeles de blit y los de lena usando la mascara generado con pcmpeq la cual dice si los píxeles de blit eran magenta o no. El resultado de la instrucción blend lo copio xmm1.
En el quinto segmento, que comienza con la etiqueta 'continuar', copio a la imagen destino el contenido de xmm1 y avanzo los puntero que uso para recorrer la imagen destino y el source. Luego me fijo en que posición estoy de la fila de píxeles que estoy recorriendo, osea que cantidad de columnas a recorrí. En caso de no estar al final salto al segmento con la etiqueta 'ciclo2'. De lo contrario me fijo si estoy en la ultima fila. En caso de no estar en la ultima fila salto al segmento 'ciclo1'. De lo contrario sigo al ultimo segmento que empieza con la etiqueta 'fin'.
Al sexto segmento, 'fin', llego únicamente después de haber recorrido toda la matriz. En este momento deshago el $stack - frame$ y retorno a la función llamadora.

2.2. Monocromatizar norma infinito

Para convertir la imagen en color a una imagen en escala de grises utilizaremos la función:

$$I_{out_infinito}(p) = max(R, G, B)(\epsilon = \infty)$$

Por lo tanto lo que se debe hacer es encontrar el máximo valor entre los colores del píxel y colocar ese valor para los tres colores.

Para el algoritmo que se utiliza tendremos algunas máscaras guardadas en registros, estas serán las siguientes representadas como cuatro DW:

- XMM0 se utilizará para tomar el último byte de los DW en los registros XMM. Se verá de la siguiente manera:

XMM0:	0xFF	0xFF	0xFF	0xFF

- XMM7 se encontrará una máscara de negativos, la cual se usara para negar otros registros, se vera de la siguiente manera:

XMM7:	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF	0xFFFFFFFF

Lo primero que se hará dentro del ciclo sera levantar cuatro píxeles de memoria, ya que será el máximo a traer en un llamado a memoria, dentro del registro los píxeles se verán representados como sigue

Alpha	Red	Green	Blue	X4

Luego se separaran los cuatro colores en distintos XMM utilizando operaciones de shift y de blend, como en el siguiente ejemplo:

0x0	0x0	0x0	Blue	X4

Resultado de **pblendvb**

0x0	Alpha	Red	Green	X4

Resultado de **psrld**

0x0	0x0	0x0	Green	X4

Resultado de **pblendvb**

```
; xmm1 = |A|R|G|B|A|R|G|B|A|R|G|B|A|R|G|B|
pblendvb xmm2, xmm1 ; xmm2 = |0|0|0|B|0|0|0|B|0|0|0|B|0|0|0|B|

psrld xmm1, 8         ; xmm1 = |0|A|R|G|0|A|R|G|0|A|R|G|0|A|R|G|
pblendvb xmm3, xmm1 ; xmm3 = |0|0|0|B|0|0|0|B|0|0|0|B|0|0|0|B|
```

Una vez teniendo todos los elementos del píxel por separado debemos encontrar el máximo de ellos, para ello se utilizara la instrucción pcmpgtd que se le dará dos registros XMM y este pondrá en el registro destino 1 donde encuentre que el DW de esa posición es mayor y 0 en el caso contrario. Se podrá ver de mejor en el ejemplo, donde se trabajara con XMM con cuatro DW:

Green	Green	Green	Green
Red	Red	Red	Red

Registros a utilizar en **pcmgtd**

0x0	0xFF	0x0	0xFF

Resultado

Esta máscara se utilizara la utilizaremos para tomar los máximos de cada registro utilizando instrucciones como pand, por, pxor:

0x0	Green	0x0	Green

Resultado de **pand**

0xFF	0x0	0xFF	0x0

Resultado **pxor** con XMM7

Red	Green	Red	Green

Resultado de **por** con los registros ya filtrados (max(R,G))

Una vez obtenidos todos los máximos de cada píxel se copiarán en las tres posiciones de R, G y B, se traerá Alpha del registro en el que quedo guardado y se guardara en la posición de la memoria.

2.3. Efecto Ondas

Este filtro requería que trabajemos principalmente con números de punto flotante para calcular para cada píxel la función prof.
En cada iteración del ciclo principal que comienza con la etiqueta 'ciclo', procesamos de a 4 píxeles.
Comenzamos poniendo, con la instrucción insert, en el registro xmm1 la posición horizontal de cada uno de los 4 píxeles en xmm0 y en xmm2 la posición vertical de cada uno de los mismos píxeles de la siguiente forma:

XMM0:	$pixel_{x,y}$	$pixel_{x+1,y}$	$pixel_{x+2,y}$	$pixel_{x+3,y}$

Luego en xmm1 y xmm2 tengo lo siguiente:

XMM1:	x	x+1	x+2	x+3

XMM2:	y	y	y	y

El paso siguiente es aplicar en paralelo a cada uno de los 4 valores en xmm1 y xmm2 el algoritmo provisto por la cátedra, siempre teniendo cuidado de usar instrucciones de enteros cuando trabajamos con enteros e instrucciones de punto flotante en otros casos.
Por ejemplo para el calculo de d_x, d_y usamos instrucciones para enteros pero de ese momento en adelante convertimos los valores d_x y d_y a single-percision floats con la instrucción cvtdq2ps para poder calcular $d_{x,y}$.
Después de haber hecho todos los cálculos de la función terminamos con los siguientes valores en xmm4:

XMM4:	$prof_{x,y} * 64$	$prof_{x+1,y} * 64$	$prof_{x+2,y} * 64$	$prof_{x+3,y} * 64$

El siguiente paso es sumarle al componente R, G y B del $pixel_{x,y}$ el valor $prof_{x,y} * 64$.
El problema que surge es que $prof_{x,y} * 64$ es un single-precisión float y los componentes de $pixel_{x,y}$ son unsigned ints de un byte.
Para solucionar este problema lo primero que hacemos es convertir $prof_{x,y} * 64$ a un integer de 4 bytes con signo usando la instrucción cvttps2dq.
Luego tenemos que convertir a cada componente R, G y B de cada uno de los 4 pixeles que estamos procesando a integer de 4 bytes con signo.
Esto lo hacemos desempaquetando de byte a word y después de word a double-word pero sin preservar los signos ya que R, G y B son sin signo. Luego vamos a tener un registro xmm para los contenidos de cada píxel.
Pero a cada uno de estos no le puedo sumar el contenido de xmm4 ya que para un dado píxel tengo un único valor de prof que sumarle.
Para poder hacer las sumas correctas uso la instrucción shufd xmm3, xmm4 de manera que me queden xmm3 de la siguiente manera:

XMM3:	0X00000000	$prof_{x,y} * 64$	$prof_{x,y} * 64$	$prof_{x,y} * 64$

Y después de desempaquetar los contenidos de xmm0, tengo en xmm0, xmm6, xmm2 y xmm7 lo siguiente:

XMM0:	$Aplha_{x,y}$	$R_{x,y}$	$G_{x,y}$	$B_{x,y}$

XMM6:	$Aplha_{x+1,y}$	$R_{x+1,y}$	$G_{x+1,y}$	$B_{x+1,y}$

XMM2:	$Aplha_{x+2,y}$	$R_{x+2,y}$	$G_{x+2,y}$	$B_{x+2,y}$

XMM7:	$Aplha_{x+3,y}$	$R_{x+3,y}$	$G_{x+3,y}$	$B_{x+3,y}$

Luego queda sumar xmm0 con xmm3 con la instrucción paddusd ya que quiero sumar saturando sin signo.
Después para poder sumarle $prof_{x+1,y} * 64$ a cada elemento de xmm6 tengo que volver a usar la instrucción shufd xmm3, xmm4; para que xmm3 quede de la siguiente manera:

XMM3:	0X00000000	$prof_{x+1,y} * 64$	$prof_{x+1,y} * 64$	$prof_{x+1,y} * 64$

Esto lo repito para hacer las sumas de xmm7 con xmm3 y xmm2 con xmm3.
Por último queda empaquetar de double word a word el contenido de xmm2 con xmm7 y xmm0 con xmm6, y de word a byte el contenido de xmm0 con xmm2.
El resultado de esta cuenta esta en xmm0 por lo cual el último paso de cada ciclo es mover el contenido de xmm0 a la matriz destino y después incrementar los punteros que uso para recorrer ambas imágenes por 16 ya que proceso de a 16 bytes por iteración.

2.4. Edge

Para este filtro deberemos definir que es un borde, para ello tenemos el operador de Laplace, cuya matriz es la siguiente:

$$M = \begin{pmatrix} 0,5 & 1 & 0,5 \\ 1 & -6 & 1 \\ 0,5 & 1 & 0,5 \end{pmatrix}$$

Esta matriz la utilizaremos posicionándonos en el centro de ella y se realizara esta función:

$$dst(x,y) = \sum_{k=0}^{2}\sum_{l=0}^{2} src(x+k-1, y+l-1) * M(k,l)$$

Aunque existirán lugares donde no se puede utilizar dicha función, estos serán los bordes, por lo tanto se dejara el píxel de la misma manera en la que se encontró.

Por lo tanto, al saber que los bordes se dejan de la misma forma, y ademas sabiendo como se recorre la imagen, se decidió no utilizar el ciclo ni para la primer fila ni para la última.

Por otra parte, sabemos que la imagen es múltiplo de 4 y con un ancho mayor a 16 bytes, y que los píxeles medirá 1 byte, lo cual decidimos que lo óptimo será trabajar con 12 píxeles por ciclo, ya que es múltiplo de 4 y con 16 píxeles no se podría trabajar sin hacer otra llamada a memoria para pedir los vecinos de los píxeles bordes.

Al no recorrer la primer y la última fila nos quedaran los bordes de la primer y última columna, entonces tendremos 3 casos, cuando se levanta la primer columna, el caso general del medio, y cuando se levante la última.

A su vez la imagen no es múltiplo de 12, por lo tanto esto nos lleva a tener que pensar que existirán 3 casos diferentes, que sea módulo de 12, que sea módulo 4 o que sea módulo 8, por lo tanto este desfazaje lo arreglaremos en la primer levantada, solo colocando los cantidad de píxeles que den como resto al dividir la cantidad total de píxeles por 12.

Entonces el ciclo comenzará levantando píxeles de memoria , los píxeles que se modificarán, los de arriba y los de abajo

Ar	Ar	Ar	Ar	Ar	Ar	Ar	Ar	Ar	Ar	Ar	Ar	Ar	Ar	Ar	Ar
P_{15}	P_{14}	P_{13}	P_{12}	P_{11}	P_{10}	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0
Ab	Ab	Ab	Ab	Ab	Ab	Ab	Ab	Ab	Ab	Ab	Ab	Ab	Ab	Ab	Ab

Píxeles a levantar

Cómo siempre se trabajara en orden, exceptuando el momento de hacer la sumatoria, todo lo que se le haga a un registro XMM se le hará a los otros 2. Dicho esto, tendremos un caso diferente, el cual será el caso que levante el comienzo de la fila, que se hará un shift a la izquierda. De esta manera tendremos que modificar para un caso los píxeles [2:13], mientras que en el otro se modificarán los píxeles [0:11].

P_{13}	P_{12}	P_{11}	P_{10}	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	0	0

Luego de shift izquierdo

Lo siguiente a hacer será aplicarles la función, pero no podremos trabajar con los datos así como están, ya que cada píxel estaría ocupando 1 byte y suponiendo el caso en el cual el píxel tenga un valor de 255(valor máximo en byte), habrá un problema al multiplicarlo por -6 ya que el resultado no entrará en 1 byte, por ello la sumatoria se hará en dos partes y desempaquetando los bytes a words.

```
; xmm0 máscara con 0 | xmm2 píxeles | xmm3 píxeles
punpcklbw xmm2, xmm0
; xmm2 = | P8 | P7 | P6 | P5 | P4 | P3 | P2 | P1 | "ó"
; xmm2 = | P6 | P5 | P4 | P3 | P2 | P1 | P0 |  0 |

punpckhbw xmm3, xmm0
; xmm3 = | P14 | P13 | P12 | P11 | P10 | P9 | P8 | P7 | "ó"
; xmm3 = | P12 | P11 | P10 | P9  | P8  | P7 | P6 | P5 |
```

Una vez desempaquetado se llamará a la función auxiliar sumatoria, en donde se hacen diferentes shift y sumas de tal manera que devuelve los datos ya procesados.
Luego se empaquetará juntos los datos, con **packuswb** que saturará y convertirá los words signados a bytes sin signo de la siguiente manera:

```
; xmm8 resultado de xmm2 | xmm2 resultado de xmm3
packuswb xmm8, xmm2
; xmm8 = | 0 |P14|P13|P12|P11|P10|P9 |P8 |P7 |P6 |P5 |P4 |P3 |P2 |P1 | 0 | "ó"
; xmm8 = | 0 |P12|P11|P10|P9 |P8 |P7 |P6 |P5 |P4 |P3 |P2 |P1 |P0 | 0 | 0 |
```

Aplicando a este registro shifts lógicos le quito los píxeles basura y me quedará de la siguiente manera

0	0	P_{13}	P_{12}	P_{11}	P_{10}	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	0	0
0	0	P_{11}	P_{10}	P_9	P_8	P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	0	0

Luego de shift izquierdo

Finalmente lo último que se debe hacer es restaurar los píxeles que no fueron modificados, ponerlos en la posición de memoria que se deben colocar y avanzar el puntero los bytes necesarios.

Para lograr esto tendremos una máscara de 1 que se utilizará para tomar los píxeles originales y luego se juntarán en un mismo registro, existirán 5 casos diferentes que dependerán de en que posición me encuentre en la columna y del resto que quedaba al dividir el largo de la columna por 12. A continuación un ejemplo con el caso que sea resto 8 y se quiera poner la primer columna:

```
; r10 cantidad de píxeles a avanzar | r8 cantidad de píxeles que se procesaron en el ciclo | xmm14 máscara de 1

mov r10, 6 ; Voy a querer avanzar sólo 6
mov r8, 8

psrldq xmm8, 2 ; xmm8 = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |P7 |P6 |P5 |P4 |P3 |P2 |P1 |P0 |
pslldq xmm14, 9  ; xmm14= | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
psrldq xmm14, 8  ; xmm14= | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
pand xmm8, xmm14 ; xmm8 = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |P7 |P6 |P5 |P4 |P3 |P2 |P1 | 0 |
pxor xmm14, xmm15 ; xmm14= | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
pand xmm14, xmm10 ; xmm14= |        originales x8         | 0 | 0 | 0 | 0 | 0 | 0 | 0 |ox1|
por xmm8, xmm14 ; xmm8 = |         originales x8         |       modificados x7      |ox1|
; El borde se deja sin modificar
```

2.5. Temperature

En este filtro tendremos que modificar el valor de cada color dependiendo el resultado de hacer el promedio de los tres colores, el cual será llamado t.

$$\mathsf{t}_{(i,j)} = \lfloor (\mathsf{src}.r_{(i,j)} + \mathsf{src}.g_{(i,j)} + \mathsf{src}.b_{(i,j)})/3 \rfloor$$

$$\mathsf{dst}_{(i,j)} < r, g, b >= \begin{cases} < 0, 0, 128 + t \cdot 4 > & \text{si t} < 32 \\ < 0, (t-32) \cdot 4, 255 > & \text{si } 32 \leq \text{t} < 96 \\ < (t-96) \cdot 4, 255, 255 - (t-96) \cdot 4 > & \text{si } 96 \leq \text{t} < 160 \\ < 255, 255 - (t-160) \cdot 4, 0 > & \text{si } 160 \leq \text{t} < 224 \\ < 255 - (t-224) \cdot 4, 0, 0 > & \text{si no} \end{cases}$$

Para lograr esto comenzamos de la misma manera que con $Monocromatizar$ infinito, levantamos de memoria cuatro píxeles y le separamos los colores y $alpha$ en distintos registros. Luego de esto se suman los tres colores y se obtiene el valor t que se usara para decidir que píxel se modifica y cual no.
Tendremos cinco intervalos de t, donde para cada uno se debe hacer distintas operaciones, haremos siempre todas las operaciones y solo tomaremos los datos en caso de que el t de ese píxel se encuentre dentro del intervalo.
Para lograr los intervalos de los t tendremos una máscara que se conformara de cuatro double words con el valor 32 y otra con el valor 1.

32	32	32	32

Máscara 32

1	1	1	1

Máscara 1

A partir de esta obtendremos las demás a partir de unas operaciones aritméticas, a continuación un ejemplo quitado del código:

```
; Obtención máscara para los t menores a 160.
; xmm15 = máscara 32 | xmm14 = máscara 1
movdqu xmm2, xmm15 ; xmm2 = |  32 |  32 |  32 |  32 |
pslld xmm2, 2 ; xmm2 = | 128 | 128 | 128 | 128 |
paddd xmm2, xmm15 ; xmm2 = | 160 | 160 | 160 | 160 |
psubd xmm2, xmm14 ; xmm2 = | 159 | 159 | 159 | 159 |
```

Utilizaremos estas máscaras con la instrucción **pcmgtd** que nos dirá los t mayores a la máscara, luego la negamos para obtener los menores y quitaré los t que ya fueron utilizados con la función **pxor**, para lograr esto tendremos en el registro xmm5 los t ya utilizados y en xmm11 una mascara repleta de unos. Sigamos con el ejemplo:

```
; xmm11 = máscara para negar
pcmpgtd xmm8, xmm2 ; xmm3 = |  T>159 |  T>159 |  T>159 |  T>159 | "Máscara"
pxor xmm8, xmm11 ; xmm3 = | 0<T<160| 0<T<160| 0<T<160| 0<T<160| "Máscara"
pxor xmm8, xmm5 ; xmm3 = |96<=T<160|96<=T<160|96<=T<160|96<=T<160| "Máscara"
```

25	150	200	234

Bytes en xmm8

XMM8:	0x0	0x0	0xFFFFFFFF	0xFFFFFFFF

Resultado de obtener los mayores a 159

XMM8:	0xFFFFFFFF	0xFFFFFFFF	0x0	0x0

Resultado de negar xmm8

XMM8:	0x0	0xFFFFFFFF	0x0	0x0

Resultado de quitar los t ya obtenidos

Luego para cada píxel le haré las operaciones del intervalo en el que se encuentra para luego tomar solo los píxeles que tengan el valor t en ese intervalo, terminando así el ejemplo del intervalo $96 \leq t < 160$.

```
; xmm10 = Valor t del píxel

movdqu xmm3, xmm10 ; xmm3 = T
psubd xmm3, xmm2 ; xmm3 = |  T-96  |  T-96  |  T-96  |  T-96  |
pslld xmm3, 2 ; xmm3 = |(T-96)*4|(T-96)*4|(T-96)*4|(T-96)*4| [0|0|0|(T-96)*4](PIXEL)

movdqu xmm2, xmm3 ; xmm2 = xmm3 = (T-960)*4
pslld xmm2, 8 ; [0|0|(T-96)*4|0](PIXEL)
por xmm2, xmm12 ; [0|0|(T-96)*4|255](PIXEL)
pslld xmm2, 8 ; [0|(T-96)*4|255|0](PIXEL)
por xmm2, xmm12 ; [0|(T-96)*4|255|255](PIXEL)
psubd xmm2, xmm3 ; [0|(T-96)*4|255|255-(T-96)*4](PIXEL)
```

0	216	255	39

Píxel 2 dentro de XMM2 luego de hacer operaciones

```
por xmm5, xmm8 ; Máscara temporaria con los t ya tomados
pand xmm8, xmm2 ; Resultado intervalo 3
```

XMM8:	0	Píxel 2	0	0

XMM5:	0xFFFFFFFF	0xFFFFFFFF	0	0

Una vez pasado por todos los intervalos tendremos que juntar todos los resultados, que será sólo aplicar **por** a los registros XMM4, XMM6, XMM7, XMM8, XMM9, los registros donde fui depositando los resultados, como los intervalos no se pisan entre ellos podremos deducir que los píxeles estarán en un solo registro.

3. Experimentaciones

3.1. Idea de la experimentación

En la experimentación queremos ver si realmente hay una mejora entre un set de algoritmos, en este caso filtros de imágenes, escritos en C y compilados con GCC y los mismos algoritmos escritos en $assembly$ utilizando $SIMD$ para trabajar en paralelo.
También queremos ver cuan costosos son los branch penalties.
Y por último vamos a ver si realmente sirve optimizar el algoritmo o si el tiempo se va en las lecturas de memoria.

Los experimentos fueron hechos con un procesador Intel® Core™ i5-2450M Processor, que posee una $micro - arquitectura\ Sandy\ Bridge$, $3MB\ SmartCache$, 2 $cores$, 4 $threads$ y una $frecuencia$ de 2.5GHz.

3.2. Experimento comparaciones

En el primer experimento compararemos los tiempos obtenidos para los algoritmos de C optimizado con $O0$, C optimizado con $O3$ y $assembly$ con $SIMD$.
En este experimento se espera observar que el algoritmo en $assembly$ con $SIMD$ es superior a los otros 2 seguido por C optimizado con $O3$ y por último C optimizado con $O0$.
Para ello corrimos los filtros con cada uno de los algoritmos a medir 25 veces por cada imagen tomando el promedio de los tiempos.
Los tamaños utilizados fueron 128x128, 140x140, 160x160, 180x180, 200x200, 208x208, 220x220, 256x256, 300x300, 360x360, 400x400, 420x420, 480x480, 512x512, 640x640, 720x720, 800x800, 1024x1024, 2048x2048, 4096x4096.

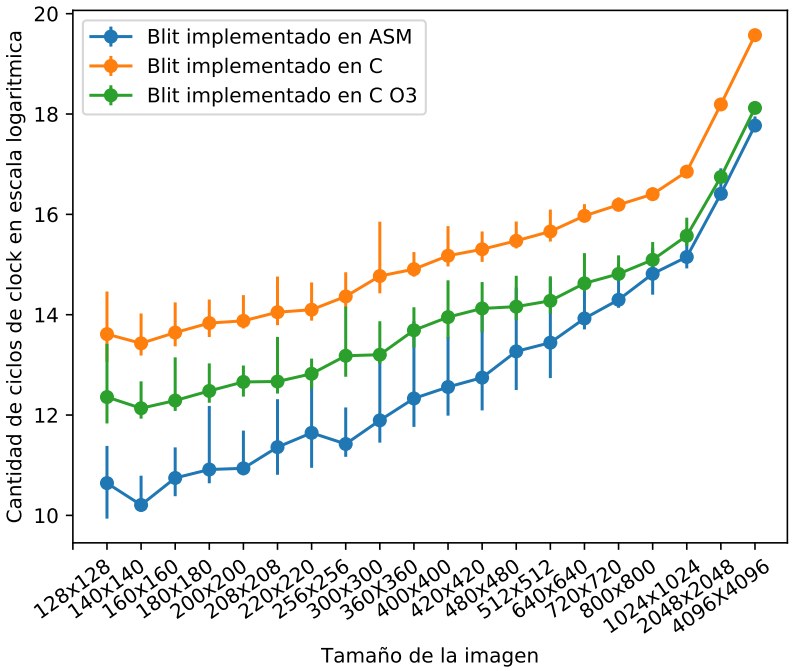


Figura 1: Comparativa para el filtro $Blit$.

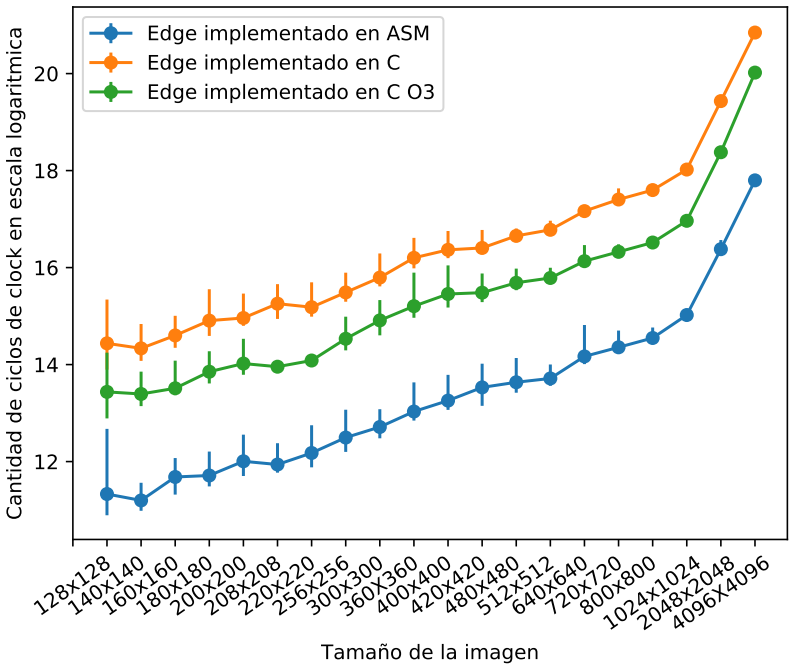


Figura 2: Comparativa para el filtro $Edge$.

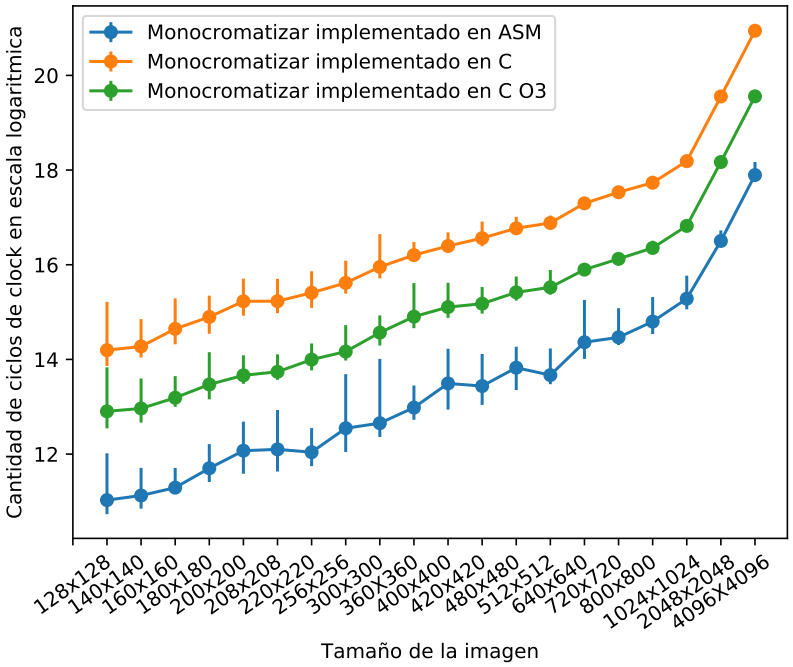


Figura 3: Comparativa para el filtro $Monocromatizar$.

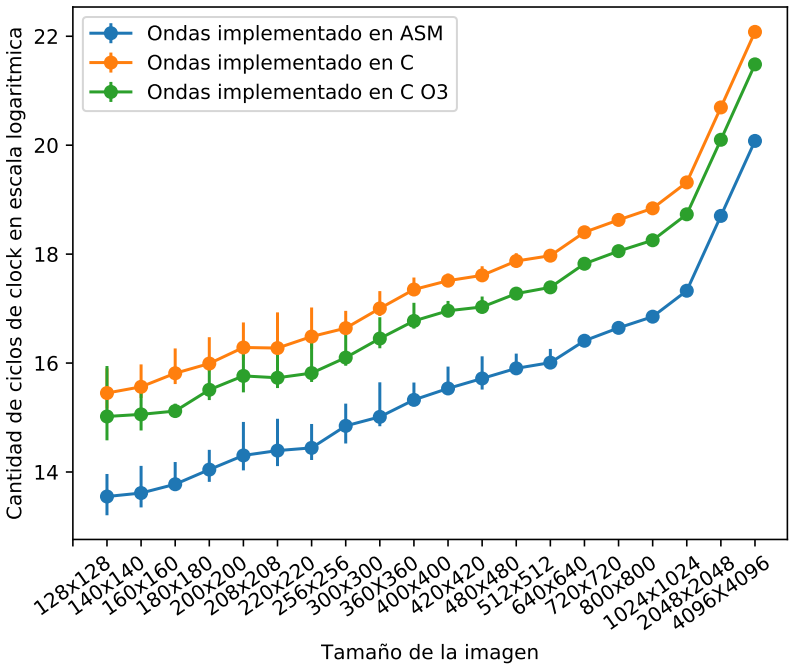


Figura 4: Comparativa para el filtro $Ondas$.

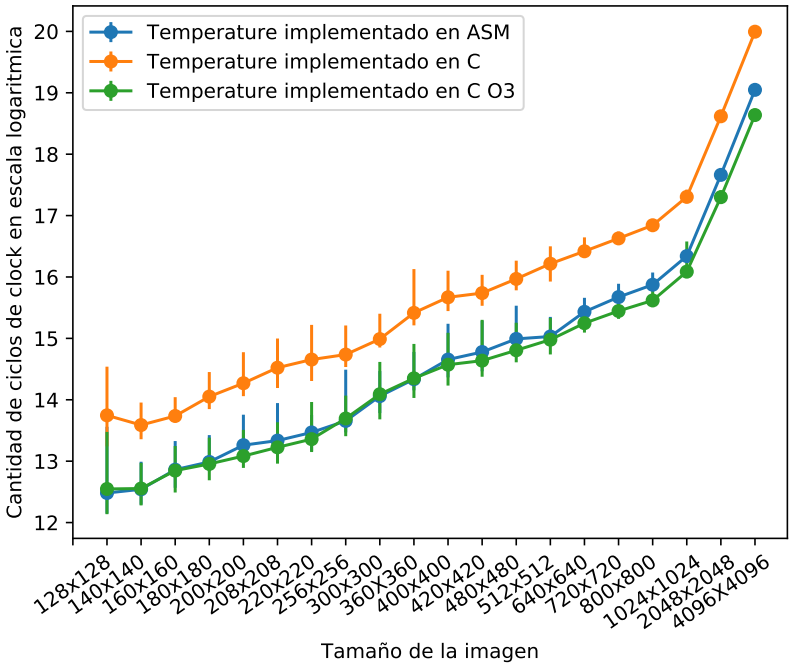


Figura 5: Comparativa para el filtro $Temperature$.

A partir de las Figura 1, 2, 3, 4 y 5 podemos concluir que como se esperaba los tiempos conseguidos a través del algoritmo de C optimizado con $O0$ es el que mas tarda, seguido siempre por el algoritmo de C optimizado con $O3$ y por último el que menos tarda es el algoritmo en $assembly$ con $SIMD$, aunque varia según el caso hasta haber uno en que estos últimos 2 tardan lo mismo. Creemos que el filtro temperatura, se podría optimizar haciendo comparaciones de t con cada intervalo, para así no tener que hacer las operaciones para los 5 casos en todos los ciclos, de esta manera quizás podría haber tardado menos tiempo que $O3$.

3.3. Experimento loop unrolling

Este experimento surge de la idea de los $branch\ penalty$, este es un grave problema porque un $branch$ es una perdida del flujo de ejecución, lo que provoca que el $pipeline$ se vacié y deba transcurrir n-1 ciclos de clock para el próximo resultado (siendo n la cantidad de etapas del $pipeline$). Entonces querremos deshacernos de estos $branch\ penaltys$, que lo haremos quitando los $Jumps\ condicionales$, estos se encuentran mayoritariamente al final de los ciclos.
Por lo tanto hicimos una serie de experimentos, basándonos en el filtro $monocromatizar$, en el cual convertimos el cuerpo del ciclo en un macro y lo pondremos seguido, esto causará sacar tantos $Jumps$ como macros seguidos pongamos.
Creemos que posiblemente baje el tiempo a medida que pongamos mas código seguido, pero que esto no será una mejora significativa ya que el algoritmo del $branch\ predictor$ posiblemente logre predecir la condición.
Igualmente al quitar las instrucciones **cmp** y **jne** ahorraremos ciclos.
Creamos 12 casos distintos en donde en cada caso se calculan n*4 píxeles por ciclo, siendo n el parámetro observado en el eje X. Mientras que la imagen que se correrá será siempre lena de 256x256 píxeles, se mantiene la misma imagen para no generar ninguna clase de ruido entre las distintas mediciones.
Cabe remarcar que no tendría sentido hacer este experimento con imágenes que tengan menos píxeles de los que se pueden calcular en un ciclo, ya que nunca se completaría el mismo.

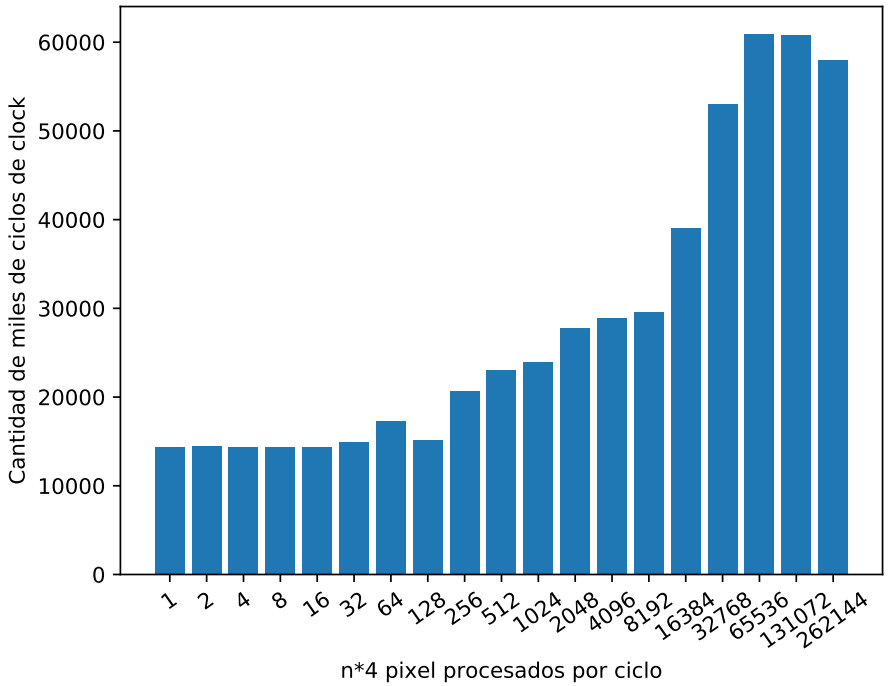


Figura 6: Tiempos obtenidos al hacer diferentes cantidades de $loop\ Unroll$.

A partir de la Figura 6 podemos concluir que la cantidad de ciclos disminuye a partir del quitado de

$jumps$ condicionales, pero nos encontramos con otro problema, un crecimiento abrupto a partir de el caso 256 en el cual de mucho analizar creemos que es causado por el agotamiento de espacio del cache, lo que genera que para el momento en el que hace el $jump$ al primer macro este ya no se encuentra en la memoria cache y debe buscarlo a memoria, lo que es realmente costoso.

Para poder confirmar esta hipótesis agregamos la instrucción NOP 1000 veces en el ciclo al que ”desenrollamos”. La razón por la cual agregamos la instrucción NOP es que esta toma una cantidad despreciable de ciclos de clock para ejecutarse y ocupa un espacio en memoria que será para llenar la cache.

Por lo tanto esperaremos que el gráfico crezca en casos mas pequeños.

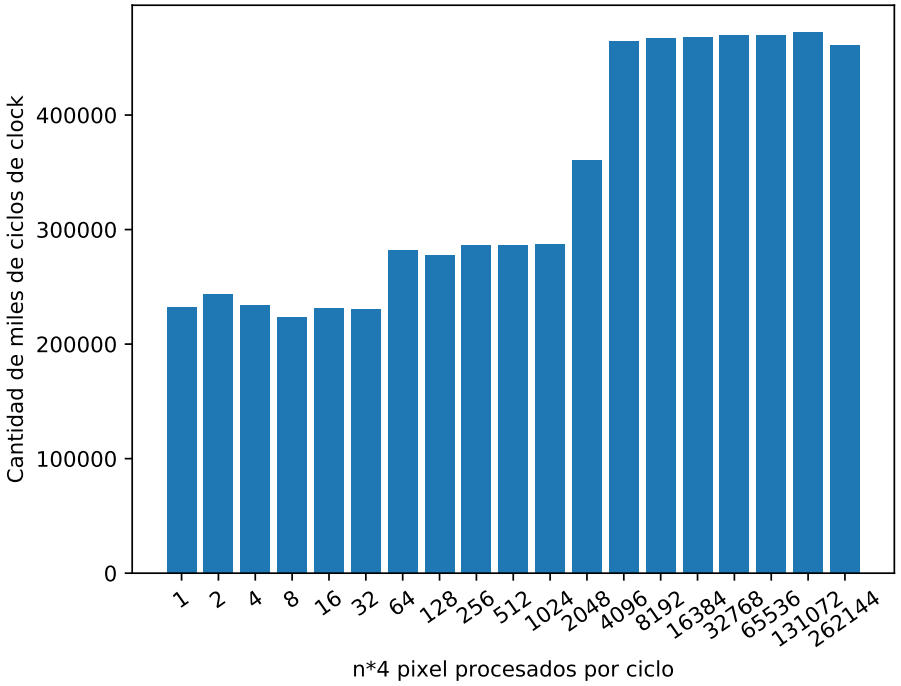


Figura 7: Tiempos obtenidos al hacer diferentes cantidades de $loop\ Unroll$ con NOP

En la figura 7 se puede ver que efectivamente al agregar la instrucción NOP 1000 veces por ciclo, causa que se llene el cache mas rápido que antes. Ademas se puede ver que existe un punto a partir del cual el gráfico se aplana por mas instrucciones seguidas que ejecutemos.

Ademas queríamos ver si hay una tendencia para los diferentes tamaños de imágenes. Para poder ver esto corrimos el experimento anterior pero para las imágenes con los siguientes tamaños: 512x512, 1024X1024, 2048x2048, 4096x4096, 8192x8192.

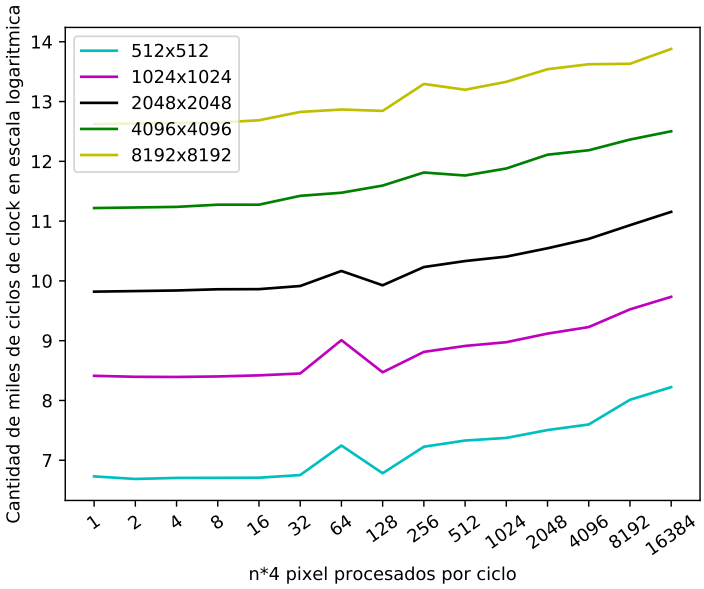
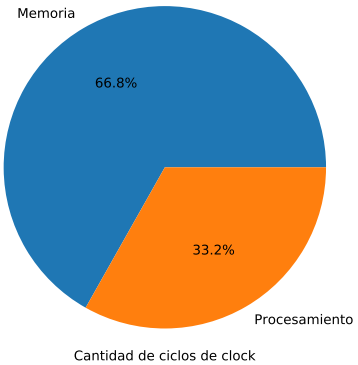


Figura 8: Tiempos obtenidos al hacer loop unroll con distintos tamaños de imágenes

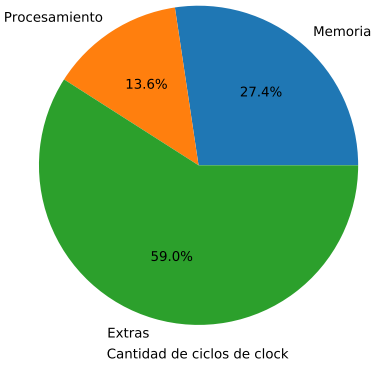
Finalmente encontramos que en la figura 8 se observa que la tendencia es la misma para los distintos tamaños, la cual es una pendiente hacia arriba.

3.4. Experimento ciclos totales

Este pequeño experimento es para comprobar que cuesta más cantidad de ciclos de clock si los llamados a memoria, el procesamiento de los datos u otras instrucciones que hace C.
Al saber los resultados podríamos ver si hay posibilidades de mejorar el algoritmo para procesar los datos y si esto puede ser un cambio significante.
Para ello modificamos el filtro $monocromatizar$ para que utilice la instrucción **rdtscp**, esta instrucción escribe el $timestamp\ counter$ del procesador en los registros $edx - eax$ y además espera a que todas las instrucciones previas sean ejecutadas, lo que es de mucha ayuda para contemplar la cantidad de clocks que se toman diferentes instrucciones. Mientras que el tiempo total lo obtenemos de lo que nos dice el ejecutable $tp2$.



(a) Porcentajes obtenidos viendo solo memoria y procesamiento.



(b) Porcentajes obtenidos con la totalidad.

En la figura 9a Encontramos que el procesamiento de la imagen utiliza solo un tercio de los clocks necesarios para obtener el filtro. Pero esto es el resultado solo comparando lo obtenido en $assembly$. En la figura 9b vemos que la mayor cantidad de ciclos de clock se realizan fuera del script de $assembly$

(extras), que contiene la configuración a la llamada del script, el seteo de los parámetros de entrada de las funciones, entre otros, y esta diferencia es tan grande que el procesamiento de datos pasa a ser de solo el $13,6\,\%$, que si uno quisiera mejorar el algoritmo parecería que no influiría demasiado en el total.

4. Conclusiones

Al finalizar este trabajo práctico podemos concluir las siguientes cosas:

- Del primer experimento podemos concluir que utilizando instrucciones de SIMD se consigue mejor rendimiento que no utilizándolas por mas de que se compile optimizadamente con O3.
- Del segundo experimento encontramos evidencia que nos lleva a concluir que al "$desenrollar$" ciclos reducimos la cantidad de instrucciones lo que concluye en una mejora de los tiempos de ejecución, pero el abuso de esto nos lleva a llenar la cache y obtener unos tiempos de ejecución mayores al original.
- Como la mayor parte del tiempo de ejecución del los filtros son los llamados a memoria, una opción posible para optimizar un algoritmo es reducir lo mas posible la cantidad de lecturas de memoria, esta podría ser una de las razones por la cual $SIMD$ utiliza menos ciclos de clock. Esto no quita que haya que optimizar la cantidad de instrucciones restantes en los casos que no haya tantos llamados a memoria.