



**DEPARTAMENTO  
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico II

## Modelo de procesamiento SIMD

Organización del Computador II  
Segundo Cuatrimestre de 2016

**Grupo: El Arquitecto**

Integrante	LU	Correo electrónico
Freidin, Gregorio	433/15	gregoriofreidin@gmail.com
Taboh, Sebastián	185/13	sebi_282@hotmail.com
Romero, Lucía Inés	272/15	luciainesromero@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

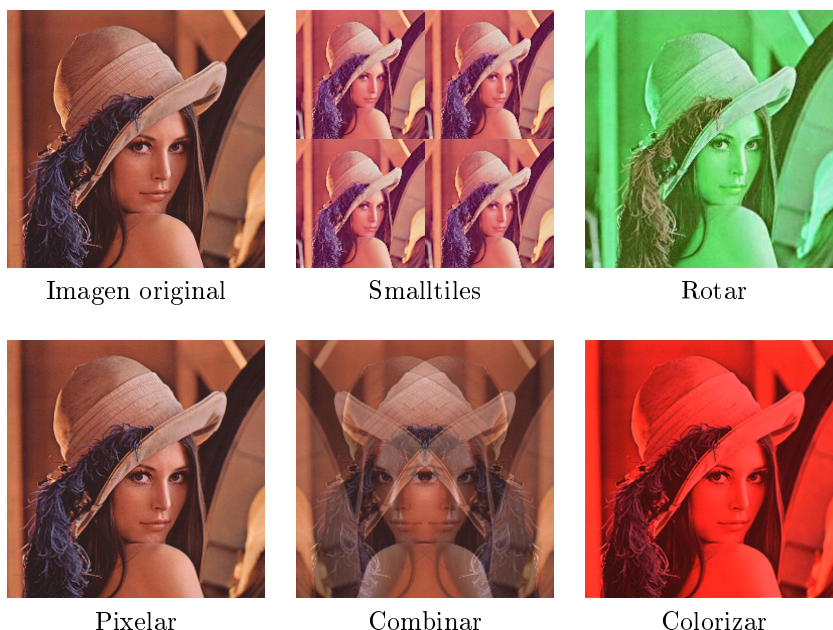
<b>1. Introducción</b>	<b>3</b>
1.1. Experimentación	3
<b>2. Smalltiles</b>	<b>4</b>
2.1. Código C	4
2.2. Código ASM	4
2.3. Experimentación	5
2.3.1. Idea	5
2.3.2. Hipótesis	5
2.3.3. Resultados	5
<b>3. Rotar canales</b>	<b>7</b>
3.1. Código C	7
3.2. Código ASM	7
3.3. Experimentación	7
3.3.1. Idea	7
3.3.2. Hipótesis	7
3.3.3. Resultados	8
<b>4. Pixelar</b>	<b>9</b>
4.1. Código C	9
4.2. Código ASM	9
4.3. Experimentación 1	10
4.3.1. Idea	10
4.3.2. Resultados	10
4.4. Experimentación 2	10
4.4.1. Idea	10
4.4.2. Hipótesis	11
4.4.3. Resultados	11
4.4.4. Conclusión	11
<b>5. Combinar</b>	<b>12</b>
5.1. Código C	13
5.2. Código ASM	13
5.3. Experimentación	15
5.3.1. Hipótesis	15
5.3.2. Idea	15
5.3.3. Resultados	16
<b>6. Colorizar</b>	<b>18</b>
6.1. Código C	18
6.2. Código ASM	18
6.3. Experimentación 1	20
6.3.1. Idea	20
6.3.2. Hipótesis	21
6.3.3. Resultados	21
6.3.4. Conclusión	22
6.4. Experimentación 2	22
6.4.1. Idea	22
6.4.2. Hipótesis	22
6.4.3. Resultados	23
6.4.4. Conclusión	23
6.5. Experimento 3	23
6.5.1. Idea	23
6.5.2. Hipótesis	23
6.5.3. Resultados	24
6.5.4. Conclusión	24

## 1. Introducción

Este trabajo propone observar las diferencias en los tiempos de cómputo debidas al aprovechamiento del modelo de procesamiento SIMD con respecto a implementaciones en C.

Se implementaron diversos filtros de los cuales se expone una breve explicación a continuación y luego se detallan en sus respectivas secciones.

- **Smalltiles:** genera una imagen del mismo tamaño de la original que la contiene 4 veces, una en cada cuadrante.
- **Rotar canales:** intercambia los colores de modo que lo azul pasa a ser rojo, lo rojo, verde y lo verde, azul.
- **Pixelar:** como su nombre lo indica, pixela la imagen disminuyendo la definición.
- **Combinar:** consiste en combinar 2 imágenes en función de un parámetro.
- **Colorizar:** intensifica los colores predominantes en cada píxel.



Además de implementar los filtros en C y en Assembler se llevaron a cabo distintos experimentos para confirmar la hipótesis que motivó el trabajo, el hecho de que el procesamiento SIMD es más eficiente y permite resultados mucho más veloces que los obtenidos con las implementaciones de C.

### 1.1. Experimentacion

En nuestra experimentación elegimos testear los tiempos de ejecución de nuestros algoritmos corriendo cada implementación una cantidad fija de veces y luego, en el caso de los gráficos de barras, nos quedamos con el promedio y también plotamos el desvío estandar. En el resto de los gráficos siempre nos quedamos con el mínimo valor obtenido, ya que creemos que este representa una corrida donde el procesador intervino lo menos posible.

## 2. Smalltiles

Este filtro consiste en replicar 4 veces la imagen original achicada. De esta manera, si enumeramos los píxeles a partir del 0, siempre estaremos utilizando los píxeles de número par de la imagen original para generar las 4 más pequeñas.

### 2.1. Código C

En el código de C recorreremos el equivalente a una de las 4 fotos pequeñas. En el píxel de la posición  $(i,j)$  guardamos el contenido del de la posición  $(2*i,2*j)$  en la imagen original. A la vez cargamos este contenido en las otras 3 imágenes.

A continuación mostramos el pseudocódigo de Smalltiles:

---

#### Algorithm 1 Smalltiles

---

```

function SMALLTILES(src: *unsigned char, dst: *unsigned char, cols: int, filas: int, srcRowSize: int, dstRowSize: int)
    unsigned char (*srcMatrix)[srcRowSize] = (unsigned char(*)[srcRowSize]) src
    unsigned char (*dstMatrix)[dstRowSize] = (unsigned char(*)[dstRowSize]) dst
    int ancho ← col/2
    int largo ← filas/2
    for f ← 0 .. largo - 1 do
        for c ← 0 .. ancho - 1 do
            bgrat * ps ← (bgrat*) & srcMatrix[f][c * 4]
            for i ← 0 .. 1 do
                bgrat * pd ← (bgrat*) & dstMatrix[f][(c + ancho * i) * 4]
                (pd → b) ← (ps → b)
                (pd → g) ← (ps → g)
                (pd → r) ← (ps → r)
                (pd → a) ← (ps → a)
            for i ← 0 .. 1 do
                bgrat * pd ← (bgrat*) & dstMatrix[f + largo][c * 4]
                (pd → b) ← (ps → b)
                (pd → g) ← (ps → g)
                (pd → r) ← (ps → r)
                (pd → a) ← (ps → a)

```

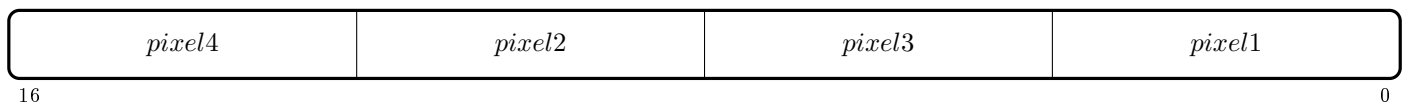
---

### 2.2. Código ASM

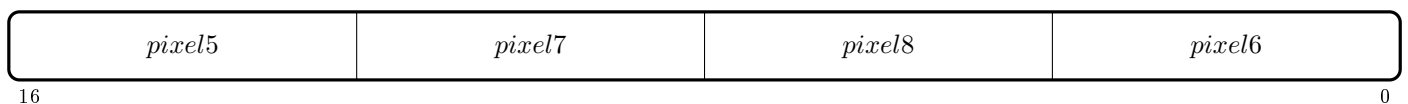
En la implementación de ASM recorreremos las filas de la imagen; tenemos dos ciclos, el interno recorre una fila iterando sobre sus columnas, y el externo itera sobre todas las filas de la imagen.

En el ciclo interno cargamos 8 píxeles, nos encargamos de ubicarlos en un registro xmmi a nuestra conveniencia utilizando **pshufd**, **psrldq**, **pslldq** y **paddb**.

En un principio cargamos los primeros 4 píxeles en **xmm1** y luego **pshud xmm1, xmm1, 0xd8. xmm1**:

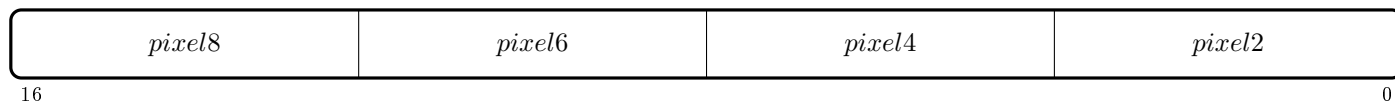


Luego cargamos los siguientes 4 píxeles en **xmm10** y también **pshud xmm10, xmm10, 0x2d. xmm10**:



Luego shifteamos convenientemente y sumamos los xmmi.

**xmm10:**



Ya tenemos en **xmm10** la información tal cual queremos guardarla, ahora simplemente la guardamos en memoria manipulando algunos índices para cargarla en los 4 cuadrantes de la imagen destino.

## 2.3. Experimentación

### 2.3.1. Idea

Nuestra implementación de ASM consiste en un ciclo que itera sobre las filas, por ende tuvimos la idea de experimentar sobre eso. Es decir, la cantidad de filas es un factor clave que influye bastante en la cantidad de ciclos de ejecución del filtro. Entonces nuestro experimento se basó en analizar los distintos tiempos de ejecución de imágenes con igual cantidad de píxeles totales, pero con distinto ancho y largo.

### 2.3.2. Hipótesis

Al comparar dos imágenes donde el ancho de una es la altura de la otra y viceversa, la aplicación del filtro a la imagen con menor altura tomaría menos ciclos. Cuanto más cercanos sean los valores de altura y ancho, menos varía la cantidad de ciclos.

### 2.3.3. Resultados

Efectivamente eso es lo que podemos observar en el siguiente gráfico. La cantidad de ciclos varía más en las imágenes de 200x1600 y 1600x200, donde podemos observar una diferencia de más de 4 millones de ciclos. En cambio en el par de imágenes de 640x500 y 500x640 la diferencia es de tan solo **1 millón y medio** (exactamente **155981** ciclos). Estos experimentos se realizaron con la misma imagen rotada y sin rotar, aunque en el caso de este filtro, las cualidades de los píxeles no modifican los resultados.

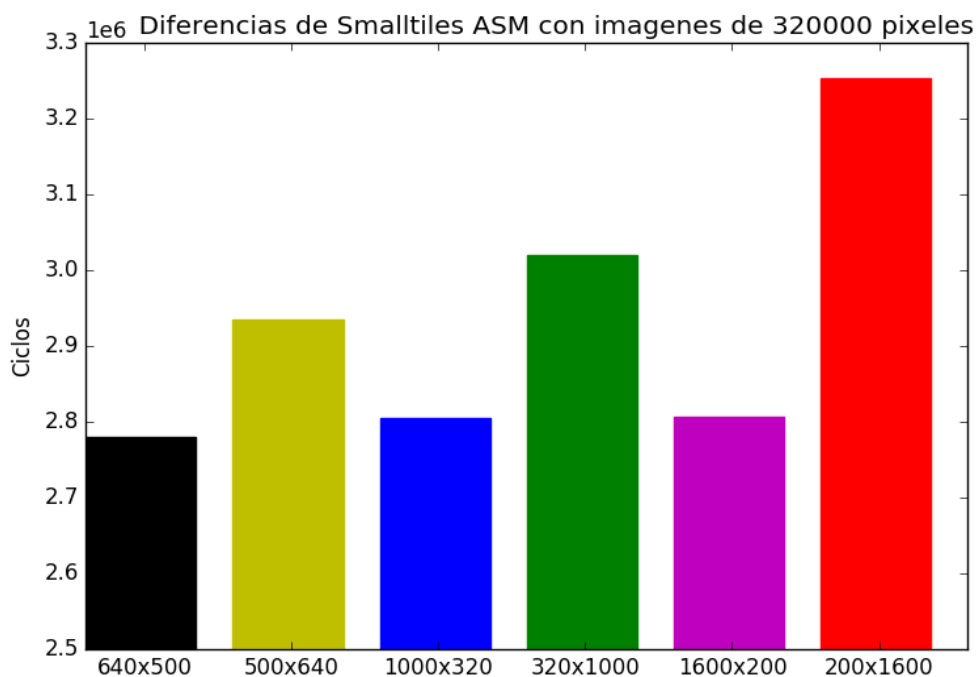


Figura 1: Gráfico de barras que da cuenta de los ciclos que implica cada corrida del filtro para imágenes de distintos tamaños con una determinada cantidad fija de píxeles.

A continuación adjuntamos el gráfico que compara la cantidad de ciclos que tarda en ejecutarse el filtro implementado en ASM y el implementado en C.

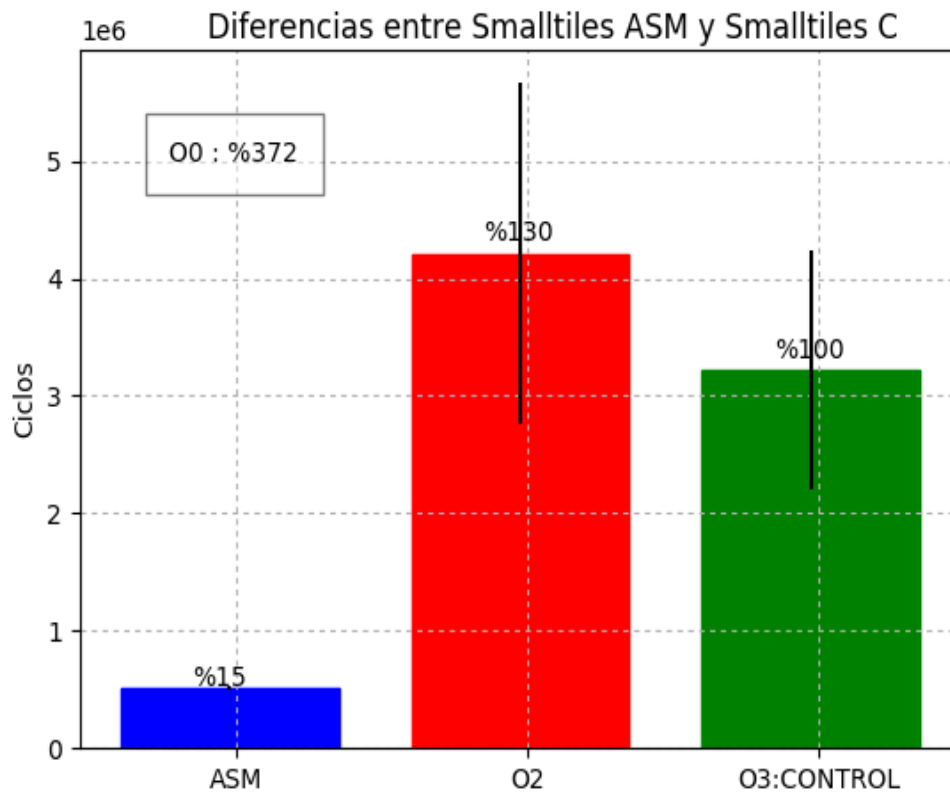


Figura 2: Comparación de las implementaciones y compilaciones con distintos niveles de optimización en cuanto a la rapidez para el procesamiento de una imagen.

### 3. Rotar canales

El filtro consiste en una rotación de canales en cada píxel, como bien indica su nombre. Y la rotación se da de esta manera:

$$\begin{aligned} \mathbf{R} &\longrightarrow \mathbf{G} \\ \mathbf{G} &\longrightarrow \mathbf{B} \\ \mathbf{B} &\longrightarrow \mathbf{R} \end{aligned}$$

#### 3.1. Código C

En el código de C recorreremos la matriz iterando sus filas y columnas y modificando un píxel a la vez. A continuación presentamos su pseudocódigo

---

##### Algorithm 2 Rotar

---

```
function ROTAR(src: *unsigned char, dst: *unsigned char, cols: int, filas: int, srcRowSize: int, dstRowSize: int)
    unsigned char (*srcMatrix)[srcRowSize] = (unsigned char(*)[srcRowSize]) src
    unsigned char (*dstMatrix)[dstRowSize] = (unsigned char(*)[dstRowSize]) dst
    for f ← 0 .. filas − 1 do
        for c ← 0 .. cols − 1 do
            bgrat * ps ← (bgrat*) & srcMatrix[f][c * 4]
            bgrat * pd ← (bgrat*) & dstMatrix[f][c * 4]
            pd → b ← ps → g
            pd → g ← ps → r
            pd → r ← ps → b
            pd → a ← ps → a
```

---

#### 3.2. Código ASM

El código de ASM recorre la matriz de la misma manera que la recorre el código de C, pero procesa 4 píxeles por iteración.

Las instrucciones propias de SIMD aceleran mucho el proceso y aportan mejoras considerables incluso al compararlo con un código de ASM sin utilizarlas.

En cada ciclo se cargan 4 píxeles en xmm0 y se aplica **psrshufb xmm0, xmm3**, donde este es:

0f	0c	0e	0d	0b	08	0a	09	07	04	06	05	03	00	02	01
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

16

0

Luego se guarda en memoria en la imagen destino, se incrementa el contador en 4 (procesamos 4 píxeles) y los respectivos punteros a las imagenes en 16 (4 píxeles ocupan 16 bytes).

#### 3.3. Experimentación

##### 3.3.1. Idea

Para la experimentación con Rotar notamos la facilidad con la que se codea y procesa el filtro gracias a las instrucciones de SIMD, por ello quisimos ver si, además, generaban alguna optimización sobre un código sin estas herramientas.

Luego comparamos la eficiencia de ASM contra el código de C y distintas optimizaciones de este.

##### 3.3.2. Hipótesis

La hipótesis que tuvimos sobre la primera idea era que efectivamente íbamos a notar un cambio ya que íbamos a estar procesando la mitad de píxeles por iteración. También suponíamos que iba a ser más complicado manipular los píxeles al no contar con instrucciones como **shuffle**, **unpacked**, **psrlx** y **psllx**.

### 3.3.3. Resultados

Efectivamente vimos una notoria diferencia entre el código en ASM con y sin SIMD, más aún, el código optimizado de C se encuentra casi a la par. **EXPLICACION DEL CODIGO SIN SIMD!!!!**

A continuación adjuntamos un gráfico en el que se puede apreciar dicha diferencia.

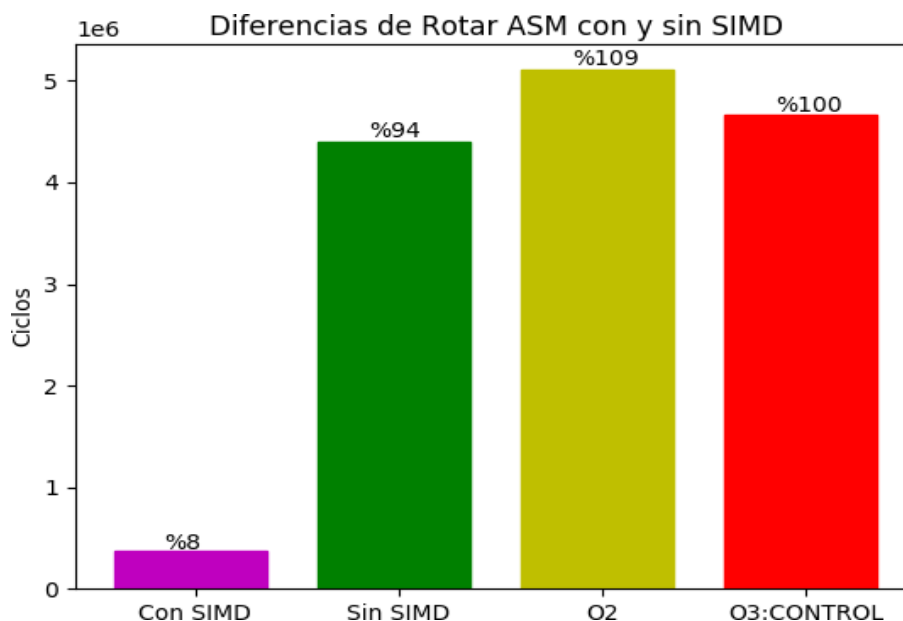


Figura 3: Gráfico que compara las velocidades de las ejecuciones de las implementaciones con y sin el uso de instrucciones SSE, y también con optimizaciones de la implementación en C.

A continuación mostramos un gráfico (armado de igual manera que el anterior) donde mostramos los tiempos de ejecución del filtro implementado en ASM y en C con distintas optimizaciones.

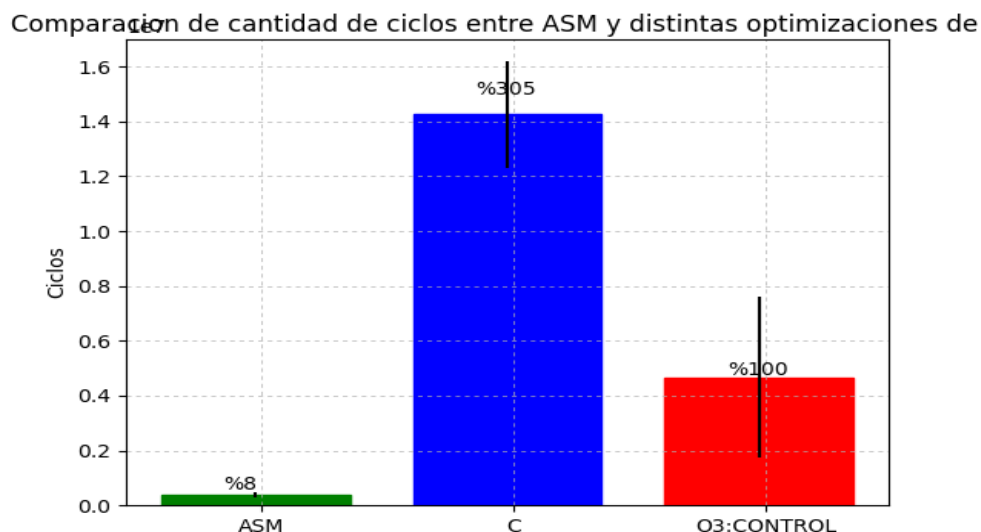


Figura 4: Diferencias en cantidad de ciclos para la ejecución del filtro con la implementación en ASM y la de C con sus optimizaciones.



## 4. Pixelar

Este filtro consiste en tomar bloques de 2x2 píxeles y asignarles a estos el promedio del bloque. De esta manera se disminuye la calidad de la imagen.

### 4.1. Código C

El código de C consiste en recorrer la imagen de a dos filas y dos columnas por iteración. En cada iteración se calcula el promedio de los canales de los píxeles.

A continuación adjuntamos el pseudocódigo.

---

#### Algorithm 3 Promedio

---

```
function PROMEDIO(a : unsigned char, b : unsigned char, c : unsigned char, d : unsigned char)
→ float
    float af ← a
    float bf ← b
    float cf ← c
    float df ← d
    return (af/4 + bf/4 + cf/4 + df/4)
```

---



---

#### Algorithm 4 Pixelar

---

```
function PIXELAR(src: *unsigned char, dst: *unsigned char, cols: int, filas: int, srcRowSize: int, dstRowSize:
int)
    unsigned char (*srcMatrix)[srcRowSize] = (unsigned char(*)[srcRowSize]) src
    unsigned char (*dstMatrix)[dstRowSize] = (unsigned char(*)[dstRowSize]) dst
    for f ← 0 .. filas - 1; f += 2 do
        for c ← 0 .. cols - 1; c += 2 do
            bgrat * ps1 ← (bgrat*) & srcMatrix[f][c * 4]
            bgrat * pd1 ← (bgrat*) & dstMatrix[f][c * 4]
            bgrat * ps2 ← (bgrat*) & srcMatrix[f + 1][c * 4]
            bgrat * pd2 ← (bgrat*) & dstMatrix[f + 1][c * 4]
            bgrat * ps3 ← (bgrat*) & srcMatrix[f + 1][(c + 1) * 4]
            bgrat * pd3 ← (bgrat*) & dstMatrix[f + 1][(c + 1) * 4]
            bgrat * ps4 ← (bgrat*) & srcMatrix[f][(c + 1) * 4]
            bgrat * pd4 ← (bgrat*) & dstMatrix[f][(c + 1) * 4]
            k ← 0.5
            b ← PROMEDIO(ps → b, ps2 → b, ps3 → b, ps4 → b)
            g ← PROMEDIO(ps → g, ps2 → g, ps3 → g, ps4 → g)
            r ← PROMEDIO(ps → r, ps2 → r, ps3 → r, ps4 → r)
            a ← PROMEDIO(ps → a, ps2 → a, ps3 → a, ps4 → a)
            for i ← 1 .. 4 do
                (pdi → b) ← b
                (pdi → g) ← g
                (pdi → r) ← r
                (pdi → a) ← a
```

---

### 4.2. Código ASM

El código en ASM consiste en una unión de ciclos, uno exterior para iterar sobre las filas y otro interior para iterar sobre los elementos de la misma (columnas). Lo que hace este código es en cada iteración, lee de memoria dos veces, empezando en la misma columna pero de dos filas diferentes. De esta forma levanta como una matriz de 8 elementos (4x2). Lo que se puede obtener de esta información va a servir para sobrecribir 8 píxeles.

El ciclo luego procede en extender los signos de cada byte, luego guardar la sumatoria de los elementos 1,2,6,7 (mirándolo como una matriz de 4x2), y la sumatoria de 4,5,8,9 en dos registros aparte, calcular el promedio de

esto, o sea dividirlos por cuatro. Y luego a esta información la volvemos a agrupar juntas en un mismo registro que se vería de esta manera:

**XMM:**

$Pp1_a$	$Pp1_r$	$Pp1_g$	$Pp1_b$	$Pp1_a$	$Pp1_r$	$Pp1_g$	$Pp1_b$	$Pp2_a$	$Pp2_r$	$Pp2_g$	$Pp2_b$	$Pp2_a$	$Pp2_r$	$Pp2_e$	$Pp2_b$
---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------	---------

16

0

$Pp1$  : promedio píxeles 1,2,6,7.

$Pp2$  : promedio píxeles 4,5,8,9

Finalmente a este registro lo escribimos en la imagen destino, en los lugares de los píxeles que levantamos en la imagen src.

Por último movemos los current en columnas 4 píxeles que es la cantidad sobre esa fila q procesamos en cada iteración. Y si terminamos de procesar la fila, nos movemos dos filas, ya que vamos procesando dos juntas a la vez.

### 4.3. Experimentación 1

#### 4.3.1. Idea

Comparamos la eficiencia del código en ASM, el de C y el de C optimizado.

#### 4.3.2. Resultados

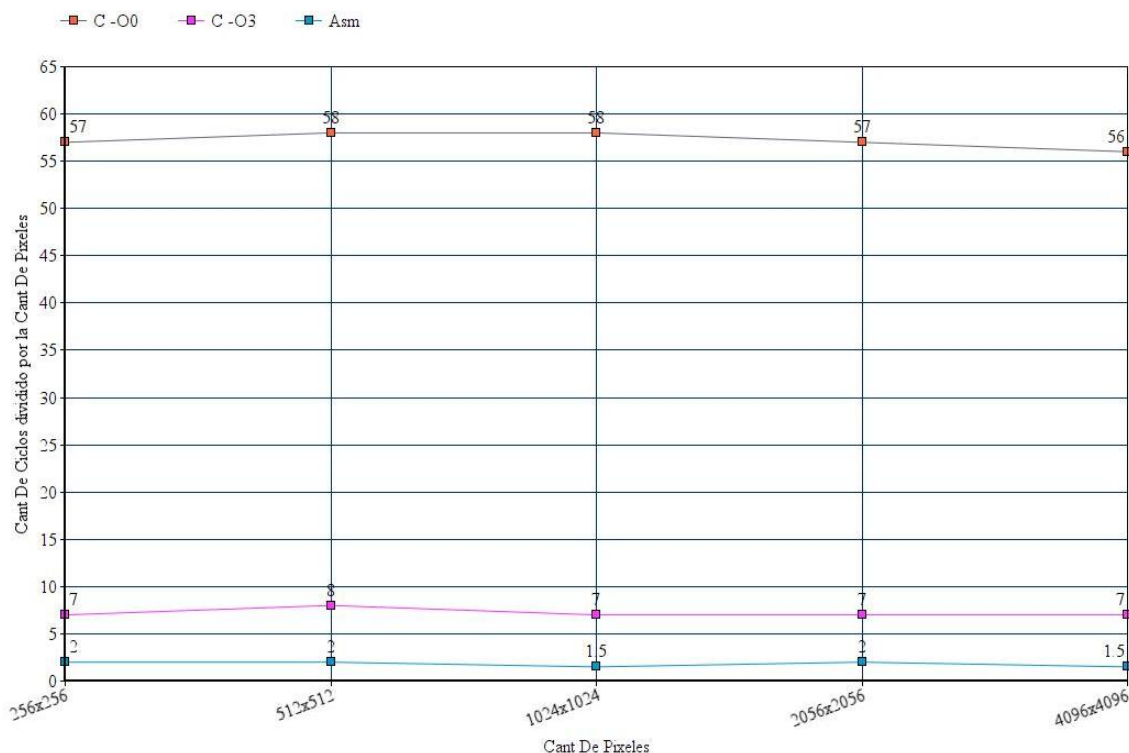


Figura 5: Comparación  $\frac{\text{ciclos totales}}{\text{cant. de píxeles}}$  - diferentes implementaciones de C y ASM

### 4.4. Experimentación 2

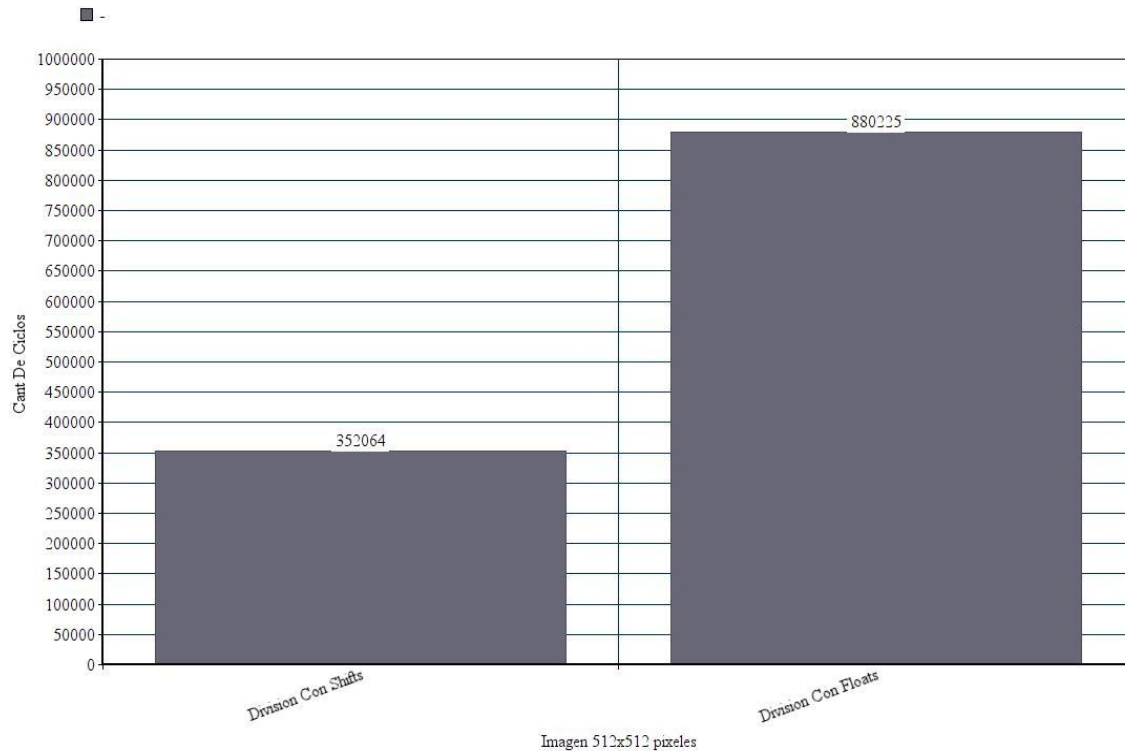
#### 4.4.1. Idea

Comparamos la eficiencia entre un código que utiliza instrucciones como shift, para dividir los píxeles, contra otro que usa las instrucciones para dividir floats, que implican además tener que hacer un traspaso de formato.

#### 4.4.2. Hipótesis

Nuestra idea es que va a ser mucho más eficiente el código donde utilizamos las instrucciones de shift para dividir los números porque son operaciones básicas que se encarga el registro de hacerlas, en cambio **el pasar a float, dividir luego y volver a pasar a float**, son todas operaciones sin implementación primitiva (por esto nos referimos a que están compuestas de varias otras instrucciones) y esto da lugar a que la ejecución de cada una demore varios ciclos más que un simple shift.

#### 4.4.3. Resultados



#### 4.4.4. Conclusión

Definitivamente el código en el cual dividimos con shifts fue mucho más eficiente por lo ya explicado en la hipótesis.

## 5. Combinar

Este filtro consiste en realizar una combinación de dos imágenes que dependa de un número real  $\alpha$  entre 0 y 255.

Cabe destacar que este filtro permite reutilizar cálculos debido a dos factores. Uno de ellos es la forma que tiene cada píxel generado en la imagen resultante, que consiste en que si se tienen una imagen  $A$  y una imagen  $B$  de tamaño  $m \times n$  entonces el píxel de la imagen generada  $I_{AB}^{i,j}$  se calcula de la siguiente forma:

$$I_{AB}^{i,j} = \frac{\alpha * (I_A^{i,j} - I_B^{i,j})}{255,0} + I_B^{i,j}$$

El otro factor es que nuestro filtro está optimizado para casos en los que la imagen  $B$  es el reflejo vertical de la imagen  $A$ . Es decir que se da que  $I_A^{i,j} = I_B^{i,n-j+1}$  como se puede apreciar en la siguiente figura.

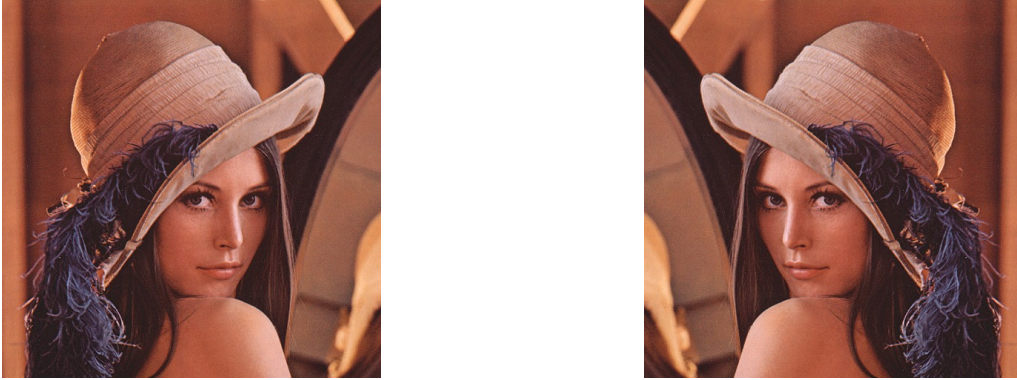


Figura 6: Se muestra la imagen  $A$  del lado izquierdo con la imagen  $B$ , el reflejo vertical de  $A$ , en la parte derecha.

Entonces se tiene que

$$\begin{aligned} I_{AB}^{i,j} &= \frac{\alpha * (I_A^{i,j} - I_B^{i,j})}{255,0} + I_B^{i,j} \text{ por la fórmula del filtro} \\ &= \frac{\alpha * (I_A^{i,j} - I_A^{i,n-j+1})}{255,0} + I_A^{i,n-j+1} \text{ dado que } I_A^{i,j} = I_B^{i,n-j+1} \end{aligned} \quad (1)$$

y análogamente

$$\begin{aligned} I_{AB}^{i,n-j+1} &= \frac{\alpha * (I_A^{i,n-j+1} - I_B^{i,n-j+1})}{255,0} + I_B^{i,n-j+1} \\ &= \frac{\alpha * (I_A^{i,n-j+1} - I_A^{i,j})}{255,0} + I_A^{i,j} \end{aligned} \quad (2)$$

Así, se obtiene

$$\begin{aligned} I_{AB}^{i,n-j+1} &= -1 * \frac{\alpha * [-1 * (I_A^{i,n-j+1} - I_A^{i,j})]}{255,0} - I_A^{i,n-j+1} + I_A^{i,n-j+1} + I_A^{i,j} \\ &= -1 * \left[ \frac{\alpha * (I_A^{i,j} - I_A^{i,n-j+1})}{255,0} + I_A^{i,n-j+1} \right] + I_A^{i,n-j+1} + I_A^{i,j} \\ &= -1 * (I_{AB}^{i,j} - I_A^{i,n-j+1}) + I_A^{i,j} \text{ usando la igualdad de (2)} \end{aligned} \quad (3)$$

Estos cálculos muestran que luego de hacer el procesamiento para generar un píxel de la parte izquierda de la imagen resultante se puede obtener el píxel que corresponde a la mitad derecha con pocos cálculos más. Más aún, si se denomina

$$P = \frac{\alpha * (I_A^{i,j} - I_A^{i,n-j+1})}{255,0}$$

se consigue

$$I_{AB}^{i,j} = P + I_A^{i,n-j+1}$$

y

$$I_{AB}^{i,n-j+1} = -P + I_A^{i,j}$$

dando lugar a menos cálculos necesarios.

### 5.1. Código C

A continuación se presenta el pseudocódigo detallado del filtro.

Clamp es una función que controla la saturación.

---

#### Algorithm 5 Clamp

---

```

function CLAMP(pixel : float) → float
  if pixel < 0,0 then
    return 0,0
  else
    if pixel > 255,0 then
      return 255,0
    else
      return pixel

```

---

Combine realiza los cálculos correspondientes determinados por el filtro.

---

#### Algorithm 6 Combine

---

```

function COMBINE(a : unsigned char, b : unsigned char,  $\alpha$  : float) → float
  float af ← a
  float bf ← b
  return  $\frac{\alpha * (af - bf)}{255,0} + bf$ 

```

---

Combinar llama a las dos funciones ya explicadas y obtiene 2 píxeles con los que operar para luego dejar el resultado en la imagen destino.

---

#### Algorithm 7 Combinar

---

```

function COMBINAR(src: *unsigned char, dst: *unsigned char, cols: int, filas: int, srcRowSize: int, dstRowSize: int,  $\alpha$ : float)
  unsigned char (*srcMatrix)[srcRowSize] = (unsigned char(*)[srcRowSize]) src
  unsigned char (*dstMatrix)[dstRowSize] = (unsigned char(*)[dstRowSize]) dst
  for f ← 0 .. filas - 1 do
    for c ← 0 .. cols - 1 do
      bgrat * psa ← (bgrat*) & srcMatrix[f][c * 4]
      bgrat * psb ← (bgrat*) & srcMatrix[f][(cols - c - 1) * 4]
      bgrat * pd ← (bgrat*) & dstMatrix[f][c * 4]
      pd->b ← CLAMP(COMBINE(psa->b, psb->b,  $\alpha$ ))
      pd->g ← CLAMP(COMBINE(psa->g, psb->g,  $\alpha$ ))
      pd->r ← CLAMP(COMBINE(psa->r, psb->r,  $\alpha$ ))
      pd->a ← CLAMP(COMBINE(psa->a, psb->a,  $\alpha$ ))

```

---

### 5.2. Código ASM

Dado que cada píxel tiene 4 bytes y los registros XMM tienen 16 bytes, se levantan 4 píxeles contiguos desde la posición  $i, j$  y otros 4 píxeles desde la  $i, n - j + 1$  en otro registro.

Se mostrará cómo se genera el píxel  $I_{i,j}$  de la imagen resultante.

Se levantan 4 píxeles de la mitad izquierda en el registro XMM1:

**XMM1:**

$A_{j+3}$	$R_{j+3}$	$G_{j+3}$	$B_{j+3}$	$A_{j+2}$	$R_{j+2}$	$G_{j+2}$	$B_{j+2}$	$A_{j+1}$	$R_{j+1}$	$G_{j+1}$	$B_{j+1}$	$A_j$	$R_j$	$G_j$	$B_j$
16															
0															

y 4 en espejo de la mitad derecha en XMM3:

**XMM3:**

$A_{n-j+4}$	$R_{n-j+4}$	$G_{n-j+4}$	$B_{n-j+4}$	$A_{n-j+3}$	$R_{n-j+3}$	$G_{n-j+3}$	$B_{n-j+3}$	$A_{n-j+2}$	$R_{n-j+2}$	$G_{n-j+2}$	$B_{n-j+2}$	$A_{n-j+1}$	$R_{n-j+1}$	$G_{n-j+1}$	$B_{n-j+1}$
16															
0															

Teniendo

**XMM9:**

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16															
0															

se ejecuta la instrucción **punpcklbw xmm1, xmm9** que da como resultado

**XMM1:**

0	$A_{j+1}$	0	$R_{j+1}$	0	$G_{j+1}$	0	$B_{j+1}$	0	$A_j$	0	$R_j$	0	$G_j$	0	$B_j$
16															
0															

y luego **punpcklwd xmm1, xmm9** para seguir desempaquetando. De esta forma quedan 4 bytes para cada componente del píxel y se podrán convertir a floats para mayor precisión en los cálculos que implican el  $\alpha$ .

**XMM1:**

0	0	0	$A_j$	0	0	0	$R_j$	0	0	0	$G_j$	0	0	0	$B_j$
16															
0															

Se copió el contenido de XMM1 a XMM10 y con el registro que contenía al píxel  $n - j + 4$ , obtenido de forma análoga a como se obtuvo el píxel  $j$

**XMM8:**

0	0	0	$A_{n-j+4}$	0	0	0	$R_{n-j+4}$	0	0	0	$G_{n-j+4}$	0	0	0	$B_{n-j+4}$
16															
0															

se realizaron las restas entre componentes (**psubd xmm10, xmm8**) antes de convertir a float el registro XMM10 (**cvttdq2ps xmm10, xmm10**) dado que realizar la resta con float es una operación mucho más costosa que con enteros y además no había chances de perder precisión con la resta.

**XMM10:**

$A_j - A_{n-j+4}$	$R_j - R_{n-j+4}$	$G_j - G_{n-j+4}$	$B_j - B_{n-j+4}$
16			
0			

Se multiplicó por  $\alpha$  al registro XMM10 (**mulps xmm10, xmm0**) y luego se dividió por 255,0 (**divps xmm10, xmm14**) obteniéndose

**XMM10:**

$\frac{\alpha * (A_j - A_{n-j+4})}{255,0}$	$\frac{\alpha * (R_j - R_{n-j+4})}{255,0}$	$\frac{\alpha * (G_j - G_{n-j+4})}{255,0}$	$\frac{\alpha * (B_j - B_{n-j+4})}{255,0}$
16			
0			

Para esto previamente, fuera del ciclo, se había movido el  $\alpha$  que había llegado en los últimos 4 bytes de XMM0 a las otras 3 double words del registro con la instrucción **pshufd xmm0, xmm0, 00000000b** y se había puesto en XMM14 un 255.0 en cada double word declarando **mascara255: dd 255.0, 255.0, 255.0, 255.0** en **section .rodata** y luego haciendo **movdqu xmm14, [mascara255]**. De esta forma, al realizar estas operaciones fuera del ciclo, se minimizan los accesos a memoria y se evita repetir operaciones innecesarias.

Convirtiendo XMM8 a float con **cvtqd2ps xmm8, xmm8** y sumándolo a XMM10 con **addps xmm10, xmm8** se obtuvo

**XMM10:**

$\frac{\alpha * (A_j - A_{n-j+4})}{255,0} + A_{n-j+4}$	$\frac{\alpha * (R_j - R_{n-j+4})}{255,0} + R_{n-j+4}$	$\frac{\alpha * (G_j - G_{n-j+4})}{255,0} + G_{n-j+4}$	$\frac{\alpha * (B_j - B_{n-j+4})}{255,0} + B_{n-j+4}$
16			0

Antes de finalizar el procesamiento, se realizó **movups xmm15, xmm10** y **subps xmm15, xmm8** para conservar en XMM15 lo que en el desarrollo de cuentas apareció como P al explicar cómo se podían reutilizar cálculos.

Por último en el procesamiento de la parte izquierda se convirtió a entero XMM10 con **cvtps2dq xmm10, xmm10** y se empaquetó con los resultados de los demás píxeles usando instrucciones como **packusdw xmm11, xmm10** y **packusdw xmm13, xmm12** para pasar de double word a word,

**XMM11:**

$F_{A_j}$	$F_{R_j}$	$F_{G_j}$	$F_{B_j}$	$F_{A_{j+1}}$	$F_{R_{j+1}}$	$F_{G_{j+1}}$	$F_{B_{j+1}}$
16							0

luego **packuswb xmm13, xmm11** para pasar de word a byte,

**XMM11:**

$F_{A_j}$	$F_{R_j}$	$F_{G_j}$	$F_{B_j}$	$F_{A_{j+1}}$	$F_{R_{j+1}}$	$F_{G_{j+1}}$	$F_{B_{j+1}}$	$F_{A_{j+2}}$	$F_{R_{j+2}}$	$F_{G_{j+2}}$	$F_{B_{j+2}}$	$F_{A_{j+3}}$	$F_{R_{j+3}}$	$F_{G_{j+3}}$	$F_{B_{j+3}}$	
16																0

y finalmente **pshufd xmm11, xmm13, 0x1b** quedando todo listo para mover el contenido del registro a donde correspondiera.

**XMM11:**

$F_{A_{j+3}}$	$F_{R_{j+3}}$	$F_{G_{j+3}}$	$F_{B_{j+3}}$	$F_{A_{j+2}}$	$F_{R_{j+2}}$	$F_{G_{j+2}}$	$F_{B_{j+2}}$	$F_{A_{j+1}}$	$F_{R_{j+1}}$	$F_{G_{j+1}}$	$F_{B_{j+1}}$	$F_{A_j}$	$F_{R_j}$	$F_{G_j}$	$F_{B_j}$	
16																0

Para finalizar la parte derecha se multiplicaron todos los valores de las double words de XMM15 por -1 (**mulps xmm15, [menos1]** estando **menos1** definido en **rodata** como **menos1: dd -1.0, -1.0, -1.0, -1.0**) y se prosigió con las conversiones y sumas como anteriormente.

## 5.3. Experimentación

### 5.3.1. Hipótesis

Nuestra hipótesis era que la implementación en ASM iba a ser mucho más veloz que la de C con el menor nivel de optimización al compilar e incluso que las de C con mayores niveles de optimización.

Además creíamos que las ejecuciones con la implementación de C iban a presentar un porcentaje de ciclos de clock debidos accesos a memoria mucho mayor que la implementación de ASM.

### 5.3.2. Idea

Como forma de verificar esto pensamos medir la cantidad de ciclos de clock insumidos en cada caso para la ejecución del filtro. Se realizaron 1000 mediciones y se realizó un gráfico de barras con los promedios obtenidos de dividir cada total de ciclos por la cantidad de iteraciones.

Por otro lado, se midieron los ciclos de clock relacionados a accesos a memoria con la implementación de ASM y con la de C y se realizaron gráficos de torta para mostrar los porcentajes con respecto al total de ciclos insumidos.

### 5.3.3. Resultados

En cuanto a las cantidades totales de ciclos de clock se obtuvieron los siguientes gráficos.

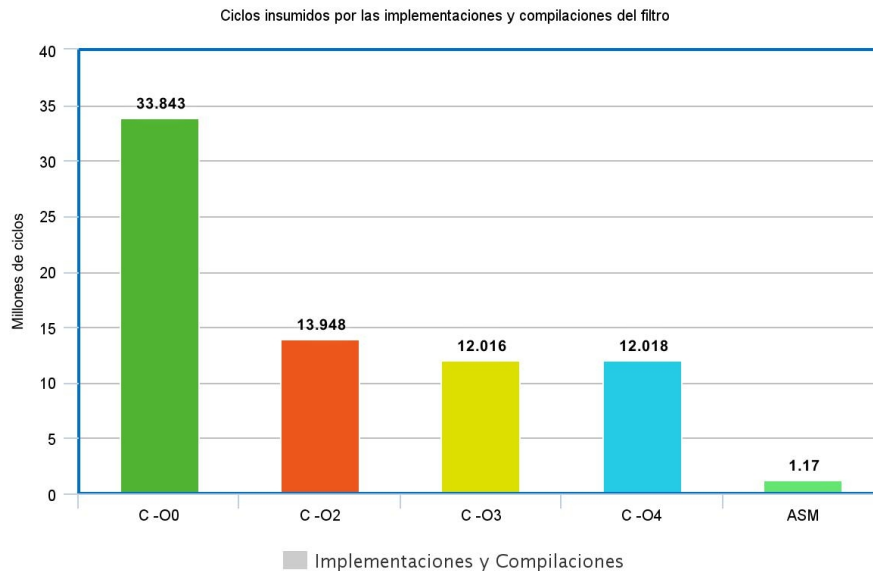


Figura 7: Gráfico de barras comparando las implementaciones y diversas optimizaciones.

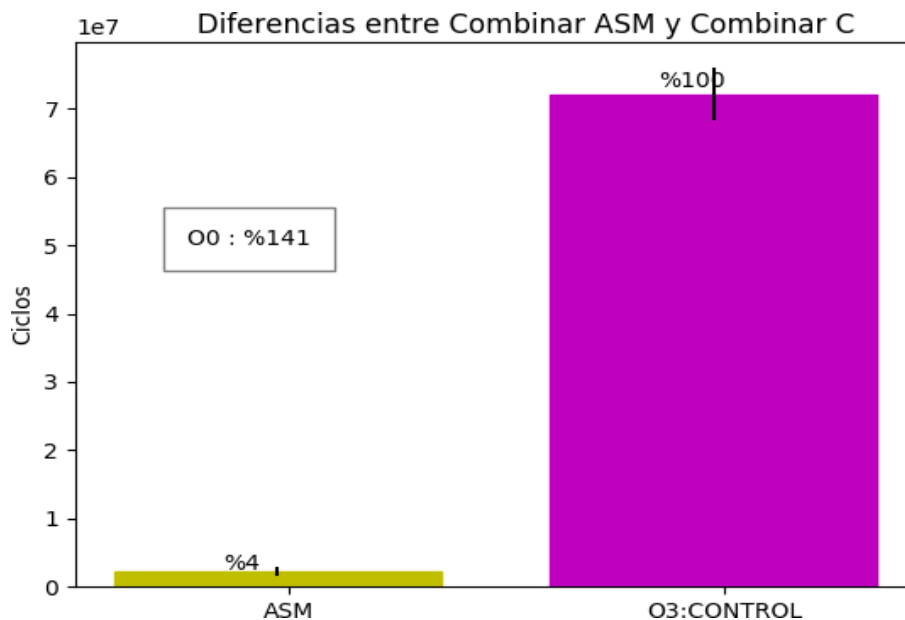


Figura 8: Gráfico de barras comparando las implementaciones y diversas optimizaciones.

Como se puede apreciar, la implementación de ASM es casi 29 veces más rápida que la menos optimizada de C y también supera ampliamente a las optimizadas, siendo más de 10 veces más rápida. Esto confirma nuestras suposiciones de la mejoría resultante del modelo de procesamiento SIMD.

Con respecto a lo planteado en cuanto a los accesos a memoria, también obtuvimos resultados esperados, visibles en la siguiente figura.



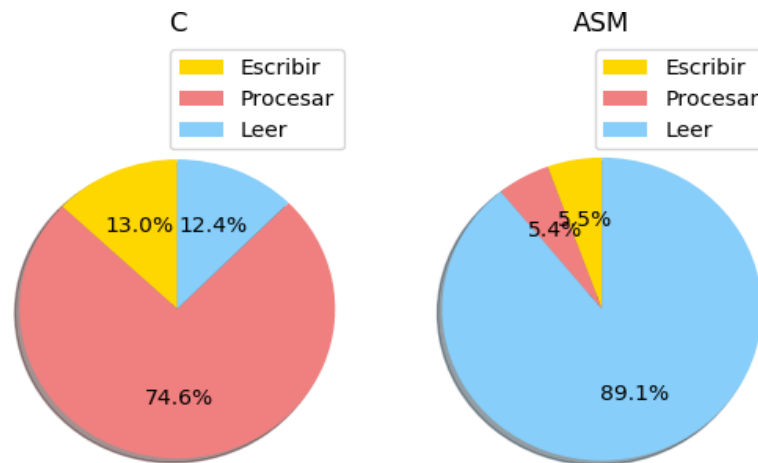


Figura 9: En estos gráficos se exponen las diferencias en las implementaciones en cuanto a la proporción de tiempo destinado a accesos a memoria y a procesamiento de la imagen.

A simple vista se observa que la implementación de ASM da como resultado que la mayor parte del tiempo de ejecución esté destinada al procesamiento de la imagen, al contrario de lo que ocurre con la implementación de C que resulta en un mayor consumo de ciclos por parte de los accesos a memoria.

Más aún, el cociente entre el procesamiento de píxeles y los accesos a memoria para la implementación de ASM da 5,5, es decir que cada 13 ciclos, 11 surgen del procesamiento y 2 de los accesos. En cambio, este cociente es 0,68 para la implementación en C, de modo que la diferencia entre ambas implementaciones en cuanto al uso de la memoria y del procesador es muy significativa.

## 6. Colorizar

### 6.1. Código C

El código C se trata de una conjunción de ciclos, el exterior que recorre desde la segunda fila hasta la ante última, y el interior que recorre desde la segunda columna hasta la última, dejando afuera a todos los bordes, tal como el enunciado pedía.

Luego en cada iteración del ciclo interior, que es donde se hacen las operaciones que modifican la imagen, lo que hacemos es crear un arreglo de unsigned chars, "res", que es donde guardamos los máximos de cada canal en comparación a todos los píxeles lindantes del píxel en el cual estemos parados (una matriz de 3x3).

- $\text{Res}[0] \leftarrow \text{MaximoLindantesAzul}$
- $\text{Res}[1] \leftarrow \text{MaximoLindantesVerde}$
- $\text{Res}[2] \leftarrow \text{MaximoLindantesRojo}$

Luego con estos tres valores calculamos el alpha correspondiente de cada canal por el cual vamos a multiplicar a cada uno. Y por último reescribimos el píxel final en la imagen src con cada canal multiplicado por dicho alpha.

### 6.2. Código ASM

El código en ASM se trata también de una unión de ciclos. El ciclo exterior recorre desde la segunda hasta la ante última fila, y el interior recorre las columnas desde la segunda hasta la ante última, pero saltando de a dos píxeles, que es la cantidad que procesamos simultáneamente con instrucciones SSE.

El ciclo interior consta varios pasos, el primero es calcular un registro en el que en las primeras dos DW guardamos los máximos valores de cada canal entre los píxeles lindantes, en sus posiciones respectivas. Se vería así:

#### ACLARACIONES:

La notación  $X_{MpN}$  es el valor máximo del Color X del el píxel N

Vamos a mostrar sólo los primero 8 bytes de los registros XMM, porque en la parte alta de cada uno hay información que vamos a descartar (Fruta).

#### XMM1:

$A_{Mp2}$	$R_{Mp2}$	$G_{Mp2}$	$B_{Mp2}$	$A_{Mp1}$	$R_{Mp1}$	$G_{Mp1}$	$B_{Mp1}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

16

0

Luego con esta información lo que hacemos es duplicar dicho registro y calcular el máximo de los máximos. Lo que hacemos para lograr esto es shiftear un byte a la derecha al registro duplicado, cosa de poder ir comparando entre ellos a los valores máximos de cada canal, y finalmente reproducimos el valor del máximo entre los máximos en todas las posiciones. El seguimiento de esta operación sería la siguiente: Vamos a utilizar para esto los registros xmm1, y xmm2

#### XMM2:

$A_{Mp2}$	$R_{Mp2}$	$G_{Mp2}$	$B_{Mp2}$	$A_{Mp1}$	$R_{Mp1}$	$G_{Mp1}$	$B_{Mp1}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

16

0

#### XMM1:

$A_{Mp2}$	$R_{Mp2}$	$G_{Mp2}$	$B_{Mp2}$	$A_{Mp1}$	$R_{Mp1}$	$G_{Mp1}$	$B_{Mp1}$
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

16

0

Shifteo un byte a la derecha xmm2 y hacemos pmaxub xmm1,xmm2 (máximo entre ambos)

**XMM2:**

$A_{Mp2}$	$R_{Mp2}$	$G_{Mp2}$	$B_{Mp2}$	$A_{Mp1}$	$R_{Mp1}$	$G_{Mp1}$	$B_{Mp1}$
16							0

**XMM1:**

<i>Fruta</i>	<i>Fruta</i>	$RG_{Mp2}$	<i>Fruta</i>	<i>Fruta</i>	<i>Fruta</i>	$RG_{Mp1}$	<i>Fruta</i>
16							0

Notar que ahora más casillas tienen información no relevante al código, es información que sera pisada prontamente

Luego volvemos a shiftear y repetir la operación pmaxub xmm1,xmm2 y queda:

**XMM2:**

$A_{Mp2}$	$R_{Mp2}$	$G_{Mp2}$	$B_{Mp2}$	$A_{Mp1}$	$R_{Mp1}$	$G_{Mp1}$	$B_{Mp1}$
16							0

**XMM1:**

<i>Fruta</i>	<i>Fruta</i>	<i>Fruta</i>	$RGB_{p2}$	<i>Fruta</i>	<i>Fruta</i>	<i>Fruta</i>	$RGB_{Mp1}$
16							0

Por último con la instrucción pshuf, dejamos en xmm1 un registro con los máximos de los máximos representado de esta forma:

**XMM1:**

$RGB_{p2}$	$RGB_{p2}$	$RGB_{p2}$	$RGB_{p2}$	$RGB_{Mp1}$	$RGB_{Mp1}$	$RGB_{Mp1}$	$RGB_{Mp1}$
16							0

En la segunda parte del código, lo que hacemos es distinguir al máximo entre los máximos. Para esto empezamos haciendo una comparación entre el registro XMM1, que tiene en todos sus valores al máximo valor del Píxel 1 y 2 como muestra la imagen anterior, y el XMM2 que tiene los máximos parciales. De esta manera, quedaría dentro del byte que representaría al color con el valor máximo una serie de sólo “FFFF”. Visualmente se puede ver, por ejemplo, sea el caso que el color con el valor máximo del píxel 2 sea el Azul , y el color con el valor máximo del píxel 1 sea el Rojo. El registro XMM1 se vería de la siguiente forma.

**XMM1:**

0	0	0	<i>FFFF</i>	0	<i>FFFF</i>	0	0
16							0

Desde este punto, lo que hacemos, primero es chequear el caso de que algun píxel tenga doble máximo, o sea que en dos bytes diferentes se hubiese impreso “FFFF”. Y esta solución es simplemente poner en ceros el byte con el color de menor prioridad.

Ahora nuestra siguiente operación es llamar a una función auxiliar donde convertimos los valores de este registro XMM1: si el byte contenía un cero, ahora va a contener un “-1” (“FFFF” en complemento a 2), y si el Byte contenía “FFFF” ahora va a contener un “1”. Se va a plasmar la información sobre el píxel 1 en el registro XMM0, y la del píxel 2 en XMM1.

Siguiendo el ejemplo anterior los registros quedarían de la siguiente forma:

**ACLARACIÓN:** Ahora los registros ahora van a ser representados completamente pero por sus valores de a doble word.

**XMM0:**

-1	1	-1	-1
----	---	----	----

16 0

**XMM1:**

-1	-1	-1	1
----	----	----	---

16 0

Ahora a estos registros los multiplicamos por el alpha pasado por parámetro, borramos el valor de Color Transparente y sumamos uno a todos los colores

**XMM0:**

1	$1 + \alpha$	$1 - \alpha$	$1 - \alpha$
---	--------------	--------------	--------------

16 0

**XMM1:**

1	$1 - \alpha$	$1 - \alpha$	$1 + \alpha$
---	--------------	--------------	--------------

16 0

Luego como último multiplicamos a cada valor por el valor respectivo del color que ocupa cada posición por lo que nos quedan los registros de la siguiente manera:

**XMM0:**

$A_{P1}$	$R_{P1} * (1 + \alpha)$	$G_{P1} * (1 - \alpha)$	$R_{P1} * (1 - \alpha)$
----------	-------------------------	-------------------------	-------------------------

16 0

**XMM1:**

$A_{P2}$	$R_{P2} * (1 - \alpha)$	$G_{P2} * (1 - \alpha)$	$R_{P2} * (1 + \alpha)$
----------	-------------------------	-------------------------	-------------------------

16 0

Los cuales son los valores ya de los píxeles resultantes. Lo siguiente simplemente es escribirlos correctamente en la imagen resultado, y listo.

## 6.3. Experimentación 1

### 6.3.1. Idea

Para el primer experimento, al igual que en el resto de los filtros, vamos a realizar la comparación de cantidad de ciclos por píxel procesado entre los códigos de C sin optimizar, C optimizado nivel 3, y ASM. En esto vamos a realizar tres comparaciones.

La primera trata de comparar entre imágenes cuadradas, cómo evoluciona el rendimiento con el tamaño de imagen que le pasemos y cómo se ven con respecto a los otros códigos.

La segunda comparación trata de comparar imágenes con la misma cantidad de píxeles, pero con diferentes dimensiones.

Por último comparamos rendimientos al tratar imágenes con el mismo color en cada píxel.

### 6.3.2. Hipótesis

En la primer comparación esperamos como siempre que el código de ASM sea el más óptimo, luego le siga el código en C optimizado, y por último el código de C sin optimizar.

En la segunda comparación, se espera que sea similar la cantidad de ciclos por píxel que se obtiene porque el ciclo no busca ninguna optimización respecto al ancho o alto de la imagen, sino que procesa por igual a todos los píxeles.

Y por último en el tercer experimento suponemos lo mismo, no buscamos ninguna optimización con respecto a los valores de los píxeles adyacentes, sino que procesamos por igual a cada píxel indistintamente de su contexto, por lo que esperamos que los valores sean constantes.

### 6.3.3. Resultados

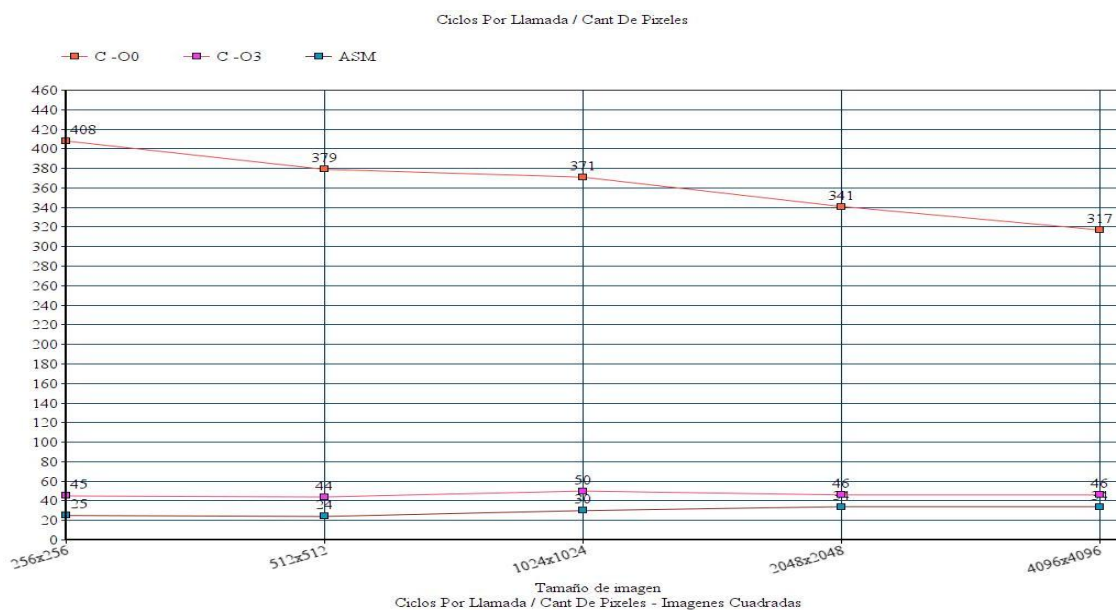


Figura 10: Comparación diversas implementaciones - Imágenes cuadradas

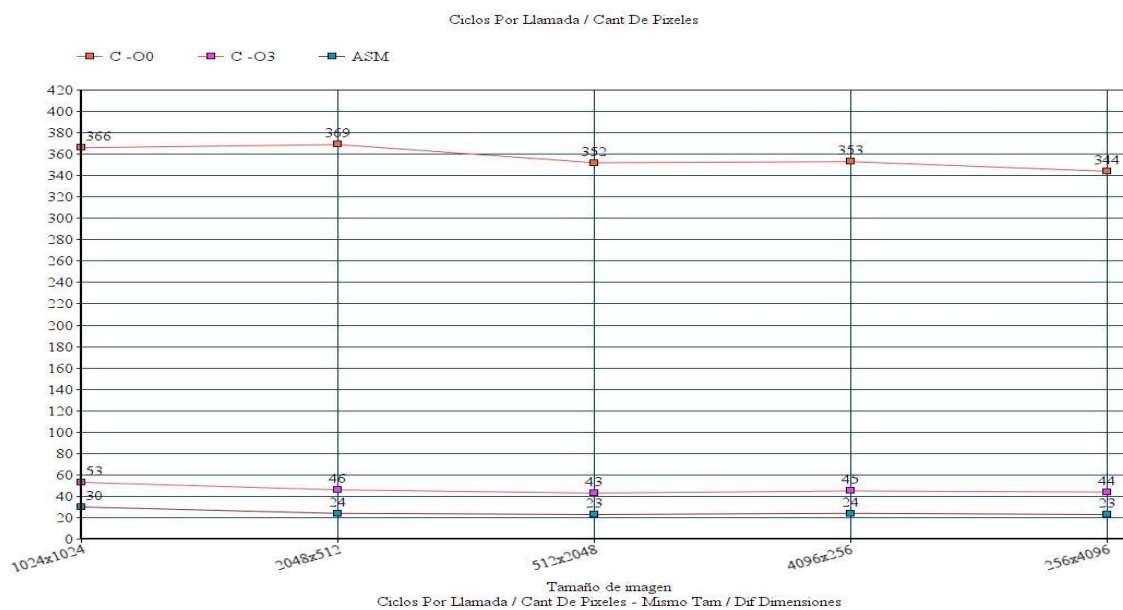


Figura 11: Comparación diversas implementaciones - Diferentes dimensiones

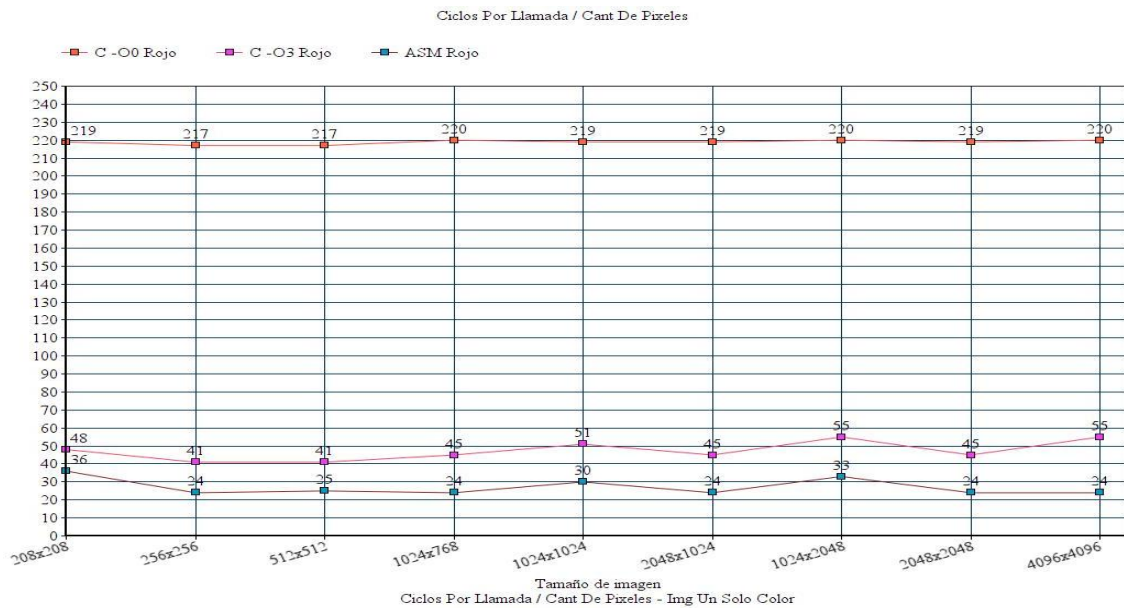


Figura 12: Comparación diversas implementaciones - Imágenes de un color

#### 6.3.4. Conclusión

La conclusión que podemos sacar de estos resultados es que, en primer lugar, como dijimos en la hipótesis, que se terminó corroborando, el código en C sin optimizar fue por lejos más lento que los otros dos. Y el código en C optimizado por más que mejoró muchísimo con respecto al anterior, siguió siendo más lento que el código en ASM.

Luego podemos ver que no nos confundimos tampoco en el momento de cambiar los tamaño de las imágenes pues los valores se mantuvieron casi constantes, justamente porque no se cambia la cantidad de operaciones que se hace respecto a las dimensiones de las imágenes, siempre se mantiene según la cantidad de píxeles.

Por último, nos llamó la atención el experimento de correr los códigos con imágenes de un solo color. Lo que pasó acá de extraño fue que las performance de los códigos en ASM y en C -O3 se mantuvieron iguales a los desarrollados con imágenes comunes, sin embargo el código en C -O0 demostró una gran mejora respecto a su performance, de la cual no podemos explicar precisamente el por qué, y de hecho esto también se ve en la primera imagen, donde para las imágenes de 4096x4096 el código en C -O0 mostró que mejoraba su rendimiento. Creemos que esto se debe a que justamente las imágenes que le pasamos tienen grandes secciones con píxeles del mismo color, porque son expansiones de una imagen de 512x512. Y se genera en estos fragmentos esta optimización que no podemos explicar sobre C -O0 en imágenes donde el color no cambia.

### 6.4. Experimentación 2

#### 6.4.1. Idea

El segundo experimento es un experimento más destructivo. Vamos a intentar molestar al jump predictor y ver cómo influye eso en la performance del programa.

El experimento en sí trata de agarrar un número arbitrario desde los datos de entrada y en diferentes puntos de parada, ejecutar un control de flujo if then else, tal que dependiendo de **ciertas condiciones random de este número arbitrario**, salte a diferentes secciones del código. Todo esto dispuesto de tal manera que sea transparente a la ejecución final del programa, es decir, si el código sigue el camino del if no cambia a si sigue el camino del else.

#### 6.4.2. Hipótesis

Nuestra hipótesis es que justamente el código no modificado va a ser bastante mejor que el código desarrollado con molestias para el jump predictor, porque en caso de funcionar estas, en cada ciclo estaríamos haciendo que flushee el pipeline, y que pierda muchos ciclos de clock extra.

### 6.4.3. Resultados

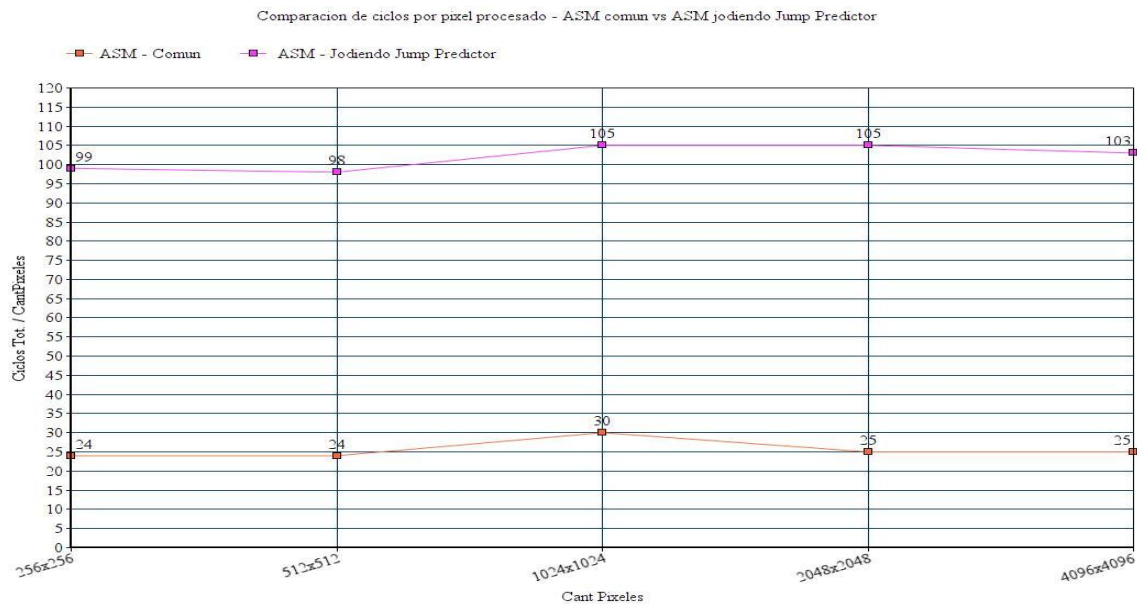


Figura 13: Comparación ASM comun vs ASM molestando Jump Predictor

### 6.4.4. Conclusión

Efectivamente los resultados fueron los esperados, por lo que nos llevamos como conclusión del experimento que influye mucho la arbitrariedad del flujo del **programa que se arma**. Si uno lo hace muy **aleatorio** como en este caso, por más que no afecte a la complejidad total el programa tarda significativamente más.

## 6.5. Experimento 3

### 6.5.1. Idea

En el tercer experimento vamos a ver en qué medida es mejor aplicar la técnica de unrolling sobre un código. Lo que planeamos es correr el código de colorizar sobre una imagen de 1024x1024 desenrollando el ciclo interior distinta cantidad de veces: vamos a empezar desenrollando 16, luego 32, y luego completamente.

### 6.5.2. Hipótesis

Lo que esperamos de este experimento es que el código desenrollado 16 y 32 veces, mantenga el nivel de performance con el que demuestra normalmente, o resultados un poco más cercanos al óptimo, y esto se debe a que en sí el código de colorizar lleva muchas instrucciones, por lo que los pasos que nos ahorramos al desenrollar el código no son significativos en el total del programa. Sin embargo en el caso de desenrollar todo el código, esperamos una mucha peor performance por parte del código debido a que el tamaño del código aumentaría enormemente su tamaño de modo que no entrara en la memoria caché y debiera empezar a hacer lecturas de memoria para leer las instrucciones.

### 6.5.3. Resultados

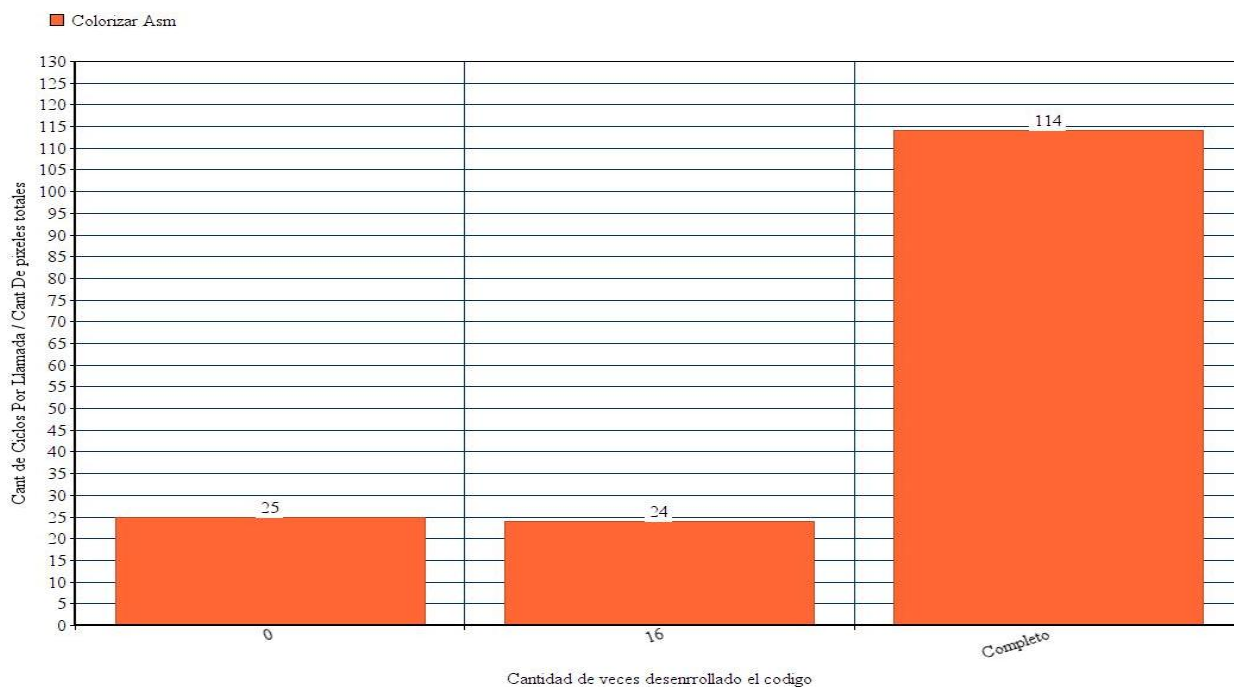


Figura 14: Comparación ASM común vs ASM unrolling 16 vs ASM Unrolling Completo

### 6.5.4. Conclusión

La conclusión que sacamos de este experimento es que la técnica de desenrollar código es efectiva para pequeños whiles, porque se eliminan algunas instrucciones innecesarias, sin embargo para un while de un tamaño mayor a lo usual, la optimización generada por esto se diluye hasta el punto de no notarse. Además de que efectivamente factores como la memoria caché son limitantes en esta situación, donde se da que los problemas de lectura de memoria dejan de ser parte de cómo es el algoritmo y pasa a ser a simplemente poder leerlo.