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

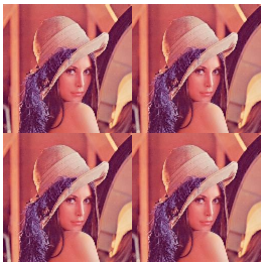
1. Introducción

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.


Imagen original


Smalltiles


Rotar


Pixelar


Combinar


Colorizar

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.

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 utiplizando los píxeles de número par de la imagen original para generar las 4 más pequeñas.

2.1. Código C

En el código de C recorremos 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 imagenes. A continuación mostramos el pseudocódigo de Smalltiles:

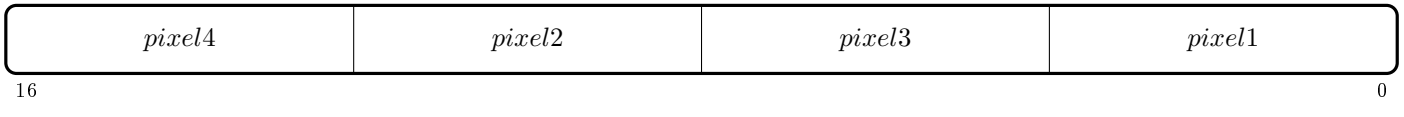
Algorithm 1 Smalltiles

function SMALLTILES(src: *unsigned char, dst: *unsigned char, cols: int, filas: int, srcRowSize: int, dstRowSize: int)

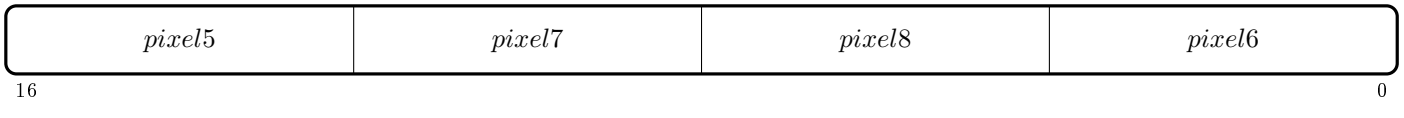
$unsigned\ char\ (*srcMatrix)[srcRowSize] = (unsigned\ char(*)[srcRowSize])\ src$
$unsigned\ char\ (*dstMatrix)[dstRowSize] = (unsigned\ char(*)[dstRowSize])\ dst$
int ancho $\leftarrow$ col/2
int largo $\leftarrow$ filas/2
for $f \leftarrow 0\ ..\ largo - 1$ **do**
  for $c \leftarrow 0\ ..\ ancho - 1$ **do**
    $bgra_t * p_s \leftarrow (bgra_t*)\ \&\ srcMatrix[f][c * 4]$
    for $i \leftarrow 0\ ..\ 1$ **do**
      $bgra_t * p_d \leftarrow (bgra_t*)\ \&dstMatrix[f][(c + ancho * i) * 4]$
      $(p_d \rightarrow$b$) \leftarrow (p_s \rightarrow b)$
      $(p_d \rightarrow$g$) \leftarrow (p_s \rightarrow g)$
      $(p_d \rightarrow$r$) \leftarrow (p_s \rightarrow r)$
      $(p_d \rightarrow$ a$) \leftarrow (p_s \rightarrow a)$
    for $i \leftarrow 0\ ..\ 1$ **do**
      $bgra_t * p_d \leftarrow (bgra_t*)\ \&dstMatrix[f + largo][c * 4]$
      $(p_d \rightarrow$b$) \leftarrow (p_s \rightarrow b)$
      $(p_d \rightarrow$g$) \leftarrow (p_s \rightarrow g)$
      $(p_d \rightarrow$r$) \leftarrow (p_s \rightarrow r)$
      $(p_d \rightarrow$ a$) \leftarrow (p_s \rightarrow a)$

2.2. Código ASM

En la implementación de ASM recorremos 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:

*pixel*8	*pixel*6	*pixel*4	*pixel*2

16 0

Ya tenemos en **xmm10** la información tal cual queremos guardarla, ahora simplemente la guardamos en memoria manipulando algunos ínidices 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 imagenes con igual cantidad de píxeles totales, pero con distinto ancho y largo.

2.3.2. Hipótesis

Al comparar dos imagenes 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 imagenes de 200x1600 y 1600x200, donde podemos observar una diferencia de más de 4 millones de ciclos. En cambio en el par de imagenes 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.

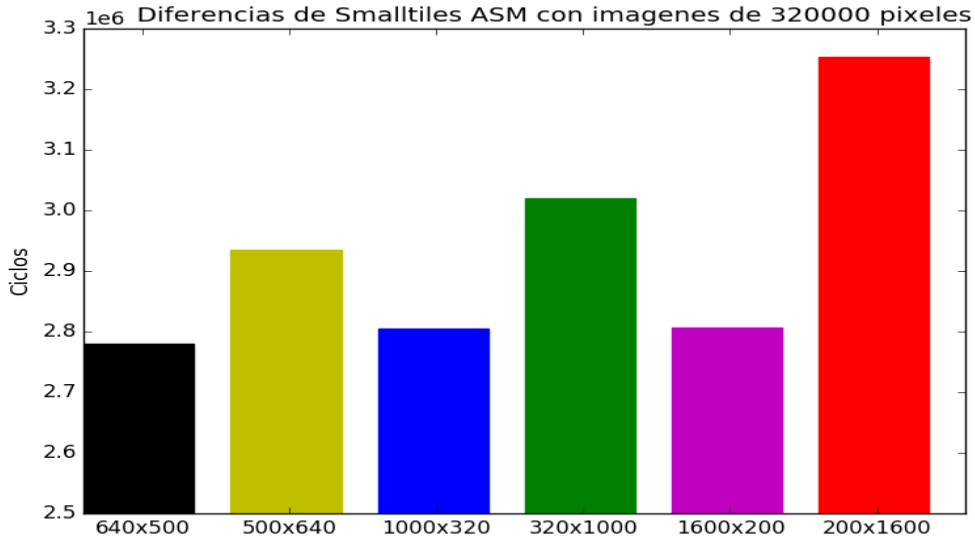


Figura 1: Gráfico de barras que da cuenta de los ciclos que implica cada corrida del filtro para imágenes de distintos tamaños con una determinada cantidad fija de píxeles.

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.

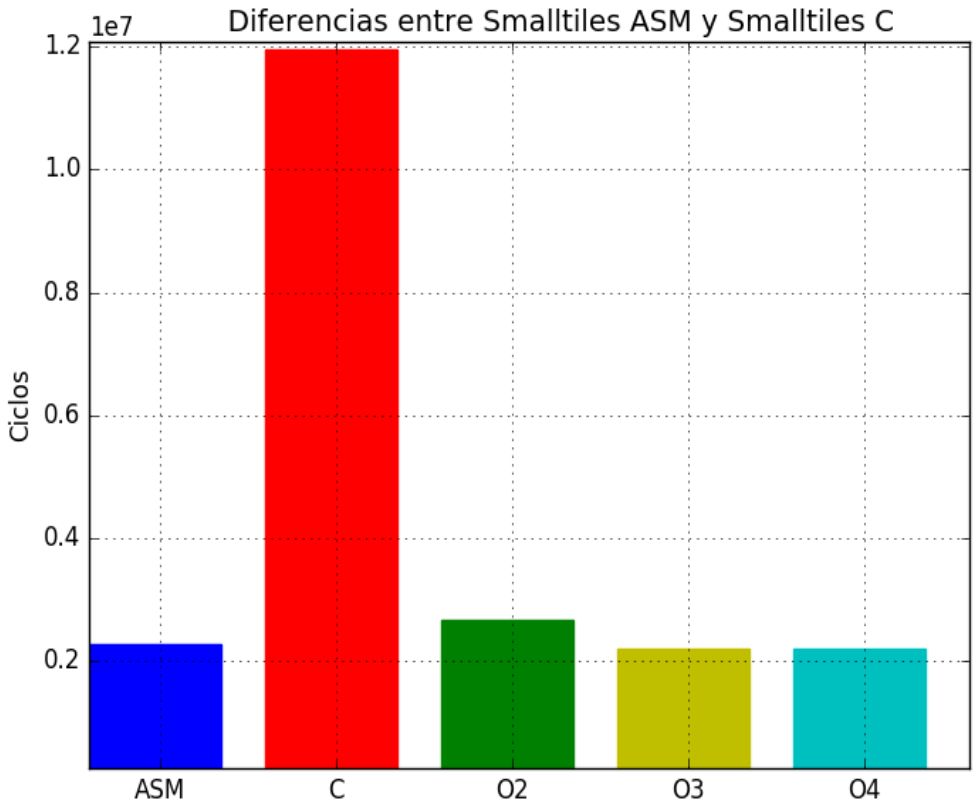


Figura 2: Comparación de las implementaciones y compilaciones con distintos niveles de optimización en cuanto a la rapidez para el procesamiento de una imagen.

3. Rotar canales

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

$$\mathbf{R} \longrightarrow \mathbf{G}$$
$$\mathbf{G} \longrightarrow \mathbf{B}$$
$$\mathbf{B} \longrightarrow \mathbf{R}$$

3.1. Código C

En el código de C recorremos la matriz iterando sus filas y columnas y modificando un pixel 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 $f \leftarrow 0\ ..\ filas - 1$ **do**
for $c \leftarrow 0\ ..\ cols - 1$ **do**
$bgra_t * p_s \leftarrow (bgra_t*)\ \&\ srcMatrix[f][c*4]$
$bgra_t * p_d \leftarrow (bgra_t*)\ \&dstMatrix[f][c*4]$
$p_d \rightarrow$b $\leftarrow p_s \rightarrow g$
$p_d \rightarrow$g $\leftarrow p_s \rightarrow r$
$p_d \rightarrow$r $\leftarrow p_s \rightarrow b$
$p_d \rightarrow$ a $\leftarrow p_s \rightarrow a$

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 pixeles 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 pixeles en xmm0 y se aplica **pshufb 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 hipotesis que tuvimos sobre la primera idea era que efectivamente ibamos a notar un cambio ya que ibamos a estar procesando la mitad de pixeles 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 notaria diferencia entre el código en ASM con y sin SIMD, más aún, el código optimizado de C es más eficiente que éste último. A continuación adjuntamos un gráfico en el que se puede apreciar dicha diferencia.

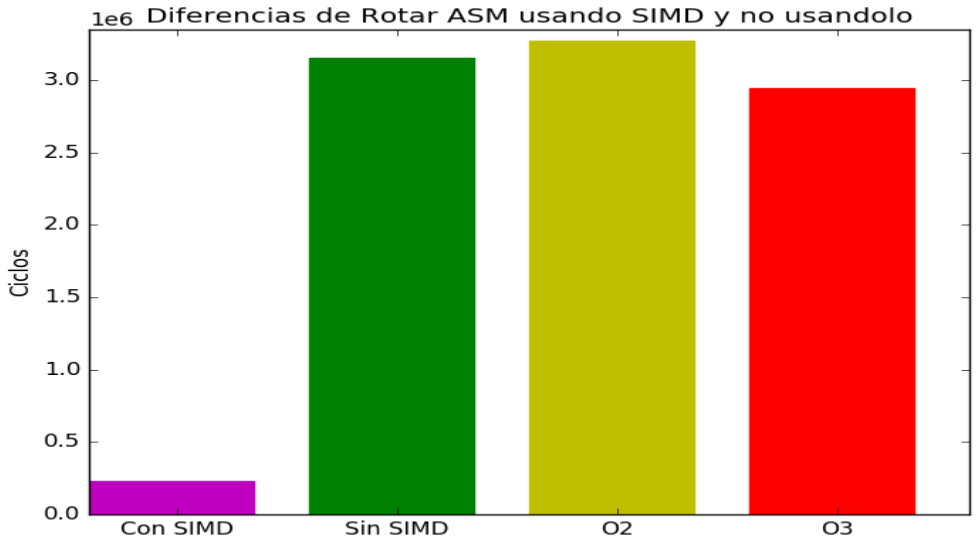


Figura 3: Gráfico que compara las velocidad de las ejecuciones de las implementaciones con y sin el uso de instrucciones SSE, y también con optimizaciones de la implementación en C.

El gráfico se armó en base a un conjunto de corridas de cada implementación del filtro, de donde se tomó la cantidad mínima de ciclos; con esto buscamos reducir la proporción de tiempo que consideramos que nuestro programa no estaba corriendo, si no que el scheduler estaba ejecutando otro programa.

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.

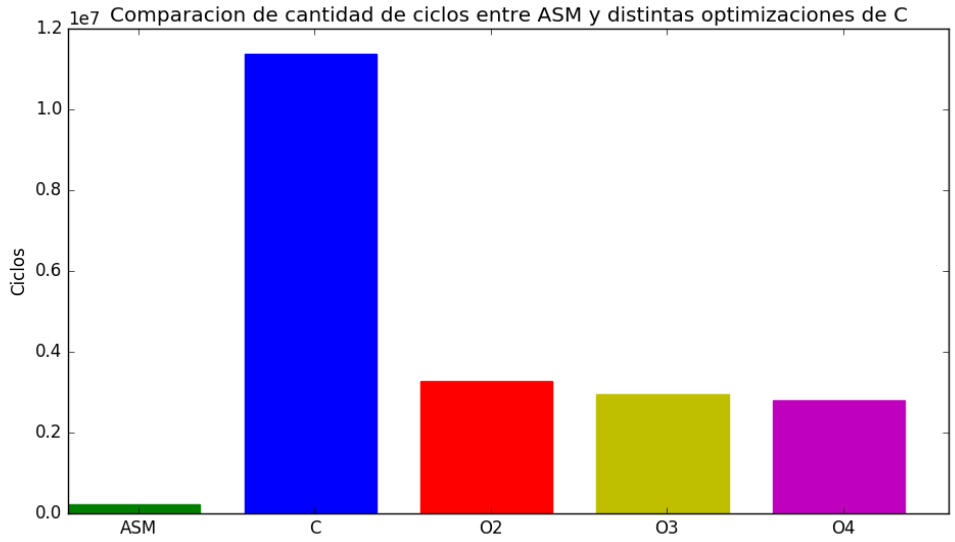


Figura 4: Diferencias en cantidad de ciclos para la ejecución del filtro con la implementación en ASM y la de C con sus optimizaciones.

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 cálidad de la imágen.

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`
  $float\ af \leftarrow a$
  $float\ bf \leftarrow b$
  $float\ cf \leftarrow c$
  $float\ df \leftarrow d$
  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 $f \leftarrow 0\ ..\ filas - 1; f+ = 2$ **do**
    for $c \leftarrow 0\ ..\ cols - 1; c+ = 2$ **do**
      $bgra_t * p_s1 \leftarrow (bgra_t*)\ \&\ srcMatrix[f][c * 4]$
      $bgra_t * p_d1 \leftarrow (bgra_t*)\ \&dstMatrix[f][c * 4]$
      $bgra_t * p_s2 \leftarrow (bgra_t*)\ \&\ srcMatrix[f + 1][c * 4]$
      $bgra_t * p_d2 \leftarrow (bgra_t*)\ \&dstMatrix[f + 1][c * 4]$
      $bgra_t * p_s3 \leftarrow (bgra_t*)\ \&\ srcMatrix[f + 1][(c + 1) * 4]$
      $bgra_t * p_d3 \leftarrow (bgra_t*)\ \&dstMatrix[f + 1][(c + 1) * 4]$
      $bgra_t * p_s4 \leftarrow (bgra_t*)\ \&\ srcMatrix[f][(c + 1) * 4]$
      $bgra_t * p_d4 \leftarrow (bgra_t*)\ \&dstMatrix[f][(c + 1) * 4]$
      k ← 0.5
      b ← PROMEDIO$(p_s \rightarrow b, p_s2 \rightarrow b, p_s3 \rightarrow b, p_s4 \rightarrow b)$
      g ← PROMEDIO$(p_s \rightarrow g, p_s2 \rightarrow g, p_s3 \rightarrow g, p_s4 \rightarrow g)$
      r ← PROMEDIO$(p_s \rightarrow r, p_s2 \rightarrow r, p_s3 \rightarrow r, p_s4 \rightarrow r)$
      a ← PROMEDIO$(p_s \rightarrow a, p_s2 \rightarrow a, p_s3 \rightarrow a, p_s4 \rightarrow a)$
      for $i \leftarrow 1\ ..\ 4$ **do**
        $(p_di \rightarrow$b) ← b
        $(p_di \rightarrow$g) ← g
        $(p_di \rightarrow$r) ← r
        $(p_di \rightarrow$ a) ← a

4.2. Código ASM

El codigo en ASM consiste en una conjuncion de cilos, uno exterior para iterar sobre las filas y otro interior para iterar sobre los elemntos de la misma (columnas). Lo que hace este codigo es en cada iteracion, 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 informacion va a servir para sobre escribir 8 pixeles.
El ciclo luego procede en exteneder los signos de cada byte, luego guardar la sumatoria de los elementos 1,2,6,7 (mirandolo como una matriz de 4x2), y la sumatoria de 4,5,8,9 en dos registros aparte, calcular el promedio de

esto, osea dividirlos por cuatro. Y luego a esta informacion la volvemos a agrupar juntas en un mismo registro que se veria 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 pixeles 1,2,6,7. Pp2 : promedio pixeles 4,5,8,9

.

Y finalmente a este registro lo escribimos en la imagen destino, en los lugar de los cual levantamos en la imagen src.

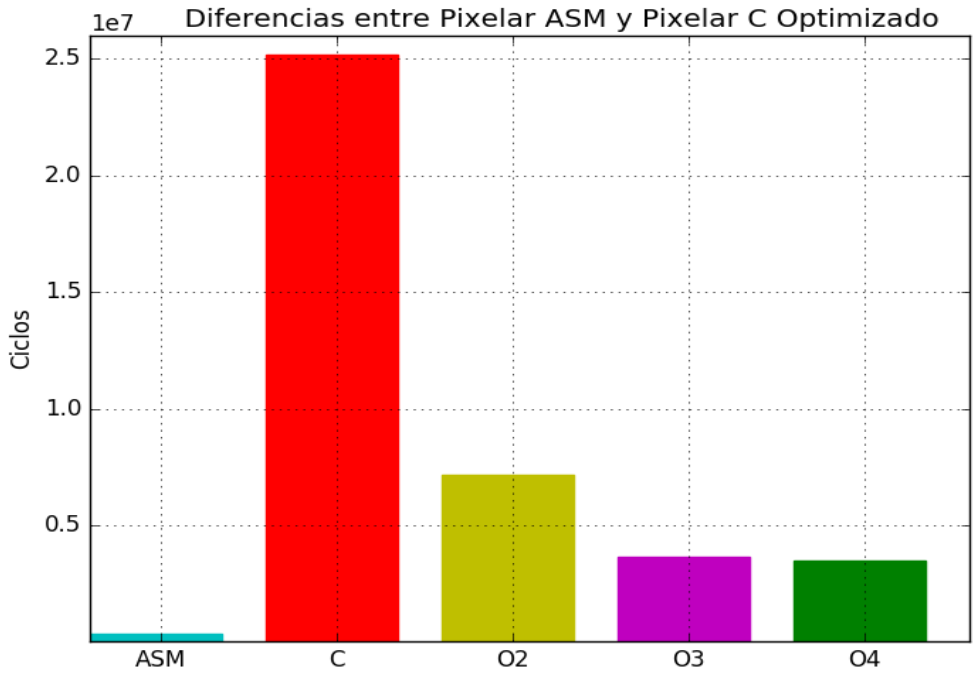
Por ultimo movemos los current en columnas 4 pixeles que es la cantidad sobre esa fila q procesamos en cada iteracion. Y si terminamos de procesar la fila, nos movemos dos filas, ya que vamos procesando dos juntas a la vez.

4.3. Experimentación

4.3.1. Idea

Comparamos la eficacia del código en ASM contra C y C optimizado.

4.3.2. Resultados



5. Combinar

Este filtro consiste en realizar una combinación de dos imágenes que dependa de un número real α 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.

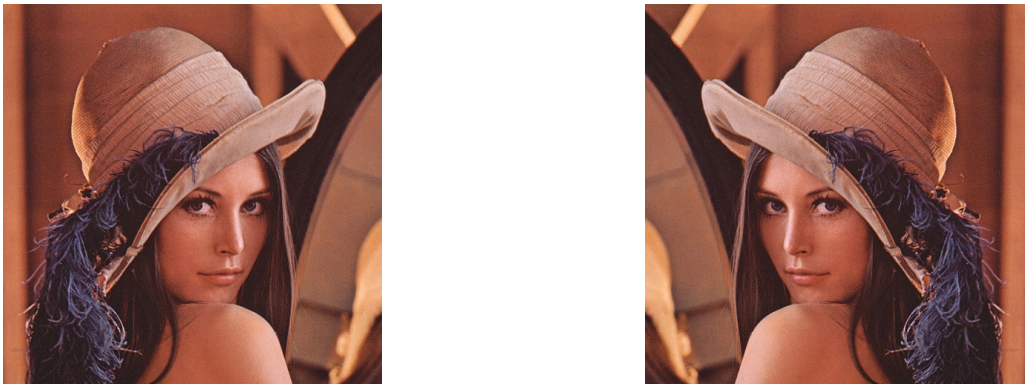


Figura 5: Se muestra la imagen A del lado izquierdo con la imagen B, el reflejo vertical de A, en la parte derecha.

Entonces se tiene que

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

y análogamente

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

Así, se obtiene

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

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, α : float) → float
    float af ← a
    float bf ← b
    return α * (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, α: 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
            bgra_t * p_sa ← (bgra_t*) & srcMatrix[f][c * 4]
            bgra_t * p_sb ← (bgra_t*) &srcMatrix[f][(cols − c − 1) * 4]
            bgra_t * p_d ← (bgra_t*) &dstMatrix[f][c * 4]
            p_d->b ← CLAMP(COMBINE(p_sa->b, p_sb->b, α))
            p_d->g ← CLAMP(COMBINE(p_sa->g, p_sb->g, α))
            p_d->r ← CLAMP(COMBINE(p_sa->r, p_sb->r, α))
            p_d->a ← CLAMP(COMBINE(p_sa->a, p_sb->a, α))
```

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 α.
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 (**cvtdq2ps 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 α 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 α 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 **cvtdq2ps 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_{Aj}	F_{Rj}	F_{Gj}	F_{Bj}	F_{Aj+1}	F_{Rj+1}	F_{Gj+1}	F_{Bj+1}

16 0

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

XMM11:

F_{Aj}	F_{Rj}	F_{Gj}	F_{Bj}	F_{Aj+1}	F_{Rj+1}	F_{Gj+1}	F_{Bj+1}	F_{Aj+2}	F_{Rj+2}	F_{Gj+2}	F_{Bj+2}	F_{Aj+3}	F_{Rj+3}	F_{Gj+3}	F_{Bj+3}

16 0

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

XMM11:

F_{Aj+3}	F_{Rj+3}	F_{Gj+3}	F_{Bj+3}	F_{Aj+2}	F_{Rj+2}	F_{Gj+2}	F_{Bj+2}	F_{Aj+1}	F_{Rj+1}	F_{Gj+1}	F_{Bj+1}	F_{Aj}	F_{Rj}	F_{Gj}	F_{Bj}

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 prosiguió 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 obtuvo el siguiente gráfico.

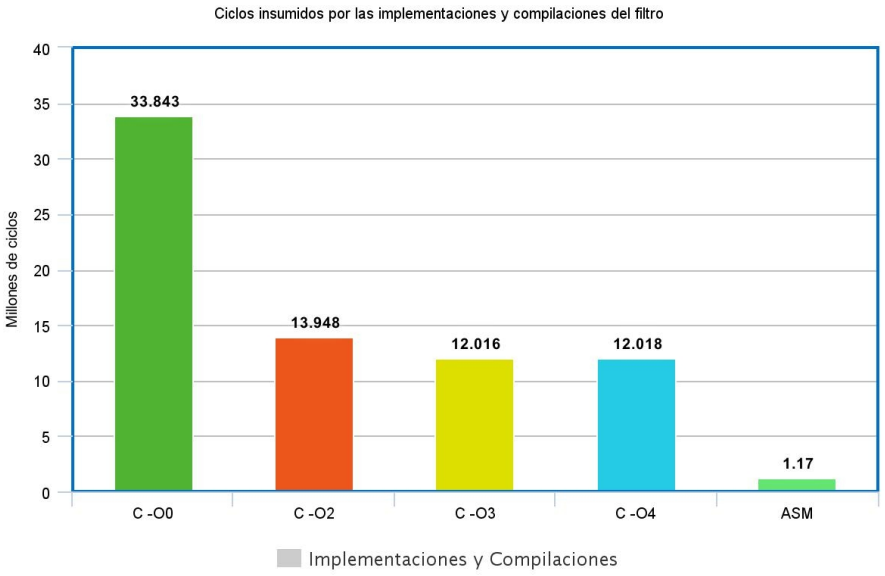


Figura 6: Gráfico de barras comparando las implementaciones y diversas optimizaciones.

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.

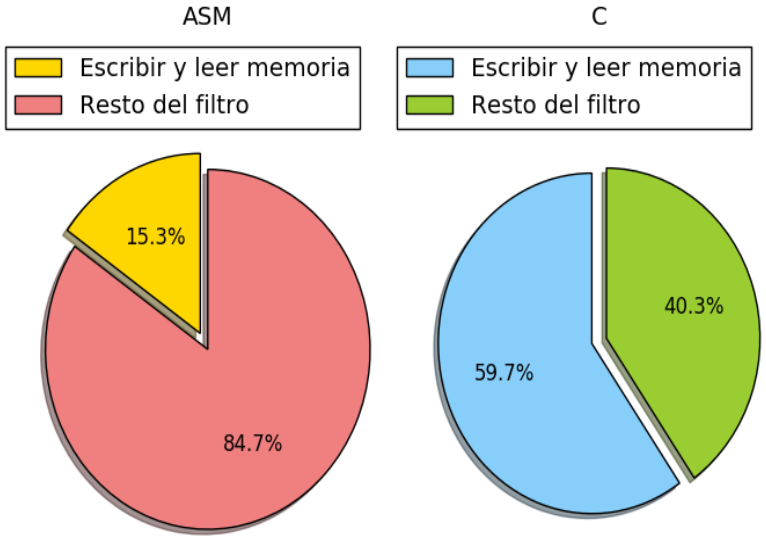


Figura 7: En estos gráficos se exponen las diferencias en las implementaciones en cuanto a la proporción de tiempo destinado a accesos a memoria y a procesamiento de la imagen.

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 opearciones que modifican la imagen, lo que hacemos es crear un arreglo de unsinged chars, res", que es donde guardamos los máximos de cada canal en comparación a todos los pixeles lindantes del píxel en el cual estemos parados (una matriz de 3x3).

- Res [0] ← MaximoLindantesAzul
- Res [1] ← MaximoLindantesVerde
- Res [2] ← 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 conjunción 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 simultaneamente con instruccciones SSE.
El ciclo interior consta de dos partes, la primera 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 asi:
Vamos a hacer referencia como "frutaçuando un byte tenga información que no nos interesa.

XMM1:

$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	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:

$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	A_Mp2	R_Mp2	G_Mp2	B_Mp2	A_Mp1	R_Mp1	G_Mp1	B_Mp1
16															0

XMM1:

$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	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:

$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	A_Mp2	R_Mp2	G_Mp2	B_Mp2	A_Mp1	R_Mp1	G_Mp1	B_Mp1
16															0

XMM1:

$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	RG_Mp2	$Fruta$	$Fruta$	$Fruta$	RG_Mp1	$Fruta$
16															0

notar que varias casillas ahora tienen mas la asignación de "fruta", y esto no es porque no sepamos que hay dentro de cada una, sino que no es información relevante a nuestras operaciones y que sera pisada si es necesario dentro de pronto

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

XMM2:

$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	A_Mp2	R_Mp2	G_Mp2	B_Mp2	A_Mp1	R_Mp1	G_Mp1	B_Mp1
16															0

XMM1:

$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	$Fruta$	RGB_{p2}	$Fruta$	$Fruta$	$Fruta$	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:

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

Y en la segunda parte del código lo que queda es que a este registro con la info del máximo de los máximos, lo transformamos en un registro con un 1 en el byte donde va el canal que contiene a este máximo entre los máximos (de cada píxel). Y ceros en el resto.
Teniendo esto se puede apreciar en el código cómo nos armamos los alphas personalizados para cada píxel y terminamos multiplicandolos y reescribiendo ambos píxeles.
Por último avanzamos nuestros currents sobre columnas en dos porque es la cantidad de píxeles que procesamos y volvemos en caso de no haber terminado toda la fila a empezar el ciclo interior.

6.3. Experimentación 1

6.3.1. Idea

El primer experimento al igua que en el resto de los filtros, vamos a realizar la comparaciones de cantidad de ciclos, por pixel procesado, entre los Codigos de C sin optimizar, C optimizado nivel 3, y Asm. En esto vamos a realizar tres comparaciones. La primera trata de comparar entre imagenes cuadradas, como evoluciona el rendimiento con el tamaño de imagen que le pasemos, y Como se ven con respecto a los otros codigos. La segunda comparacion trata de comparar imagenes con la misma cantidad d pixeles, pero con diferentes dimensiones. Y por ultimo comapramos rendimientos al tratar de imagenes con el mismo color en cada pixel.

6.3.2. Hipotesis

En la primer comparacion esperamos como siempre que el codigo de asm sea el mas optimo, luego le siga el codigo en C optimizado, y por ultimo el codigo de C sin optimizar.
En la segunda comparacion, se espera que sea similarla cantidad de ciclos por pixel que se obtiene porque el ciclo no busca ninguna optimizacion respecto al ancho o alto de la imagen, sino que procesa por igual a todos los pixeles. Y por ultimo en el tercer experimento suponemos lo mismo, no buscamos ninguna optimizacion con respecto a los valores de los pixeles adyacentes, sino que procesamos por igual a cada pixel indistintamente de su contexto, por lo que esperamos que los valores sean constantes.

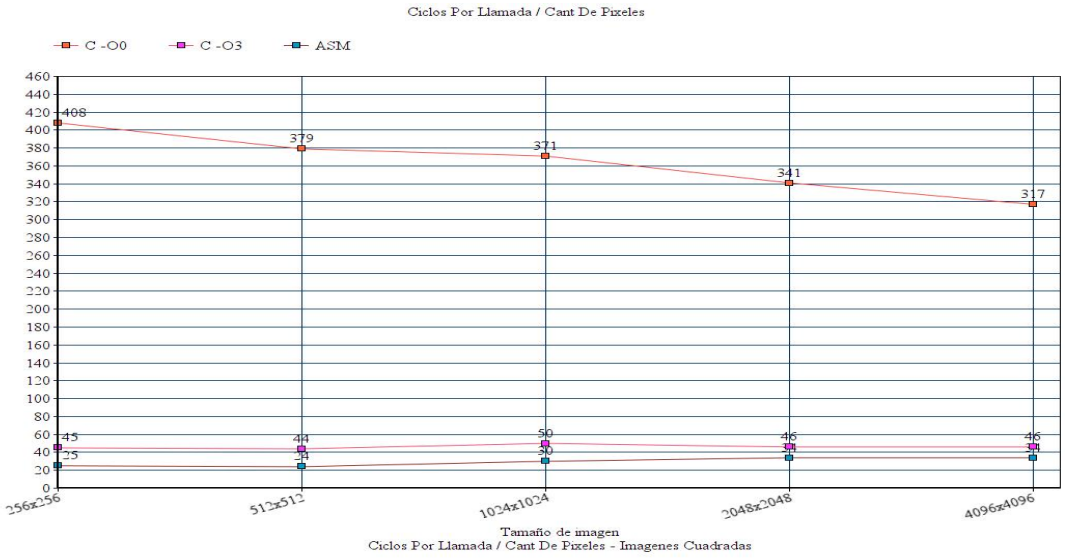


Figura 8: Gráfico de barras comparando las implementaciones y diversas optimizaciones.

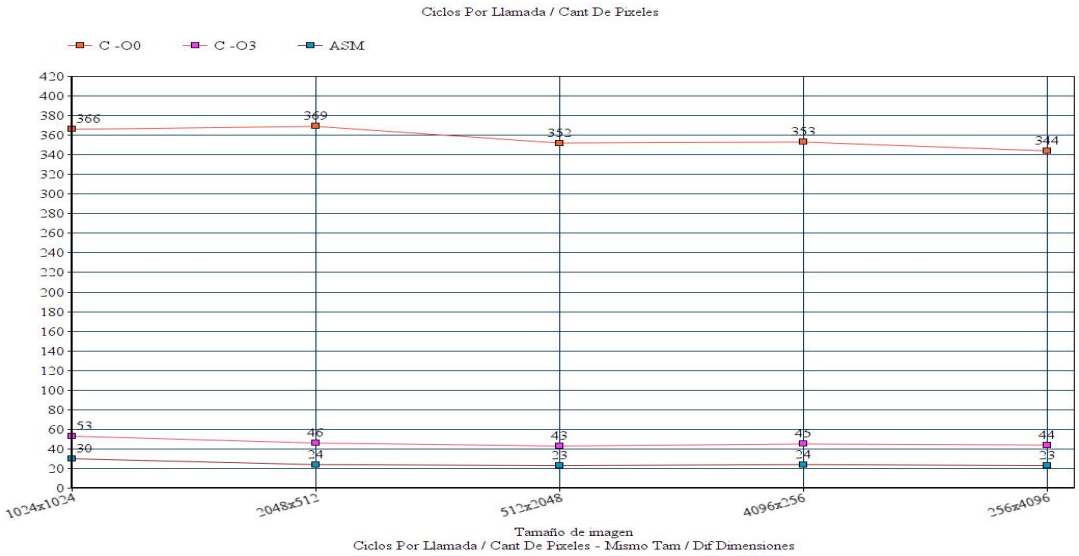


Figura 9: Gráfico de barras comparando las implementaciones y diversas optimizaciones.

6.3.3. Resultados

6.3.4. Conclucion

La conclucion que podemos sacar de estas imagenes, es que, en primer lugar, como dijimos en la hipotesis, que se termino corroborando, el codigo en C sin optimizar fue por lejos mas lento que los otros dos. Y el codigo en C optimizado por mas que mejoro muchisimo con respecto al anterior, siguio siendo mas lento que el codigo en asm.
Luego podemos ver que no nos confundimos tampoco en el momento de cambiar los tamaño de las imagenes pues los valores se mantuvieron bastantes constantes, y esto porque justamente no se cambia la cantidad de operaciones que se hace respecto a las dimensiones de las imagenes, siempre se mantiene segun la cantidad de pixeles.
Por ultimo sin embargo no llamo la atencion el experimento de correr los codigos con imagenes de un solo

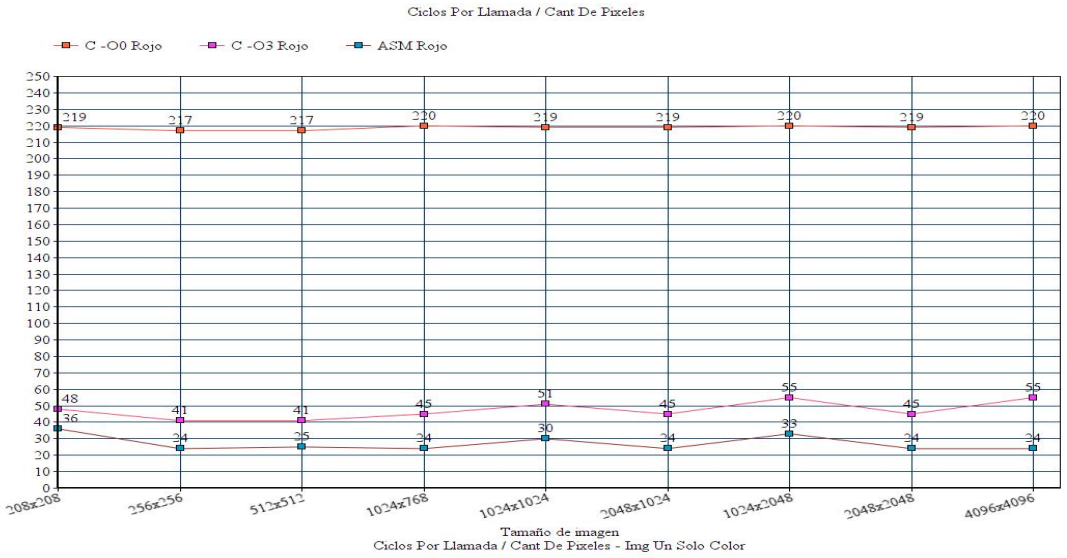


Figura 10: Gráfico de barras comparando las implementaciones y diversas optimizaciones.

color. Lo que paso aca de extraño se que bueno los codigos en asm y en C -O3 se mantuvieron cosntantes a los desarrollados con imagenes comunes, sin embargo el codigo en C -O0 demostros una gran mejora respecto a su performance, la cual no podemos explicar bien el porque de ella, y de hecho esto tambien se ve en la primer imagen, donde el las imagenes de 4096x4096 el codigo en C -O0 mostro que mejoraba su rendimiento. Y la explicacion que le asignamos a eso es que justamente porque las imagenes que le pasamos tienen grandes secciones con pixeles del mismo color, porque son eexpanciones de una imagen de 512x512. Y se genera en estos fragmentos esta optimizacion que no podemos explicar sobre C -O0 en imagenes donde el color no cambia.

6.4. Experimentacion 2

6.4.1. Idea

El segundo experimento, es un experimento mas destructivo. Vamos a intentar molestar al jump predictor y ver como influye eso en la performance del programa. El experimento en si trata de agarrar un numero 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 numero arbitrario, salte a diferentes secciones del codigo. TOdo esto dispuesto de tal manera que sea transparente a la ejecucion final del programa, osea si el codigo sigue el camino del if no cambia a si siquue el camino del else.

6.4.2. Hipotesis

Nuestra hipotesis es que justamente el codigo no modificado va a ser bastante mas optimo al codigo desarrollado con molestias para el jump predictor, porque en caso de funcionar estas, en cada ciclo estariamos haciendo que flushee el pipeline, y que pierda muchos ciclos de clock extra.

6.4.3. Resutltados

6.4.4. Conclucion

Efectivamente los resultados fueron los esperados, por lo que nos llevamos como conclucion del experimento que influye mucho la arbitrareidad del flujo del programa que armes, si uno lo hace muy random como en este caso, por mas que no afecte a la complegidad total el programa tarda significativamente mas.

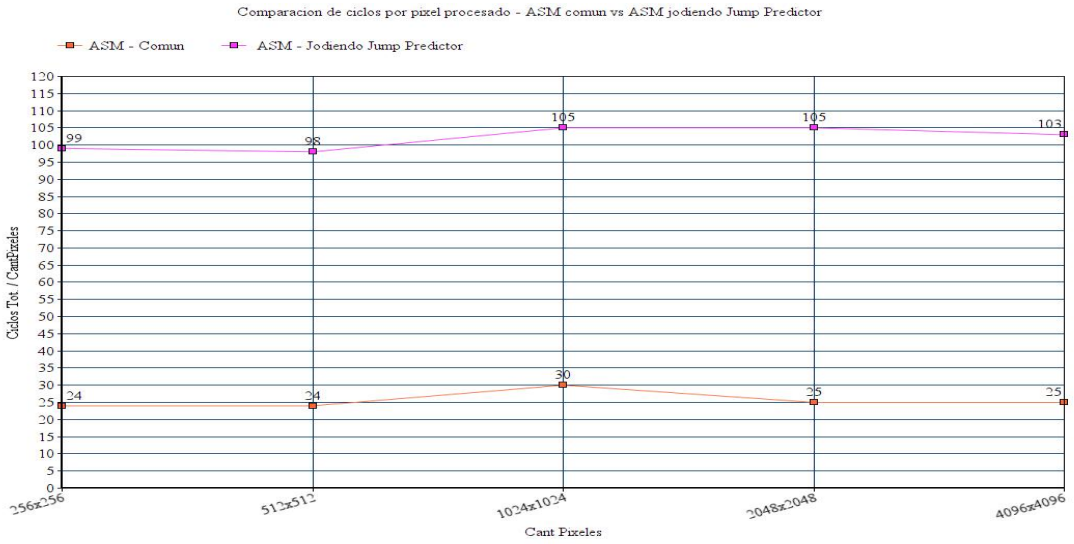


Figura 11: Gráfico de barras comparando las implementaciones y diversas optimizaciones.

6.5. Experimento 3

6.5.1. Idea

En el tercer experimento vamos a ver hasta que punto es optimo aplicar la tecnica de unrolling sobre un codigo. Osea lo que planeamos hacer es correr el codigo de colorizar, sobre una imagen de 1024x1024, desenrrollando el ciclo interior, distinta cantidad de veces, vamos a empezar desenrrollando 16, luego 32, y luego completamente.

6.5.2. Hipotesis

Lo que esperamos de este experimento es que el codigo desenrrollado 16 y 32 veces, mantenga el nivel de performance con el que demuestra normalmente, o un poco mas optimo, y esto porque en sí el codigo de colorizar lleva muchas instrucciones, por lo que los pasos que nos ahorramos al desenrrollar el codigo no es significativo en el total del programa. Sin embargo en el caso de desenrrollar todo el codigo, esperamos una mucha peor performance por parte del codigo y esto debido a que el tamaño del codigo aumentaria enrmomente su tamaño de modo que no entre en la memoria cache el mismo, y deba empezar a hacer lecturas de memoria para leer las instrucciones.

6.5.3. Resultados

6.5.4. Conclusión

La conclucion que sacamos de este esxprimento es que la tecnica de desenrrollar codigo es efectiva para pequeños whiles, porq eliminas algunas instrucciones inecesarias, sin embargo ya para un while de un tamaño mayor a lo usual, la optimizacion generada por esto se diluye hasta el punto d no notarse. Ademas de que efectivamente elementos como la memoria cache son limitantes aca, donde pasa que los problemas de lectura a memoria dejan de ser parte de como esta hecho el algoritmo y pasa a ser a simplemente poder leerlo.

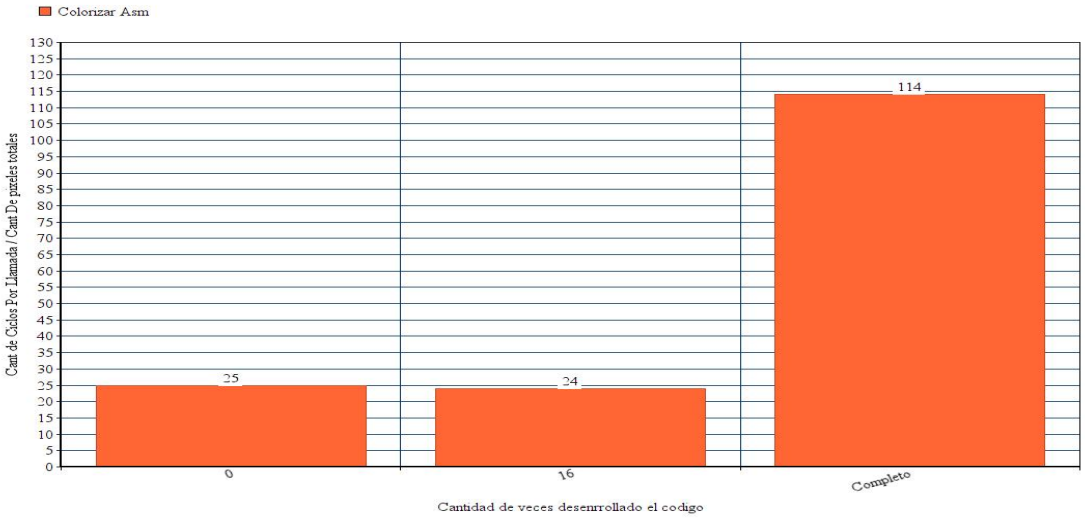


Figura 12: Gráfico de barras comparando las implementaciones y diversas optimizaciones.