DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Modelo de procesamiento SIMD

Organización del Computador II

Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Juan Lanuza	770/15	juan.lanuza3@gmail.com
Agustin Penas	668/14	agustinpenas@gmail.com
Fernando Frassia	340/13	ferfrassia@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

http://www.fcen.uba.ar

Índice

1. Introducción

En este trabajo práctico nos proponemos conocer y comprender las instrucciones de SIMD (Single Instruction Multiple Data), que permiten realizar operaciónes para varios datos en paralelo. Nosotros utilizaremos las instrucciones SSE (streaming SIMD extentions) de la familia de procesadores x86-64 de Intel para realizar distintos filtos de imágenes. Para analizar la performance del uso de instrucciones SIMD realizamos varias implementaciones del mismo algoritmo en Assembly y además, en C (sin SIMD).

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

- **RGBtoYUV:** transforma la imagen fuente de formato RGB a YUV.
- **YUVtoRGB:** transforma la imagen fuente de formato YUV a RGB.
- **FourCombine:** consiste en mover los pixeles de una imagen tal que queden ordenados en cuatro cuadrantes.
- **LinearZoom:** duplica el tamaño de la imagen fuente, realizando una interpolación lineal entre pixeles.
- **MaxCloser:** para cada pixel calcula la componente de color máxima para cada color sobre un kernel dado y en la imagen destina guarda una mezcla de este con el pixel original.

Imagen original

RGBtoYUV

LinearZoom

FourCombine

MaxCloser

1.1. Procedimiento experimental

Para realizar las mediciones de performance de nuestras implementaciones hay dos problemáticas principales:

1 La variación de la frecuencia de reloj de los procesadores.

2 La ejecución puede ser interrumpida por el scheduler para realizar un cambio de contexto. Esto implicará contar muchos más ciclos (outliers) que si nuestra función se ejecutara sin interrupciones.

Para resolver el item **1** decidimos realizar las mediciones en terminos de ciclos de clock, y no de segundos. Para el item **2**, para un experimento decidimos realizar multiples mediciones. En los casos en que hicimos gráficos en términos absolutos tomanmos el promedio luego de podar los valores más grandes (outliers). Para los gráficos relativos nos quedamos con el mínimo. De esta manera, estamos usando la medición en la que hubo menos interrupciones.

2. Convertir YUV a RGB y RGB a YUV

Los filtros descriptos a continuación modifican el formato de una imagen, de RGB a YUV y viceversa; utilizando una transformación lineal que puede ser descripta de la siguiente manera:

$$RGBtoYUV(R,G,B) = \left\{ \begin{array}{l} Y = sature(((66 \cdot R + 129 \cdot G + 25 \cdot B + 128) >> 8) + 16) \\ U = sature(((-38 \cdot R - 74 \cdot G + 112 \cdot B + 128) >> 8) + 128) \\ V = sature(((112 \cdot R - 94 \cdot G - 18 \cdot B + 128) >> 8) + 128) \end{array} \right\}$$

$$YUVtoRGB(Y,U,V) = \left\{ \begin{array}{l} R = sature((298 \cdot (Y-16) + 409 \cdot (V-128) + 128) >> 8) \\ G = sature((298 \cdot (Y-16) - 100 \cdot (U-128) - 208 \cdot (V-128) + 128) >> 8) \\ B = sature((298 \cdot (Y-16) + 516 \cdot (U-128) + 128) >> 8) \end{array} \right\}$$

2.1. Código C

En el código de C recorremos la matriz iterando sus filas y columnas y modificando un píxel a la vez. Para cada pixel, realizamos la transformación lineal de sus componentes, y luego lo escribimos en la imagen de salida. A continuación presentamos su pseudocódigo

Algorithm 1 RGBtoYUV

```
function RGBtoYUV(src: uint8_t*, srcw: uint32_t, srch: uint32_t, dst: uint8_t*, dstw: uint32_t, dsth: uint32_t)
    uint32_t i, j
    for j ← 0 .. srch − 1 do
        for i ← 0 .. srcw − 1 do
            int pos ← (j · srcw + i) · 4
            int r ← src[pos + 3]
            int g ← src[pos + 2]
            int b ← src[pos + 1]
            int a ← src[pos + 0]

            int y ← ((66 · R + 129 · G + 25 · B + 128) >> 8) + 16
            y ← min(255, y)
            y ← max(0, y)

            int u ← ((−38 · R − 74 · G + 112 · B + 128) >> 8) + 128
            u ← min(255, u)
            u ← max(0, u)

            int v ← ((112 · R − 94 · G − 18 · B + 128) >> 8) + 128
            v ← min(255, v)
            v ← max(0, v)

            dst[pos + 3] ← y
            dst[pos + 2] ← u
            dst[pos + 1] ← v
            dst[pos + 0] ← a
```

Algorithm 2 YUVtoRGB

function YUVtoRGB(src: uint8_t*, srcw: uint32_t, srch: uint32_t, dst: uint8_t*, dstw: uint32_t, dsth: uint32_t)

$uint32_t\ i, j$
for $j \leftarrow 0\ ..\ srch - 1$ **do**
  for $i \leftarrow 0\ ..\ srcw - 1$ **do**
    $int\ pos \leftarrow (j \cdot srcw + i) \cdot 4$
    $int\ y \leftarrow src[pos + 3]$
    $int\ u \leftarrow src[pos + 2]$
    $int\ v \leftarrow src[pos + 1]$
    $int\ a \leftarrow src[pos + 0]$

    $int\ r \leftarrow (298 \cdot (Y - 16) + 409 \cdot (V - 128) + 128) >> 8$
    $r \leftarrow min(255, r)$
    $r \leftarrow max(0, r)$

    $int\ g \leftarrow (298 \cdot (Y - 16) - 100 \cdot (U - 128) - 208 \cdot (V - 128) + 128) >> 8$
    $g \leftarrow min(255, g)$
    $g \leftarrow max(0, g)$

    $int\ b \leftarrow (298 \cdot (Y - 16) + 516 \cdot (U - 128) + 128) >> 8$
    $b \leftarrow min(255, b)$
    $b \leftarrow max(0, b)$

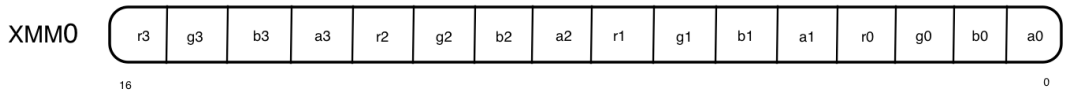
    $dst[pos + 3] \leftarrow r$
    $dst[pos + 2] \leftarrow g$
    $dst[pos + 1] \leftarrow b$
    $dst[pos + 0] \leftarrow a$

2.2. Código ASM

RGBtoYUV:

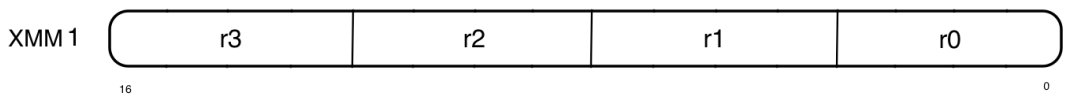
Este algoritmo recorre la matriz de la misma manera que la recorre el código de C, pero procesa 4 píxeles por iteración.

En cada ciclo, se cargan 4 píxeles en xmm0:

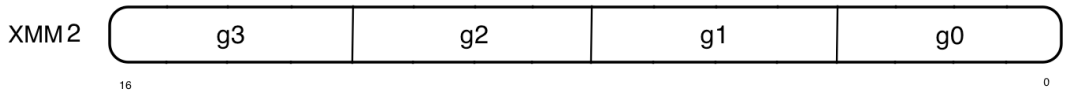


(1)

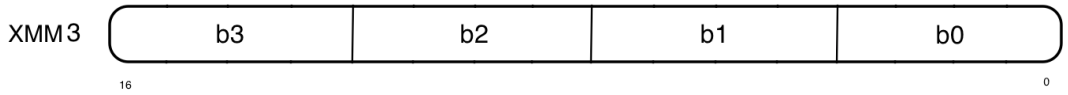
Primero, para no perder precisión, se desempaquetan las componentes R, G, B de cada pixel, de manera tal de poder procesar 4 componentes a la vez. Esto es, extender cada componente de byte a doubleWord, para esto se usa la operación **pshufb**: (notese que la componente A no es parte de la transformación lineal, por ende se deja guadada como byte y lista para el empaquetamiento final)



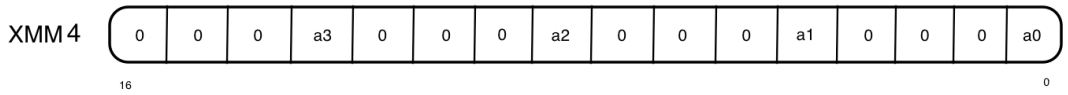
(2)



(3)

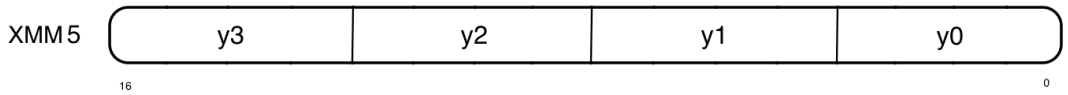


(4)

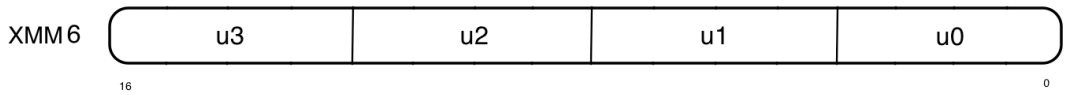


(5)

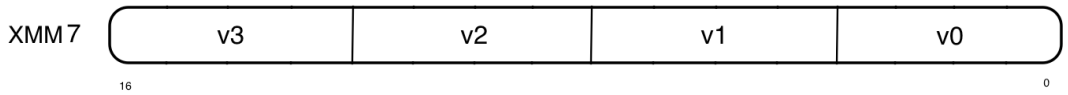
Luego se realiza la transformación lineal para cada componente -Y, U, V- y se las guarda en 3 nuevos registros:



(6)

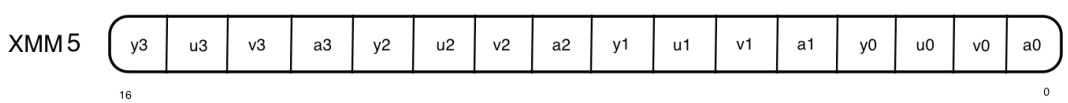


(7)



(8)

Finalmente, se empaquetan las componentes para escribirlas en memoria. Nuevamente eso es con la operación **pshufb**:



(9)

YUVtoRGB:

El algoritmo para YUVtoRGB en Assembler es análogo al de RGBtoYUV, lo único diferente es la transformación lineal usada.

2.3. Experimentación

2.3.1. Comparación C - ASM

Motivación

En este experimento compararemos la performance de nuestra implementación en C, compilada con distintos niveles de optimización, y nuestro código Assembler.

Hipótesis

Esperamos que el código Assembler sea el más performante debido a su capacidad de procesar múltiples datos en simultáneo, seguido por C compilado con O3 y luego C compilado con O0.

Resultados

Comparamos la performance de nuestro código C compilado con O0 y 03 y nuestro código de ASM. Los resultados fueron los siguientes:

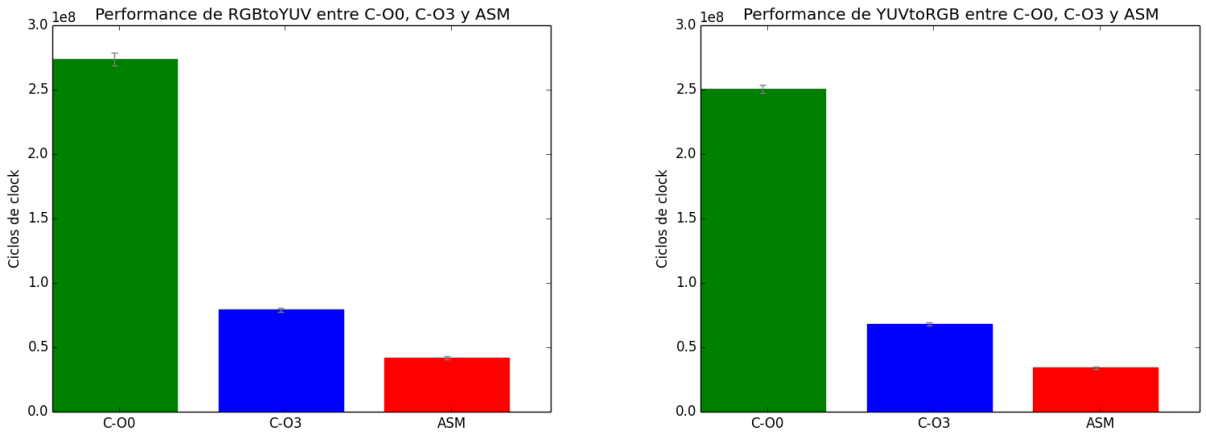


Figura 1: Performance de ambas conversiones en C compilado con -o0, C compilado con -o3 y ASM. Utilizamos una imagen de lena de tamaño 512x512. La altura de las barras representa el promedio de las 100 mediciones luego que quitar los outliers y las lineas grises, el desvío estandar.

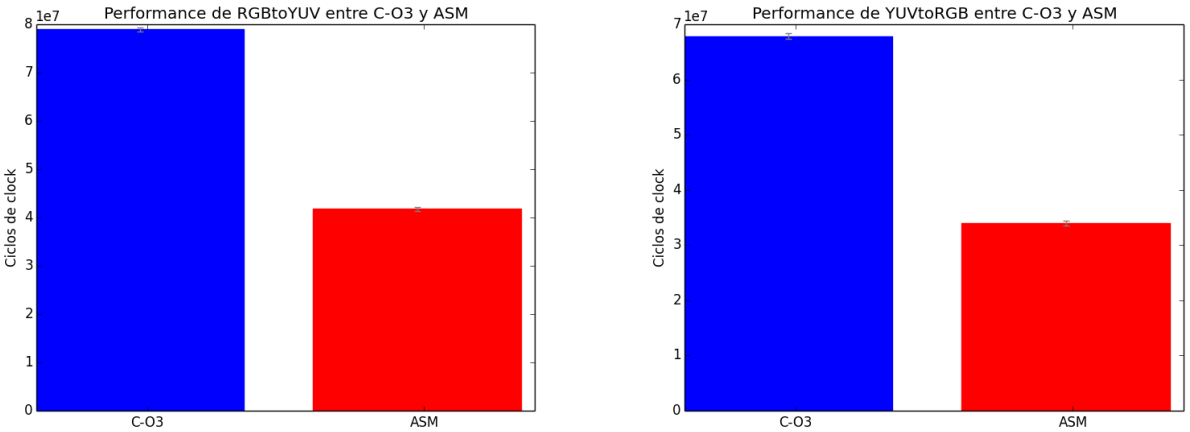


Figura 2: Acercamiento de la Figura 1, mostrando la diferencia entre C-O3 y ASM.

Podemos ver que los resultados se condicen con nuestra hipótesis, dado que en ambas conversiones C-O3 es entre 3 y 4 veces más performante que C-O0 y ASM es aproximandamente el doble de rápido que C-O3.

2.3.2. Comparación Shuffle vs Shift

Motivación:

La idea de este experimento surgió a partir de observar que por ciclo realizamos 7 operaciones de empaquetamiento y desempaquetamiento con **pshufb**. También notamos que hay muchas formas de desempaquetar y empaquetar estos datos, siendo una de ellas **psrldq** para shiftear los bytes a la posición deseada, y **pand** con una máscara para poner en 0 los demás bytes.

Hipótesis:

Nuestra hipótesis es que la instrucción **pshufb** será más rápida que **psrldq** y **pand** porque estamos reemplazando una instrucción por dos. Con lo cual el procesador, aún utilizando un pipeline, tendrá que al menos usar un ciclo de clock más para resolver la segunda operación.

Resultados:

Comparamos la performance, para cada conversión, utilizando PSHUFB contra PSRLDQ y PAND. Los resultados fueron los siguientes:

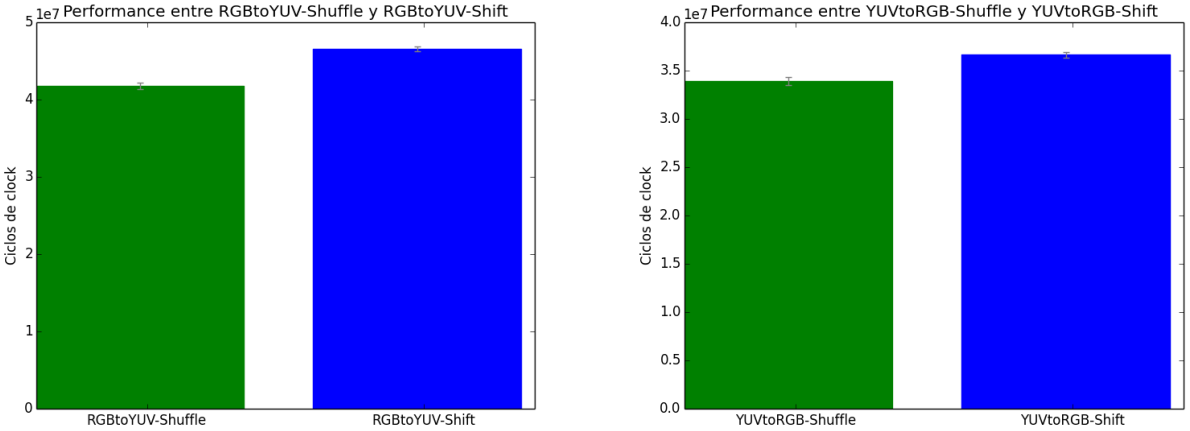


Figura 3: Cantidad de clocks de ambas conversiones en ASM con Shuffle y con Shift. Utilizamos una imagen de lena de tamaño 512x512. La altura de las barras representa el promedio de las 100 mediciones luego que quitar los outliers y las lineas grises, el desvío estandar.

Ambas conversiones tuvieron deterioros notorios de performance en la implementación con Shift - aproximadamente 10 % en RGBtoYUV y 8 % en YUVtoRGB. Tomando en cuenta que utilizamos la instrucción de shuffle 7 veces por ciclo, y en cada ciclo convertimos 4 pixeles, eso nos dice que estamos utilizando shuffle 458.752 veces para convertir una imagen de 512x512; lo cual nos da un aumento de aproximadamente 10 clocks por cada cambio en RGBtoYUV, y de 6 clocks por cada cambio en YUVtoRGB; vemos que tal diferencia tiene sentido.

Si bien no viene al caso, nos parece extraño que ambas conversiones muestren tanta diferencia entre sus mejoras de performance, dado que la única diferencia entre éstas es la transformación lineal, y que las instrucciones de shift/shuffle suceden alejadas de dicha transformación. No encontramos explicación para dicha diferencia aún.

También nos parece interesante recalcar lo que el manual de Intel dice al respecto:

SSE2 provides the PSHUFB instruction to carry out byte manipulation within a 16-byte range. PSHUFB can replace a set of up to 12 other instruction, including SHIFT, OR, AND and MOV. Use PSHUFB if the alternative code uses 5 or more instructions. [1]

Lo primero nos dice que podemos reemplazar instrucciones de - en particular - SHIFT y AND por PSHUFB, sin hablarnos de performance; y lo segundo sí es una sugerencia de performance, para cuando PSHUFB pueda reemplazar a un código de 5 o más instrucciones.

Ahora bien, en este caso nuestro código alternativo es de 2 instrucciones, con lo cuál no estamos en condiciones de seguir la premisa de Intel. Pero sí nos hace pensar que pueden suceder las siguientes cosas:

(a) Dado que lo que queremos medir es muy chico y que los errores de medición pueden ser grandes, es probable que la diferencia obtenida no sea la correcta; más aún, puede que de hacer futuras mediciones, los resultados se contradigan entre sí.

(b) Para un código de 4 instrucciones o menos, las diferencias de performance pueden ser más inciertas y por eso el manual no dice ninguna aclaración al respecto.

2.3.3. Disminuición de lecturas al guardar máscaras en registros

Motivación:

Para este experimento notamos que ambas implementaciones de ASM usaban una gran cantidad de máscaras y ninguna de ellas se encontraba guardada en un registro. Esto nos sugirió que por ciclo estábamos utilizando una gran cantidad de lecturas a memoria para operar con dichas máscaras. La idea que surgió fue la de mover la mayor cantidad de máscaras posibles a registros y así disminuir la cantidad total de lecturas a memoria.

Hipótesis:

Nuestra hipótesis es que dado que bajamos la cantidad de accesos a memoria deberíamos ver una mejora de rendimiento en ambas conversiones. Es difícil predecir en cuánto esperamos que mejore la performance de las conversiones, dado que - si bien sabemos que en RGBtoYUV bajamos los accesos en un %39,39 y en YUVtoRGB bajamos en un %32,14 - no sabemos con exactitud cuánto influyen estos accesos en el total de la performance de los algoritmos.

Resultados:

Comparamos la performance, para cada conversión, leyendo cada máscara en memoria contra leyendo la menor cantidad posible en memoria. En RGBtoYUV la diferencia de accesos a memoria por ciclo es de 33 a 13, y en YUVtoRGB es de 28 a 9. Los resultados fueron los siguientes:

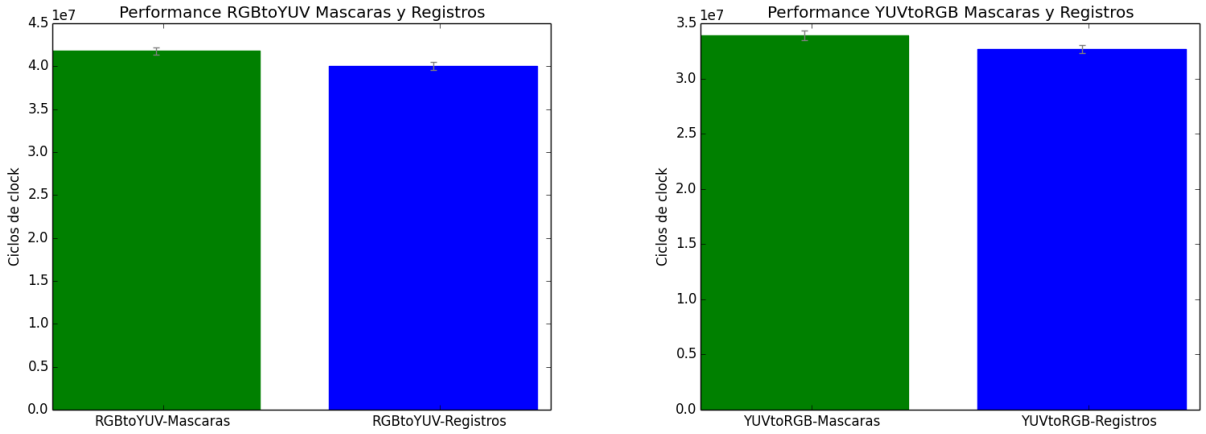


Figura 4: Cantidad de clocks de ambas conversiones en ASM con la versión de mayor cantidad de accessos ("Máscaras") y la versión de menor cantidad de accessos ("Registros"). Utilizamos una imagen de lena de tamaño 512x512. La altura de las barras representa el promedio de las 100 mediciones luego que quitar los outliers y las lineas grises, el desvío estandar.

En ambas conversiones obtuvimos una mejora de aproximadamente 4 %, lo cual se condice con nuestra hipótesis. Sin embargo, esperabamos una diferencia más marcada entre ambas implementaciones. Creemos que la diferencia es poca porque probablemente esos accesos a memoria sean a caché de Level 1, dado que en cada ciclo se están buscando los mismos datos.

3. FourCombine

Este filtro consiste en mover los pixeles de una imagen tal que queden ordenados en cuatro cuadrantes según indica el siguiente ejemplo

11	12	13	14	15	16	17	18
21	22	23	24	25	26	27	28
31	32	33	34	35	36	37	38
41	42	43	44	45	46	47	48
51	52	53	54	55	56	57	58
61	62	63	64	65	66	67	68
71	72	73	74	75	76	77	78
81	82	83	84	85	86	87	88

11	13	15	17	12	14	16	18
31	33	35	37	32	34	36	38
51	53	55	57	52	54	56	58
71	73	75	77	72	74	76	78
21	23	25	27	22	24	26	28
41	43	45	47	42	44	46	48
61	63	65	67	62	64	66	68
81	83	85	87	82	84	86	88

Figura 5: A la izquierda, la imagen antes aplicar el filtro y a la derecha, después.

Los píxeles de las filas pares (empezando a contar desde 0 desde la parte superior) se colocarán el los cuadrantes de arriba, mientras que los de las filas impares van a los cuadrantes inferiores. Los píxeles de las columnas pares irán en los cuadrantes de la izquierda y los de las impares a la derecha. Por otro lado, la posicion en x de un pixel dentro de un cuadrante se determina haciendo la división entera de la posición original en x sobre 2. La posicion en y se calcula análogamente.

3.1. Código C

Nuestra implementación en C es bastante simple. Para cada pixel de la imagen original calculamos su posición en la imagen destino siguiendo con las reglas mencionadas. A continuación presentamos la implementación en pseudocódigo:

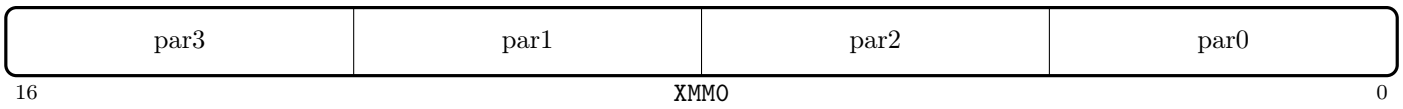
Algorithm 3 C_fourCombine

function C_FOURCOMBINE(src: uint8_t*, srcw: uint32_t, srch: uint32_t, dst: uint8_t*, dstw: uint32_t, dsth: uint32_t)
  $uint32\_t\ i, j, x, y$
  $uint32\_t\ cuadrante\_x, cuadrante\_y$
  $uint64\_t\ pos\_act, pos\_dst$
  for $j \leftarrow 0\ ..\ srch - 1$ **do**
    for $i \leftarrow 0\ ..\ srcw - 1$ **do**
      $pos\_act \leftarrow j \cdot srcw + i$
      $cuadrante\_x \leftarrow i\,\%2 == 0\ ?\ \ srcw/2$
      $cuadrante\_y \leftarrow j\,\%2 == 0\ ?\ \ srch/2$
      $x \leftarrow cuadrante\_x + i/2$
      $y \leftarrow cuadrante\_y + j/2$
      $pos\_dst \leftarrow y \cdot srcw + x$
      $((uint32\_t*)dst)[pos\_dst] \leftarrow ((uint32\_t*)src)[pos\_act]$

3.2. Código ASM

Teniendo en cuenta que los píxeles de las filas pares van a distintos cuadrantes, en nuestra implementación en Assembly decidimos recorrer la imagen de a dos filas. Así evitamos tener un if adentro del ciclo principal para decidir si se está en una fila par o en una impar. Para cada fila vamos leyendo de a 4 píxeles y los colocamos en registros **XMM**. Para las filas pares e impares realizamos las mismas operaciones, solo que en la imagen destino escribimos los píxeles en distintos cuadrantes. Para lo pares:

Hacemos un shiffle (PSHUFD xmm0, xmm0, 0xD8) para que queden los pixeles 0 y 2 por un lado (que van al cuadrante de la izquierda) y el 1 y el 3 por otro (que van al de la derecha).



Escribimos en memoria con MOVQ los primeros dos pixeles, shifteamos **XMM0** y escribimos nuevamente en otra parte de la memoria.

Para hacer las lecturas y escrituras a memoria contamos con cuatro punteros y dos contadores. Un puntero apunta a la fila par actual de la imagen fuente y otro a la fila impar. Para la imagen destino tenemos una que apunta a la fila actual de los cuadrantes superiores y otro a la de los inferiores. Además tenemos dos contadores con la posicion actual en x y en y en la imagen fuente.

En esta implementación, en cada iteración hacemos 2 lectura en memoria de 4 píxeles cada una y 4 escrituras de 2 píxeles cada una.

3.3. Experimentación

3.3.1. Comparación C - ASM

Comparamos la performance de nuestro código C compilado con O0 y 03 y nuestro código de ASM. Los resultados fueron los siguientes:

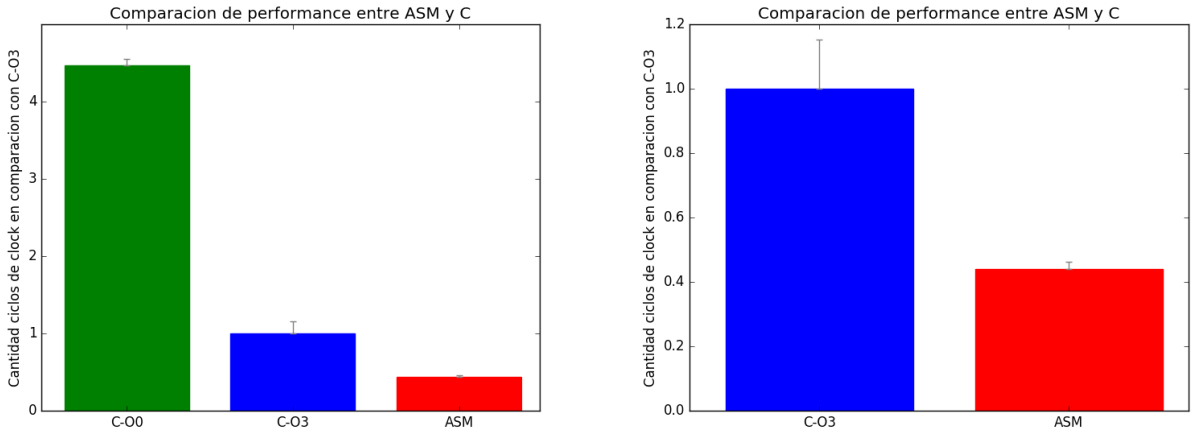


Figura 6: Performance de C compilado con -o0, C compilado con -o3 y ASM. Utilizamos una imagen de lena de tamaño 1024x1024. La altura de las barras representa el minimo de las 100 mediciones y las líneas grises representan el promedio luego que quitar los outliers.

Como era de esperarse, la compilación con C-O0 es significativamente peor que con C-O3. Por otro lado, quedamos sorprendidos que la versión ASM es solo dos veces más rápida que C-O3, teniendo en cuenta que en ASM procesamos cuatro pixeles en paralelo. Creemos que esto se debe a que en el código Assembly cuando escribimos en memoria lo hacemos de a dos píxeles, y cada escritura cuesta bastante en términos de ciclos de clock.

3.4. Comparación con escrituras de a 4 píxeles

Como consecuencia de los resultados obtenidos en el experimento anterior, realizamos una modificación a nuestro programa en ASM con el fin de mejorar su performance. En la implementación original cada vez que escribimos en memoria lo hacemos de a dos píxeles cuando se podría escribir de a cuatro y así reducir la cantidad de escrituras a la mitad. En nueva implementación en cada ciclo levantamos el ocho píxeles de cada una de las dos filas en vez de cuatro. Así, cuando hay que escribir los píxeles en la imagen destino tenemos acumulados cuatro por cada cuadrante y para cada uno los escribimos con una sola instrucción.

Esta versión tiene como contrapartida que solo funciona para imágenes de ancho múltiplo de ocho píxeles. Igualmente esto podría solucionarse facilmente con un salto condicional dentro del ciclo que se produzca cuando quedan 4 píxeles en la fila, y que en ese caso se ejecute el contido del ciclo de la implementación orginal.

Los resultados de las mediciones de performance fueron las siguientes:

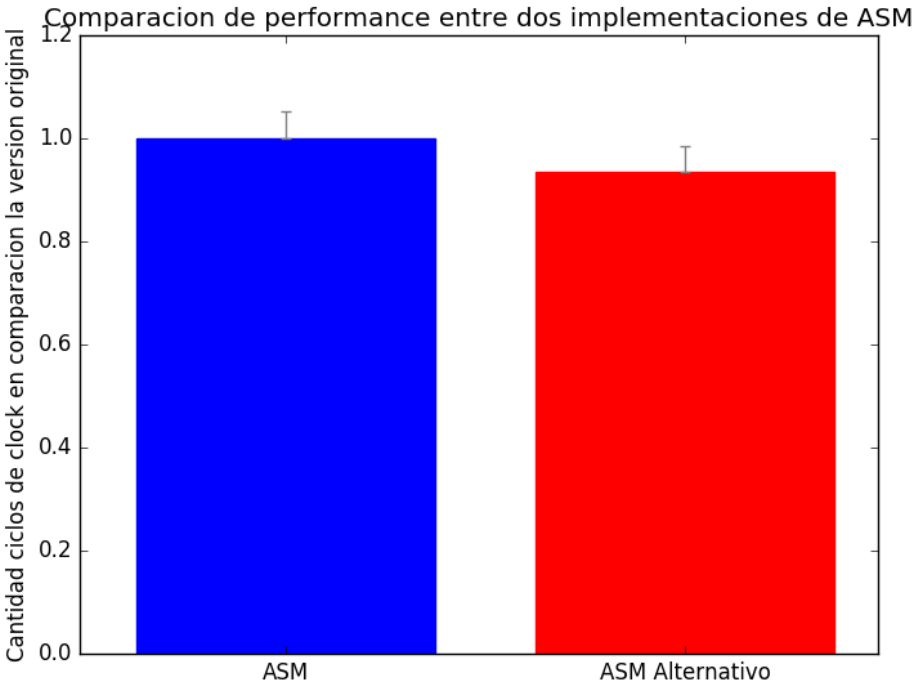


Figura 7: Performance de ambas implementaciones en Assembly. Utilizamos una imagen de lena de tamaño 1024x1024. La altura de las barras representa el minimo de las 100 mediciones sobre el mínimo de ASM original y las líneas grises representan el promedio luego que quitar los outliers sobre el mínimo de ASM original.

Como se ve en el gráfico, la nueva versión del código es aproximadamente 6 % más performante que la anterior. Sin embargo, esperabamos una mayor diferencia entre ambos. Creemos que la poca diferencia entre ambos códigos se debe principalmente a dos causas:

Primero, para poder tener los cuatro píxeles que hay que escribir en una posión en un mismo registro tuvimos que agregar varias instrucciónes extra como shifts y blends. Por otro lado, como veremos en el experimento de LinearZoom, realizamos varias lecturas consecutivas que pueden retrasar el pipeline.

3.5. Comparación con Loop Unrolling

La idea de este experimento surgió de observar que nuestro algoritmo se compone de un ciclo anidado que se ejecuta Largo · Ancho / 4 veces, y esto genera muchos saltos no condicionales en nuestro código. Creemos que podemos mejorar la performance eliminando los saltos no condicionales, es decir, haciendo loop unrolling.

Loop unrolling es una técnica de optimización de ciclos que busca mejorar la performance de un programa al reducir sus saltos condicionales; esto reduce las instrucciones de aumento de punteros, comparaciones aritméticas y jump entre iteraciones. Si bien esta técnica mejora la performance también tiene un costo, y es que el ejecutable se vuelve más pesado, porque las líneas de código aumentan en $k \cdot i$, siendo k las líneas del ciclo e i la cantidad de ciclos quitados.

Nuestra hipótesis es que efectivamente deberíamos ver mejora en la performance porque estaremos eliminando saltos en el total de la ejecución del programa, pero a medida que desenrollamos más ciclos esperamos ver un quiebre en la performance una vez que nuestro código ya no quepa en caché y se disparen los caché misses.

3.5.1. Resultados

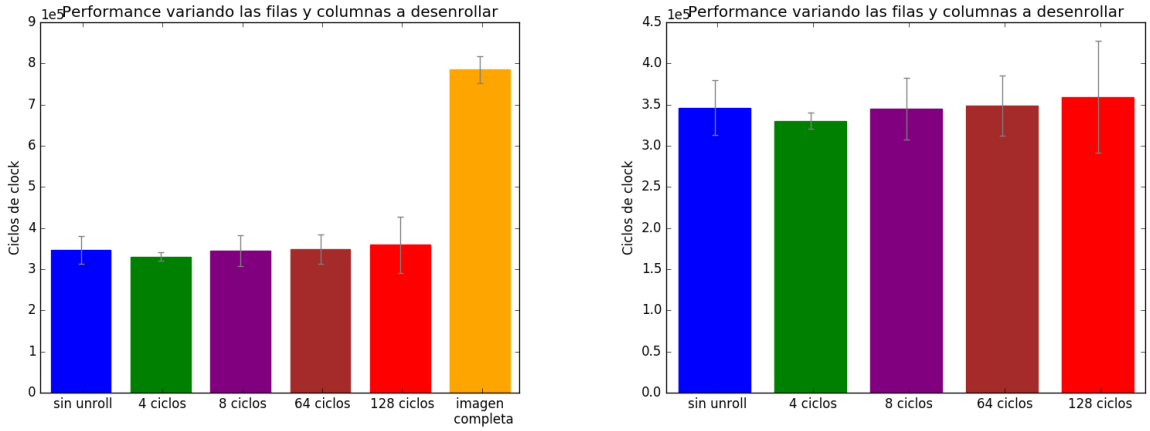


Figura 8: Comparación de varias implementaciones de Four Combine(ASM). Las variaciones se basan en la cantidad de ciclos en los que hacemos loop unrolling, comenzamos con las filas incrementalmente y finalmente hacemos unrolling de 128 filas y 128 columnas. Utilizamos una imagen de lena de tamaño 512x512. La altura de las barras representa el minimo de las 100 mediciones sobre el mínimo de ASM original y las líneas grises representan el promedio luego que quitar los outliers sobre el mínimo de ASM original.

Nos sorprendió ver que los resultados para distinta cantidad de unrolls muestren diferencias despreciables. Según nuestra experimentación el unroll de 4 ciclos parece ser el más performante, y a mayor cantidad de unrolls - pero que aún entran en caché - la performance empeora levemente. SIn embargo, las diferencias no son tan marcadas para ser concluyentes.

Esperábamos lo contrario: una mejora en performance hacia unrolls más grandes, siempre y cuando el código entre en caché. No logramos identificar por que es que pasa esto ya que por lo que habiamos averiguando, el hacer loop unrolling deberia coincidir con nuestra hipotesis.

A pesar de lo anterior, parte de nuestra hipotesis fue correcta y corroborable con el experimento. El caso del loop unrolling de la iamgen completa dio valores mayores al doble de las demas mediciones. Como dijimos en la hipotesis, esto pasa cuando el código es tan grande que no entra en la caché lvl1 y por ende se registran muchos miss.

4. Linear Zoom

Este filtro consiste en duplicar el tamaño de la imagen original, realizando una interpolación lineal de pixeles de acuerdo con la siguiente imagen:

A	$(A+B)/2$	B
$(A+C)/2$	$(A+B+C+D)/4$	$(B+D)/2$
C	$(C+D)/2$	D

Donde A, B, C y D son pixeles de la imagen fuente.

Como la cantidad de filas y de columnas de las imágenes originales son pares, los márgenes derecho e inferior no pueden seguir el patrón de la interpolación; por lo que se decidió que sean copia de su pixel lindante.

4.1. Código C

En nuestra implementación del filtro en C completamos la imagen fuente en varios pasos. Primeramente, colocamos todos lo pixeles originales en la posicón correspondiente (marcados en blanc en la Figura), en las filas y columnas impares, si empezamos a contarlas desde 0, desde la parte inferior. Posteriormente recorremos las filas impares calculando el promedio de los pixeles que ya están en la imagen. Con toda las filas impares ya completa, excepto el último pixel de cada una, podemos completar las filas pares. En cada pixel de estas colocamos el promedio de sus pixeles superior e inferior. Por último completamos los bordes con su pixel lindante.

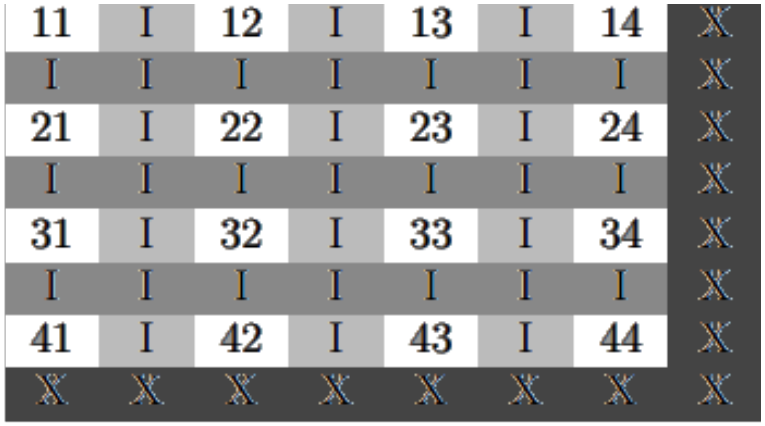


Figura 9: Matriz de pẋeles que representa la imagen destino. En nuestra implementación en C los píxeles se pintan en orden de satuacion creciente.

4.2. Código Assembly

Teniendo en cuenta que cada pixel mide 4 bytes, se pueden levantar hasta 4 píxeles a un registo XMM. Sin embargo, si procesaramos de a 4 píxeles de la imagen fuente, al avanzar a los siguientes 4 nos quedaría pendiente calcular y escribir el promedio entre el ultimo pẋel de los primeros y el primero de los segundos (el que está marcado en rojo en la imagen).

Por lo tanto decidimos levantar en cada ciclo 4 píxeles pero en vez de procesar todos solo procesamos 2, y 2 promedios. De esta manera, en la imagen destino vamos avanzando de a 4 píxeles por ciclo y la imagen fuente de a 2.

En nuestra implementación en ASM hay dos ciclos:

En el primer ciclo, completamos las ultimas dos filas, que son iguales. Levantamos 4 píxeles de memoria y los almancenamos en **XMM0**

$pixel3$	$pixel2$	$pixel1$	$pixel0$

16 0

Con un shuffle, en **XMM2** (pshufd xmm2, xmm0, 0xF9)

$pixel3$	$pixel3$	$pixel2$	$pixel1$

16 0

Y luego realizamos el promedio entre los valores de estos registros con pavgb y en **XMM2** nos queda

$pixel3$	$(pixel3 + pixel2)/2$	$(pixel2 + pixel1)/2$	$(pixel1 + pixel0)/2$

16 0

donde $(pixelx + pixely)/2$ es el promiedo entre ambos pixeles componente a componente. Por último hacemos un punpckldq xmm0, xmm2 y en **XMM0** nos queda

$(pixel2 + pixel1)/2$	$pixel1$	$(pixel1 + pixel0)/2$	$pixel0$

16 0

que es lo que escibimos en ambas filas. El cíclo lo detenemos una iteración antes de terminar de recorrer toda la fila, ya que los últimos dos píxeles tienen que quedar iguales.
Para procesar esta parte utilizamos la instrucción punpckhdq xmm1, xmm2 que no deja el **XMM1** así:

$pixel3$	$pixel3$	$(pixel3 + pixel2)/2$	$pixel2$

16 0

En **XMM1** teníamos una copia de los últimos píxeles de la imagen fuente.

El procesamiento del resto de la imagen es más compleja. En cada ciclo, primero levantamos 4 píxleles de memoria de la imagen fuente y realizamos las mismas operaciones que para las últimas dos filas. Estos datos serán colocados dos filas arriba de la última procesada. Después leemos de la imagen destino píxeles de la última fila procesada para poder hacer el promedio entre esta y la ya calculada.
Tenemos en **XMM0**

$(p_{i+2,j-2} + p_{i+1,j-2})/2$	$p_{i+1,j-2}$	$(p_{i+1,j-2} + p_{i,j-2})/2$	$p_{i,j-2}$

16 0

En **XMM1** tenemos los píxeles de dos filas mas abajo:

$(p_{i+2,j} + p_{i+1,j})/2$	$p_{i+1,j}$	$(p_{i+1,j} + p_{i,j})/2$	$p_{i,j}$

16 0

Hacemos la instrucción PAVGB xmm1, xmm0 y en **XMM1** nos queda lo que tenemos que escribir en la fila j-1.

Luego escribimos los 4 píxeles de la fila j-2 y los 4 de la fila j-1 y avanzamos i, dos píxeles. En caso de haber llegado a final de la fila, se avanzan los punteros al coniemzo de la siguiente fila, si es que hay una.

En esta implementación, en cada una de las iteraciónes del ciclo principal se realizan dos lecturas de 128bits (una de la imagen fuente y una de la destino) y se escriben 8 píxeles. Estos datos nos servirán después, cuando en la experimentación hagamos una comparación con una implementación alternativa.

4.3. Experimentación

4.3.1. Comparación C - ASM

En este experimento compararemos la performance de nuestra implementación en C, compilada con distintos niveles de optimización, y nuestro código Assembly. Suponíamos que la versión ASM será mucho más rápida que todas los compilaciones del código C, ya que utilizamos las instrucciones SSE que nos permiten procesar en paralelo.

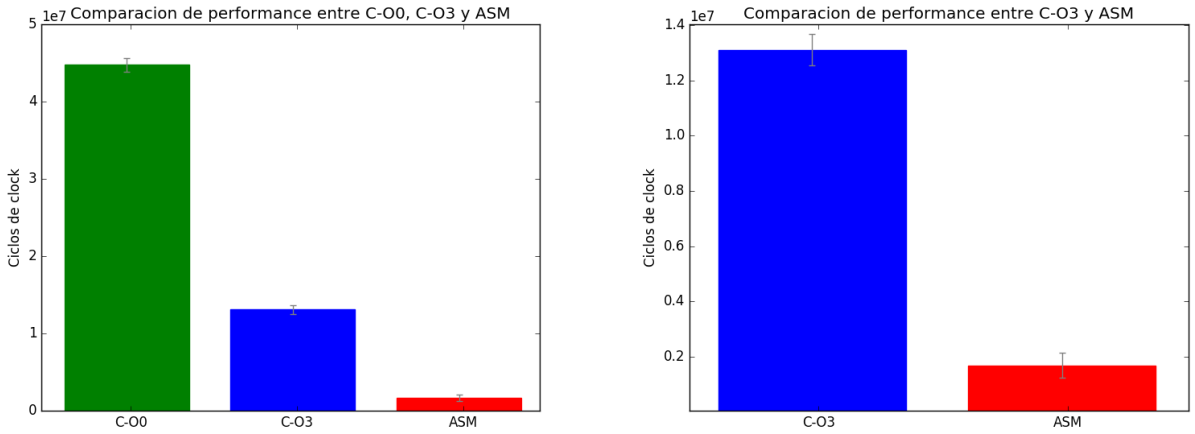


Figura 10: Performance de C compilado con -o0, C compilado con -o3 y ASM. Utilizamos una imagen de lena de tamaño 512x512. La altura de las barras representa el promedio de las 100 mediciones luego que quitar los outliers y las lineas grises, el desvío estandar.

Se puede ver claramente que la performance de compilar el código C con O3 es mucho mejor que con O0. El programa compilado con O0 tarda casi 4 veces lo que tarda O3. Por otro lado, es notable la velocidad de la implentación en ASM comparado con las otras dos. Esto se debe a que en cada iteración escribimos 4 pixeles en la imagen destino, contra 1 en la implementación en C.

4.3.2. Comparación con implementación alternativa de ASM.

Al consultar con los docente sobre nuestra implentación en ASM y la posibilidad de procesar de a mayor cantidad de píxeles, nos sugirieron una implementación que en cada iteración se procesan los 4 píxeles de la imagen fuente que se levantan de memoria (en nuestra implementación original de procesan solo 2). A continuación la explicamos en detalle:

Al comenzar cada iteración levantamos al registro **XMM0** los primeros cuatro píxeles y en **XMM2**, del 2do al 5to. **XMM0**

$pixel3$	$pixel2$	$pixel1$	$pixel0$

16 0

XMM2

$pixel4$	$pixel3$	$pixel2$	$pixel1$

16 0

Despues calculamos el promedio con PAVGB xmm2, xmm0. **XMM2** queda asi:

$(pixel4 + pixel3)/2$	$(pixel3 + pixel2)/2$	$(pixel2 + pixel1)/2$	$(pixel1 + pixel0)/2$

16 0

Luego realizamos un PUNPCKLDQ entre xmm0 y xmm2 y un PUNPCKHDQ entre xmm1(copia de xmm0) y xmm2. Esto deja en xmm1 y xmm0 lo que hay que escribir en la imagen destino **XMM1**

$(pixel2 + pixel1)/2$	$pixel1$	$(pixel1 + pixel0)/2$	$pixel0$

16 0

XMM0

$(pixel4 + pixel3)/2$	$pixel3$	$(pixel3 + pixel2)/2$	$pixel2$

16 0

Para completar las filas que tienen promedio de promedios levantamos la fila anterior, calculada en una iteración anterior, y hacemos un PAVGB con la fila superior, al igual que en la otra implementación. La forma en la que recorremos la imagen es similar. Primeramente pintamos las dos filas inferiores y despues recorremos el resto de las filas.

Comparando ambas implementaciones vemos que en esta se realiza la mitad de iteraciones. Sin embargo, en cada itereración se realiza el doble de lecturas y el doble de escrituras a memoria. Por lo tanto, en terminos absolultos se realiza la misma cantidad de accesos.

Por otro lado, teniendo en cuenta que en la versión original se realiza el doble de saltos condicionales, uno podría pensar que tardara más. Igualmente, el jump precictor va a asumir Always Taken y solo se produce una misprediction cuando se termina de recorrer la fila, y esto para tambien en la otra implementación.

Luego de realizar las mediciones correspondiente los resultados obtenidos son los siguientes:

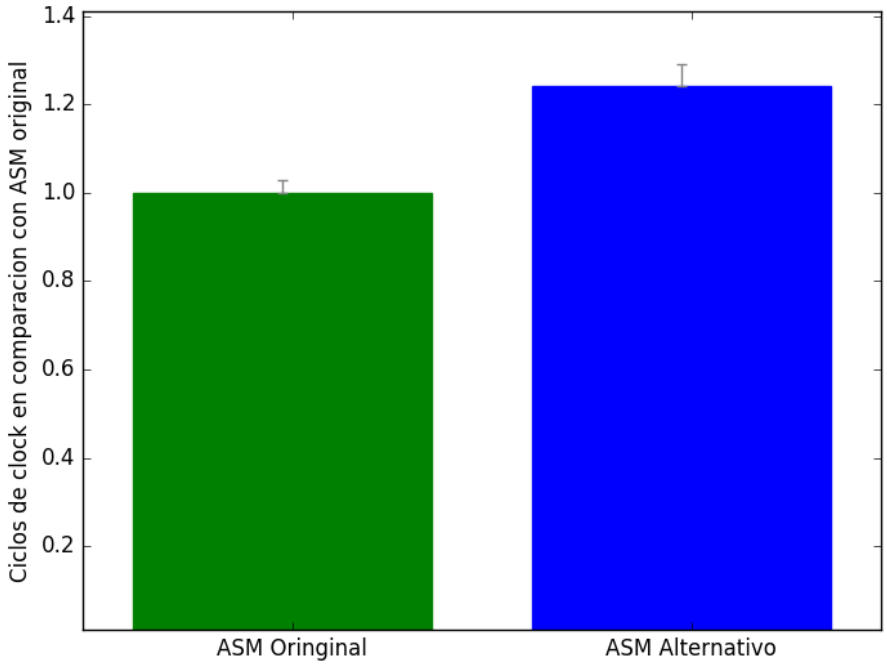


Figura 11: Cantidad de ciclos de clock en realción con la implementación orginal. Utilizamos la imagen de lena de tamaño 512x512. La altura de las barras representa al mínimo de las 100 mediciones y la altura de las lineas grises al promedio luego de podar al 10 % más alto.

Como se vé claramente en el gräfico, la implementación alternativa tarde aproximadamente un 20 % mas que la original. Los resultados no fueron los esperados. Por eso se repitió el experimento para distintos tamaños de imágenes, pero los resultados fueron los mismos.

Luego de analizar ambas implementaciones, la única razon que encontramos para explicar los resultados dados es que en la segunda versión se realizan varios accesos a memoria (tanto de lectura como de escritura) consecutivos. Esto podría provocar un obstáculo estructural en el pipeline, ya que utilizan los mismos recursos, que retrasaría la ejecución del código.

5. MaxCloser

Este filtro consiste en que, para cada pixel, se calcula el máximo de cada color dentro de un kernel de 7x7 centrado en el. En el mismo pixel de la imagen destino se escribe el resultado de la combinacion lineal entre el pixel original y el máximo. En la imagen destino se pinta de blanco 3 pixeles en los 4 costados de la imagen, pues ahí el kernel es inválido.

La combinacion lineal esta dada por la formula:

$$dst[i][j][R] = src[i][j][R] * (1 - val) + maxR * (val) \tag{10}$$

$$dst[i][j][G] = src[i][j][G] * (1 - val) + maxG * (val) \tag{11}$$

$$dst[i][j][B] = src[i][j][B] * (1 - val) + maxB * (val) \tag{12}$$

Donde val es un valor entre 0 y 1 pasado por parametro.

5.1. Código C

En el código de C, primero recorremos cada uno de los cuatro márgenes (en ciclos distintos) pintandolos de blanco. Posteriormente, para cada uno del resto de los pixeles, recorremos su kernel buscando el máximo y realizamos la combinación lineal de cada color.

Algorithm 4 MaxCloser

function MAXCLOSER(src: uint8_t*, srcw: uint32_t,srch: uint32_t , dst: uint8_t* ,dstw: uint32_t ,dsth: uint32_t , val: float)
  for $j \leftarrow 3$ .. $srch - 3; j++$ **do**
    for $i \leftarrow 3$ .. $srcw - 3; i++$ **do**
      $max_r \leftarrow 0$
      $max_g \leftarrow 0$
      $max_b \leftarrow 0$
      for $y \leftarrow j - 3$ .. $j + 4; y++$ **do**
        for $x \leftarrow i - 3$ .. $i + 4; x++$ **do**
          if $max_r < src[(y * srcw + x) * 4 + 3]$ **then**
            $max_r \leftarrow src[(y * srcw + x) * 4 + 3]$
          if $max_g < src[(y * srcw + x) * 4 + 2]$ **then**
            $max_r \leftarrow src[(y * srcw + x) * 4 + 2]$
          if $max_b < src[(y * srcw + x) * 4 + 1]$ **then**
            $max_r \leftarrow src[(y * srcw + x) * 4 + 1]$
      $pos \leftarrow (j * srcw + i) * 4$
      $dst[pos + 3] \leftarrow src[pos + 3] * (1 - val) + max_r * val$
      $dst[pos + 2] \leftarrow src[pos + 2] * (1 - val) + max_r * val$
      $dst[pos + 1] \leftarrow src[pos + 1] * (1 - val) + max_r * val$
      $dst[pos + 0] \leftarrow src[pos + 0]$

5.2. Código Assembly

Lo que hace nuestro codigo assembler es ciclar por todos los pixeles. Para cada uno primero calcula el maximo, luego hace la combinacion lineal. Para los bordes en particular no hace esto sino que pone el pixel en blanco. No analizaremos el caso de los bordes, que es trivial, sino el caso de los pixeles internos.

Primero analizaremos como calculamos el máximo. Nuestro kernel es de 7x7. Nuestro codigo levanta de a 4 pixeles, por lo que al avanzar a la siguiente columna avanzaremos solo 3 pixeles, de manera que estamos solapando 1 pixel. Sin embargo eso no influye en el calculo del maximo ya que aunque se tome en cuenta un numero más de una vez, el máximo del conjunto seguira siendo el mismo.

Acontinuacion mostramos como calculamos el maximo en cada ciclo para el color rojo. Para los demas colores sera analogo, solo se tienen que cambiar las mascaras correspondientes.

Cargamos en **XMM0** el pixel y hacemos copias de este para cada color

$pixel3$	$pixel2$	$pixel1$	$pixel0$

16 0

Hacemos un pand con una mascara para quedarnos unicamente con el color rojo

$R3000$	$R2000$	$R1000$	$R0000$

16 0

Hacemos un psrldq para mover 3 posiciones el byte rojo, de esta manera nos queda en el byte menos significativo de cada double word

$R3$	$R2$	$R1$	$R0$

16 0

Luego hacemos un pmaxub entre el registro xmm donde tenemos los colores rojos y el registro donde guardamos los máximos parciales

máximo parcial	máximo parcial	máximo parcial	máximo parcial

16 0

Al finalizar pasamos a la siguiente columna corriendonos 3 pixels. En el caso de ya estar en la segunda columna, vamos a la primera columna de la siguiente fila. Repetimos hasta completar todo el kernel

Al salir del ciclo que recorre el kernel tendremos en un registro (por ejemplo el **XMM10**) 4 maximos parciales. Lo que debemos hacer es encontrar el maximo entre estos 4

Para esto hacemos una copia de **XMM10** en **XMM9**, hacemos un psrldq xmm9, 4 y nos queda

XMM10

max_3	max_2	max_1	max_0

16 0

XMM9

0	max_3	max_2	max_1

16 0

Hacemos un pmaxub **XMM10**,**XMM9** y nos queda

?	$max(max_3, max_2)$	?	$max(max_1, max_0)$

16 0

Volvemos a copiar en **XMM9** y shifteamos 8 lugares y volvemos a hace pmaxub

XMM10

?	?	?	$max(max_3, max_2, max_1, max_0)$

16 0

Hay que repetir lo mismo para cada color. Con esto entonces tenemos el calculo de el maximo de el kernel para cada color.

Pasamos ahora a la parte del calculo lineal:

Levantamos de memoria el valor del pixel actual y separamos cada componente (o color) del pixel en distintos registros con la misma metodologia anterior. (Nuevamente solo haremos el ejemplo con el color rojo, los demas son analogos) Supongamos que guardamos el rojo en **XMM2**

Convertimos tanto el registro donde esta el maximo como el que tiene el rojo actual a float. Para esto usamos la instruccion CVTDQ2PS.

Cargamos en un registro XMM el numero 1, lo convertimos a float y a este le restamos **VAL**. De esta manera tenemos en un registro 1-**VAL**. Supongamos que este valor lo guardamos en **XMM7**

Luego hacemos mulss entre **XMM10** y **XMM0** y entre **XMM2** y **XMM7**

XMM10

?	?	?	$max_r * VAL$

16 0

XMM2

?	?	?	$Rojo * (1 - VAL)$

16 0

Luego sumamos con addss y convertimos a entero con CVTPS2DQ

Hacemos un shift left para volver a el color rojo a su posicion en el double word con PSLLDQ xmm2,3 y luego hacemos la suma entre todos ellos. En **XMM1** me queda:

?	?	?	$R\|G\|B\|alfa$

16 0

Solo queda escribir en el destino y avanzar de pixel

5.3. Experimentación

5.3.1. Comparación C - ASM

En este experimento compararemos la performance de nuestra implementación en C, compilada con distintos niveles de optimización, y nuestro código Assembly.

Creemos que la versión ASM será mucho más rápida que todas los compilaciones del código C, ya que utilizamos las instrucciones SSE que nos permiten procesar en paralelo.

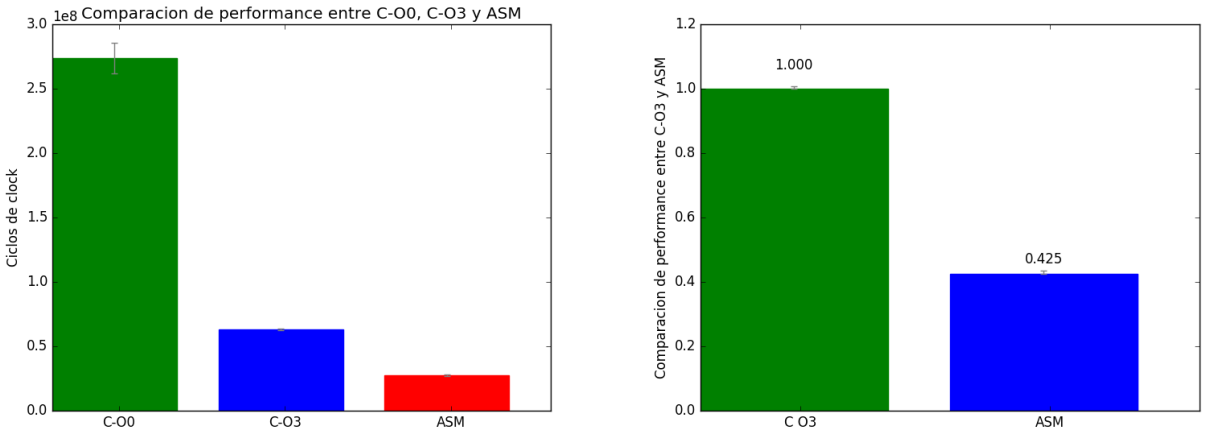


Figura 12: Performance de C compilado con -o0, C compilado con -o3 y ASM. Utilizamos una imagen de lena de tamaño 512x512. La altura de las barras representa el promedio de las 100 mediciones luego que quitar los outliers y las lineas grises, el desvío estandar. La segunda imagen muestra la comparacion en relacion con la cantidad de ciclos de C con O3

Podemos ver que el codigo en assembler es mucho más rapido que cualquier optimizacion de C. Si bien Podemos ver que C compilado con O3 es mucho más rapido que O0, este es menos optimo que el assembler, que hace el mismo trabajo en un 40 % de la cantidad los ciclos.

5.3.2. Pintar los bordes afuera del ciclo principal vs pintarlos adentro

En nuestra primera implementacion lo que haciamos era primero recorrer los bordes y pintarlos de blanco, y luego procesar el resto de la image. Luego se nos ocurrio mover esto al ciclo principal.

Esto nos llevo a pensar en si pintar los bordes en distintos ordenes impactaria en la performance entre implementaciones. Nuestra hipotesis era que al pasar a pintarlos adentro del ciclo principal nos ahorramos lecturas innccesarias y para los bordes verticales tendriamos el beneficio de ya tener en cache cada busqueda y escritura a memoria.

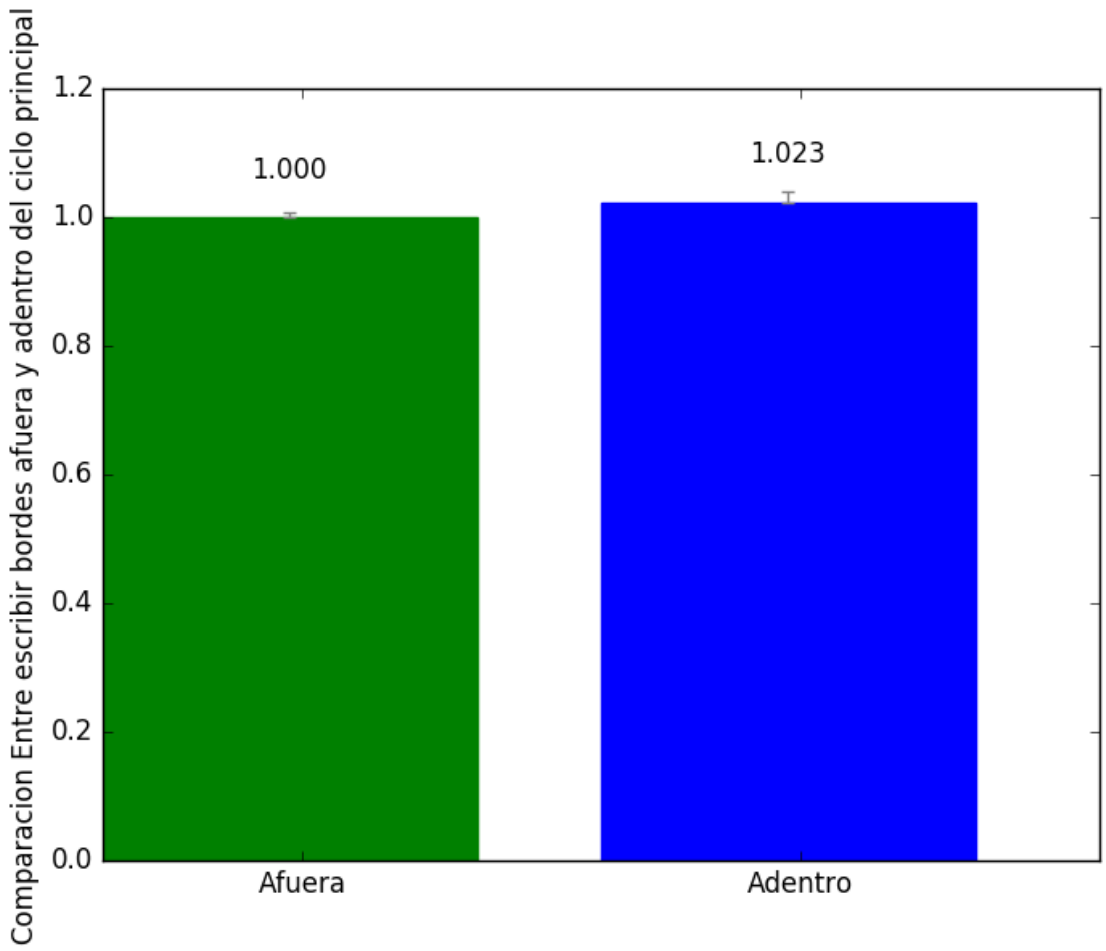


Figura 13: Comparacion de performance entre nuestras dos implementaciones de ASM. Utilizamos una imagen de lena de tamaño 512x512. La altura de las barras representa el promedio de las 100 mediciones luego que quitar los outliers y las lineas grises, el desvío estandar.

Como podemos ver la diferencia entre ambos es minima, siendo que pintar los bordes de blanco adentro del ciclo principal es un 2 % menos performante que hacerlo afuera. Con estos datos no podemos afirmar que una implementacion sea mejor que la otra.

Lo que concluimos es que como la cantidad de pixeles del borde esta en funcion de la raiz cuadrada de el total de pixeles, entonces estos son muy pocos para afectar la performance de manera significativa.

5.3.3. Combinacion lineal con floats vs enteros

La idea de este experimento es ver si hay una diferencia significativa en performance al hacer la combinacion lineal utilizando floats o utilizando enteros.

En esta nueva implementación cambia la manera en que calculamos la combinación lineal. A continuacion veremos como es esta implementacion para el color rojo, es analogo para los demas colores

Lo que hacemos es tomar el numero 255, pasarlo a float y multiplicarlo por VAL utilizando CVTDQ2PS y mulss respectivamente. Luevo volvemos este valor a bytes con CVTPS2DQ y hacemos 255 menos ese resultado. De esta manera tenemos: **XMM8**

?	?	?	$255 * VAL$

16 0

XMM9

?	?	?	$255 - (255 * VAL)$

16 0

De esta manera lo normalizamos a 255, queda entonces hacer las cuentas y dividir para normalizar a 1. En **XMM10** tenia guardado el maximo rojo y en **XMM2** tenia guardado el rojo original. Multiplicamos **XMM10** con **XMM8** para tener:
XMM10

?	?	?	$maxRojo * 255 * VAL$

16 0

Y **XMM2** con **XMM9** para tener:
XMM2

?	?	?	$Rojo * (255 - 255 * VAL)$

16 0

Hacemos ahora la suma con padd y nos queda:
XMM2

?	?	?	$Rojo * (255 - 255 * VAL) + maxRojo * 255 *$

16 0

Sacamos factor comun 255:
XMM2

?	?	?	$255(Rojo * (1 - *VAL) + maxRojo * VA$

16 0

Hacemos un shift right para deshacernos del 255 en la cuenta con psrld xmm2,8
XMM2

?	?	?	$Rojo * (1 - *VAL) + maxRojo * VAL$

16 0

Nuestra hipotesis es que al hacer menos conversiones a float y al hacer operaciones mas elementales con enteros, la cuenta con enteros sera más performante.

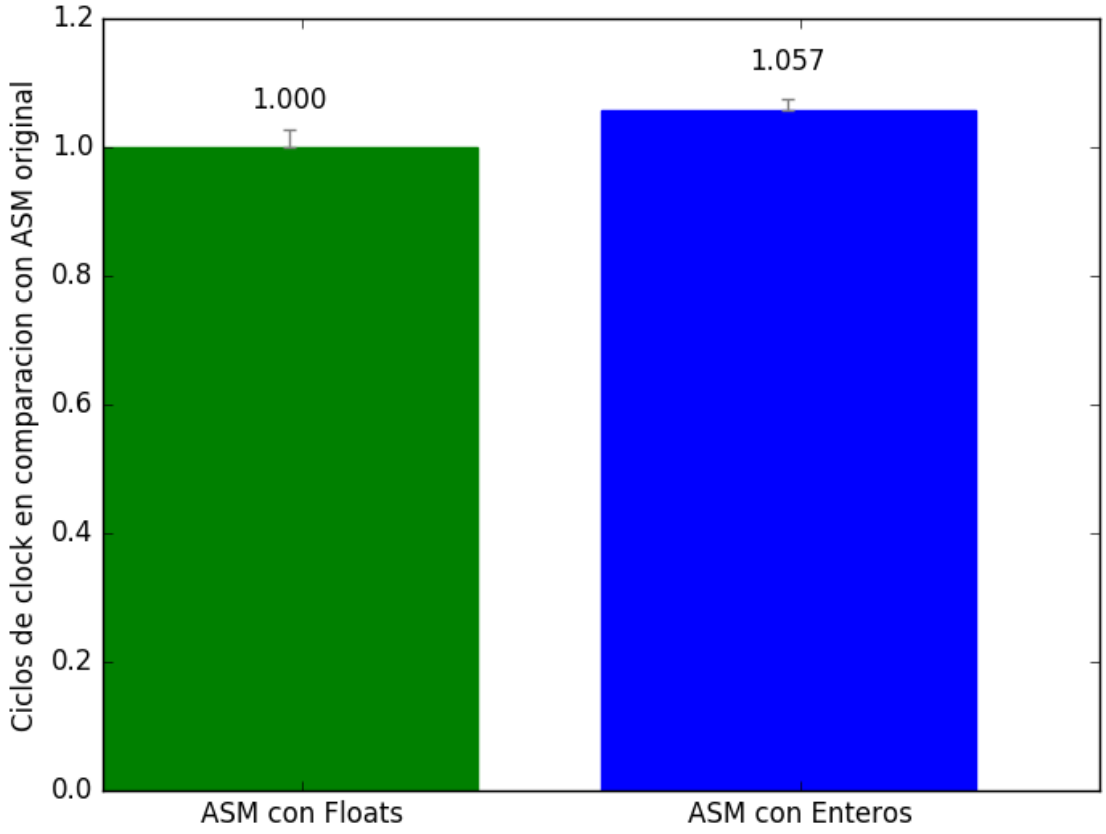


Figura 14: Comparacion de performance entre nuestras dos implementaciones de ASM. Utilizamos una imagen de lena de tamaño 512x512. La altura de las barras representa el promedio de las 100 mediciones luego que quitar los outliers y las lineas grises, el desvío estandar.

Pareceria que con enteros hay una diferencia de 5 % de performance a favor de los calculos con floats. Nos parecio una diferencia chica y poco intuitiva por lo que decidimos indagar más.

Decidimos hacer mediciones sobre las distintas partes de el codigo para conocer cuanto tarda cada una. Lo que hicimos fue comentar la parte del codigo que queriamos medir, le restamos al resultado del total el resultado de esa medición. De esta manera tenemos la cantidad de ciclos que conlleva hacer las operaciones comentadas.

Para estas mediciones usamos el mínimo ya que como estabamos contrastando entre diferentes corridas del codigo queriamos la que menos interrupciones haya tenido y por lo tanto la medicion que más se acerque al tiempo de la implementación.

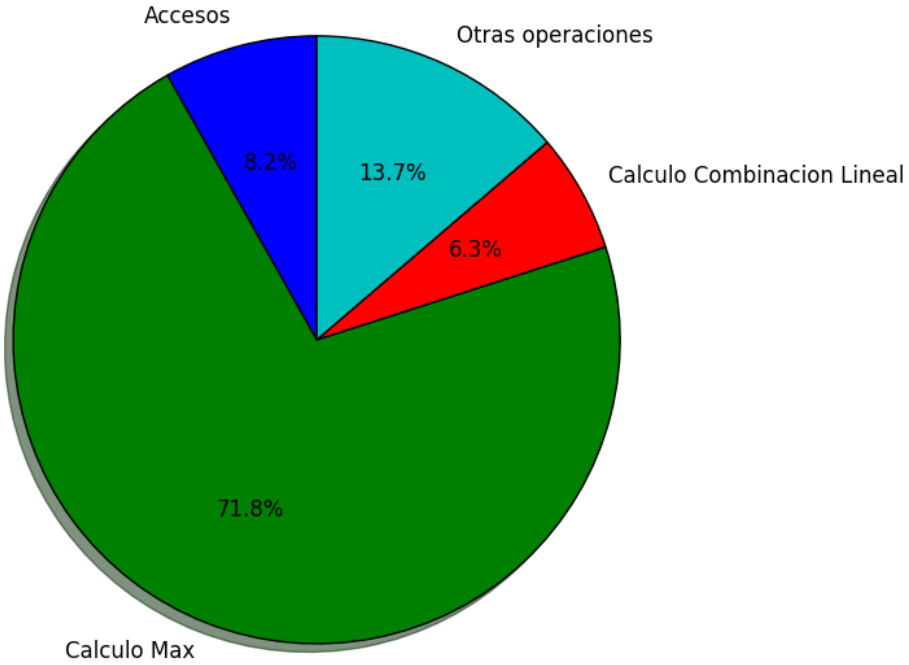


Figura 15: Comparacion de cantidad de cantidad de ciclos que conlleva cada parte importante del codigo. Calculo Max es el calculo del máximo en el kernel y otras operaciones son las operaciones que no estan directamente ligadas con los calculos ni accesos a memoria (copias de registros, jumps, el pintar los bordes de blanco y cargar las mascaras en registros).

Como conclusión podemos decir que el calculo de la combinacion lineal no aporta mucho al total de la ejecucion. Esto nos lleva a pensar que de el experimento anterior no podemos sacar una conclusion fuerte ya que la diferencia vista podria darse por un error de medicion. Como el calculo de la comnbinación impacta tan poco en la performance tambien podemos decir que no tiene sentido intentar mejorar la performance a partir de mejoras en este calculo. El cuello de botella se encuentra en el calculo del maximo en el kernel y es ahi donde se deberian buscar mejoras de perforance. Esta estambien la parte donde hemos aplicado el uso del set set instrucciones SSE y por lo tanto donde más se aprovechan. Esto se vé muy bien reflejado en el primer experimento cuando comparamos con C, donde no se usan instrucciones SSE.

6. Conclusión

En conclusión, programar en lenguaje Assembly requiere mucho más trabajo que programar en un lenguaje de más alto nivel como C. Sin embargo, el uso de SIMD en Assembly para programas paralelizables - como el procesamiento multimedia - mejora la performance drásticamente. Además, en Assembly se tiene mayor control sobre el hardware del sistema. Esto le da más herramientas al programador, pero si no las usa correctamente puede lleva a un deterioro de la performance del programa. Un ejemplo de uso incorrecto del hardware es el 2do experimento de LinearZoom, en el que nuestro código provoca un hazard en el pipeline. En lenguajes como C el compilador es el encargado de manejar estos recursos y dichos recursos se ven mejor protegidos a estos errores.

Como segunda conclusión podemos decir que se debe tener mucho cuidado al momento de hacer las mediciones. Repetir una gran cantidad de veces los experimentos es importante ya que con una corrida los resultados son muy vulnerables a ensuciarse con interrupciones del sistema operativo.

Finalmente, nos parece que hay que tener cuidado al elegir qué instrucciones se utilizan a la hora de programar en Assembly, ya que distintos grupos de operaciones pueden ser equivalentes en su resultado, pero no en su performance.

Referencias

[1] Intel, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 5.6.4 Pixel Format Conversion.