DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Organización del Computador II

Primer Cuatrimestre de 2019

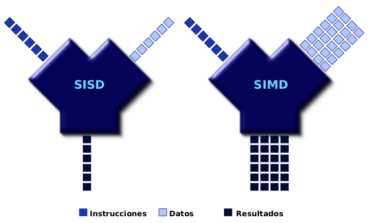
Integrante	LU	Correo electrónico
Joaquin P. Centeno	699/16	joaquinpcenteno@gmail.com
Joaquin Romera	183/16	joakromera@gmail.com
Werner Busch	380/09	w.g.busch@hotmail.com

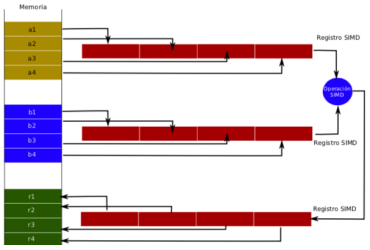
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En el presente trabajo hicimos un primer acercamiento al modelo de ejecución SIMD (Single instruction Multiple Data). Diseñado como una extensión al set de instrucciones de la arquitectura x86 e introducido por Intel en 1999, esta técnica de paralelización puede mejorar notablemente la performance de un sistema en contextos de programación donde se deben aplicar algoritmos repetitivos sobre un mismo conjunto de datos. Esto lo hace particularmente útil para procesamientos multimedia -audio, video e imágenes- donde la reproducción en tiempo real es crítica. Gracias a las instrucciones SIMD podemos ejecutar una misma operación sobre muchos datos al mismo tiempo en una sola instrucción, contrario a realizarlo en un modelo exclusivamente SISD (single instruction single data). Como veremos a lo largo del trabajo su utilización introducirá notables mejoras en performance y eficiencia de nuestros algoritmos en una gran cantidad de casos siempre que sea posible su aplicación.





1.1. Objetivos Generales

Además de la exploración del modelo SIMD se repasaron conceptos y técnicas de programación vectorizada en C y ASM dentro del campo de aplicación del procesamiento de imagenes. Para esto se llevó a cabo la implementación de los siguientes filtros para imágenes:

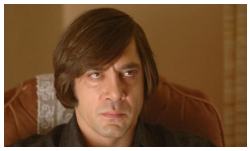
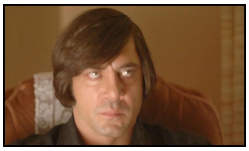


Imagen original



Cuadrados

Manchas

Offset

Sharpen

1.2. Metodología de trabajo

Al disponer de las implementaciones en C de todos los filtros, se procedió a realizar su implementación en lenguaje ensamblador para la arquitectura x86-64 de Intel. Para esto, se utilizaron las instrucciones SSE de dicha arquitectura, que aprovechan el ya mencionado modelo SIMD para procesar datos en forma paralela.

Una vez realizadas estas implementaciones, fueron sometidas a un proceso de comparación para extraer conclusiones acerca de su rendimiento. Con este fin, se experimentó con variaciones tanto en los datos de entrada como en detalles implementativos de los mismos algoritmos. De esta manera, se pudo recopilar datos sobre el comportamiento de cada implementación, y contrastar estos resultados con diversas hipótesis previamente elaboradas.

A continuación introducimos los filtros y sus respectivas implementaciones, luego describimos los tests realizados y los resultados obtenidos, y finalmente a partir de estos datos otorgamos algunas conclusiones sobre la experiencia realizada.

2. Filtros

2.1. Cuadrados

2.1.1. Descripción

Esta operación genera un efecto *geométrico*, borrando curvas y generando un efecto de *cuadrados*. Esto se logra reemplazando cada pixel por el máximo en un cuadrado con vértice en dicho pixel.

$$cuadrado(p) = \max_{0 \leq i \leq 3, 0 \leq j \leq 3} A[p_0 + i][p_1 + j]$$

2.1.2. Implementación ASM

En este filtro procesamos de a 4 píxeles: tenemos dos ciclos, uno que recorre las columnas de la imagen y otro que recorre las filas. A continuación el pseudocódigo de la implementación *SIMD*:

- Primero se arma el stack frame y se guarda en los registros de propósito general las entradas. (Líneas 30 a 45)
- Luego se dibuja el borde negro de 4 píxeles con el llamado a la función auxiliar *CompletarConCeros* en la línea 52.
- Antes de comenzar el ciclo se limpian los registros usados para evitar errores. (Líneas 62 a 70)
- Se setea el comienzo de las imágenes fuente y de destino luego del borde negro, y se trae de memoria la máscara a usar. (Líneas 72 a 77)
- Iniciamos los índices de columnas, filas y offset a usar para los ciclos del algoritmo. (Líneas 80 a 91)
- Condiciones de los ciclos de filas y columnas: si la fila llegó al final, saltar al final del ciclo de filas, y análogo con las columnas.
- El cuerpo del ciclo principal del filtro es donde se realiza el proceso del filtro. En los registros XMM se carga la matriz desde la memoria.

```
103 ;Loading 4x4 matrix on XMM registers.
104 movdqu first_row, [src]
105 movdqu second_row, [src+src_row_size]
106 movdqu third_row, [src+src_row_size*2]
107 movdqu fourth_row, [src+fourth_row_offset]
```

$row1[0]$	$row1[1]$	$row1[2]$	$row1[3]$

XMM0 ← row1

$row2[0]$	$row2[1]$	$row2[2]$	$row2[3]$

XMM1 ← row2

$row3[0]$	$row3[1]$	$row3[2]$	$row3[3]$

XMM2 ← row3

$row4[0]$	$row4[1]$	$row4[2]$	$row4[3]$

XMM3 ← row4

- Una vez que el algoritmo carga la matriz en los registros XMM0-XMM3, aplica el algoritmo para hallar el máximo. Este se encontrara en la parte baja del registro XMM4, y se copia a memoria en la imagen de destino.

```
;Find maximums and store in the vector_maximum register.
110 jmp .hallarMaximos

112 .retornarDeMaximos:
;Save maximums on the destination image.
114 movss [dst], vector_maximum
```

- Al final de estas cuatro comparaciones, el vector maximum tendrá en cada píxel, el máximo de su correspondiente columna.

```
;Algorithm for finding maximums:
137 .hallarMaximos:
138 pxor vector_maximum, vector_maximum

;Find maximum comparing row by row on each column.
141 pmaxub vector_maximum, first_row
142 pmaxub vector_maximum, second_row
143 pmaxub vector_maximum, third_row
144 pmaxub vector_maximum, fourth_row
```

$max(Column1)$	$max(Column2)$	$max(Column3)$	$max(Column4)$

vector_maximum (XMM4)

- Se rota el vector de máximos 3 veces y se hacen tres comparaciones, para obtener el máximo general de la matriz:

```
146 movdqu rotated_vector_maximum, vector_maximum

;Rotate vector one pixel to the right and compare.
;Repeat four times to find the maximum among all pixels in the vector.
150 pshufb rotated_vector_maximum, mask
151 pmaxub vector_maximum, rotated_vector_maximum
152 pshufb rotated_vector_maximum, mask
153 pmaxub vector_maximum, rotated_vector_maximum
154 pshufb rotated_vector_maximum, mask
155 pmaxub vector_maximum, rotated_vector_maximum
156 jmp .retornarDeMaximos
```

$max(Column2)$	$max(Column3)$	$max(Column4)$	$max(Column1)$

rotated_vector_maximum (XMM6)

...	...	...	$max(Column4, Column1)$

vector_maximum (XMM4)

$max(Column3)$	$max(Column4)$	$max(Column1)$	$max(Column2)$

rotated_vector_maximum (XMM6)

...	...	...	$max(Column4, Column1, Column2)$

vector_maximum (XMM4)

$max(Column4)$	$max(Column1)$	$max(Column2)$	$max(Column3)$

rotated_vector_maximum (XMM6)

...	...	...	max

vector_maximum (XMM4)

- De la línea 117 a la 134 se obtienen los próximos píxeles a procesar y cómo pasar a la siguiente fila cuando es necesario.
- Para terminar, se desarma el stack frame.

2.2. Manchas

2.2.1. Descripción

El objetivo de este filtro es afectar la luminosidad de los píxeles en la imagen produciendo un patrón de manchas.

Se cuenta con una función que dada la posición del pixel genera un valor llamado *tono* el cual se suma con igual peso a los tres componentes de color del pixel. Se define *tono* como:

$$\texttt{tono}[i,j] = \operatorname{sen}\left(\frac{2\pi(i\,\%n)}{n}\right) * \cos\left(\frac{2\pi(j\,\%n)}{n}\right) * 50 - 25 \qquad (1)$$

donde i y j son las posiciones vertical y horizontal respectivamente y n es el diámetro de las manchas en píxeles.

2.2.2. Implementación SISD en C

A continuación el pseudocódigo de la implementación *SISD*:

```
Para i de 0 a height - 1:
    Para j de 0 a width - 1:

        ii = 2 * PI * (i % n) / n
        jj = 2 * PI * (j % n) / n
        tono = sen(ii) * cos(jj) * 50 - 25;

        dst[i][j].b = SAT(src[i][j].b + tono)
        dst[i][j].g = SAT(src[i][j].g + tono)
        dst[i][j].r = SAT(src[i][j].r + tono)
```

2.2.3. Implementación SIMD en ASM

A continuación el pseudocódigo de la implementación *SIMD*:

- Se computa un arreglo de floats `sen_ii[i]` con el valor de $\text{sen}\left(\frac{2\pi(i\,\%n)}{n}\right)$ para cada valor de i entre 0 y `height`.
- Se computa un arreglo de floats `cos_jj[j]` con el valor de $\cos\left(\frac{2\pi(j\,\%n)}{n}\right)$ para cada valor de j entre 0 y `width`:
- para cada valor de i en $\{0, \ldots, \texttt{height} - 1\}$.
  - para cada valor de j en $\{0, 4, \ldots, \texttt{width} - 4\}$ (de 4 en 4).
    - Construye vector de 4 `doublewords` con el valor de `tono`

`tono[i,j+3]`	`tono[i,j+2]`	`tono[i,j+1]`	`tono[i,j+0]`

      Por simplicidad, de ahora en mas:

T_3	T_2	T_1	T_0

    - Convierte con saturación la parte baja a un vector de 8 `words` repitiendo 4 veces el valor de tono para los pixeles 1 y 0.

$\texttt{sat}(T_1)$	$\texttt{sat}(T_1)$	$\texttt{sat}(T_1)$	$\texttt{sat}(T_1)$	$\texttt{sat}(T_0)$	$\texttt{sat}(T_0)$	$\texttt{sat}(T_0)$	$\texttt{sat}(T_0)$

      tono_lo

    - Idem para la parte alta, los pixeles 3 y 2.

$\texttt{sat}(T_3)$	$\texttt{sat}(T_3)$	$\texttt{sat}(T_3)$	$\texttt{sat}(T_3)$	$\texttt{sat}(T_2)$	$\texttt{sat}(T_2)$	$\texttt{sat}(T_2)$	$\texttt{sat}(T_2)$

      tono_hi

    - Lee 4 pixeles de la memoria a partir de la posicion (i, j) hasta la posicion $(i, j + 3)$.

A3	R3	G3	B3	A2	R2	G2	B2	A1	R1	G1	B1	A0	R0	G0	B0

      px

    - Desempaqueta a `word` las partes alta y baja de `px`.

A1	R1	G1	B1	A0	R0	G0	B0

      px_lo

A3	R3	G3	B3	A2	R2	G2	B2

      px_hi

      A partir de ahora pensamos los pixeles como enteros con signo.

2.3. Offset

2.3.1. Descripción

Este filtro obtiene la imagen final desplazando cada uno de los componentes RGB de los píxeles de la imagen original por un offset -en la implementación actual el valor del mismo es 8 píxeles-. El resultado es la imagen original con estelas de colores alrededor de las figuras, producto de este desfasaje fijo de cada componente. Este efecto se logra respetando el siguiente pseudocódigo:

```
Para i de 8 a height - 9:
    Para j de 8 a width - 9:
        dst[i][j].b = src[i+8][j].b;
        dst[i][j].g = src[i][j+8].g;
        dst[i][j].r = src[i+8][j+8].r;
```

A su vez se deja un marco de 8 píxeles de color negro alrededor de toda la imagen.

2.3.2. Implementación ASM

Esta implementación procesa de a 4 píxeles por cada iteración sobre la imagen original y destino.

Cálculo del offset

Vamos a dar una breve explicación de las partes más relevantes del cálculo del offset en assembly. Para filtrar y separar en distintos registros los componentes de cada píxel se utilizaron las siguientes máscaras -almacenadas en los filtros *XMMx* detallados a continuación-, las cuales luego se aplicaron mediante la instrucción lógica *PAND* para filtrar y finalmente *POR* para combinar:

0xFF	0	0	0	0xFF	0	0	0	0xFF	0	0	0	0xFF	0	0	0

xmm10 ← canal alfa

0	0xFF	0	0	0	0xFF	0	0	0	0xFF	0	0	0	0xFF	0	0

xmm11 ← canal rojo

0	0	0xFF	0	0	0	0xFF	0	0	0	0xFF	0	0	0	0xFF	0

xmm12 ← canal verde

0	0	0	0xFF	0	0	0	0xFF	0	0	0	0xFF	0	0	0	0xFF

xmm13 ← canal azul

En cada iteración se cargan 4 píxeles de memoria y se guardan en un registro *XMMx*. Consideramos que en una iteración *i* y *j* corresponden a la fila y columna actual respectivamente.

```
55  movdqu xmm0, [rax + r11]
```

Los mismos quedan almacenados de la siguiente manera:

A3	R3	G3	B3	A2	R2	G2	B2	A1	R1	G1	B1	A0	R0	G0	B0

xmm0

Se traen los 4 píxeles correspondientes a la fila *i+8* y columna *j+8*. Con la máscara roja se conserva este componente. (Graficamos únicamente el primer píxel por espacio y comodidad, la lógica se repite)

```
61  movdqu xmm1, [r12 + rdx * 8]
62  pand xmm1, xmm11
```

...	...	...	...	...	...	...	...	...	...	...	...	0	src[i+8][j+8].r	0	0

xmm1

Se traen los 4 píxeles correspondientes a la misma fila y la columna *j+8*. Con la máscara verde se conserva este componente.

```
68  movdqu xmm2, [r12]
69  pand xmm2, xmm12
```

...	...	...	...	...	...	...	...	...	...	...	...	0	0	src[i][j+8].g	0

xmm2

Se traen los 4 píxeles correspondientes a la próxima fila y la misma columna. Con la máscara azúl se conserva este componente.

```
74  movdqu xmm3, [r12 + rdx * 8]
75  pand xmm3, xmm13
```

...	...	...	...	...	...	...	...	...	...	...	...	0	0	0	src[i+8][j].b

xmm3

Por último, se combinan los componentes de cada registro y se guarda el resultado en la dirección de memoria destino.

```
77  pxor xmm0, xmm0
78  por xmm0, xmm1
79  por xmm0, xmm2
80  por xmm0, xmm3
81  por xmm0, xmm10

83  movdqu [r8 + r11], xmm0
```

Mostramos el resultado para el primer píxel con el filtro calculado -se repite en los píxeles siguientes la misma lógica-.

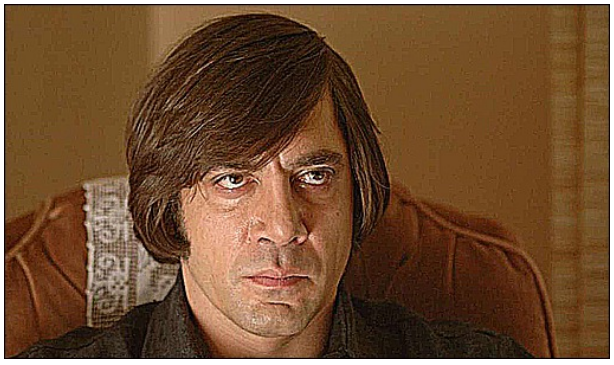
...	...	...	...	...	...	...	...	...	...	...	...	0xFF	src[i+8][j+8].r	src[i][j+8].g	src[i+8][j].b

src ← xmm0

De esta manera, en cada iteración procesamos 4 píxeles al mismo tiempo aprovechando las posibilidades de paralelismo que ofrece *SIMD*.

2.4. Sharpen

2.4.1. Descripción



Sharpen pertenece a un conjunto de filtros que se realizan aplicando operaciones matriciales (operadores) sobre los píxeles. El efecto de este filtro en particular es de aumentar la *nítidez* percibida de la imagen. Una implementación secuencial del mismo respeta el siguiente pseudocódigo:

```
                          | -1, -1, -1 |
float sharpen[3][3] =     | -1,  9, -1 |
                          | -1, -1, -1 |

Para i de 0 a height - 3:
    Para j de 0 a width - 3:
        totalB = 0, totalG = 0, totalR = 0
        Para ii de 0 a 2:
            Para jj de 0 a 2:
                totalB = totalB + sharpen[ii][jj] * src[i+ii][j+jj].b
                totalG = totalG + sharpen[ii][jj] * src[i+ii][j+jj].g
                totalR = totalR + sharpen[ii][jj] * src[i+ii][j+jj].r
        dst[i+1][j+1].b = SAT(totalB);
        dst[i+1][j+1].g = SAT(totalG);
        dst[i+1][j+1].r = SAT(totalR);
```

El filtro además, deja un marco negro de 1 píxel alrededor de toda la imagen.

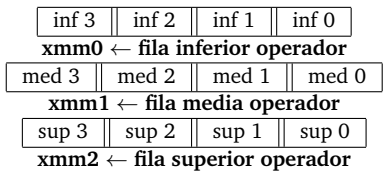
2.4.2. Implementación ASM

En nuestra implementación SIMD de Sharpen procesaremos de a 2 píxeles en simultáneo. Daremos la explicación de las partes más significativas del código assembler. En cada iteración del código se realizan los siguientes pasos:

Primero se traen de memoria los píxeles para tener el operador necesario para procesar de a dos píxeles. Podemos tener por registro *XMMx* cuatro píxeles, para los tres píxeles inferiores se usan para el primer píxel a aplicar el filtro, los tres píxeles superiores se utilizan para el segundo píxel procesado.

```
44  movdqu xmm0, [rax + r11]
45  add r11, r8
46  movdqu xmm1, [rax + r11]
47  add r11, r8
48  movdqu xmm2, [rax + r11]
```

De esta manera tenemos cuatro píxeles por cada uno de estos registros.



Hacemos los cálculos correspondientes a la fila inferior del operador matricial. En el registro *XMM11* acumulamos los resultados parciales.

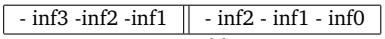
```
54  pxor xmm10, xmm10
55  movdqu xmm3, xmm0
56  punpckhbw xmm3, xmm10
57  movdqu xmm4, xmm0
58  punpcklbw xmm4, xmm10

60  pxor xmm11, xmm11
61  psubw xmm11, xmm3

63  psubw xmm11, xmm4
64  pslldq xmm3, 8
65  psrldq xmm4, 8
66  por xmm3, xmm4

68  psubw xmm11, xmm3
```

En *XMM11* tenemos dos píxeles con el siguiente cálculo parcial:



xmm11

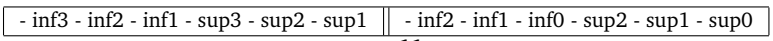
Luego, se efectúan los cálculos correspondientes a la fila superior del operador matricial.

```
71  movdqu xmm3, xmm2
72  punpckhbw xmm3, xmm10
73  movdqu xmm4, xmm2
74  punpcklbw xmm4, xmm10

76  psubw xmm11, xmm3

78  psubw xmm11, xmm4
79  pslldq xmm3, 8
80  psrldq xmm4, 8
81  por xmm3, xmm4
82  psubw xmm11, xmm3
```

El estado de *XMM11* ahora es:



xmm11

Para terminar con el cálculo del operador matricial de los dos píxeles que se están procesando en esta iteración se opera sobre la fila media del operador.

```
85  movdqu xmm3, xmm1
86  punpckhbw xmm3, xmm10
87  movdqu xmm4, xmm1
88  punpcklbw xmm4, xmm10

90  movdqu xmm5, xmm3
91  movdqu xmm6, xmm4
92  pslldq xmm5, 8
93  psrldq xmm6, 8
94  por xmm5, xmm6
95  times 9 paddw xmm11, xmm5
104 psubw xmm11, xmm3
105 psubw xmm11, xmm4
```

Tenemos en *XMM11* el resultado deseado, correspondiente a aplicar el operador en su totalidad:

-inf3 -inf2 -inf1 -sup3 -sup2 -sup1 -med3 + (med2 * 9) -med1	-inf2 -inf1 -inf0 -sup2 -sup1 -sup0 -med2 + (med1 * 9) -med0

xmm11

Por último empaquetamos *XMM11* de word a byte y movemos los dos píxeles a memoria.

```
108     packuswb xmm11, xmm11
109     movq r13, xmm11

111     mov qword [rsi + r11 * 4], r13
```

Antes de empaquetar:

sharpen(P1)	sharpen(P0)

xmm11

Después:

sharpen(P1)	sharpen(P0)	sharpen(P1)	sharpen(P0)

xmm11

En la línea 111 movemos únicamente la parte baja de *XMM11* a memoria, escribiendo dos píxeles procesados. Obteniendo de esta manera el resultado en la dirección de memoria destino. Al terminar de ciclar por todos los píxeles de la imagen original el filtro termina de ser aplicado a a la misma.

3. Experimentación

3.1. Mediciones y metodología de experimentación

Para poder realizar las mediciones correspondientes a los experimentos realizados utilizamos la instrucción de assembler *rdtsc*. Con ella podemos obtener el valor de Time Stamp Counter (TSC) del procesador -un registro de 64 bits presente en todos los procesadores x86 desde los Pentium-, como este se incrementa en uno con cada ciclo del procesador podemos obtener la duración en cantidad de ticks de una llamada a una función calculando la diferencia de este registro al principio y al final de su ejecución. Ya que esta medida no es constante entre llamadas, por cada caso de test (ejecución de un filtro, con una implementación y para una imagen con un tamaño específico) se realizaron 2500 llamadas. Al mismo tiempo, para poder tener las mediciones individuales de cada iteración y poder calcular la varianza y filtrar outliers se modificó el framework de la cátedra para tener disponible esta información. La modificación se realizó en el ciclo principal de la función *correr_filtro_imagen* del archivo *tp2.c*:

```
MEDIR_TIEMPO_START(start)
for (int i = 0; i < config->cant_iteraciones; i++) {
    unsigned long long start_iter, end_iter, iteraciones;
    MEDIR_TIEMPO_START(start_iter)
    aplicador(config);
    MEDIR_TIEMPO_STOP(end_iter)
    iteraciones = end_iter - start_iter;
    fprintf(fp, ";%llu", iteraciones);
}
MEDIR_TIEMPO_STOP(end)
```

Luego, una vez que se contó con los tiempos de cada iteración se lidió con los outliers calculando una media 25 % podada. La computadora en la cual se corrieron los experimentos tiene las siguientes especificaciones:

Cuadro 1: Especificaciones cpu

model name	Intel(R) Core(TM) 14-nm ”Broadwell ULTi5-5350U CPU @ 1.80GHz
cpu MHz	1799.948
l2 cache size	256 KB
l3 cache size	3072 KB
mem total	8 GB 1600 MHz DDR3

A su vez, cuenta con los flags detallados a continuación: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht syscall nx rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc eagerfpu pni pclmulqdq monitor ssse3 cx16 pcid sse4_1 sse4_2 movbe popcnt aes xsave avx rdrand lahf_lm abm 3dnowprefetch invpcid_single fsgsbase avx2 invpcid rdseed flush_l1d

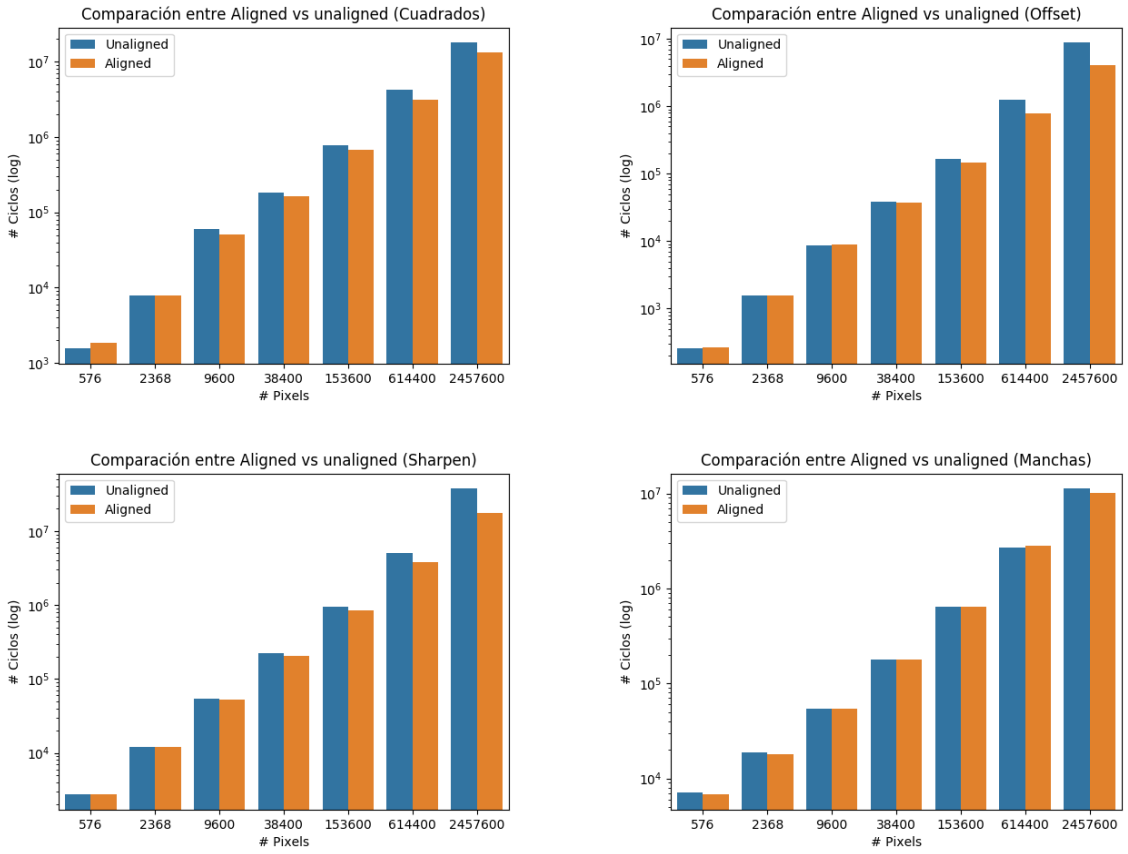
Se utilizó la siguiente combinación de sistema operativo y compiladores: Ubuntu/Xenial64 16.04.4 LTS, GCC versión 5.4.0 y NASM versión 2.11.08. Para compilar el código C se usaron los flags *-ggdb -Wall -Wno-unused-parameter -Wextra -std=c99 -pedantic -m64 -O0* y para ASM *-felf64 -g -F dwarf* conforme a los archivos proporcionados por la cátedra de la materia.

La imagen utilizada para todos los experimentos fue la correspondiente a *No Country For Old Men*, proporcionada por la cátedra. Otros detalles de las mediciones se dan específicamente por cada experimento producido.

3.2. Comparación entre accesos a memoria alineados y desalineados

En la extensión SSE, se cuenta con dos versiones para la mayoría de las instrucciones que realizan accesos a memoria. Una versión *aligned* y una versión *unaligned*. Las instrucciones alineadas prometen un acceso mas rápido a los datos requiriendo que las direcciones sean múltiplos de 16 bytes. Las instrucciones no-alineadas permiten acceder a datos en cualquier posición de memoria a cambio de un costo mayor en el tiempo de acceso al dato. Eso se debe a la necesidad de realizar mas accesos para completar la operación de lectura o escritura del dato.

Como experimento nos propusimos cuantificar el costo extra de usar instrucciones *unaligned* frente a una implementación que haga uso, siempre que haya garantía de tener los datos alineados, de las instrucciones *aligned*. Nuestra hipótesis es que una implementación que no hace uso de instrucciones alineadas siempre que sea posible, debe requerir una cantidad mucho mayor de ciclos de CPU para completar la misma tarea que una implementación que haga uso de instrucciones alineadas.



Filtro	Tiempo Unaligned	Tiempo Aligned	Diferencia porcentual
Cuadrados	1.0	0.66	-34.26 %
Offset	1.0	0.35	-65.16 %
Sharpen	1.0	0.37	-63.38 %
Manchas	1.0	0.85	-15.42 %

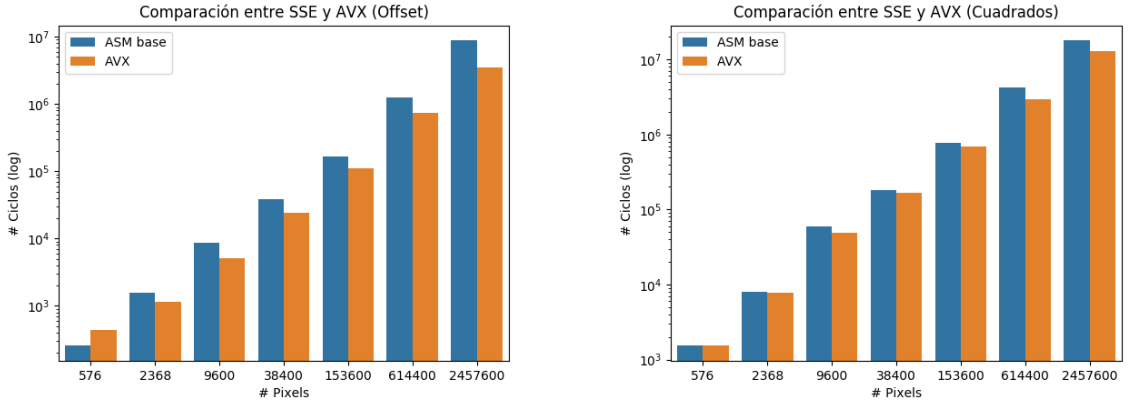
En los filtros Cuadrados, Offset y Sharpen, puede verse una mejora en el rendimiento. Para el tamaño mas grande de imagen, 2048×1200, se logro una mejora del $34{,}26\,\%$ en cuadrados, para Offset, una mejora del $65{,}16\,\%$ entre la cantidad de ciclos promedio entre ambas implementaciones. Para Sharpen, la diferencia obtenida fue de $63{,}38\,\%$ a favor de la implementación alineada. Para el filtro Manchas, el resultado fue menor que en los otros filtros, pero aun se obtuvo una diferencia de 15.42 % a

Concluimos que el alineamiento de los datos mejora notablemente el rendimiento de los filtros, por lo que debe usarse siempre que esto sea posible, para obtener una mejor performance.

3.3. Comparación entre uso de registros SSE y AVX

Advanced Vector Extensions o *AVX* es una extensión a la arquitectura *x86* que permite extender las operaciones *SIMD* de 128 bits a 256 bits. Esto significa que para un mismo ciclo de *fetch-decode-execute* podemos procesar el doble de datos que se procesaban anteriormente con la extensión *SSE*.

Como experimento, hicimos implementaciones alternativas para los filtros *Cuadrados* y *Offset* haciendo uso de la extensión *AVX*. Esperamos ver una mejora notable en la impementación *AVX* frente a la implementación base *SSE* ya que se procesa el doble de datos en una cantidad parecida de instrucciones.



Filtro	Tiempo Unaligned	Tiempo Aligned	Diferencia porcentual
Cuadrados	1.0	0.73	-27.13 %
Offset	1.0	0.40	-59.85 %

Para el filtro *Offset*, la mejora es significativa. En el tamaño de imagen mas grande, 2048×1200, se tiene una mejora del $58{,}85\,\%$ en la cantidad de ciclos de CPU promedio. Para *Cuadrados* puede verse también una mejoría, de menor grado pero también importante, arrojando una mejora del $27{,}13\,\%$ entre la cantidad de ciclos promedio consumida por cada impelementación en la imagen mas grande.

Se ve un beneficio importante al hacer uso de los registros *AVX*. En Offset podemos ver que la mejora incluso supera el 50 % teórico que resulta de procesar el doble de datos.

3.4. Loop unrolling: Sharpen y Offset

Loop unrolling es una técnica de optimización en la que se transforma un ciclo para mejorar la velocidad de un programa a expensas del tamaño de su binario. El objetivo es aumentar el rendimiento de un programa reduciendo o eliminando las instrucciones que controlan el ciclo, como comparaciones, reduciendo penalidades por branching y saltos en la ejecución del código. Para realizar esto los loops se escriben como una secuencia repetida del cuerpo del mismo. Probamos la efectividad de esta técnica con los filtros *Offset* y *Sharpen*.

La forma de realizarlo fue usando la instrucción de loops del preprocesador que ofrece NASM, *%REP*, la misma toma un argumento numerico y un bloque de código que es replicado la cantidad de veces indicada por el argumento. Tomamos el ciclo más interior de los filtros (el que hace los accesos a memoria, los procesa y almacena) y lo encerramos en uno de estos *%REP*. A su vez, incluímos el *%REP* dentro de otro loop para poder ir de un extremo donde el ciclo original del filtro es expandido en su totalidad resultando en ninguna repetición de ciclos, hasta otro donde se realizan todos los ciclos necesarios para no tener código repetido o expandido (lo cual equivale a nuestra implementación final de los algoritmos).

Utilizamos la imagen de No Country For Old Men en su versión de 2048 x 1200 píxeles para tener la mayor cantidad de pruebas expandiendo los ciclos. Mostramos los resultados obtenidos en la siguiente tabla:

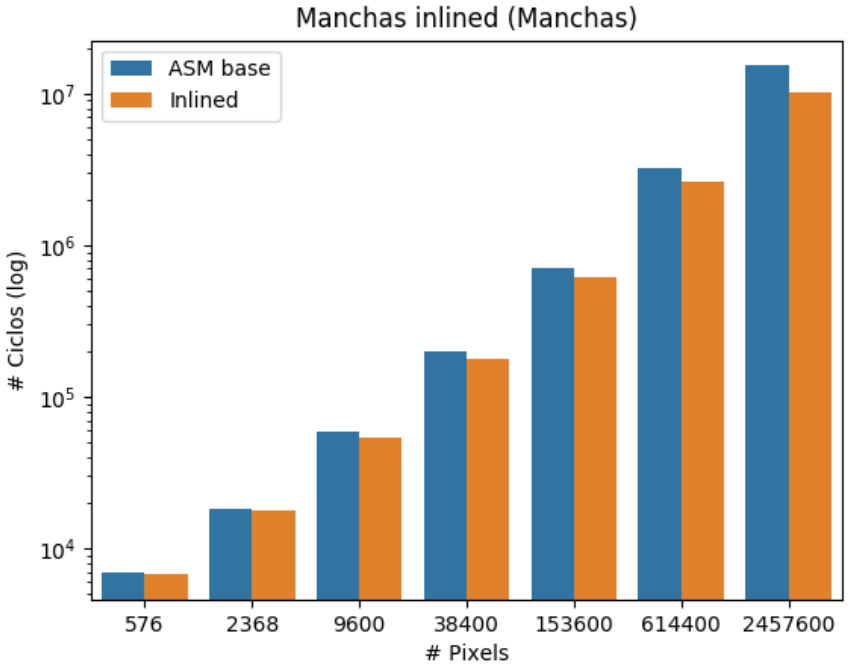
Filtro	Iteraciones Loop	Repeticiones Unrolled	Tamaño del binario (bytes)	Radio del tamaño respecto a implementación base	Ciclos de reloj	Radio de los ciclos de reloj respecto a implementación base
Offset	512	1	3696	1,00	4009672	1,00
Offset	256	2	3792	1,03	3669473	0,92
Offset	128	4	3984	1,08	3767671	0,94
Offset	64	8	4368	1,18	3750223	0,94
Offset	32	16	5136	1,29	3673374	0,92
Offset	16	32	6672	1,81	3653327	0,91
Offset	8	64	9744	2,64	3612854	0,90
Offset	4	128	15888	4,30	3683683	0,92
Offset	2	256	28192	7,63	3930615	0,98
Offset	1	512	52688	14,26	4294925	1,07
Sharpen	1024	1	3296	1,00	18122512	1,00
Sharpen	512	2	3536	1,07	17643338	0,97
Sharpen	256	4	4048	1,23	17556130	0,97
Sharpen	128	8	5072	1,54	17133416	0,95
Sharpen	64	16	7104	2,16	17242764	0,95
Sharpen	32	32	19344	5,87	17569156	0,97
Sharpen	16	64	19344	5,87	18084884	1,00
Sharpen	8	128	35664	10,82	18520954	1,02
Sharpen	4	256	68304	20,72	17943204	0,99
Sharpen	2	512	133584	40,53	19556586	1,08
Sharpen	1	1024	264144	80,14	29322494	1,62

Los resultados muestran cómo el tamaño de los binarios crece cada vez que se expande más el ciclo, dependiendo del filtro el radio al tamaño de la implementación base varió hasta ser 80 a 1 en el caso de Sharpen. La cantidad de ciclos de reloj promedio disminuyó constantemente hasta cierta combinación de ciclos y repeticiones, pero luego volvió a aumentar, llegando a ser incluso mayor que en la implementación base. La explicación de este comportamiento puede ser debido a que el código -o parte del mismo- es cacheado, o incluso el branch predictor puede predecir repeticiones del ciclo que no puede realizar cuando está expandido. La leve mejora del código se contrapone al incremento en el tamaño del binario, si no se realiza de manera transparente, el código resultante es muy extenso y dificil de leer y mantener, conteniendo claramente, una cantidad inaceptable de código repetido. Su provecho como técnica de optimización de compiladores puede resultar ventajosa, pero hay que tener en cuenta el tamaño de los binarios si el espacio es una preocupación o requerimiento.

3.5. Manchas inlined

Inlinear una función es una optimización habitual que realizan los compiladores, se espera que esto resulte en alguna mejora de performance en detrimento de la modularidad y legibilidad del código fuente del programa. Esta hipótesis se basa en el hecho de que un llamado a una función mediante la instrucción *call* implica un overhead que no existiría si el cuerpo de la función invocada estuviese disponible en el código siendo leído por el procesador: pushear la instrucción de retorno, armar el stack para el cambio de contexto y buscar el código a ser ejecutado, que incluso puede resultar en penalizaciones si no está disponible en memoria. Para poner a prueba esta optimización reemplazamos los llamados a funciones con el código de las mismas en el lugar donde son invocadas. Las funciones que fueron expandidas en el códidgo fueron: *computar_sen_ii*, *computar_cos_jj*, y *tono*. Los resultados obtenidos se muestran en la siguiente table y el siguiente gráfico:

Pixeles	Promedio Ciclos de reloj			Desviación estándar		Coeficiente de variación (desvío/promedio)	
	Normal	Inlined	Radio (normal/inlined)	Normal	Inlined	Normal	Inlined
576	6,943.24	6,739.50	97.07%	235.26	8.64	0.0339	0.0013
2368	18,469.65	17,906.93	96.95%	628.98	9.63	0.0341	0.0005
9600	59,310.22	53,760.07	90.64%	2,151.17	895.08	0.0363	0.0166
38400	201,143.67	176,148.92	87.57%	13,425.18	4,222.49	0.0667	0.0240
153600	708,893.06	623,077.44	87.89%	52,188.34	15,681.02	0.0736	0.0252
614400	3,215,846.95	2,637,839.17	82.03%	254,921.58	168,200.34	0.0793	0.0638
2457600	15,266,415.69	10,087,612.96	66.08%	2,128,886.31	388,576.97	0.1394	0.0385



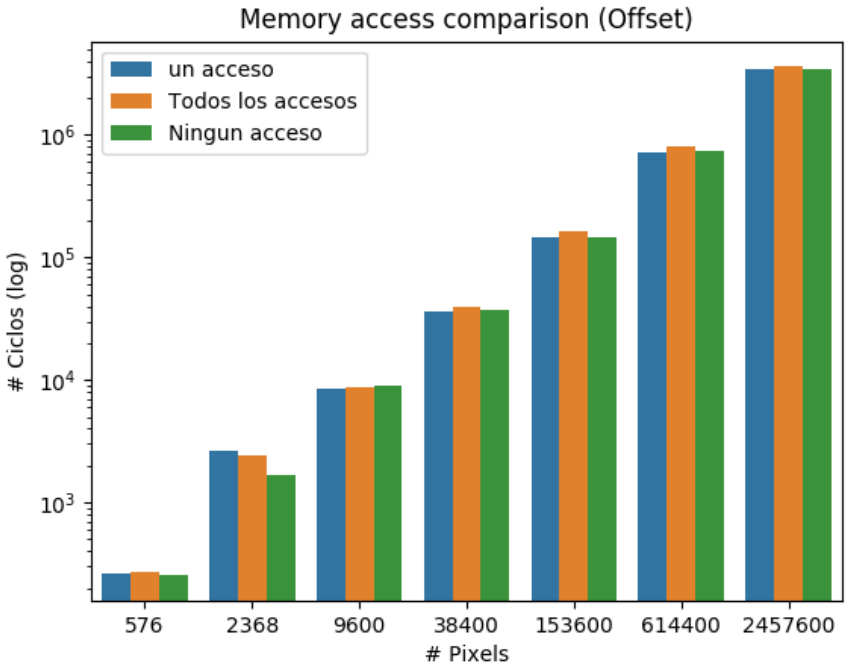
Los resultados obtenidos concuerdan con nuestras expectativas. Se observan mejoras en la cantidad promedio de ciclos insumidos para todos los tamaños de imágenes, siendo incluso más acentuada la diferencia entre inlinear la función y no hacerlo a medida que se incrementa el tamaño de la imagen de prueba. También observamos que el desvío estándar fue menor al inlinear las funciones, por lo que los resultados están menos dispersos alrededor de la media. Para la imagen más grande obtuvimos que la implementación inlined consume un 66 % de los ciclos de reloj de la implementación original. Como contrapartida se obtiene un código fuente más monolítico, de menor legibilidad, que impondría mayores dificultades de mantenimiento. Si bien hubo mejoras en la performance, ninguna fue significativa, por lo que su utilidad debería ser evaluada contraponiendo mantenimiento y performance. En estos casos no se observó que el rendimiento empeorase ni que el tamaño del binario se incremente, dos efectos que tal vez podrían suceder si el código introducido fuese más grande que el cache de instrucciones o si el mismo proveniese de una librería o archivo externo -particularmente en nuestro caso, las funciones inlineadas ya eran parte del archivo utilizado, por lo que el binario mantuvo su tamaño a pesar de la modificación-.

3.6. Accesos a memoria para armar máscaras

Las *máscaras* son de mucha utilidad para filtrar el contenido de un registro o dirección de memoria, en algunos casos incluso son necesarios y la única opción para realizar determinados procesos (varias instrucciones de la arquitectura Intel devuelven máscaras con el resultado de una operación). Dentro del filtro *Offset* utilizamos máscaras para separar los componentes de color de cada pixel y así poder lograr el efecto de desfasaje -la explicación de cómo lo realizamos está en la sección correspondiente del filtro-. En nuestra primera implementación guardamos en la sección *.data* las máscaras que necesitamos y luego las cargamos en registros haciendo accesos a memoria al inicio del programa, sin embargo existe la opción de armar las mismas con diferentes combinaciones de instrucciones:

```
pcmpeqq xmm10, xmm10
pslld xmm10, 24       ; máscara para canal alfa
movdqu xmm11, xmm10
psrld xmm11, 8        ; máscara para canal rojo
movdqu xmm12, xmm11
psrld xmm12, 8        ; máscara para canal verde
movdqu xmm13, xmm12
psrld xmm13, 8        ; máscara para canal azul
```

Motivados por estas posibilidades decidimos comparar el rendimiento del filtro en ambas, sumando una tercera opción en la cual en vez de cargar en registros las máscaras una única vez, lo repetimos en cada iteración del ciclo principal del algoritmo, provocando una cantidad mayor de accesos a memoria. Los resultados obtenidos fueron los siguientes:



Se puede comprobar que el efecto de la penalización de acceder a memoria en nuestros casos de prueba solo mostró grandes diferencias en el segundo caso (para imagen de 2368 píxeles). Llama la atención para el mismo caso que acceder a memoria una vez fuera del ciclo haya resultado menos performante que hacerlo en cada iteración. Luego, el resto de los casos de prueba dio resultados semejantes en todas sus variantes, nuestra hipótesis es que esto se debe a que estas posiciones de memoria son cacheadas y luego de varios accesos ya no impacta en el rendimiento promedio. Además tratándose de imágenes más grandes, esta penalización de ciclos de reloj se puede ver enmascarada por el resto del proceso del algoritmo, esto podría explicar por qué a medida que se agranda el tamaño de la imagen, la diferencia en cantidad de ciclos de reloj en promedio de cada implementación se mantiene relativamente constante. En promedio la implementación con más accesos a memoria consume más ciclos de reloj, pero esta diferencia no parece significativa.

3.7. Cuadrados usando pshuffle o right shifts

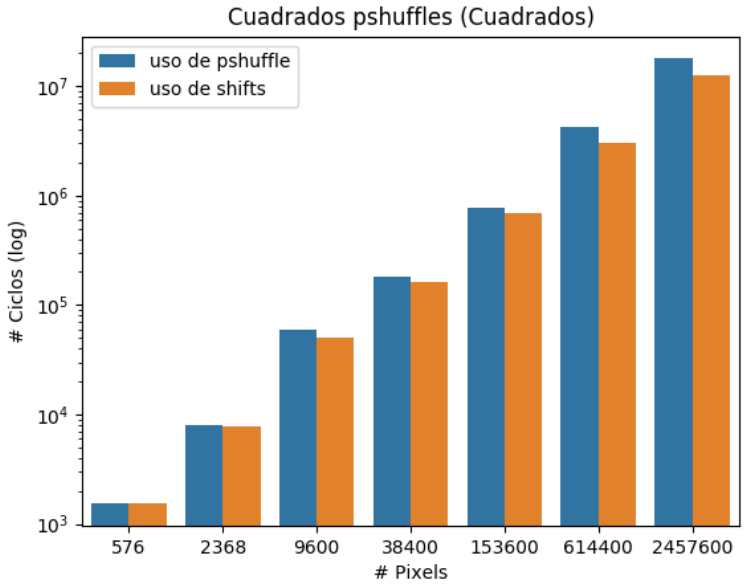
En este experimento realizamos la siguiente modificación al algoritmo. Primero se muestra el algoritmo original y luego la modificación:

```
;Rotate vector one pixel to the right and compare.
;Repeat four times to find the maximum among all pixels in the vector.
pshufb rotated_vector_maximum, mask
pmaxub vector_maximum, rotated_vector_maximum
pshufb rotated_vector_maximum, mask
pmaxub vector_maximum, rotated_vector_maximum
pshufb rotated_vector_maximum, mask
pmaxub vector_maximum, rotated_vector_maximum
jmp .retornarDeMaximos
```

```
psrldq vector_maximum_copy, 4
pmaxub vector_maximum, vector_maximum_copy
psrldq rotated_vector_maximum, 4
pmaxub vector_maximum, vector_maximum_copy
psrldq rotated_vector_maximum, 4
pmaxub vector_maximum, vector_maximum_copy
jmp .retornarDeMaximos
```

Buscamos analizar la diferencia en performance del algoritmo “Cuadrados” usando pshuffle y usando right shifts. El primer caso rota el registro un pixel de izquierda a derecha. La idea es que rotando cuatro veces el vector que contiene los máximos de cada columna y comparando consigo misma 3 veces, obtendremos el máximo general del vector.

El segundo algoritmo shiftea a derecha un pixel y compara. Si bien compara todo el vector, solamente el pixel más significativo es relevante, ya que será donde se alojará el máximo general luego de finalizado el proceso. Al ser el shift una operación más sencilla, y dado que no se tiene que ir a buscar la máscara a memoria, esperamos que la versión con pshuffles sea más lenta que la que usa shifts. Como unidad de medición de nuestros experimentos usamos la cantidad de ciclos de reloj contra la cantidad de píxeles de la imagen.

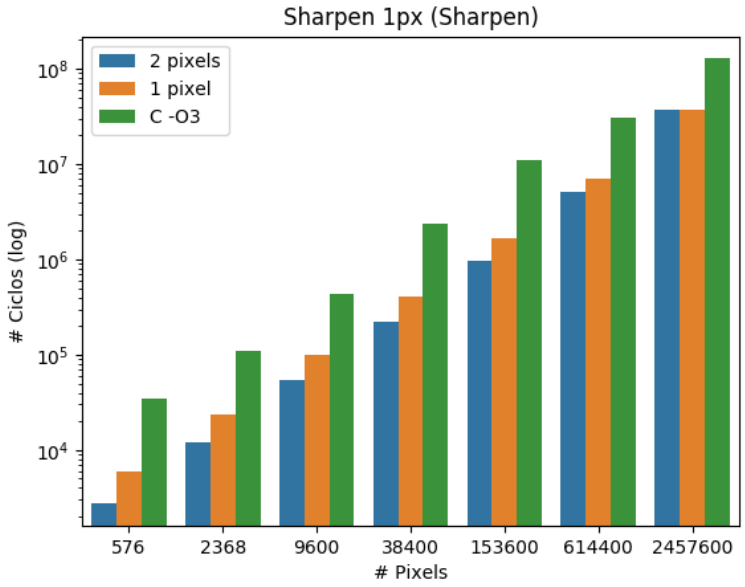


Pixeles	Promedio Ciclos de reloj			Desviación estándar		Coeficiente de variación (desvío/promedio)	
	Pshuffle	Shifts	Radio (shift/pshuffle)	Pshuffle	Shifts	Pshuffle	Shifts
576	1,554.65	1,530.78	98.46%	7.08	4.46	0.0046	0.0029
2368	7,930.21	7,862.01	99.14%	16.23	12.90	0.0020	0.0016
9600	59,773.94	50,384.72	84.29%	8,720.33	1,620.34	0.1459	0.0322
38400	183,568.65	163,137.51	88.87%	15,914.24	123.41	0.0867	0.0008
153600	781,494.62	682,253.11	87.30%	78,681.44	12,037.99	0.1007	0.0176
614400	4,263,108.30	3,010,163.97	70.61%	555,219.49	82,811.07	0.1302	0.0275
2457600	17,783,111.34	12,495,929.33	70.27%	1,105,249.85	350,007.89	0.0622	0.0280

Los resultados que obtuvimos respaldaron nuestra hipótesis: la versión que usa solamente pshifts finaliza en una menor cantidad de ciclos de reloj que la versión que usa pshuffle. Sin embargo, la diferencia solamente se hace significativa para tamaños “grandes” de imagenes (alrededor de 70 % menos ciclos de reloj); para tamaños pequeños, la performance es casi la misma. Esto seguramente se deba a que para tamaños chicos la diferencia en instrucciones sacada por usar right shift no es grande, mientras que para imágenes de mayor cantidad de píxeles y por lo tanto, muchos ciclos, el ahorro acumulado de unas pocas instrucciones se vuelve notable.

Por otra parte, usar shift es intuitivamente más sencillo que usar pshuffle, ya que esta última involucra un algoritmo más complejo que simplemente mover los bits del registro a la derecha. Por otra parte no hace llamados a memoria, lo cual inmediatamente implica una mejora en performance.

3.8. Sharpen procesando de a 2 píxeles y de a 1 píxel



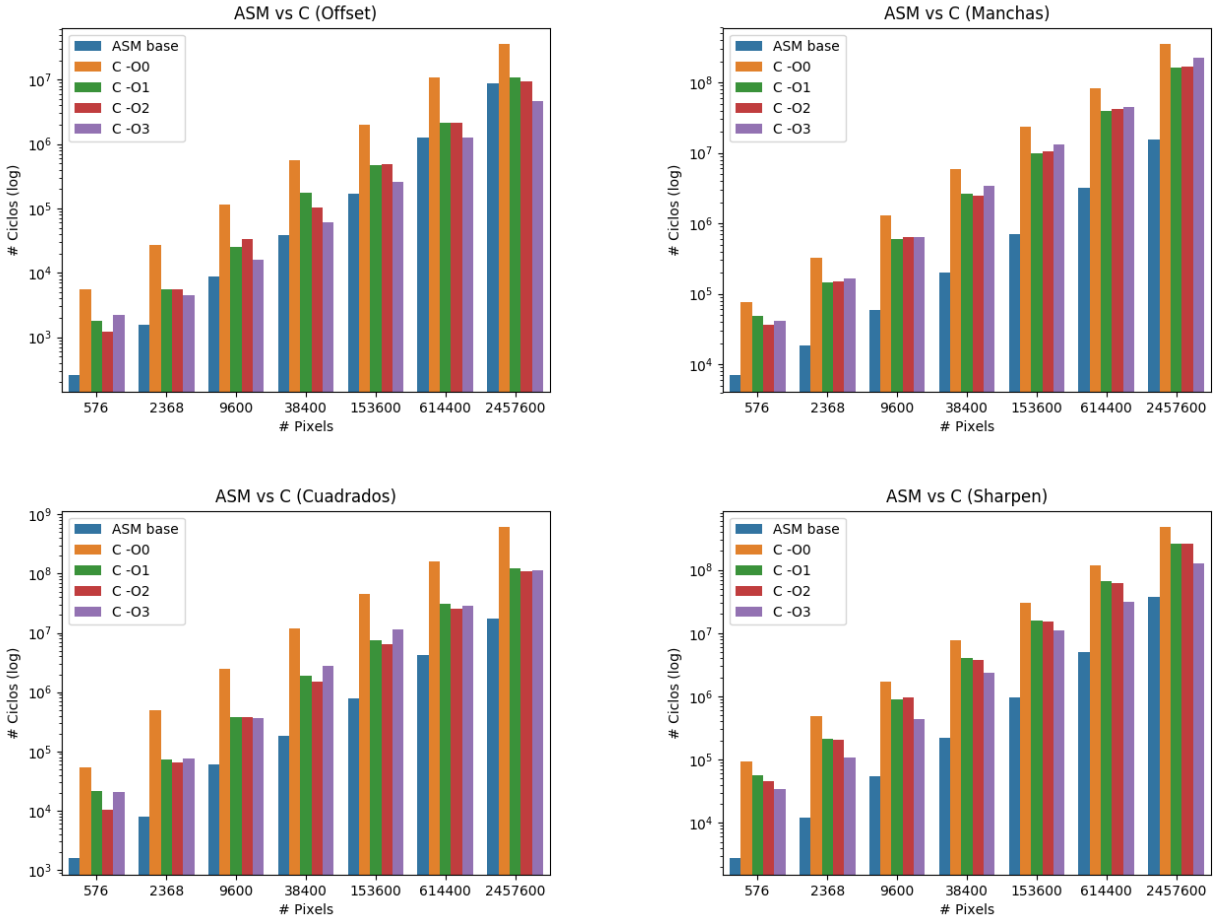
En este experimento modificamos Sharpen para que en vez de procesar de a 2 píxeles, procese de a 1 pixel. Nuestro objetivo es analizar la mejora del algoritmo “Sharpen” procesando de a dos píxeles en su versión ASM respecto de versiones que procesan de a 1, tanto un ASM como en C. Nuestra expectativa es no solo que procesando de a dos píxeles, el algoritmo sea el doble de rápido, sino que incluso procesando de a un píxel, la versión en ASM tenga mejor rendimiento que la versión en C (compilada con todas las optimizaciones, O3). Como unidad de medición de nuestros experimentos usamos la cantidad de ciclos de reloj contra la cantidad de píxeles de la imagen.

Observando los resultados, y teniendo en cuenta que la escala es logarítmica, podemos comprobar que efectivamente en promedio el algoritmo es al menos el doble de rápido procesando de a 2 píxeles en ASM con respecto de C. En algunos casos es incluso notoriamente mayor, como en el caso de imágenes pequeñas (de 576 píxeles).

Un resultado sorprendente que obtuvimos es que en tamaños de imágenes muy grandes, la diferencia de velocidad parece ser casi despreciable. Como hipótesis podemos conjeturar que al tratarse de imágenes más grandes pueden ocurrir dos fenómenos: por un lado el procesador accede más veces a memoria y por lo tanto la cache puede almacenar información correspondiente a los píxeles siguientes al que se está procesando, y por otro lado el mismo proceso del filtro puede estar enmascarando en ciclos de reloj la diferencia de las implementaciones ASM.

Finalmente, ambos algoritmos son muy superiores al algoritmo en C, logrando ventajas de casi 10 veces menos ciclos de reloj.

3.9. Implementaciones ASM y C



En este experimento buscamos comparar el rendimiento de los algoritmos escritos en Assembler y aprovechando los recursos SIMD del microprocesador, contra los algoritmos escritos en C sin utilizar esos recursos. Nuestra hipótesis de trabajo es que las versiones en Assembler deberían ser superior, dado que los registros XMM permiten paralerizar cálculos sobre matrices y vectores. Como unidad de medición de nuestros experimentos usamos la cantidad de ciclos de reloj contra la cantidad de píxeles de la imagen.

La primera observación es que los resultados que obtuvimos respaldaron nuestra hipótesis: efectivamente la versión en Assembler resultó ser más rápida que la versión en C (sin optimizaciones). Las mejoras llegaron a ser del orden de 10 veces menos cantidad de ciclos de reloj (por ejemplo en el caso del algoritmo Sharpen). Esta mejora muy significativa se nota en todos los algoritmos (a excepción del filtro Offset). Resulta aún más notable para imagenes "pequeñas" (desde 1 a 2500 píxeles) donde la diferencia es mucho más significativa.

Pasando al código en C con optimizaciones, vale la pena separar entre optimización O1 y O2 por un lado, y O3 por el otro. Las optimizaciones O1 y O2 fueron, en líneas generales, muy similares, con escasa diferencia entre ambas. Un factor común en todos los algoritmos es que O1 y O2 fueron decididamente más rápidos que el código en C base (O0), pudiendo a lo sumo, igualarlo en un único caso (Offset).

En el caso de la optimización O3, la situación es diferente. Este nivel de optimización en muchos casos *no es* más rápida que las optimizaciónes O1 y O2. En los casos en que lo es, la diferencia es notable pero pequeña. En los casos de Cuadrados y Manchas, O3 es igual, o peor que O1 y O2. En Offset es igual o algo mejor, y en Sharpen es igual o mejor, siempre dependiendo del tamaño de la imagen. Entonces, no podemos decir que universalmente y sin discusión sea mejor que O1 u O2.

Para el filtro Offset encontramos resultados que contradijeron nuestra hipótesis: para imágenes del tamaño más grande (de más de 2 millones de píxeles), el código en C con optimización O3 es superior a la versión en Assembler. Este comportamiento (donde O3 es igual o más rápido que assembler), sólo se

nota para imagenes muy grandes. En los experimentos anteriores exploramos algunas optimizaciones habituales de los compiladores (loop unrolling, inlinear funciones), es posible que les mismas introducidas con el flag O3 en estos casos sean más significativas por la lógica misma del algoritmo.

4. Conclusiones y trabajo futuro

Gracias a los experimentos realizados en el presente trabajo, se pudo llegar a la conclusión de que las ventajas que puede brindar el paradigma **SIMD** a la hora de implementar programas que realicen operaciones altamente paralelizables, como el procesamiento de imágenes, son verdaderamente significativas. Esto queda reflejado en la gran brecha de rendimiento que se observa entre algunas implementaciones realizadas con dicho paradigma y las que utilizan el lenguaje de programación C.

Esto siempre debe contraponerse a otro hecho que se hizo presente durante el proceso de implementación: realizar un programa en lenguaje ensamblador resulta, por lo general, considerablemente más difícil que hacerlo en un lenguaje de más alto nivel. El código resultante es menos legible, es más sencillo cometer errores y el proceso de *debugging* se vuelve considerablemente más arduo. Por eso es importante analizar de antemano las características del contexto particular de aplicación, para poder decidir si este esfuerzo adicional realmente vale la pena.

Incluso una vez realizada una implementación en ASM, es necesario comparar el rendimiento de ambas versiones con distintos tamaños de imágenes para ver que efectivamente haya una mejora y el grado de la misma -por ejemplo, en nuestras implementaciones Offset para tamaños grandes obtuvo peores resultados que la versión en C con O3-. Otros aspectos a tener en consideración es el tamaño de los binarios involucrados, que puede ser decisivo en determinados escenarios.

También se intentó la realización de una optimización manual dentro del código ensamblador, con distintos resultados. Por ejemplo: expander los ciclos, inlinear llamadas a funciones, analizar instrucciones individuales para encontrar alternativas, tuvieron distintos beneficios en algunos casos y en otros no. Estas optimizaciones son realizadas por el compilador de C al setear distintas las opciones del flag O, las distintas técnicas de optimización que realizan los mismos pueden servir de inspiración a la hora de intentar mejorar el rendimiento de un programa, pero su justificación siempre tendrá que ser acompañada de métricas que las respalden.

Por último, pudimos hacer una pequeña prueba con la extensión del modelo de SIMD propuesto por la tecnología AVX2 cuyos resultados confirmaron las ventajas de su utilización en la programación vectorial a bajo nivel.

Esperamos que la experiencia nos provea de nuevos recursos a la hora de afrontar problemáticas similares en el futuro.