DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Procesamiento SIMD

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Nuñez Morales Carlos Daniel	732/08	cdani.nm@gmail.com
Salvador Alejo Antonio	467/15	alelucmdp@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

Resumen

En el presente trabajo se analiza la diferencia en performance que pueden provocar los misses en la cache y un analisis de performance y calidad al utilizar el set de instrucciones SIMD.

Índice

1. Objetivos generales

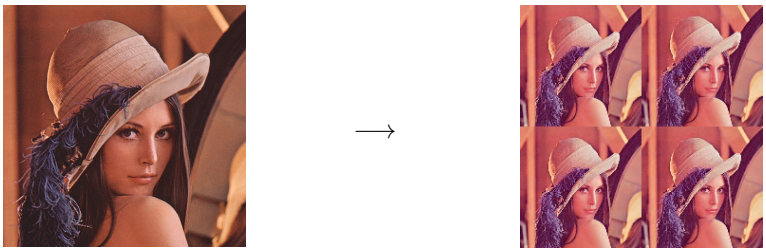
El objetivo de este Trabajo Práctico es analizar la importancia que puede tener la cache en la performance de un programa. También se analizará la mejora de la perfomance obtenida al utilizar el set de instrucciones SIMD en un codigo ASM por sobre un codigo C compilado con O3 (para asi poder comparar el rendimiento del codigo de assembler con el mejor código posible de C). Por otro lado, analizamos la precisión que se pierde al bajar la precición de los floats para así procesar más de manera simultanea.

2. Contexto

Se implementará una serie de fitros tanto en C como en ASM (en este ultimo caso, utilizando el set de instrucciones SIMD) en los cuales se tomará una imagen de entrada en formato BMP y se obtendrá una imagen resultante en ese mismo formato, con el objetivo de utilizarlas para realizar el análisis descripto en los objetivos generales. A continuación se procedera a describir dichos filtros.

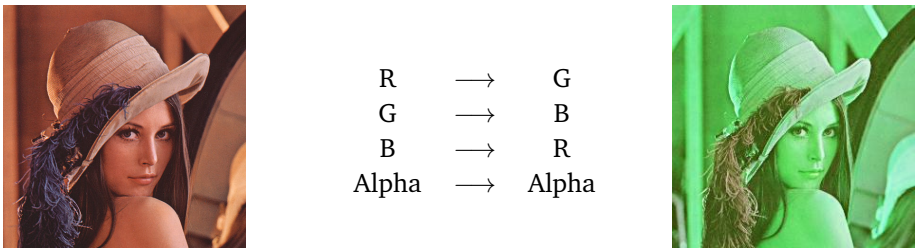
2.1. Smalltiles

El filtro consiste en generar cuatro miniaturas de una imagen fuente en una misma imagen final. Cada miniatura posee la mitad de dimensiones.



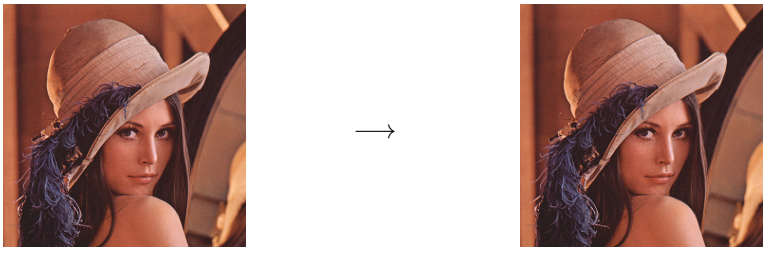
2.2. Rotar

El filtro consiste en intercambiar los valores de los canales entre si. En el resultado final, los canales quedan ordenados de la siguiente manera:



2.3. Pixelar

El filtro consiste en tomar cuadrados de 2x2 pixeles de una imagen base, calcular el promedio entre ellos y copiar el pixel obtenido a los 4 pixeles correspondientes de la imagen destino.



2.4. Combinar

El filtro consiste en tomar dos imagenes, A y B, y combinarlas segun un parametro alpha (entre 0 y 255) que indica cuanto de la imagen B se aplicará. Un valor máximo de 255 no modifica la imagen A, mientras que un valor de 0 dejará unicamente la imagen B.


2.5. Colorizar

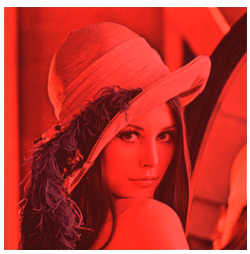
El filtro consiste en analizar cada pixel y sus vecinos obteniendo el maximo para cada canal entre todos ellos. Luego, para cada canal, se analiza de acuerdo a las siguientes reglas:

$$\phi_R(i,j) = \begin{cases} (1+\alpha) & si\ max_R(i,j) \geq max_G(i,j) y max_R(i,j) \geq max_B(i,j) \\ (1-\alpha) & si\ no \end{cases}$$

$$\phi_G(i,j) = \begin{cases} (1+\alpha) & si\ max_R(i,j) < max_G(i,j) y max_G(i,j) \geq max_B(i,j) \\ (1-\alpha) & si\ no \end{cases}$$

$$\phi_B(i,j) = \begin{cases} (1+\alpha) & si\ max_R(i,j) < max_B(i,j) y max_G(i,j) < max_B(i,j) \\ (1-\alpha) & si\ no \end{cases}$$

Una vez obtenido cada ϕ_c, se calcula el mínimo entre $\phi_c * max_c(i,j)$ y 255 y se utiliza este como valor para el canal c del pixel (i,j) destino.

3. Implementacion

3.1. Smalltiles

Para aplicar el filtro, se recorre la matriz de a filas (ignorando las filas pares, ya que serian las impares de la imágen al estar invertidas y por como esta definido el filtro solo interesan los pixeles en fila y columna par) y de a 4 columnas/pixeles (dado que es el maximo que se puede guardar en un registro xmm). En el ciclo exterior recorremos cada fila impar, mientras que en el interior procesamos las columnas. Como podría suceder que la cantidad de columnas de la imagen sea de la forma 4K+2 con K entero y el algoritmo recorre de a 4 por fila, debe hacerse un caso especial en el cual de pasar eso se recorren los ultimos 2 elementos de forma independiente, lo cuál no se hara con SIMD ya que no se ganaría nada al solo tener que usar el segundo elemento de los 2 al momento de escribir en los lugares correspondientes.

El algoritmo de procesamiento es el siguiente:

- Buscamos en memoria 4 pixeles, px1, px2, px3, y px4. figura 1

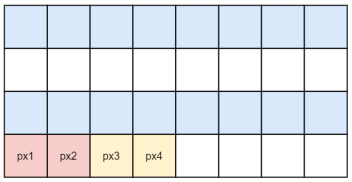


Figura 1: Etiquetas de pixels

- Ahora haremos un shuffle para conseguir que los elementos px2 y px4 queden en la low-word. figura 2

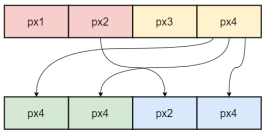


Figura 2: Shuffle

- Ahora guardo los 2 elementos de la low-word en las 4 posiciones que corresponde en la imagen de destino. figura 3

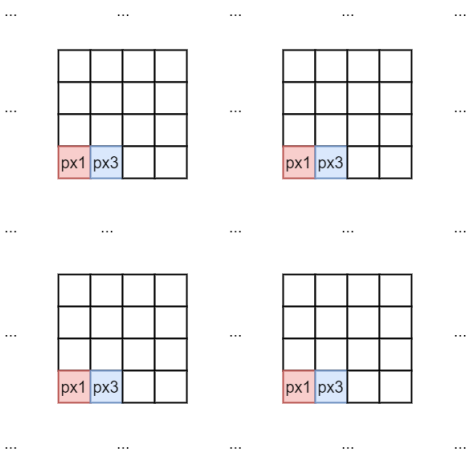


Figura 3: Guardar

Codigo fuente:

```
.ciclo_columna:
cmp rdx, 0
je .fin
mov rdi, r10
mov rsi, r11
mov rbx, r12
mov rcx, r13
sar rcx, 1 ; proceso dos pixels por vez
.ciclo_fila:
movdqu xmm0, [rdi]
pshufd xmm1, xmm0, 0x08
movq [rsi], xmm1
movq [rsi+r13*4], xmm1
movq [rbx], xmm1
movq [rbx+r13*4], xmm1
add rdi, 16
add rsi, 8
add rbx, 8

loop .ciclo_fila

mov rcx, r13
shr rcx, 1
shl rcx, 1
sub rcx, r13
cmp rcx, 0
je .salir_c_fila
mov eax, [rdi]
mov [rsi], eax
mov [rsi+r13*4], eax
mov [rbx], eax
mov [rbx+r13*4], eax

.salir_c_fila:
lea r10, [r10+r8*2]
lea r11, [r11+r9]
lea r12, [r12+r9]
dec rdx
jmp .ciclo_columna
```

3.2. Rotar

Para aplicar el filtro, se recorre la imagen de una fila y de a 4 columnas/pixeles (dado que es el maximo que se puede guardar en un registro xmm). En el ciclo exterior recorremos las filas, mientras que en el interior procesamos las columnas. En caso de que las filas tengan una cantidad de pixles que no sea multiplo de 4 se seguira haciendo exactamente el mismo shufle, pero solamente se copiaran la cantidad de elementos que corresponda en el último paso. El proceso de determinar la congruencia modulo 4 se hace simplemente usando un compare.

El algoritmo de procesamiento para cada iteración en la fila es el siguiente:

- Buscamos en memoria 4 pixeles, px1, px2, px3, px4 figura 4

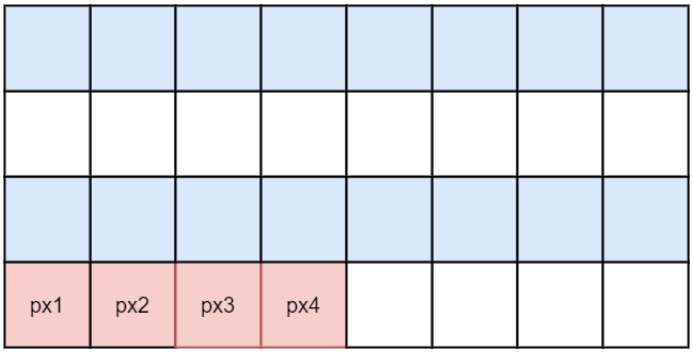


Figura 4: Etiquetas de pixels

- Hacemos un shufle de forma tal que se roten los colores de cada uno de dichos pixeles. figura 5

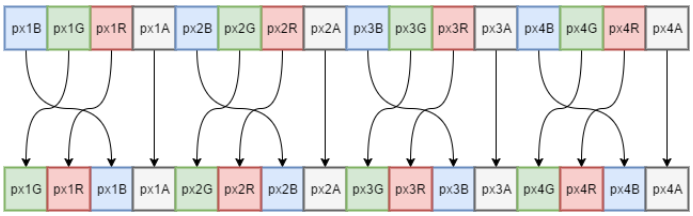


Figura 5: Shuffle

- Guardamos en el resutlado lo obtenido con el shuffle.

Codigo fuente:

```
cicloFilas:
    cmp rax, 0 ;comparo el alto con 0 a ver si termine de procesar todas las filas
   je finRotar
   mov rdi, r10
   mov rsi, r11
   mov rcx, rdx
   shr rcx, 2 ;proceso de a 4 pixeles

cicloColumnas:

   movdqu xmm0, [rdi] ;xmm0= |a15|....|a0| ;uso xmm0 para rojo
   pshufb xmm0,xmm1
   movdqu [rsi], xmm0 ;muevo a memoria

   add rdi, 16
   add rsi, 16
   loop cicloColumnas

;sino es multiplo de 4 ntonces me faltan procesar 3, 2 o 1 pixel
mov rcx, rax
shr rcx, 2
shl rcx, 2
sub rcx, rax ;esta resta puede ser: 0, -1, -2 o -3
cmp rcx, 0
je .no_faltan
cmp rcx, -1
je .faltan_1
cmp rcx, -2
je .faltan_2
;si no salta entonces faltaba 3 pixeles:

movdqu xmm0, [rdi] ;xmm0= |a15|....|a0| ;uso xmm0 para rojo
pshufb xmm0,xmm1
movq [rsi], xmm0 ;muevo a memoria 2 pixeles
add rsi, 8
psrldq xmm0, 8
movd [rsi], xmm0
jmp .no_faltan

.faltan_1:
movd xmm0, [rdi] ;xmm0= |a15|....|a0| ;uso xmm0 para rojo
pshufb xmm0,xmm1
movd [rsi], xmm0 ;muevo 1 pixel memoria
jmp .no_faltan

.faltan_2:
movq xmm0, [rdi] ;xmm0= |a15|....|a0| ;uso xmm0 para rojo
pshufb xmm0,xmm1
movq [rsi], xmm0 ;muevo a memoria
jmp .no_faltan


.no_faltan:

   add r10, r8 ;a rdi le sumo el ancho para apuntar a la proxima fila
   add r11, r9
   dec rax ;decremento el contador de filas
   jmp cicloFilas
```

3.3. Pixelar

Para aplicar el filtro, se recorre la imagen de a dos filas (ya que para ello se necesitas dos pixeles de la primera, y sus inmediatos superiores de la segunda) y de a 4 columnas/pixeles (dado que es el maximo que se puede guardar en un registro xmm). En el ciclo exterior recorremos cada par de filas, mientras que en el interior procesamos las columnas.

El algoritmo de procesamiento es el siguiente:

- Buscamos en memoria 4 pixeles por cada linea, px11, px12, px21, px22 y px13, px14, px23, px24. figura 6

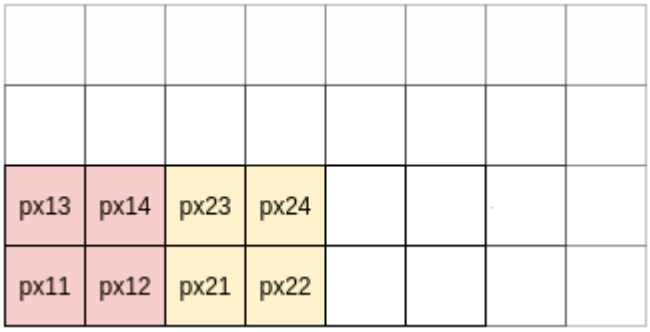


Figura 6: Etiquetas de pixels

- Desempaquetamos extendiendo cada canal, de byte a word, con ceros para obtener 4 registros xmm con cada par de pixeles.
- Utilizamos SIMD para sumar cada componente de los pixels entre si.
- Copiamos cada suma parcial a otro registro y hacemos un shuffle para cruzar los datos. figura 7

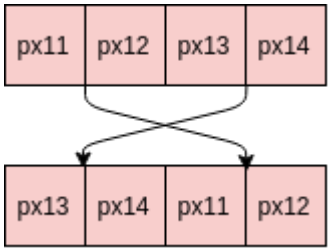


Figura 7: Shuffle

- Utilizando SIMD nuevamente sumamos estas sumas parciales y obtenemos dos registros con las sumas totales de cada grupo de pixeles.
- Hacemos un shift para dividir por cuatro las sumas y asi obtener el promedio entre los pixeles.
- Una vez obtenidos los promedios, desempaquetamos todo en un solo registro xmm, el cual contendra los pixeles finales a copiar. figura 8

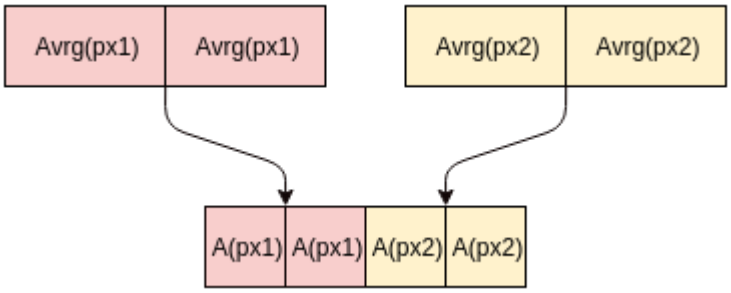


Figura 8: Unpack

- Una vez listo, copiamos los nuevos valores a la imagen destino. figura 9

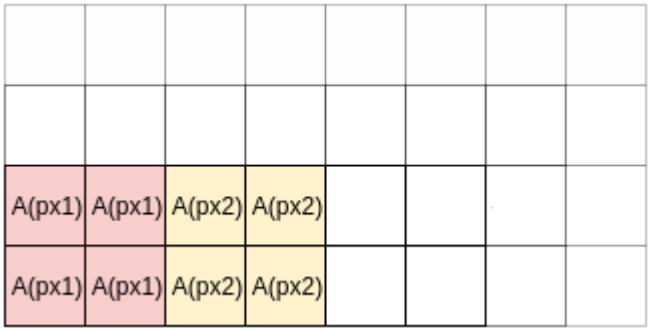


Figura 9: Destino

Codigo fuente:

```
98  movdqu xmm7, [rdi] ; |px11|px12|px21|px22|
99
100 movdqu xmm1, xmm7
101 punpcklbw xmm1, xmm6 ; |px11|px12|
102 movdqu xmm2, xmm7
103 punpckhbw xmm2, xmm6 ; |px21|px22|
104
105 movdqu xmm7, [rdi + rdx * 4] ; |px13|px14|px23|px24|
106
107 movdqu xmm3, xmm7
108 punpcklbw xmm3, xmm6 ; |px13|px14|
109 movdqu xmm4 , xmm7
110 punpckhbw xmm4, xmm6 ; |px23|px24|
111
112 paddw xmm1, xmm3 ; |px11 + px13|px12 + px14|
113 paddw xmm2, xmm4 ; |px21 + px23|px22 + px24|
114
115 movdqu xmm3, xmm1
116 shufpd xmm3, xmm1, 00000001b ; |px12 + px14|px11 + px13|
117
118 movdqu xmm4, xmm2
119 shufpd xmm4, xmm2, 00000001b ; |px22 + px24|px21 + px23|
120
121 paddd xmm1, xmm3 ; |px11 + px12 + px13 + px14|px11 + px12 + px13 + px14|
122 paddd xmm2, xmm4 ; |px21 + px22 + px23 + px24|px21 + px22 + px23 + px24|
123
124 psrlw xmm1, 2 ; |avrg(px1)|avrg(px1)|
125 psrlw xmm2, 2 ; |avrg(px2)|avrg(px2)|
126
127 packuswb xmm1, xmm2 ; |avrg(px1)|avrg(px1)|avrg(px2)|avrg(px2)|
128
129 movdqu [rsi], xmm1
130 movdqu [rsi + rdx * 4], xmm1
131
132 add rdi, 16
133 add rsi, 16
```

3.4. Combinar

La implementacion del filtro consiste en dos partes. En la primera, se procede a invertir la imagen. Para ello se recorre la fuente y, utilizando una mascara, se invierte el orden de los pixeles y se escribe en el destino. Una vez obtenido el espejo de la fuente en el destino, se procede a aplicar el filtro propiamente dicho. Dado que en los calculos de este filtro se utilizan floats, y mas allá de que por cada ciclo procesamos 4 pixels, el proceso simultaneo del filtro es de a un pixel; osea, un float de 4 bytes por cada canal. Para ahorrar instrucciones, antes de iniciar el ciclo, se calcula la division entre el alpha recibido y 255. Con este resultado, copiamos a un segundo registro y por medio de un shuffle lo replicamos en todo un registro, dejandolo listo para utilizar con operaciones de SIMD. El algoritmo de procesamiento es el siguiente:

- Buscamos en memoria 4 pixeles de cada imagen y los almacenamos en 2 registros xmm.
- Por medio de un desempaquetado, extendemos con ceros obteniendo 4 registros con 2 pixeles cada uno. figura 10

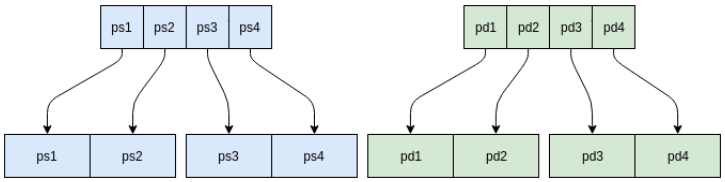


Figura 10: Detalle de desempaquetado

- Con los valores desempaquetados, utilizamos SIMD para realizar la resta entre los canales de los pixels.
- Extendemos nuevamente, esta vez de word a double. Para ello, utilizamos una comparación con cero, del cuál obtenemos una máscara para extender con signo las restas.
- Realizamos una conversion de cada registro a float.
- Una vez tenemos los datos en formato de float de presicion simple, realizamos la multiplicacion con el valor precalculado anteriormente.
- Con los pixeles y procesados, realizamos una nueva conversion para obtener nuevamente enteros.
- Por medio de dos instrucciones de pack (ambas utilizando saturacion y signo), pasamos los datos de double words a bytes. figura 11

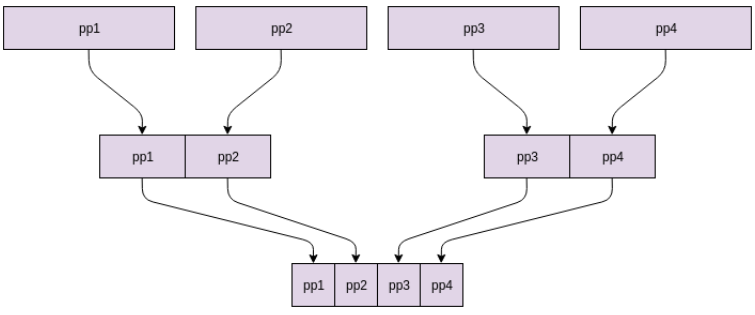


Figura 11: Detalle de empaquetado

- Realizamos una suma entre los datos empaquetados y los pixeles originales obtenidos de la imagen destino.
- En este punto ya poseemos todos los pixeles con el filtro aplicado, por lo que solo resta escribirlos en la imagen destino y continuar el ciclo.

Codigo fuente:

```
movdqu xmm1, [rdi] ; xmm1 = |px1s|px2s|px3s|px4s|
movdqu xmm2, [rsi] ; xmm2 = |px1d|px2d|px3d|px4d|
movdqu xmm5, xmm2  ; Lo guardo para utilizar luego.

pxor xmm7, xmm7

movdqu xmm8, xmm1
punpcklbw xmm8, xmm7  ; xmm8  = | px1s | px2s |
movdqu xmm9, xmm1
punpckhbw xmm9, xmm7  ; xmm9  = | px3s | px4s |

movdqu xmm12, xmm2
punpcklbw xmm12, xmm7 ; xmm12 = | px1d | px2d |
movdqu xmm13, xmm2
punpckhbw xmm13, xmm7 ; xmm13 = | px3d | px4d |

psubw xmm8, xmm12     ; xmm8  = | px1s - px1d | px2s - px2d |
psubw xmm9, xmm13     ; xmm9  = | px3s - px3d | px4s - px4d |

movdqu xmm1, xmm8
movdqu xmm2, xmm8
pxor xmm7, xmm7
pcmpgtw xmm7, xmm1
punpcklwd xmm1, xmm7  ; xmm1 = | px1s - px1d |
punpckhwd xmm2, xmm7  ; xmm2 = | px2s - px2d |

movdqu xmm3, xmm9
movdqu xmm4, xmm9
pxor xmm7, xmm7
pcmpgtw xmm7, xmm3
punpcklwd xmm3, xmm7  ; xmm3 = | px3s - px3d |
punpckhwd xmm4, xmm7  ; xmm4 = | px4s - px4d |

cvtdq2ps xmm1, xmm1  ; xmm1 = | f(px1s - px1d) |
cvtdq2ps xmm2, xmm2  ; xmm2 = | f(px2s - px2d) |
cvtdq2ps xmm3, xmm3  ; xmm3 = | f(px3s - px3d) |
cvtdq2ps xmm4, xmm4  ; xmm4 = | f(px4s - px4d) |

mulps xmm1, xmm0  ; xmm1 = | f(px1s - px1d) * d |
mulps xmm2, xmm0  ; xmm2 = | f(px2s - px2d) * d |
mulps xmm3, xmm0  ; xmm3 = | f(px3s - px3d) * d |
mulps xmm4, xmm0  ; xmm4 = | f(px4s - px4d) * d |

; p() = procesado = ((pxXs - pxXd) / d)
cvtps2dq xmm1, xmm1  ; xmm1 = | p(px1) |
cvtps2dq xmm2, xmm2  ; xmm2 = | p(px2) |
cvtps2dq xmm3, xmm3  ; xmm3 = | p(px3) |
cvtps2dq xmm4, xmm4  ; xmm4 = | p(px4) |

packssdw xmm1, xmm2  ; xmm1 = | p(px1) | p(px2) |
packssdw xmm3, xmm4  ; xmm3 = | p(px3) | p(px4) |

packsswb xmm1, xmm3  ; xmm1 = | p(px1) | p(px2) | p(px3) | p(px4) |

paddb xmm1, xmm5 ; xmm1 = | p(px1) + px1d | p(px2) + px2d | p(px3) + px3d | p(px4) + px4d |

movdqu [rsi], xmm1 ; Guardo el resultado final en memoria
```

3.5. Colorizar

Este filtro pocesa nueve pixeles de la imagen fuente para obtener uno en la imagen destino (cada pixel necesita de los 8 pixeles vecinos). Por esta razón en cada iteración se obtiene un pixel en la imágen destino.
Suponiendo que se quiera procesar un pixel de una imagen al que por comodidad llamaremos $pixel(5)$ (va a representar al pixel a procesar en todo momento, independientemente de qué pixel se esté procesando en un momento dado). Entonces se considera la siguiente matriz formada por el $pixel(5)$ y sus vecinos: figura 12 Entonces en cada iteración se extrae la fila del $pixel(5)$ y la fila anterior y posterior a ella. Para

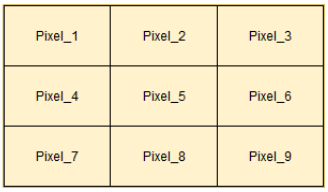


Figura 12: Representa el pixel 5 y sus vecinos

no perder presición se hacen conversiones a floats. Una vez procesados los 9 pixeles se obtiene como resultado el pixel que irá en la misma posición que el $pixel(5)$ en la imagen destino. Antes de el ciclo se calcula el $1 + alpha$ y el $1 - alpha$, que son dos resultados que se usará en cada iteración del ciclo.

El algoritmo de procesamiento es el siguiente:

- Buscamos en memoria los 9 pixeles de la matriz (3 filas de 3 pixeles) y los almacenamos en 3 registros xmm.
- Por cada fila de la matriz se hace exactamente lo mismo:
  - Cada fila de la matriz contiene 4 pixeles (interesan los 3 pixeles de la parte menos significativa), entonces se desempaquetan de byte a word para que cada canal (r,g,b) ocupe un word como muestra la figura 10.
  - Una vez desempaquetado a word se hacen comparaciones de a words para ir quedándose con el mayor. Como ejemplo esta es la comparación para la primera fila (pixeles 1,2,3 de la matriz). figura 13

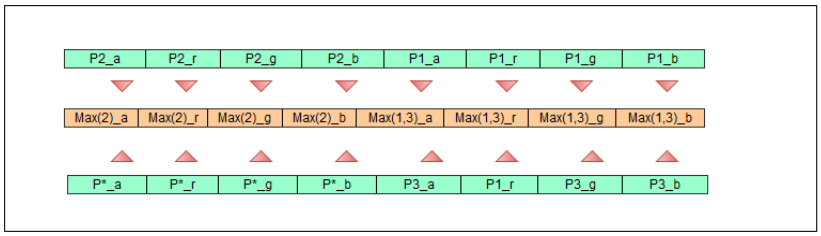


Figura 13: Comparacion de Pixeles 1,2 y 3

  - Al resultado de la comparación de la primera fila se compara con el resultado de la segunda y luego con el de la tercera, como muestra la figura 14

- Luego de comparar todas los pixeles de la matriz se desempaqueta a dword quedando cada componente del pixel de tamaño dword y dentro de un registro xmm como se muestra a continuación: 15
- Luego para cada componente por separado se le aplica la función phi usando el resultado de de los máximos obtenidos. Para lograr mayor presición se hace una conversión a floats.
- Una vez terminado el paso anterior se vuelve a convertir a enteros y se empaqueta todo el pixel (cada componente es un dword) de dword a word, luego de word a byte, saturando sin signo, de esta manera si la componente pasa el valor máximo, quedará saturada.

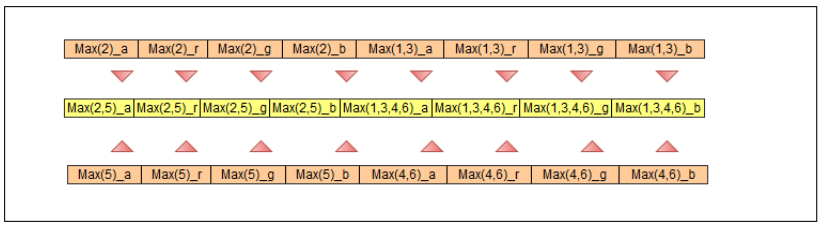


Figura 14: Comparacion entre filas 1 y 2

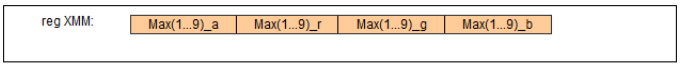


Figura 15: Resultado los máximos por componente

- Se guarda el pixel resultante en la matriz destino en la posición correspondiente.

Codigo fuente:

```
; xmm0, xmm1 y xmm2 van a guardar las tres filas de la matriz que contienen los pixeles
movdqu xmm1, [rdi] ;xmm1 = | ---- | pixel3 | pixel2 | pixel1 |
movdqu xmm2, [rdi+r8] ;xmm2 = | ---- | pixel6 | pixel5 | pixel4 |
movdqu xmm3, [rdi+r8*2] ;xmm3 = | ---- | pixel9 | pixel8 | pixel7 |

movdqu xmm4, xmm1 ;xmm4 = | --- | a3,r3,g3,b3 | a2,r2,g2,b2 | a1,r1,g1,b1 |
punpcklbw xmm4, xmm7 ;xmm4 = |Pixel2(a,r,g,b) | Pixel1(a,r,g,b)|
pslldq xmm1, 4
psrldq xmm1, 12 ;xmm1 = | *** | *** | *** | Pixel3(a,r,g,b) |
punpcklbw xmm1, xmm7 ;xmm1 = | *** | Pixel3(a,r,g,b) |
movdqu xmm5, xmm4
pcmpgtw xmm5, xmm1 ;xmm5 = el resultado de la comparacion (máscara de la comparacion)
pand xmm4, xmm5
pxor xmm5, xmm8 ;invierto los bits del resultado
pand xmm1, xmm5 ;el resultado de ls componentes mayores entre pixeles 1,2 y 3
por xmm1, xmm4 ; xmm1 = | Pixel2(a,r,g,b) | max_pixel_1_3(a,r,g,b) |
.....
.se hace lo mismo para xmm2 y xmm3 que guardan la 2da y tercera fila de la matriz
....
....
.Una vez obtenido los maximos se hace lo siguiente
; aplico las funciones fi con cada componente
movq r12, xmm1 ;r12 = el maximo b
movq r13, xmm2 ;r13 = el maximo g
movq r14, xmm3 ;r14 = el maximo r
pxor xmm1, xmm1
call fib
pxor xmm2, xmm2
call fig
pxor xmm3, xmm3
call fir

.....
....
donde cada fi hace lo siguiente
.....
.....
;xmm11 = 1+alpha    xmm12= 1 - alpha
fib:
cmp r14, r12
jge .no3
cmp r13, r12
jge .no3
movdqu xmm1, xmm11
jmp .finb
.no3:
movdqu xmm1, xmm12
.finb:
ret

fig:
cmp r14, r13
jge .no2
cmp r13, r12
jl .no2
movdqu xmm2, xmm11
jmp .fing
.no2:
movdqu xmm2, xmm12
.fing:
ret

fir:
```

4. Análisis de las implementaciones

5. Hipotesis General

Se espera que el rendimiento de los filtros realizados en ASM sea mejor que los hechos en C, ya que las operaciones SIMD deberían mejorar el rendimiento de forma notable. Además se espera que los filtros colorizar y combinar pierdan precisión ya que para poder computar más datos a la vez conn la instrucciones SIMD se trabaja con floats en lugar de double lo cual baja la precisión de dichos calculos.

5.1. Diseño experimental

Para ejecutar los experimentos, mas adelante detallados, se utilizó una máquina virtual en VirtualBox corriendo bajo Ubuntu 16.04. Dicha máquina se encuente corriendo sobre un equipo con un procesador Intel Core I7-4700HQ bajo Windows 10. Los detalles del procesador son los siguientes:

```
Datos extraídos de lscpu:

Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                1
On-line CPU(s) list:   0
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Model name:            Intel(R) Core(TM) i7-4700HQ CPU @ 2.40GHz
Stepping:              3
CPU MHz:               2394.466
BogoMIPS:              4788.93
Hypervisor vendor:     KVM
Virtualization type:   full
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K

Datos extraidos de cachegrind:

I1 cache:         32768 B, 64 B, 8-way associative
D1 cache:         32768 B, 64 B, 8-way associative
LL cache:         6291456 B, 64 B, 12-way associative

I1 = Instruction L1 caché
D1 = Data L1 caché
LL = L3 caché
```

Para evitar ruido en las mediciones, se cerraron todas las aplicaciones de usuario dejando unicamente una terminal abierta; donde se corrieron varias veces los mismos experimentos con 100 iteraciones.
Donde no se especifica los parametros de combinar se usó como parametro para combinar 128 y para colorizar 0.4

A continuacion describimos cómo tomamos las mediciones de nuestros experimento:

- Para capturar el tiempo demorado por el programa, se utilizó el comando time, asegurandonos de correr varias iteraciones por en cada ejecucion, para obtener una medida lo más precisa posible.

- En el caso de los ciclos utilizamos lo ya provisto por la cátedra en el código brindado.
- Para medir las estadísticas de caché utilizamos la herramienta cachegrind de Valgrind, de donde obtuvimos el detalle de hits y misses por archivo de codigo fuente para cada nivel de caché. Para medir la cantidad de instrucciones que se realizan utilizamos la herramienta cachegrind de Valgrind.

5.2. C vs ASM

Se realizará la comparación de cada implementación de ASM con su respectiva implementación de C para asi verificar la mejoría de rendimiento que tiene la primera implementacíon sobre la segunda. Para realizar esta comparación del rendimiento se compará el tiempo de ejecución con el tamaño de la imagen. El tamaño de la imagen de entrada irá aumentando de forma tal que se mantenga las proporciones cuadradas de la imágen.

5.2.1. Comparación de tiempo de ejecución de las implementaciones en C y ASM

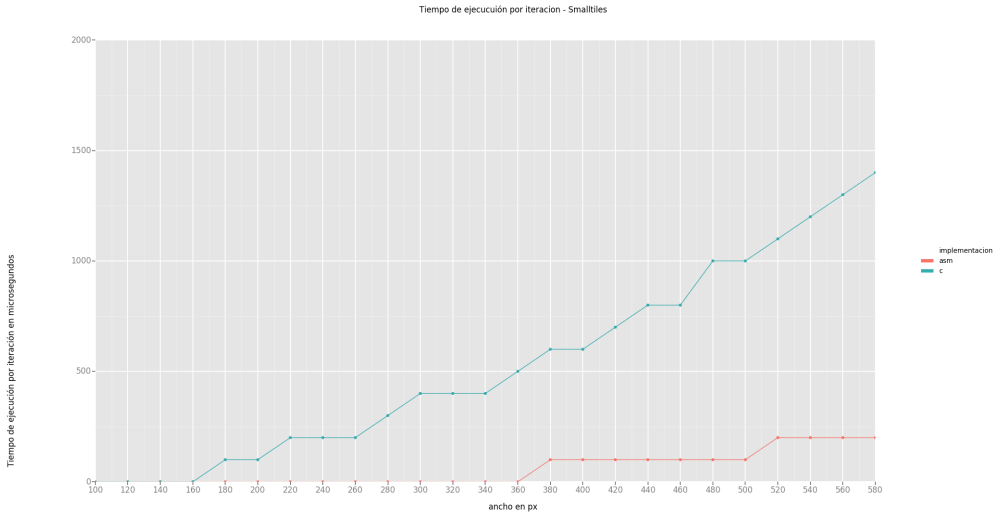


Figura 16: Comparación de tiempo de ejecución en C vs ASM de Smalltiles

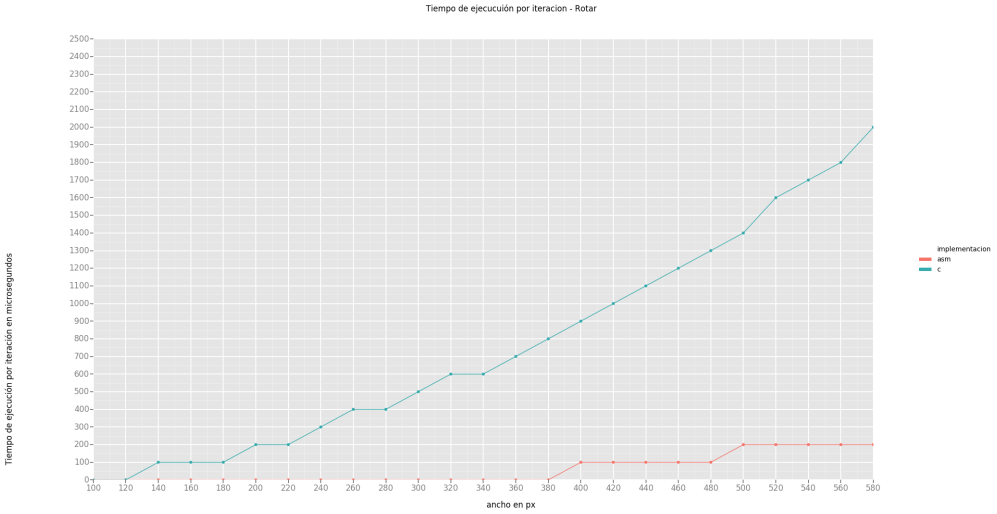


Figura 17: Comparación de tiempo de ejecución en C vs ASM de Rotar

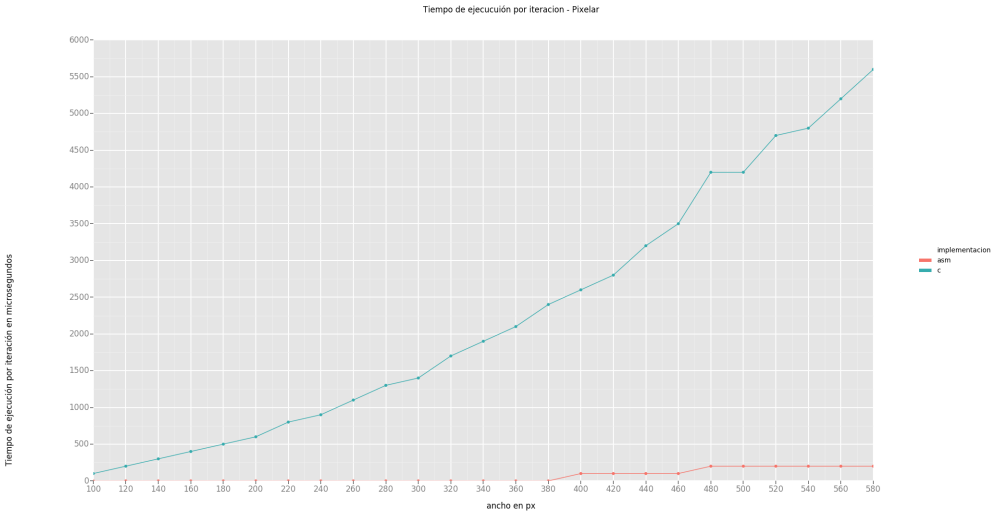


Figura 18: Comparación de tiempo de ejecución en C vs ASM de Pixelar

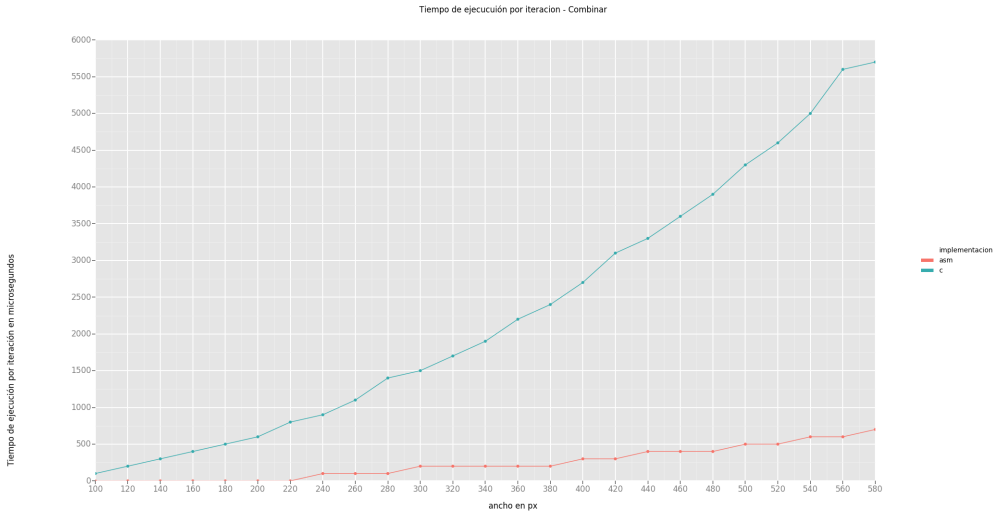


Figura 19: Comparación de tiempo de ejecución en C vs ASM de Combinar

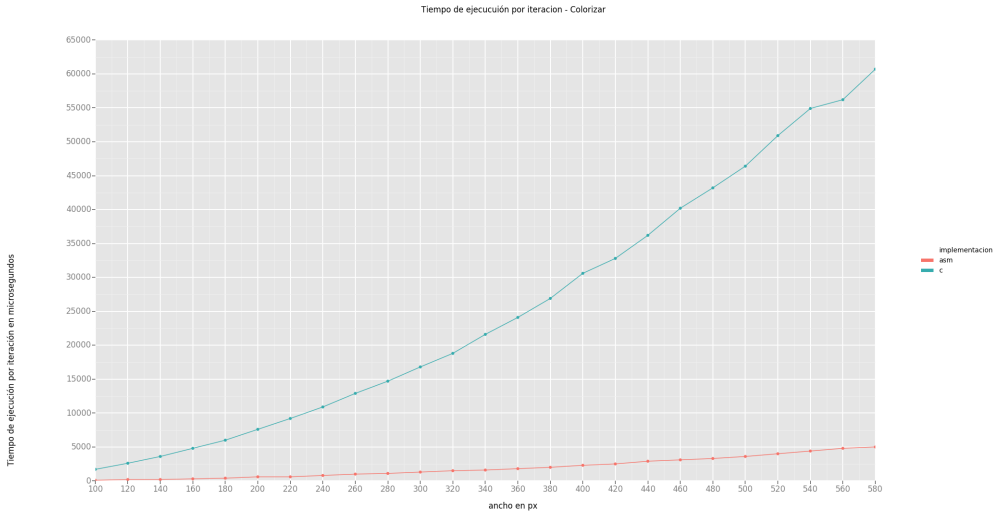


Figura 20: Comparación de tiempo de ejecución en C vs ASM de Colorizar

Estos gráficos son extremadamente similares entre ellos con la única diferencia notable siendo la escala. En estos gráficos se puede comprobar que efectivamente hay una mejora notable en el rendimiento de modo tal que la implementación en ASM rinde mucho mejor que la de C automaticamente optimizada por el compilador con la opcion O3. Esta diferencia se vuelve mas evidente cuanto mayor es el tamaño de la imágen.

5.2.2. Comparacion de la cantidad de instrucciones por pixel de cada una de las implementaciones

Se espera que en todos los casos la implementación en ASM tengan menor cantidad de instrucciones por pixel y que la cantidad de instrucciones crezca de forma lineal con la cantidad de pixeles de la imagen. Para verificar eso se utilizara un grafico de cantidad de pixeles en relacion a cantidad de instrucciones. Para hacerlo las imágenes usadas serán cuadradas.

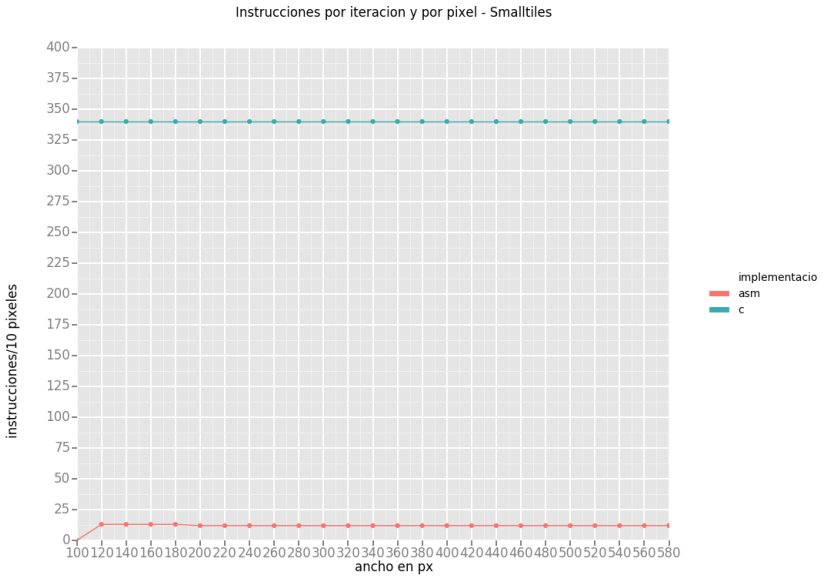


Figura 21: instrucciones por pixel en C vs ASM de Smalltiles

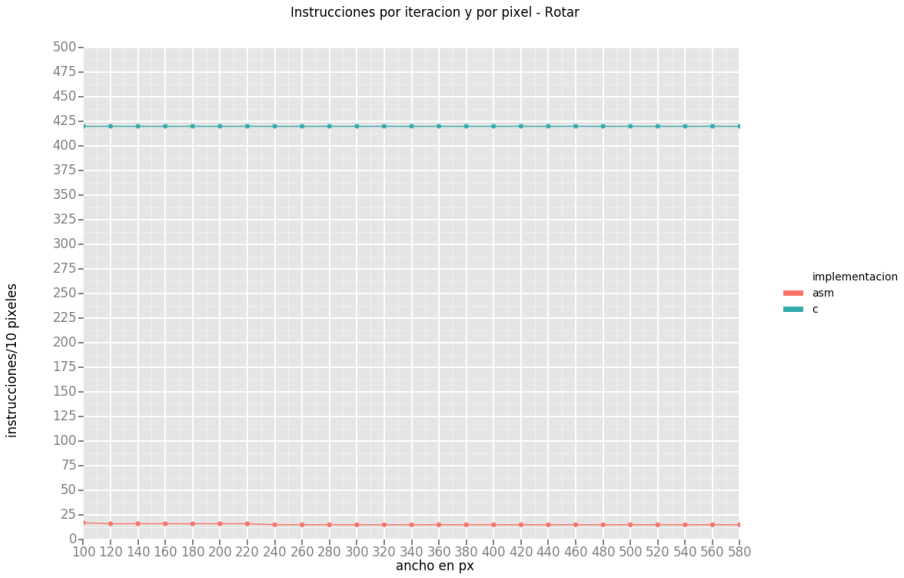


Figura 22: instrucciones por pixel en C vs ASM de Rotar

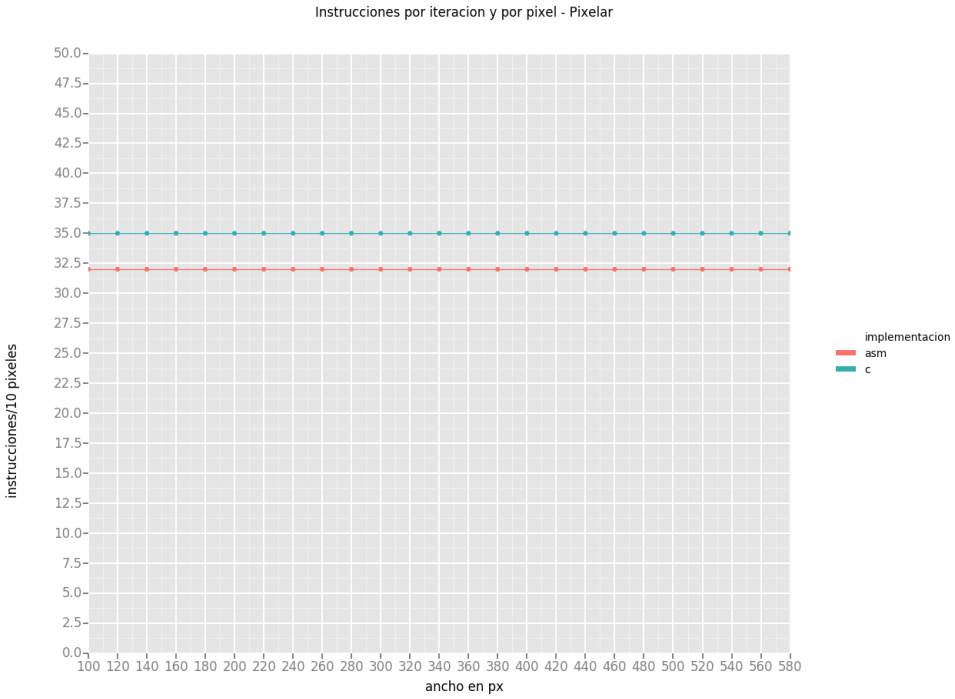


Figura 23: instrucciones por pixel en C vs ASM de Pixelar

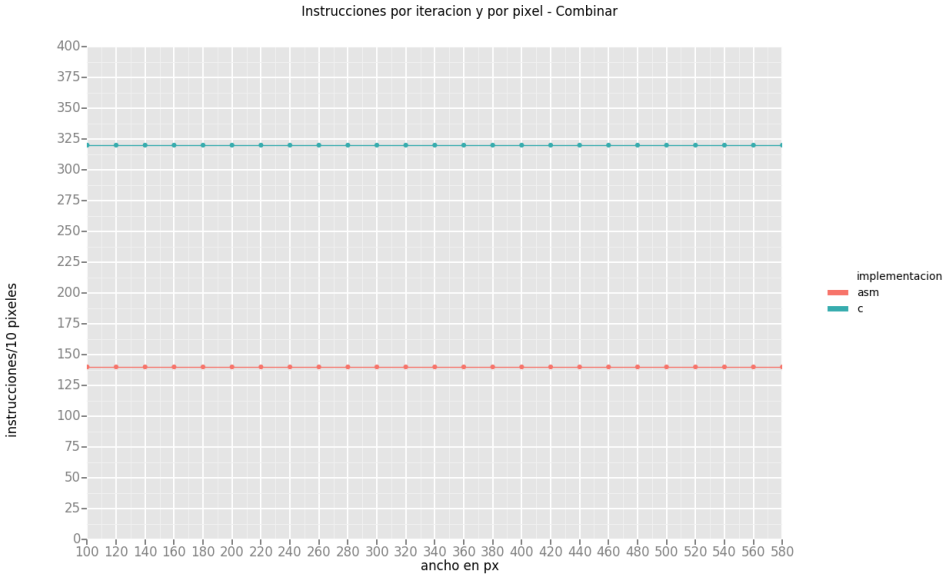


Figura 24: instrucciones por pixel en C vs ASM de Combinar

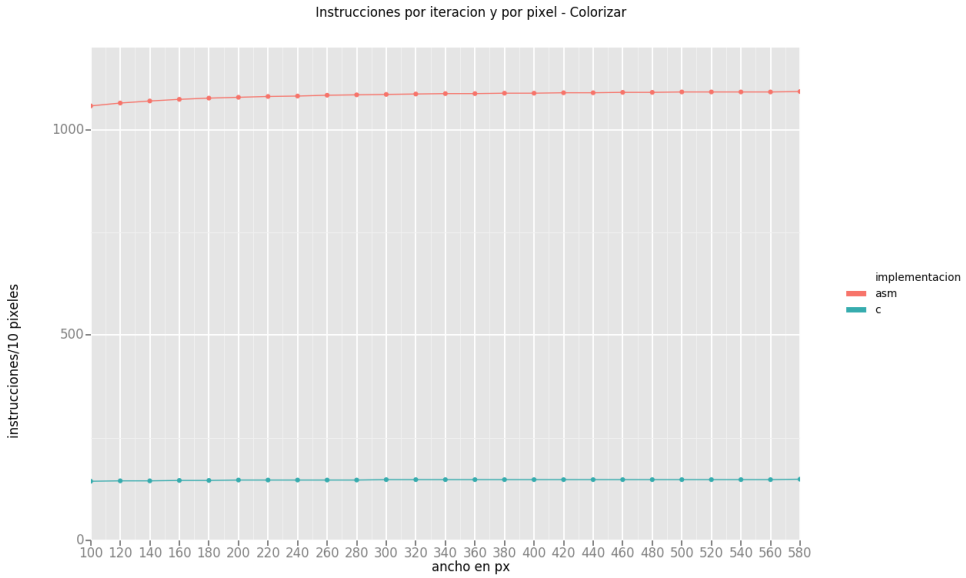


Figura 25: instrucciones por pixel en C vs ASM de Colorizar

Efectivamente la relación entre cantidad de pixeles y cantidad de instrucciones es lineal pero lo mas interesante que se extraé de esto es el hecho de que la implementacion en ASM de colorizar ejecuta más instrucciones pero igualmente es más rápida lo cuál indica que las instrucciones en la implementación en ASM de dicha función se ejecutan en promedio mucho más rapida que en la versión de C.

5.3. Diferencias en la imagen final generada con el código en C y ASM al tababajar con puntos flotantes

Es importante notar que solo se trabaja con puntos flotantes para las implmentaciones de los filtros combinar y colorizar por lo que son los único en los que debería presentarse diferencias al comparar las imagenes obtenidas en C y en ASM con el comparador proveido por la catedra. Al correr el comparador con la imagenes obtenidas al correr cada implementación filtro con la imagen de Lena proporcionada, se llega a que no se presentan ninguna diferencia en los filtros Smalltiles, Rotar y Pixelar, verificando de esa manera la hipótesis. Por otro lado al comparar las obtenidas en Combinar y Colorizar no se presentan diferencias al correr el comparador con tolerancia de 1 pero al correrlo sin tolerancia si se presentan varias. Eso indica que al bajar la precisión de los floats la imágen efectivamente se altero pero no de forma apreciable ya que una variación de 1 en la intensidad de algún color es indistinguible a simple vista. En conclusión, la perdida de precisión producida es despreciable para los propósitos con los que usualmente se usa una imágen.

5.4. Overhead de llamado a las funciones en C

Se supondría que el overhead de llamado a estas funciones es despreciable en comparación al tiempo que tardan al correr con imágenes de tamaño razonable. Igualmente se verificará que efectivamente es así. No se podrá medir realmente el valor del overhead pero se tomará un valor chico que incluye al overhead por lo que se establecera una cota superior para el overhead. De ser ese valor despreciable el overhead también lo sería. Para hacer esto se tomará una imagen de lena de 1x1 convertida de Lena que es la mínima que aceptan las funciones Smalltiles, Rotar, Pixelar y Combinar. También se usará una imagen de 3x3 para Colorizar ya que rechaza tamaños mas chicos. Una vez tomada las mediciones de tiempo de las mismas se compará con el tiempo que tardan en correr con la entrada siendo la imagen de lena en 20 x 20. Las mediciones de tiempo se harán en base a la cantidad de ciclos insumidos por llamada de la función, las cuales serán corriedas 100000 veces para reducir los errores de medición.
Al hacerlo se obtiene:
Smalltiles: 30 ciclos en 1x1 contra 5370 ciclos en 20x20
Rotar 40 ciclos en 1x1 contra 7130 en 20x20
Pixelar: 200 ciclos en 1x1 contra 17950 ciclos en 20x20
Combinar:95 ciclos en 1x1 contra 18870 ciclos en 20x20
Colorizar:450 ciclos en 3x3 contra 119520 ciclos en 20x20
Al mirar estos resultados efectivamente se puede comprobar que el overhead del llamado a las funciones es absoultamente despreciable como se esperaba.

5.5. Overhead de llamado a las funciones en ASM

De la misma manera que en el caso de las funciones de C las de ASM deberian tener un overhead despreciable. Igualmente se verificará que efectivamente es así. No se podrá medir realmente el valor del overhead pero se tomará un valor chico que incluye al overhead por lo que se establecera una cota superior para el overhead. De ser ese valor despreciable el overhead también lo sería. Para hacer esto se tomará una imagen de lena de 4x4 convertida de Lena ya es un tamaño en que todas las funciones corren adecuadamente. Una vez tomada las mediciones de tiempo de las mismas se compará con el tiempo que tardan en correr con la entrada siendo la imagen de lena en 20 x 20. Las mediciones de tiempo se harán en base a la cantidad de ciclos insumidos por llamada de la función, las cuales serán corriedas 100000 veces para reducir los errores de medición.
Al hacerlo se obtiene:
Smalltiles: 95 ciclos en 4x4 contra 750 ciclos en 20x20

Rotar 100 ciclos en 4x4 contra 535 en 20x20
Pixelar: 95 ciclos en 4x4 contra 730 ciclos en 20x20
Combinar:200 ciclos en 4x4 contra 1975 ciclos en 20x20
Colorizar:265 ciclos en 4x4 contra 12510 ciclos en 20x20
Al mirar estos resultados se puede notar que la diferencia entre una implementación y otra es menos notable que en la comparación de C pero probablemente eso se deba al tamaño minimo de imagen usado. Sin embargo la diferencia sigue siendo bastante apreciable por lo que el overhead puede considerarse que no tiene importancia.

5.6. Caché y alineamiento de memoria

Hipótesis

Sabemos que el objetivo de la caché de un procesador es mejorar el rendimiento del mismo, pudiendo evitar el desperdicio de ciclos mientras se espera que la memoria retorne los datos requeridos. Si el dato que necesitamos de la memoria principal, ya se encuentra en caché lo llamamos un hit; caso contrario, decimos que tenemos un miss.

Debido a limitaciones obvias de costo, no podemos almacenar todo lo que querriamos en la caché, por lo tanto, siempre vamos a tener algún miss de caché en casi cualquier algoritmo (con la excepcion de aquellos que donde todos los datos necesarios entran en caché). Bajo esta premisa, podemos pensar en analizar qué pasaría si exprimimos la caché al máximo y que resultados obtendriamos en cuanto a la mejora de rendimiento de nuestro programa.

Lo que nos proponemos a analizar es:

- Como varía la cantidad de hits y misses de acuerdo a los tamaños de entrada de nuestras imagenes fuente.
- Cuanto rendimiento podemos obtener si maximizamos la proporcion de hits en un programa.

5.6.1. Experimentacion preliminar

En primera instancia, realizamos pruebas preliminares sobre los filtros realizados en su implementación ASM para tener una primera imagen de cómo se comportaban en cuanto a tiempos de ejecución y estadísticas de caché.

Todas las imagenes de entrada poseran casi la misma cantidad de pixeles totales (tomamos como base una imagen de 512x512), para mantener integridad entre los resultados. La variación se realizó ajustando el ancho y el alto de la imagen para cumplir que $0 \leq (ancho * alto - 262144) \leq 64$ (consideramos que en una imagen de 262144 pixeles, un delta de 64 pixeles es despreciable). Normalizando los tamaños de las imagenes de entrada, podemos comparar fehacientemente cómo influye el tamaño de entrada en el rendimiento de la caché, asi como poder utilizar estas mismas imagenes más adelante para comparar los tiempos entre cada filtro.

Los resultados obtenidos en la primera prueba (figura 26) resaltan la diferencia que hace el algoritmo utilizado en la proporcion de misses que tendrá el programa.

En nuestras implementaciones de combinar y colorizar, las cuales hacen el mayor uso posible de la localidad espacial (ya que procesa todos los datos de fila en fila), se ve cómo se mantiene casi completamente constante la proporción de misses.

Por otro lado, tenemos los casos de pixelar y rotar los cuales siguen un patron bastante similar. Por su lado, rotar, debido a que en nuestra implementación procesamos de a columnas, se observa una diferencia notable con respecto a combinar, con quien compartiría mayor semejanza en caso de haber optado por procesar de a filas. Es decir que de haber trabajado con filas se hubiera obtenido un aumento del rendimiento al tener una menor cantidad de misses.

El caso más interesante es el de smalltiles, dado que ya de por sí sabemos que iba a tener un comportamiento muy particular debido a los grandes saltos de memoria que hace al escribir la imagen final. En el grafico se puede observar facilmente la notable diferencia que se logra una vez que el ancho de la imagen supera los 16 pixeles, que casualmente concuerda con el tamaño de una linea de caché (64 Bytes).

También es importante mencionar que colorizar tiene un hitrate anormalmente alto lo cual probablemente se debe a que al utilizar los datos 9 veces, cada vez luego de la primera ya estaría en cache por lo cual se conseguiría mejor hitrate.

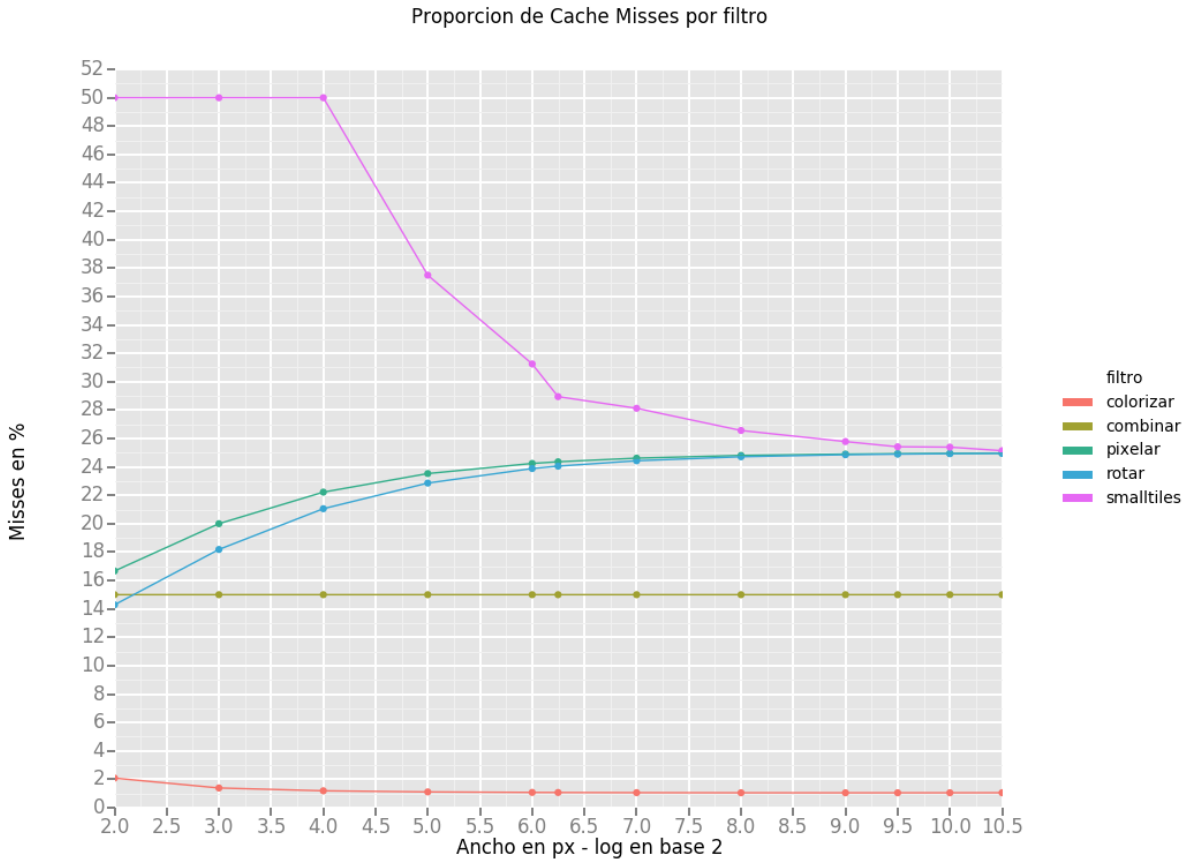


Figura 26: Comparación de misses entre filtros

Analisis de tiempos, combinar es esperable que tarde mas tiempo porque tiene mas procesamiento y floats. Puntos interesantes, pixelar y rotar es casi un espejo invertido comparado con el cache. Smalltiles tiene comportamientos particulares en algunos puntos. Mientras tanto colorizar crece en su tiempo de ejecución enormemente a medida que la imagen se va volviendo mas cuadrada aunque el uso de cache aparenta mantenerse parejo cuando se observa esta imágen pero cuandoo se ve en más detalle en los valores chivos se ve que la cantidad de misses decrece extremadamente rápido lo cual conicide con el comportamiento del tiempo de ejecución recién visto.

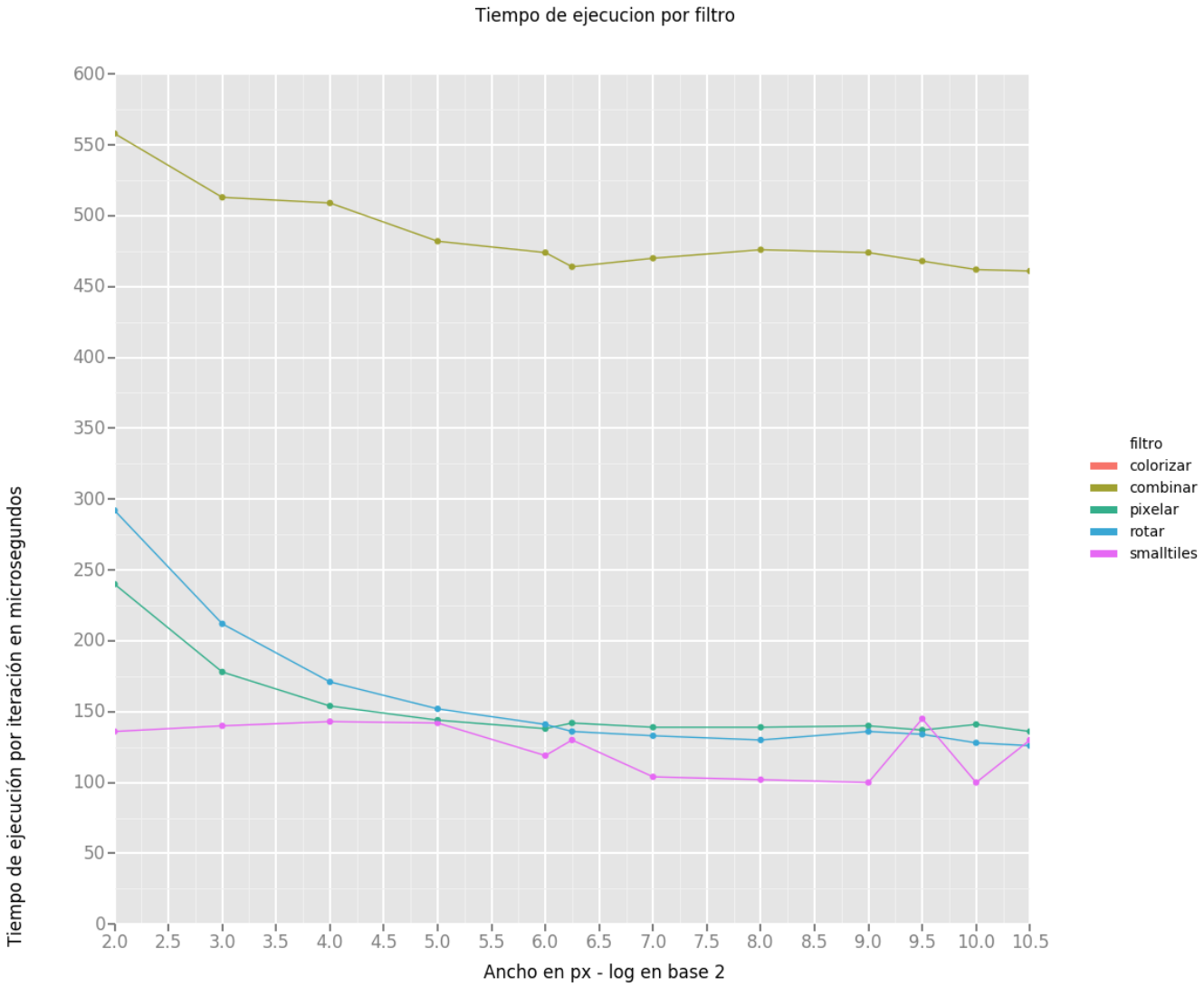


Figura 27: Tiempo de ejecucion entre filtros

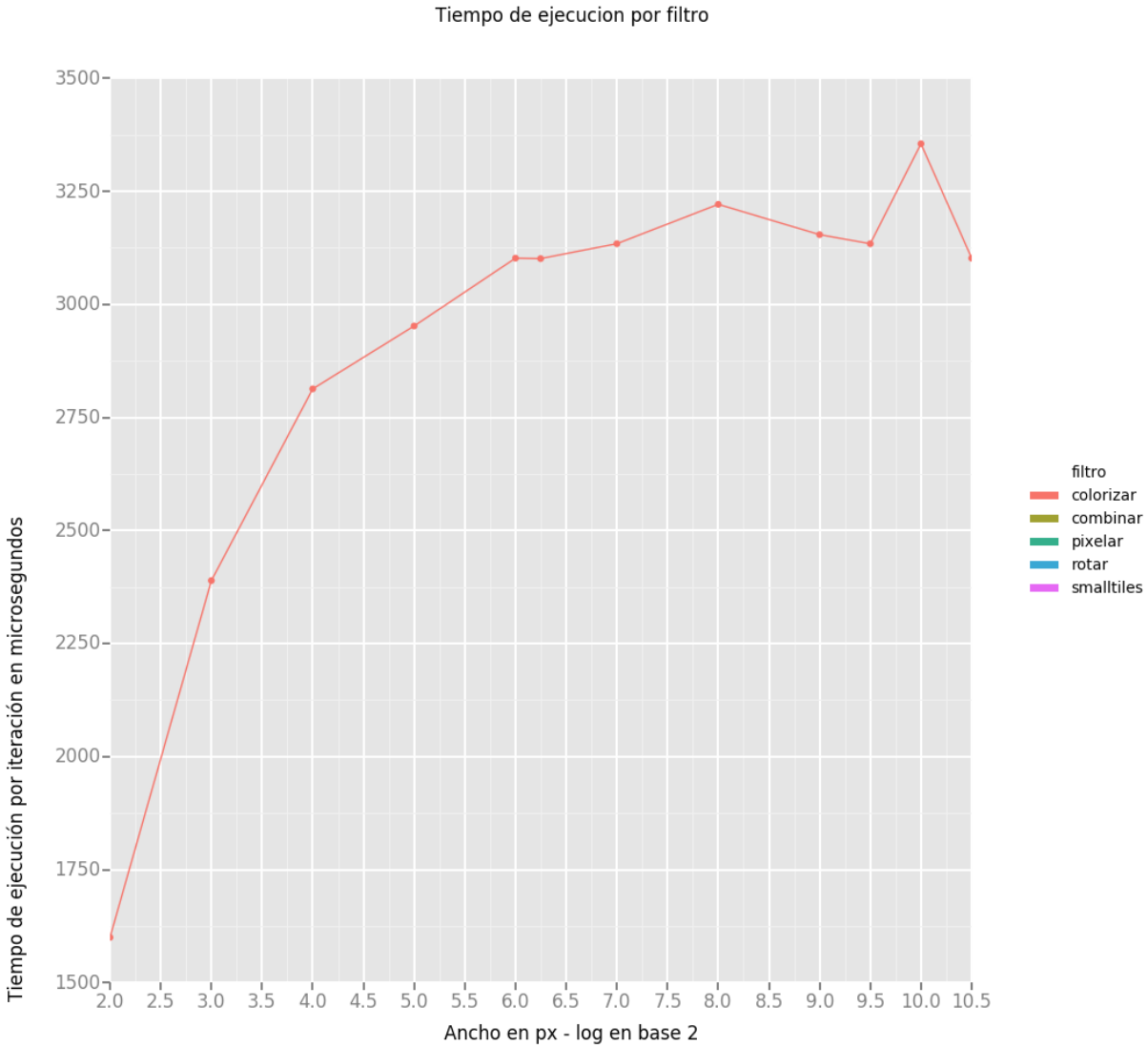


Figura 28: Tiempo de ejecucion colorizar

Comportamiento en anchos muy pequeños

Ahora se analizará el comportamiento en anchos pequeños lo cuál servira para ver el efecto de la distribución de los datos en memoria, influyendo en que no sea menos informacion en una lectura por el padding o se lean datos de forma desalineada, sobre el porcentaje de misses y el tiempo de ejecución.

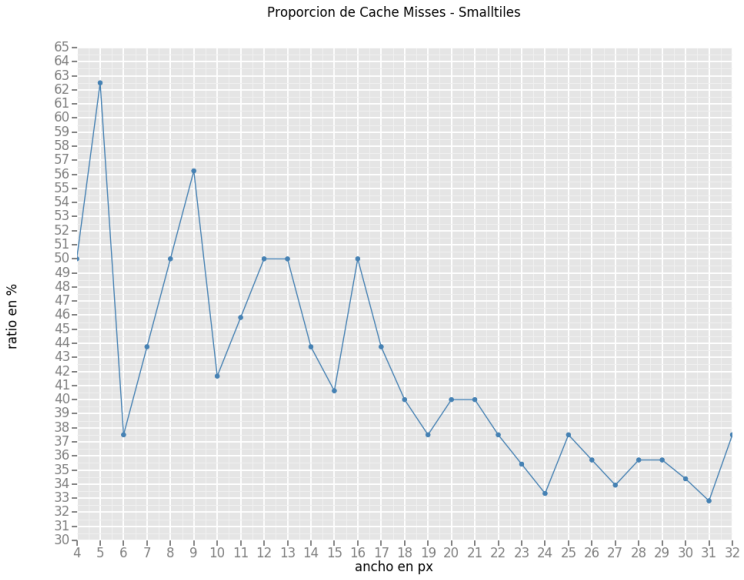


Figura 29: Misses-smalltiles

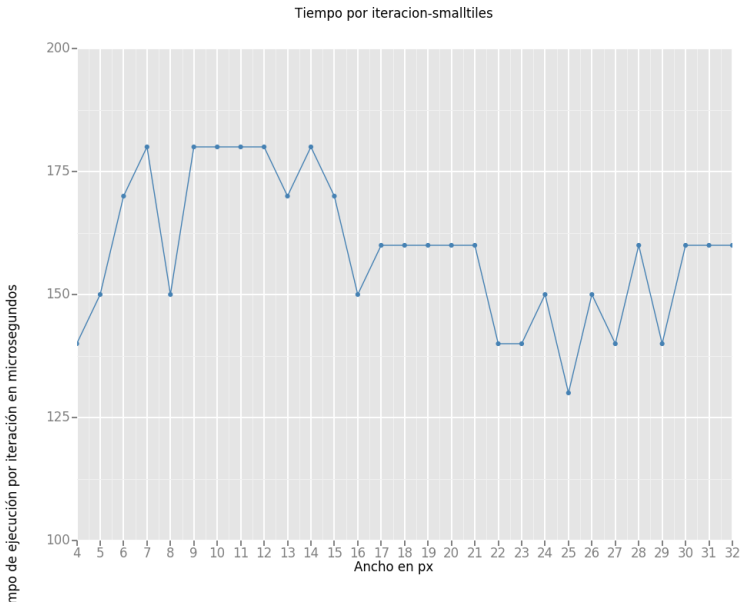


Figura 30: Tiempo de ejecucion-smalltiles

En estas figuras puede verse que la cantidad de misses en smalltiles tiene cierta correlacion con el tiempo de ejecución pero no es directa

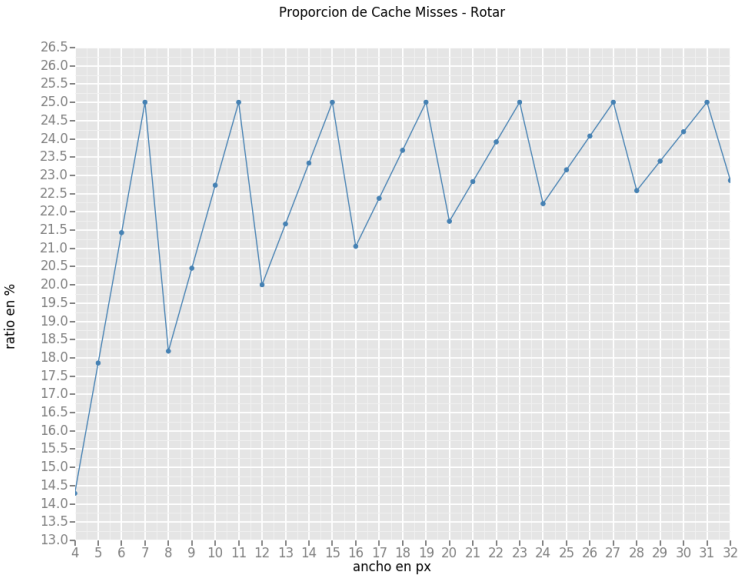


Figura 31: Misses-rotar

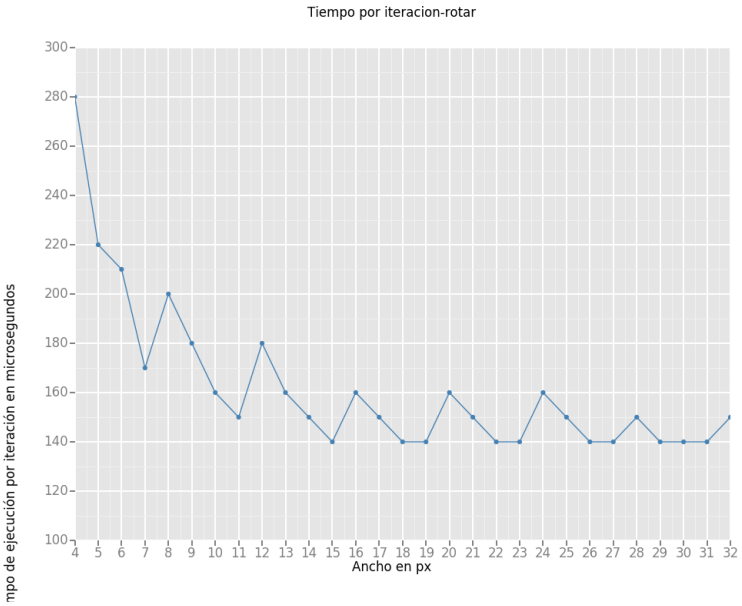


Figura 32: Tiempo de ejecucion-rotar

En estas figuras se puede ver que el tiempo de ejecución de rotar es un espejo con la cantidad de misses.

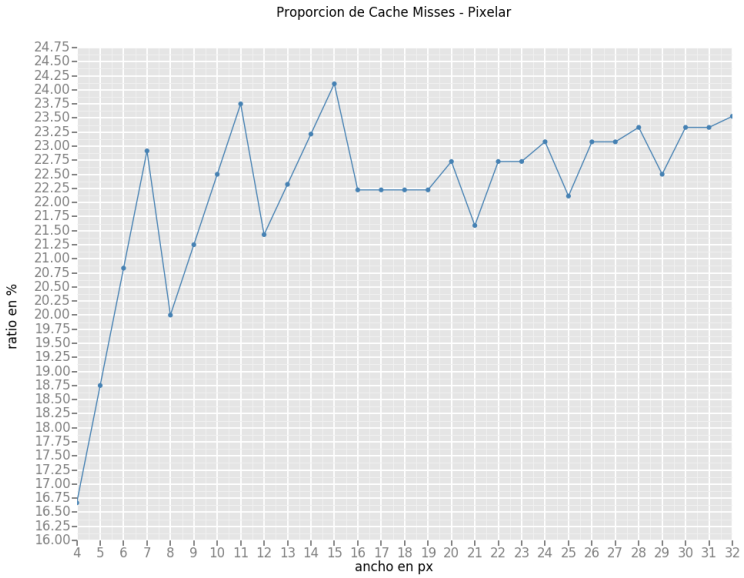


Figura 33: Misses-pixelar

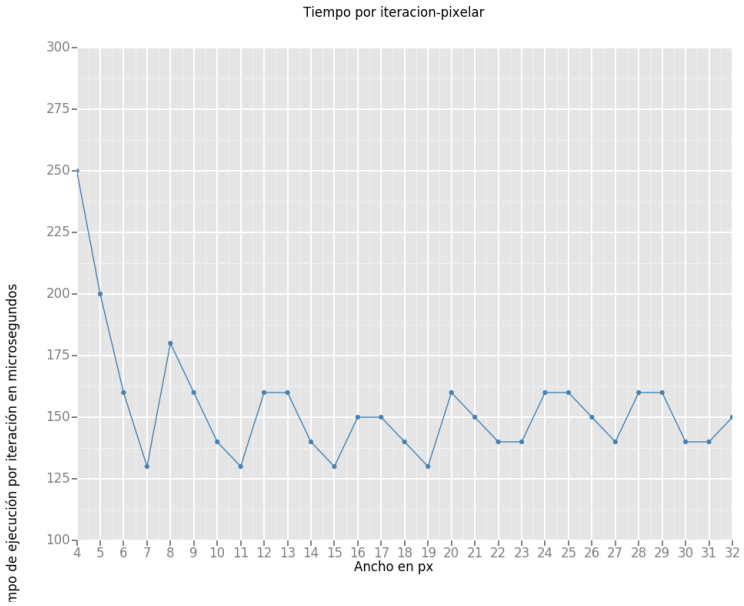


Figura 34: Tiempo de ejecucion-pixelar

Qquí tambíen es un espejo el tiempo de ejecución con la cantidad de misses.

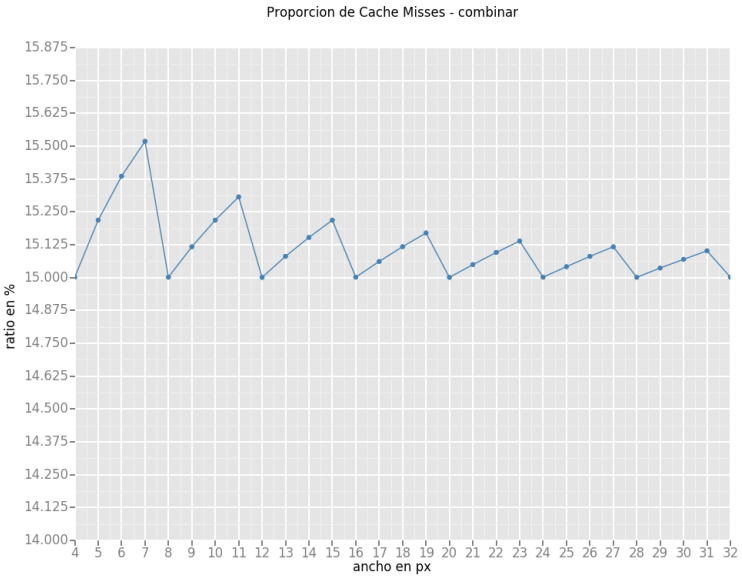


Figura 35: Misses-combinar

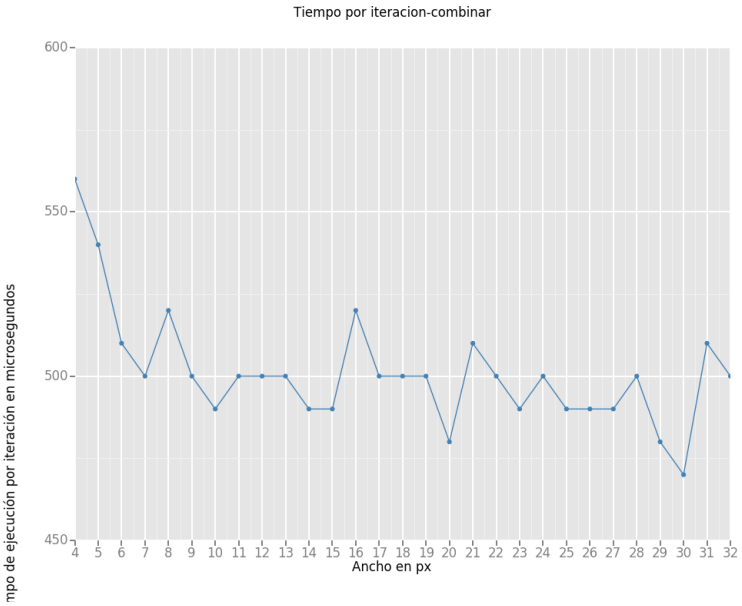


Figura 36: Tiempo de ejecucion-combinar

Combinar tiene un comportamiento similar pero no tiene una relación directa.

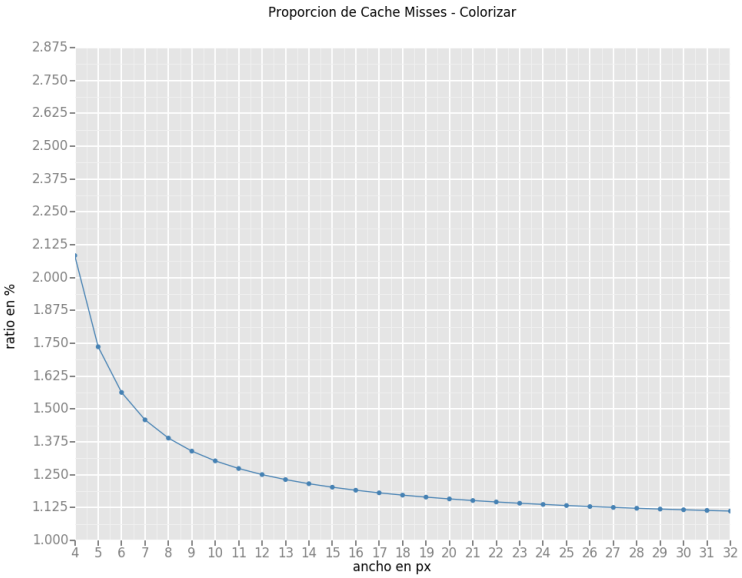


Figura 37: Misses-colorizar

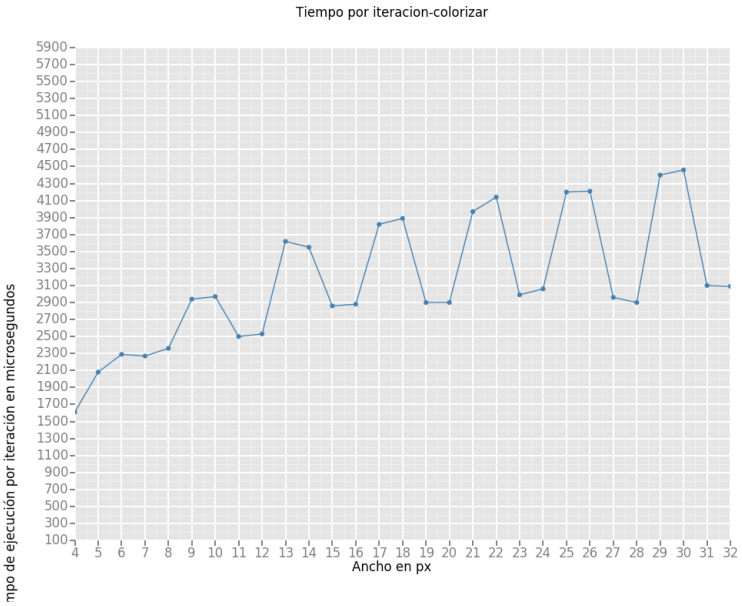


Figura 38: Tiempo de ejecucion-colorizar

Aquí se puede ver que el tiempo de ejecición de colorizar no tiene ningun parecido con el porcentaje de misses.

Se puede observar como en la mayoría de estos gráficos la cantidad de misses sube y baja periodicamente reptiendose cada 4 de ancho, lo cual tiene sentido por la diferencia en el alineamiento. El único que tiene un comportamiento muy particular es colorizar pero este se debe a que lós bordes de la imagen son leidos menos veces ya que forman parte del promedio de menor cantidad de elementos por lo que cuanto más cuadrada sea la imagen menor sería la cantidad de misses, ya que un menor porcentaje de los pixeles formarán parte del borde.
Al analizar los tiempos de ejecución se ve que el tiempo de ejecución va subiendo y bajando por como varía el ancho también.

6. Conclusiones y trabajo futuro

Al realizar este analisis exhaustivo se llega a las siguientes conclusiones:

- Es mucho mas eficiente un codigo en ASM usando las instrucciones SIMD en caso de ser esto posible
- Es importante acceder a la memoria de forma adecuada para tratar de máximizar la cantidad de hits y minimizar el tiempo de ejecución. Además hay que prevnir acceder a memoria de forma desalineada.
- La precisión perdida al recudir la precision de los floats para asi poder trabajar con mas pixeles a la vez usando SIMD no influye mucho en el resultado salvo que se requiera de una precisión absoulta.