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	grego_kpo@gmail.com
Taboh, Sebastián	185/13	sebi_282@hotmail.com
Romero Lucía	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

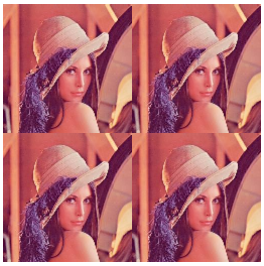
1. Introducción

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

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.

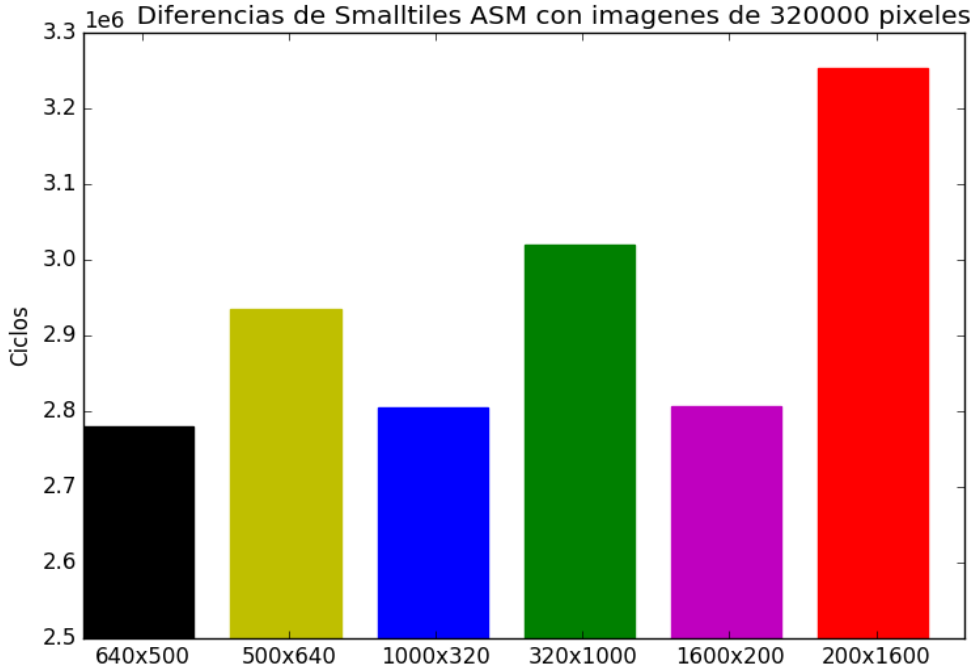


Figura 1: •

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.

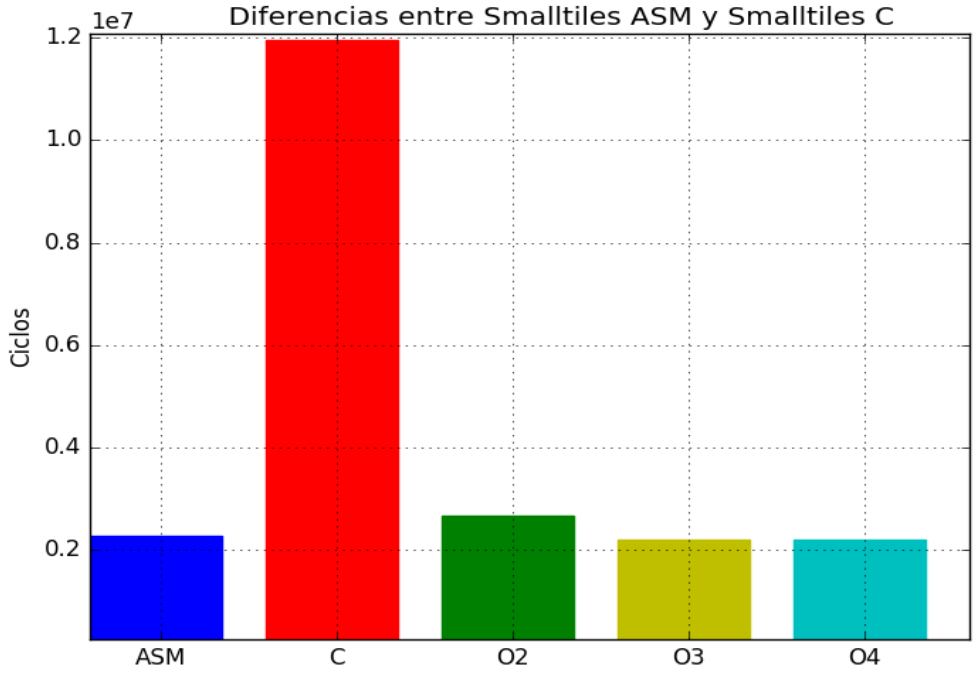


Figura 2: •

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.

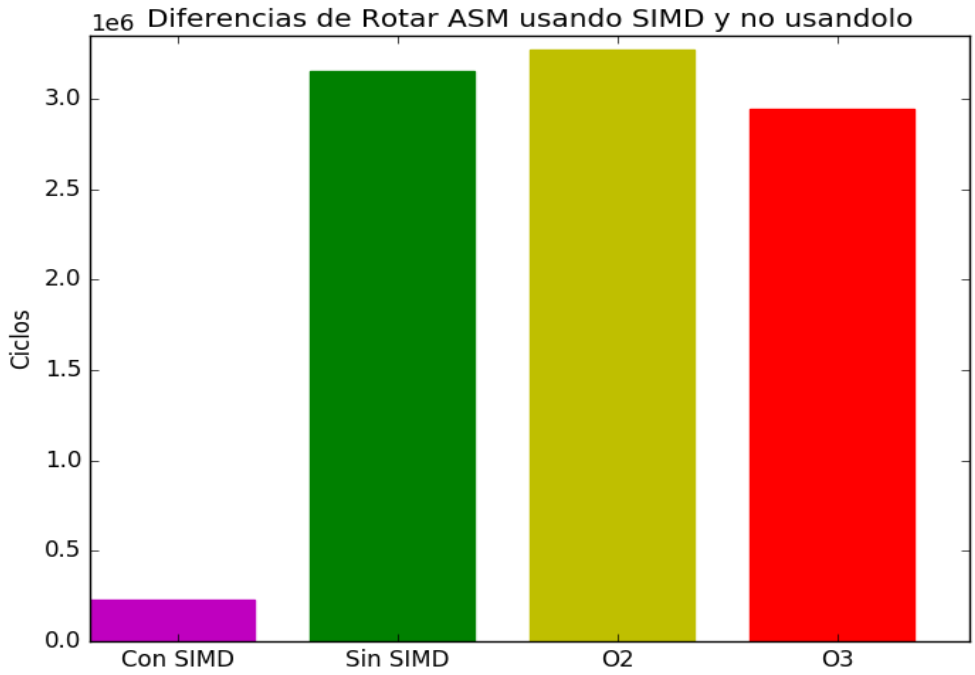
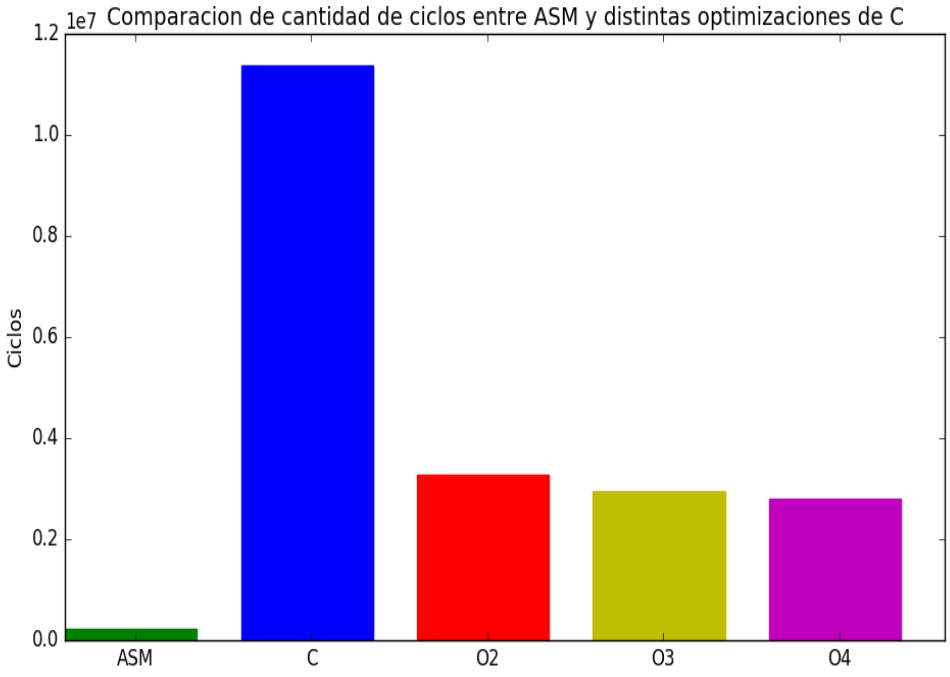


Figura 3: •

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 lo s tiempos de ejecución del filtro implementado en ASM y en C con distintas optimizaciones.


Comparacion de cantidad de ciclos entre ASM y distintas optimizaciones de C
1e7
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Ciclos
ASM
C
O2
O3
O4

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

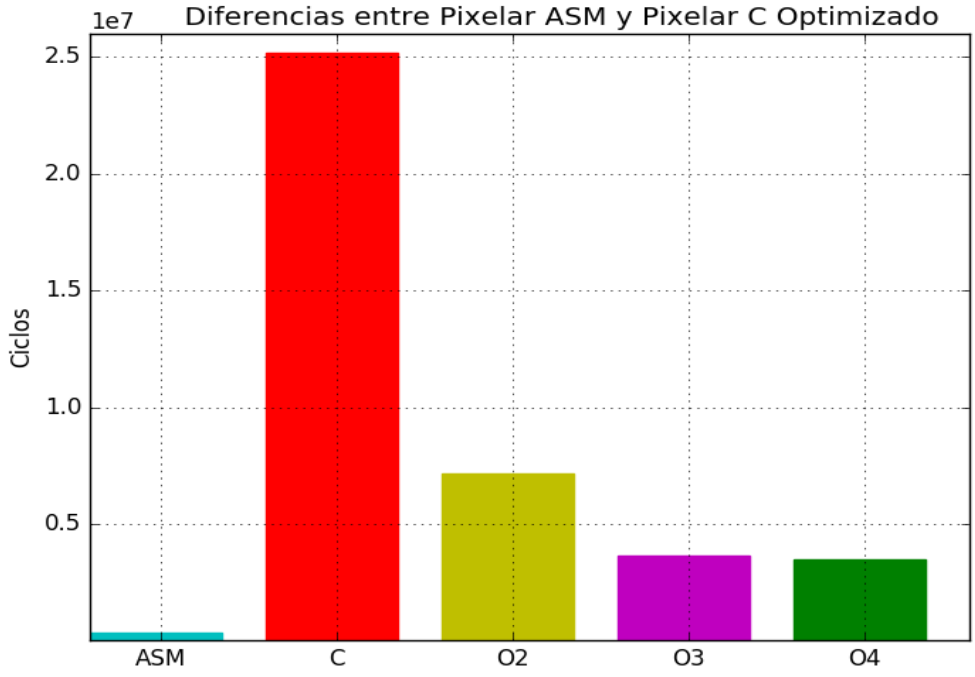
4.2. Código ASM

4.3. Experimentación

4.3.1. Idea

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

4.3.2. Resultados


Diferencias entre Pixelar ASM y Pixelar C Optimizado
1e7
2.5
2.0
1.5
1.0
0.5
Ciclos
ASM
C
O2
O3
O4

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.

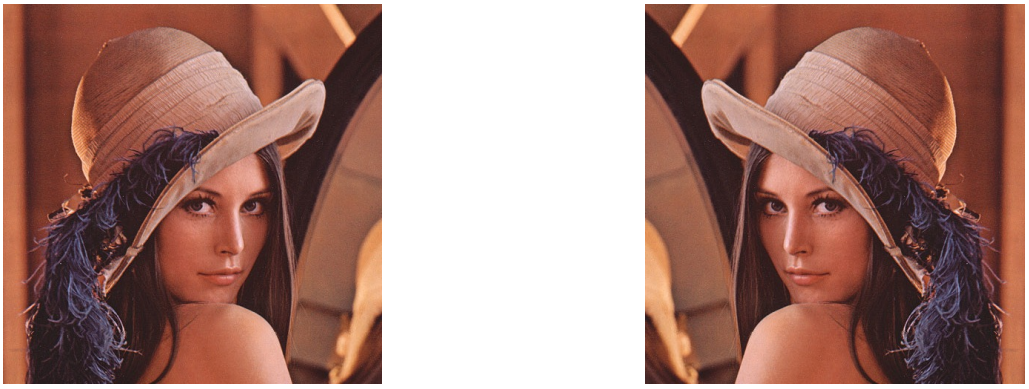


Figura 4: Se muestra la imagen A del lado izquierdo con la imagen B, el reflejo vertical de A, en la parte derecha.

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.

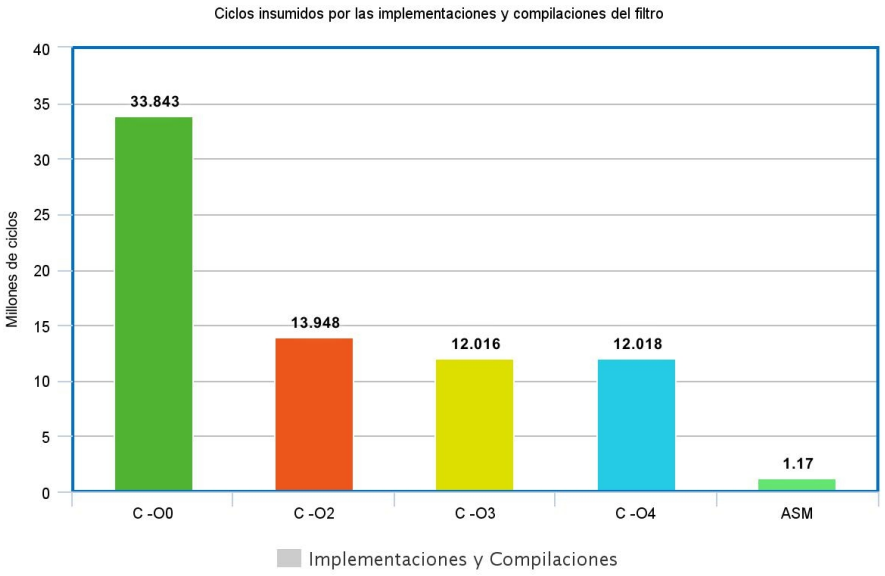
5.3.2. Idea

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.

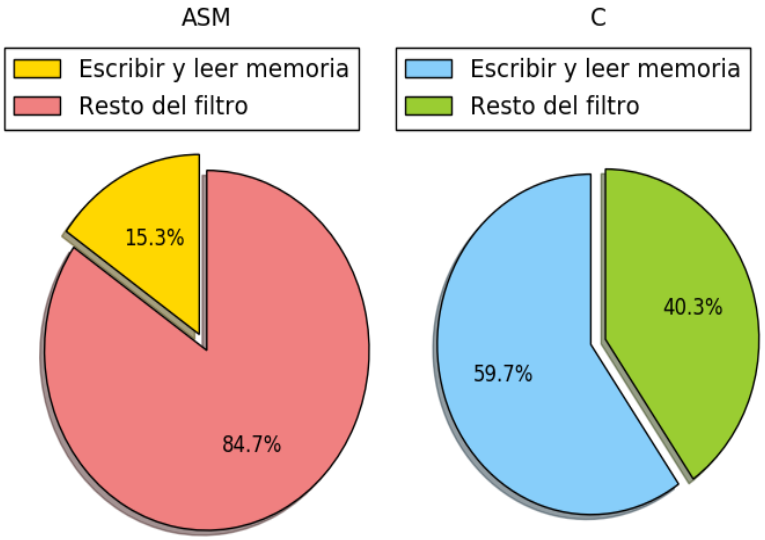
5.3.3. Resultados

En cuanto a las cantidades totales de ciclos de clock se obtuvo el siguiente gráfico.



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.



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 conjuncion de Fors, el exterior que recorre desde la segunda fila hasta la ante ultima, y el interior que recorre desde la segunda columna hasta la ultima, dejando asi afuera a todos los bordes, tal como el enunciado decia. Luego en cada iteracion del ciclo interior, que es donde se hace las opearciones que modifican la imagen, lo que hacemos es crear un arreglo de unsinged chars, res", que es donde guardamos los maximos de cada canal en comparacion a todos los pixeles lindantes del pixel en el cual estemos parado.

- Res [0] ← MaximoLindantesAzul
- Res [1] ← MaximoLindantesVerde
- Res [2] ← MaximoLindantesRojo

Luego con estos tres valores calculamos el alpha correspondiente de cada canal por el cual voy a multiplicar a cada uno. Y por ultimo reescribimos el pixel final, en la imagen source con cada canal multiplicado por dicho alpha.

6.2. Código ASM

El código en ASM se trata tambien de una conjuncion de ciclos. El recorre las filas desde la segunda hasta la anteUltima, y el interior recorre las columnas desde la segunda hasta la anteultima, pero saltando de a dos pixeles, que es la cantidad que procesamos simultaneamente con instruccciones SSE. El ciclo interior consta de tres partes, la primera es tan pronto se levanta de memoria los pixeles a procesar y todos sus lindantes, calculamos en dos registros los maximos de cada canal, de ambos pixeles a procesar con respecto a todos sus lindantes, y los guardamos ambos en un registro xmm. Luego la segunda parte es calcular el maximo de los maximos de ambos al mismo tiempo. Luego atravez de una par de operaciones, muy especificas para explicar, logro tener un registro xmm de dw, con cada dw representando un canal de cada pixel, en el orden establecido (argb), donde tengo un 1-alpha en la posicion del canal que no teine el maximo de los maximos, y un 1+alpha en la posicion que tiene al maximo de los maximos, y luego concluyo multiplicando a cada pixel por su registro con los alphas indicados, lo reduzco a tamaño de 32 bits por pixel, los uno y los escribo en el destino.

6.3. Experimentación

6.3.1. Idea

En la experimentacion de este filtro al igual que en el resto vamos a comparar el rendimiento respecto a los ciclos de clock, que tiene la funcion colorizar en C desde -o0 a -o3 contra asm.
Sin embargo luego vamos a contrastar la funcion de asm, contra sigo mismo pero primero agragando jumps de forma que no influya el flujo del programa solo para molestar al jump predictor. Luego vamos a correrlo tal cual esta, y de apoco vamos a ir desenrollando el codigo. Como dijimos tiene un ciclo externo y uno interno, por lo que vamos a desenrrollar primero el interno 2 veces, luego 4,16,32 y ver que pasa, y finalmente vamos a saltar a desenrrollarlo completamente, cosa de no dejar casi ninguno controlador de flujo.

6.3.2. Hipótesis

Nuestra hipotesis es que el rendimiento va a ir mejorando a medida que vayamos cambiando los programas respectivamente a como los fui mencionando, osea el mas lento va a ser el codigo de asm, molestando al jmp predictor, y el mas rapido el desenrollando el codigo 32 veces. El ultimo test, por mas que desenrollemos todo el programa creemos justamente que va a ser el mas lento, por el tamaño del codigo, creemos que al hacer un codigo de tamaño mayor al de la cache, en un momento el pc va a estar haciendo muchisimos mas miss en la cache, que los que ahorra sacando los controladores de flujo.

6.3.3. Resultados

Graficos lindos de lucia :D ciclos promedio colorizar c -o0 100 iteraciones : 101.898.160 ciclos promedio colorizar c -o3 100 iteraciones : 21.567.776 ciclos promedio colorizar original 1000 iteraciones: 5.980.263 ciclos promedio colorizar JodiendoJumps1 1000 iteraciones: 5.977.453 ciclos promedio colorizar Unrolling Ocho ciclos 100 iteraciones : 5.873.712

ciclos promedio colorizar Unrolling lena 1024x768 todos los cilcos 100 iteraciones : 35.285.958
ciclos promedio colorizar original lena 1024x768 100 iteraciones: 21.050.994

6.3.4. Conclusión

Conclusion Todo es una mierda. Conclusion posta: Primera gran conclusión reveladora, c -o0 demuestra ser mucho mas lento que optimizando, el código mejoro 5 veces su cantidad de ciclos por llamada al compilar con -O3. Luego sin embargo la función hecha en asm le sigue ganando por bastante a C, en una proporción de 3.5 veces mas rápido en cantidad de ciclos todavía. Ahora encuanto al experimento personalizado, hicimos una comparación con distintos códigos para medir la influencia del jump predictor en la ejecución del código. Primera prueba fue ejecutar el mismo código que teníamos con el agregado de que en el ciclo interior (el que se repite mas veces) metíamos una serie de saltos transparentes con diferentes comparaciones, para tratar de romper con pipeline del procesador. Sin embargo por mas duro que itnentasemos descubrimos que el algoritmo que dirige al jump predictor es bastante mas sofisticado de lo que pensábamos, por lo que no pudimos influenciar en lo mas mínimo la eficiencia del código. Luego para contrastar el ultimo experimento mencionado, intentamos implementar lo contrario, desenrollar el código de manera que nunca influya un miss hit del jump predictor, en la ejecución del programa, sin embargo ya sacando la conclucion de lo mencionado recién, de que el sistema que lo regula es mas sofisticado del esperado, el desenrollar el código principal 2,4,8 veces, no estuvo mostrando ninguna diferencia con respecto al original. Por ultimo pasamos a otro experimento con desenrollar, en el cual se pretendia desenrollar completamente el codigo, generando un resultado en caunto a la cantidad de ciclos consumidos por llamada mucho mayor, por el tema de que un codigo tan grande no entraria completametne en la cache porl o que a partir de un punto el PC estaria generando constatnes miss hits en la cache y contrarestrando lo logrado evitar por el jump predictos. Y De hecho este experimento funciono muy bien, lo intentamos con una imagen mayor de lena, de 1024x768 para que el ciclo pueda ser desenrrollado mas cantidad de veces, y asi se ve que el codigo modificado tardo en promedio al rededor de unos 14.000.00 de ciclos mas que el original.