



DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Sin Filtro

Organización del Computador II
Primer Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Ignacio Alonso Rehor	195/18	arehor.ignacio@gmail.com
Manuel Panichelli	72/18	panicmanu@gmail.com
Vladimir Pomsztein	364/18	blastervla@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>

Resumen

Este paper cubre un análisis de rendimiento y efectividad del uso de instrucciones *SIMD* por sobre instrucciones normales o implementaciones en C.

Índice

1. Introducción	3
2. Desarrollo	3
2.1. Metodología	3
2.2. Filtros	3
2.2.1. Nivel	3
2.2.2. Bordes	4
2.2.3. Rombos	5
3. Experimentos	6
3.1. Sobre la performance del filtro de Nivel	6
3.2. Impacto de loop-unrolling en el rendimiento del filtro Bordes	6
3.3. Precálculo de valores en Rombos	7
4. Resultados y discusión	7
4.1. Performance del filtro de Nivel	7
4.2. Impacto de loop-unrolling en el rendimiento del filtro Bordes	9
4.3. Precálculo de valores en el filtro de Rombos	10
4.4. Posibles mejoras a implementaciones	11
4.4.1. Nivel	11
4.4.2. Bordes	11
4.4.3. Rombos	11
5. Conclusiones	12

1. Introducción

El procesamiento de imágenes resulta un interesante e importante foco de estudio en tiempos modernos, teniendo ésta área múltiples aplicaciones en distintos campos de la ciencia y el marketing o entretenimiento; es fundamental para la programación de inteligencias artificiales y también puede ser utilizado como un divertimento cuando aplicamos filtros en nuestra aplicación de celular favorita. Esta experimentación se avoca a analizar 3 filtros distintos que son aplicables sobre imágenes, dos de ellos sobre imágenes de 4 canales (3 de color y un cuarto de *Alpha*, u opacidad) y un tercero que se aplicará sobre imágenes de 1 canal (es decir, en blanco y negro).

De cada filtro se realizarán distintos experimentos, según amerite su comportamiento y complejidad.

2. Desarrollo

2.1. Metodología

Para la implementación de los distintos filtros, se hizo uso del lenguaje Assembly para la arquitectura x86 de procesadores (en particular los de 64 bits). Aprovechando las herramientas dispuestas por la tecnología utilizada, hicimos fuerte uso de las instrucciones **SIMD** (o Single Instruction Multiple Data) para optimizar los filtros y garantizar ejecuciones más rápidas de los mismos. Será uno de nuestros intereses analizar, luego, cuánto mejor funcionan nuestras implementaciones gracias a su uso.

A su vez, contamos con implementaciones de los algoritmos que describiremos a continuación escritas en C. Usaremos estas implementaciones como benchmarks, o puntos de partida sobre los que analizar la performance de nuestras implementaciones en Assembly.

2.2. Filtros

A continuación, se presentan los 3 filtros implementados, con una breve explicación de cada uno y algunas aclaraciones, relevantes según el caso, acerca de la implementación en particular.

Como fue mencionado en la introducción, dos de estos filtros trabajan con imágenes de 4 canales mientras que uno de ellos utiliza imágenes de canal único.

Las imágenes multi-canal utilizan los canales Rojo, Verde, Azul y Alpha (de 1 Byte cada uno, representando números enteros no signados), en ese orden. Las imágenes de canal único poseen simplemente el canal de Brillo (de 1 Byte representando un entero sin signo también).

2.2.1. Nivel

El más sencillo de los 3 filtros, la idea del filtro de Nivel es sencilla; detectar qué bits están en 1 en la representación binaria de cada canal, para cada píxel. Así, el objetivo es proveer un número i que actuará de índice, y saturar (para cada píxel) todos los canales en los que el valor del bit i sea 1, y llevarlos a 0 en caso contrario.

Puesto que es el filtro más sencillo, no hay demasiadas cosas para destacar de él. Su implementación con instrucciones SIMD realiza los siguientes pasos:

1. Se crea una máscara de 128 bits en un registro xmm donde todos los bits se encuentran en 0, exceptuando el bit del nivel indicado por parámetro (para cada byte del registro).
Por ejemplo, para el nivel 3, la máscara quedaría: 0x8888888888888888 (00001000 para cada byte).
2. Se procede a levantar 4 píxeles (con sus respectivos 4 canales de 1 Byte cada uno) en simultáneo.
3. Se realiza un **AND** con la máscara creada.
4. Luego, se compara cada byte para saber si es igual a 0. De esta manera, negando el resultado de la comparación obtenemos (casi) los bytes a escribir.
5. Se sanitiza los canales Alpha con una operación **OR**, consiguiendo como resultado dejar intactos todos los otros canales y configurar el Alpha en 255

6. Finalmente, se escribe el resultado descripto en la imagen destino.

A fines de demostrar su funcionamiento, podemos apreciar la siguiente imagen:



Figura 1: Imagen original (izq.) vs imagen con filtro de nivel 6 aplicado (med.) vs imagen con filtro de nivel 7 aplicado (der.).

2.2.2. Bordes

El filtro de bordes resulta particularmente interesante dados sus campos de aplicación. Es frecuentemente utilizado para *Computer Vision*, por lo que este filtro forma parte del cinturón de herramientas básico de cualquier programador de sistemas inteligentes de reconocimiento de características en imágenes.

El objetivo del mismo es el de obtener una imagen en la que se remarquen únicamente los bordes de los objetos presentes en la imagen original. Veamos la siguiente imagen, de ejemplo:



(a) Imagen original



(b) Imagen con el filtro bordes aplicado

Un aspecto que cabe destacar respecto de este filtro es que trabaja con imágenes de canal único (en blanco y negro), por lo que resulta necesario transformar cualquier imagen a este espectro previo a su utilización. En particular, para la implementación, se utilizó el *Operador Sobel* (o de Realzado). La idea, resumida, es realizar las siguientes operaciones (para *src* y *dst* las imágenes fuente y destino, respectivamente):

Sean

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Con *height* y *width* la altura y el ancho de la imagen (respectivamente):

```

1: for i in 1...height - 2 do
2:   for j in 1...width - 2 do
3:     totalGx ← 0
4:     totalGy ← 0

5:     for ii in 0...2 do
6:       for jj in 0...2 do

7:         totalGx ← totalGx + Gx[ii][jj] × src[i + ii - 1][j + jj - 1]
8:         totalGy ← totalGy + Gy[ii][jj] × src[i + ii - 1][j + jj - 1]
9:       end for
10:      end for

11:     dst[i][j] ← SAT(|totalGx|, |totalGy|)
12:   end for
13: end for

```

La implementación de este filtro con SIMD consiste en mantener 3 punteros apuntando a las filas anterior, actual y posterior de los píxeles a procesar. De esta manera, podemos leer los píxeles vecinos simplemente sumándole los offsets correspondientes a estos punteros. Una vez que tenemos los píxeles vecinos, podemos calcular $totalG_x$ y $totalG_y$, de todos los píxeles que estamos procesando, sumando los registros con los vecinos correspondientes. Tenemos que tener cuidado al hacer esto, pues el resultado de estos cálculos podría irse del rango representación que tenemos para los valores. Es por eso que debemos utilizar instrucciones de desempaquetado con cada uno de los registros que contienen a los píxeles vecinos.

Una vez que calculamos $TotalG_x$ y $TotalG_y$, los sumamos en un registro saturando sin signo y escribimos el resultado en la imagen destino.

Cabe destacar que luego de procesar todo el resto de la imagen, escribimos todos los píxeles del borde de la misma de color blanco.

2.2.3. Rombos

Finalmente, se encuentra el filtro de Rombos, cuyo objetivo es el de distorsionar levemente la imagen de manera tal que aparezcan unos tenues rombos de tamaño específico sobre la imagen.

Esto lo logra sumando un valor que se calcula exclusivamente a partir de la posición del píxel dentro de la imagen, sin dar importancia a su valor.

A su vez, internamente parte la imagen en sectores cuadrados del tamaño de rombo especificado, de manera que al final el valor calculado solo depende de la posición *relativa* del píxel dentro de su respectivo cuadrado *padre*.

Esto implica que todos los píxeles que estén en la misma posición pero en diferentes sectores tendrán sumados el mismo valor. De esta manera, destacamos, dicho valor podría ser precalculado.

Sin embargo, la implementación *inicial* que fue realizada para este filtro se limita a levantar únicamente de a dos píxeles realizando, a su vez, el cálculo en cada iteración.

```

1: size ← 64
2: for i in 0...height - 1 do
3:   for j in 0...width - 1 do
4:     ii ← ( $\frac{\text{size}}{2}$  - (i % size)) > 0?( $\frac{\text{size}}{2}$  - (i % size)) : - $\frac{\text{size}}{2}$  - (i % size))
5:     jj ← ( $\frac{\text{size}}{2}$  - (j % size)) > 0?( $\frac{\text{size}}{2}$  - (j % size)) : - $\frac{\text{size}}{2}$  - (j % size))
6:     x ← (ii + jj -  $\frac{\text{size}}{2}$ ) >  $\frac{\text{size}}{16}$ ?0 : 2 × (ii + jj -  $\frac{\text{size}}{2}$ )
7:     dst[i][j].b ← SAT(src[i][j].b + x)
8:     dst[i][j].g ← SAT(src[i][j].g + x)
9:     dst[i][j].r ← SAT(src[i][j].r + x)
10:    dst[i][j].a ← 255
11:  end for
12: end for

```

Ejemplo del filtro en acción:



(a) Imagen original

(b) Imagen con el filtro rombos aplicado

3. Experimentos

A continuación, se presentan los experimentos planteados para cada filtro, cada uno con su correspondiente hipótesis y motivación.

3.1. Sobre la performance del filtro de Nivel

Con el objetivo de obtener una idea de cuánto aumenta el rendimiento de las implementaciones al utilizar SIMD (en sus distintos exponentes, operando con más o menos datos en simultáneo) y en comparación con los distintos niveles de optimización del compilador GCC (O0, O1, O2 y O3), decidimos aprovechar el filtro más sencillo para realizar un *benchmarking* de estas mejoras de rendimiento.

Así, proponemos ejecutar 100 veces cada implementación (con el objetivo de tener varias mediciones que nos permitan ver las diferencias de performance independientemente del ruido generado por el sistema operativo) y luego comparar resultados. Se ejecutarán la implementación de *C*, compilada con los 4 distintos niveles de optimización mencionados previamente (O0, O1, O2, O3) y 3 implementaciones distintas en *ASM*, donde se variará la cantidad de píxeles trabajados en simultáneo con SIMD (tendremos de 1, 2 y 4 píxeles a la vez).

Esperamos que la peor implementación en *C* y la mejor en *ASM* estén a dos niveles distintos, siendo la de *ASM* más performante. Sin embargo, consideramos razonable que la peor implementación de *ASM* esté razonablemente cerca en rendimiento que la mejor de *C*.

3.2. Impacto de loop-unrolling en el rendimiento del filtro Bordes

La técnica de *loop-unrolling* (o *desenroscamiento de bucles*) consiste en disminuir la cantidad de iteraciones que debe realizar un ciclo repitiendo el código de cada iteración. A simple vista, la pregunta ¿Afecta al rendimiento del programa hacer *loop-unrolling*? parecería no tener mucho sentido de análisis,

pues abstrayéndonos de la microarquitectura del sistema, desenroscar un ciclo o no parecería ser simplemente una decisión de mera expresividad; de cómo uno decide escribir su algoritmo. Pero, teniendo en cuenta la microarquitectura del sistema, nos gustaría ver qué tanto impactaría esto al rendimiento del programa.

Con esta motivación, planteamos tres implementaciones en *ASM*, donde cada una reduce las iteraciones realizadas por el ciclo principal del programa a $\frac{1}{2}$, $\frac{1}{4}$ y $\frac{1}{8}$ de la original. Quisiéramos observar si el impacto en el rendimiento es perceptible en términos prácticos y, de ser así, qué tanto puede aumentar en consecuencia.

En un principio, esperaríamos que el rendimiento del programa mejore con cada factor de *desenroscamiento*, puesto que parece razonable pensar que al tener menos saltos condicionales en el programa disminuya la necesidad del buen funcionamiento del predictor de saltos, lo que resulta en menos posibles predicciones erróneas. Esto implicaría menos descarte de instrucciones pre-procesadas con el pipeline.

3.3. Precalculo de valores en Rombos

La implementación inicial de este filtro realiza los cálculos para cada píxel en cada iteración. Esto implica que se repite muchas veces el mismo cálculo, innecesariamente.

El experimento que proponemos consiste en *precalcular* previo al ciclo principal los valores que podrá tener cada posición de cada píxel y almacenarlos, por ejemplo, en una matriz. De esta forma, puede ser obtenido simplemente indexando la matriz desde el ciclo principal.

Uno tendería a suponer que con estos cambios la implementación funcionaría mucho más rápido, ya que se están ahorrando muchos cálculos. Pero, esto se hace bajo el costo de tener que ir a memoria una vez más (recordemos que ya tenemos que levantar el píxel) por iteración. Además, los cálculos que se realizan son relativamente simples; no hay operaciones costosas como, por ejemplo, la división.

No obstante, intuimos que será más rápido de todas formas, puesto que la porción precalculada es de pequeño tamaño (32KB) y por lo tanto entraría en una línea de la memoria caché, por lo que la penalización por ir a buscarlo debería ser poco relevante.

Por último, destacamos que por la naturaleza de la mejora, sus efectos deberían hacerse más evidentes mientras más grande sea la imagen de entrada, ya que de esta manera mayor será la cantidad de cálculos que se están precalculando.

Además, para imágenes chicas, debería ser mucho más notoria la penalidad de la ida a memoria adicional, a tal punto que seguramente sea peor.

4. Resultados y discusión

4.1. Performance del filtro de Nivel

En la figura 4, se presenta una imagen en el que se podrá apreciar gráficamente las diferencias de rendimiento para cada implementación, donde cada una de ellas fue ejecutada 100 veces sobre la misma imagen y con el mismo nivel (7). Dejar de lado, de momento, la variación *ASM(1 byte)*, hablaremos de ella en breve.

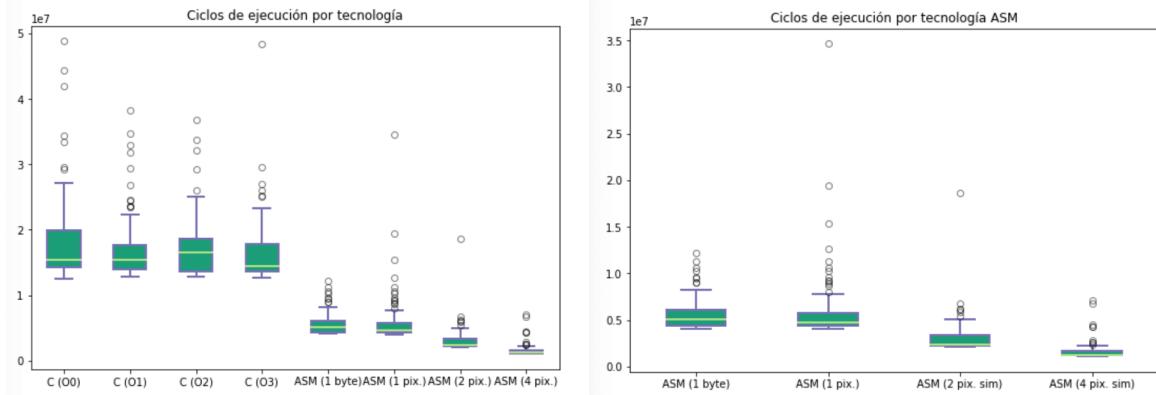


Figura 4: Ciclos de ejecución para cada implementación en *C* y *ASM* con implementación de *ASM* procesado de a 1 byte (izq.) y ciclos de ejecución para implementaciones de *ASM* (der.). Notar diferencias en escalas del eje coordenado

Como puede apreciarse en el gráfico, la diferencia de rendimiento entre las distintas implementaciones en *C* y las de *ASM* es altamente notoria. A esta gran diferencia adjudicamos dos posibles explicaciones:

- La implementación de *C* utiliza dos ciclos, por lo que se nos ocurre que los saltos podrían estar influenciando
- La implementación de *C* levanta un byte a la vez, mientras que la de *ASM* más lenta levanta 1 píxel entero a la vez (por lo tanto, 4 bytes).

Para comprobar esta última hipótesis, decidimos crear una cuarta implementación de *ASM* que levante de a byte, tal y como la implementación en *C* lo hace. Sin embargo, los resultados no fueron los esperados. Esta es la variación *ASM* (1 byte) que mencionábamos previamente y que puede observarse en la figura 4.

Como podemos ver en la figura 4, apenas si hay diferencia entre la implementación de *ASM* con SIMD de 1 píxel en comparación con la que levanta de a bytes. Intuimos que la memoria caché debe ser la responsable de esta falta de diferencia. A su vez, notamos que aún hay una gran diferencia entre la mejor implementación de *C* y esta última, por lo que proponemos utilizar branches (if's) en lugar de shifts aritméticos, como venimos haciendo en la implementación actual. Como veremos en el gráfico ??, esto hace que la implementación empeore levemente, pero no lo suficiente como para acercarse considerablemente a la implementación en *C*.

Además, observando los gráficos de la figura 5, resulta muy notoria la mejora de rendimiento de la implementación con SIMD que levanta de a 4 píxeles. En términos de la mediana, esta implementación corre en $\frac{1}{4}$ del tiempo que la *ASM* (1 byte) y en $\frac{1}{10}$ del tiempo que cualquier implementación de *C*. Podremos ver todas estas comparaciones en la figura 5.

Cabe destacar, por otro lado, que los resultados de rendimiento del filtro para otros niveles fueron los mismos, por lo que el su performance resulta independiente del nivel que se utilice.

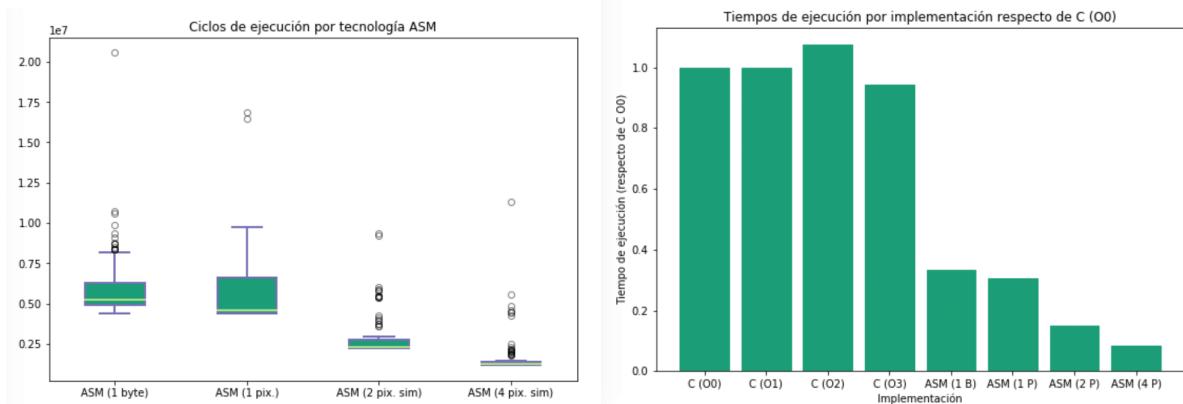


Figura 5: Ciclos de ejecución para implementaciones de *ASM* utilizando branches para la implementación que trabaja de a byte (izquierda) y Tiempo de ejecución para cada implementación respecto de *C* (00) (derecha). Resultados según la mediana de cada medición

Concluimos, luego, que las mejoras en rendimiento obtenidas al comenzar a trabajar con *SIMD* son excepcionales, como pudimos ver. Por lo tanto, resulta súmamente conveniente utilizarlo de ser posible. Más aún, pudimos comprobar que el uso de *ASM* por sobre *C* tiene sus beneficios, incluso sin aprovechar tecnologías como *SIMD*.

4.2. Impacto de loop-unrolling en el rendimiento del filtro Bordes

A continuación presentamos una imagen con las muestras del experimento realizado. Como veremos en ella, el hecho de hacer loop-unrolling mejora notablemente el rendimiento del programa. Sin embargo, cuando el factor de desarrollo del ciclo principal alcanza 8, no solo no notamos una mejoría en el rendimiento del programa, sino que este baja en comparación con la implementación de factor de desarrollo = 4.

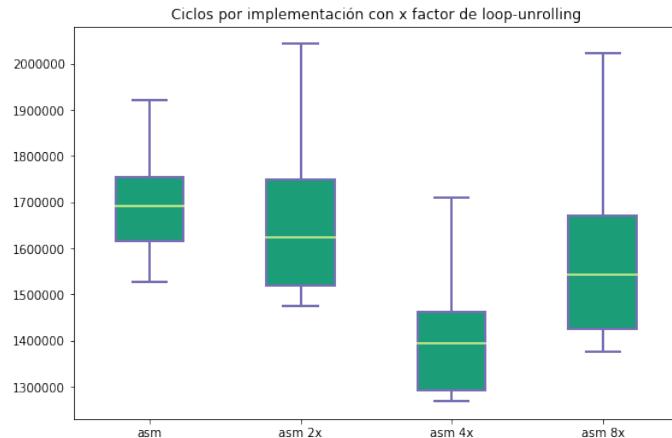


Figura 6: Cantidad de ciclos para las distintas implementaciones utilizando distintos grados de loop-unrolling

Además, cuando comparamos la mediana para las distintas implementaciones, (como puede verse en la siguiente figura) podemos apreciar que la implementación que más aumenta el rendimiento del programa, *ASM 4x*, se ejecuta en $\frac{1}{5}$ del tiempo de lo que toma la implementación sin loop-unrolling.

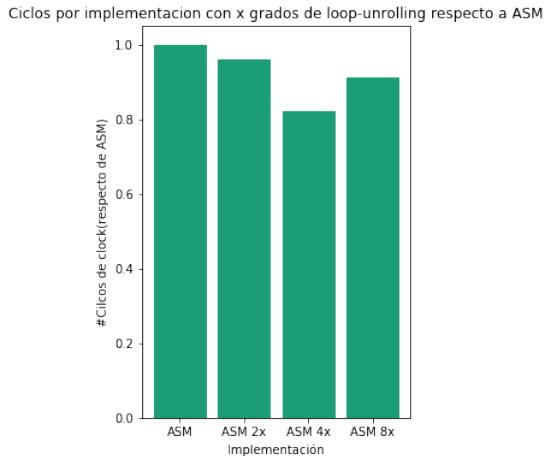


Figura 7: Rendimiento comparado al de control de las distintas implementaciones usando distintos grados de loop-unrolling

4.3. Precálculo de valores en el filtro de Rombos

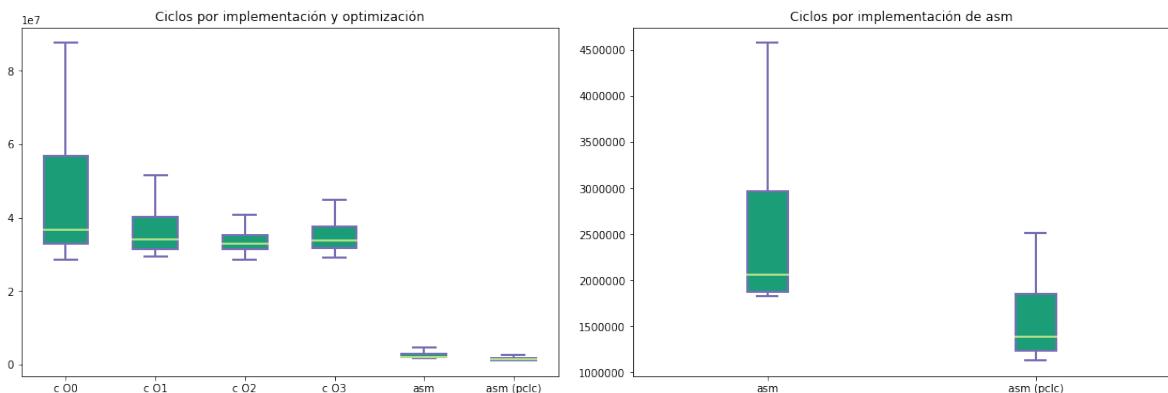


Figura 8: Ciclos ejecución para cada implementación / optimización en *C* y *ASM*

En estas figuras podremos apreciar varios aspectos sobre la implementación de rombos. La metodología usada en este caso fue iterar 100 veces una corrida del filtro sobre una imagen de 1024x600 para cada implementación. Destacamos, luego, las siguientes observaciones:

- La implementación en *ASM* resulta más veloz que cualquier optimización de *C*, como ya era de esperarse luego de los resultados de las experimentaciones del filtro de Nivel.
- Notemos que la implementación que realiza el precálculo resulta más performante en términos de tiempo hasta para una imagen relativamente chica como con la que se realizó la prueba. Y, ¿qué pasará con imágenes de distintos tamaños?

Para responder esa última pregunta, corrimos cada implementación en *ASM* 150 veces para diversos tamaños de imagen.

Los resultados pueden verse en la figura 9.

- Para imágenes chicas, la implementación que no realiza el precálculo es más veloz que la que sí, lo cual se ajusta a lo predecido anteriormente.
- A medida que las imágenes se hacen cada vez más grande, empieza a notarse más la diferencia en velocidad, y importa cada vez menos el overhead generado por el precálculo y la ida a memoria.

De esta manera, comprobamos que el hecho de tener que ir una vez más por iteración a memoria no produjo mayores impactos en la eficiencia temporal. Por lo tanto, podríamos pensar que la caché está ayudando fuertemente a esta implementación en ese aspecto.

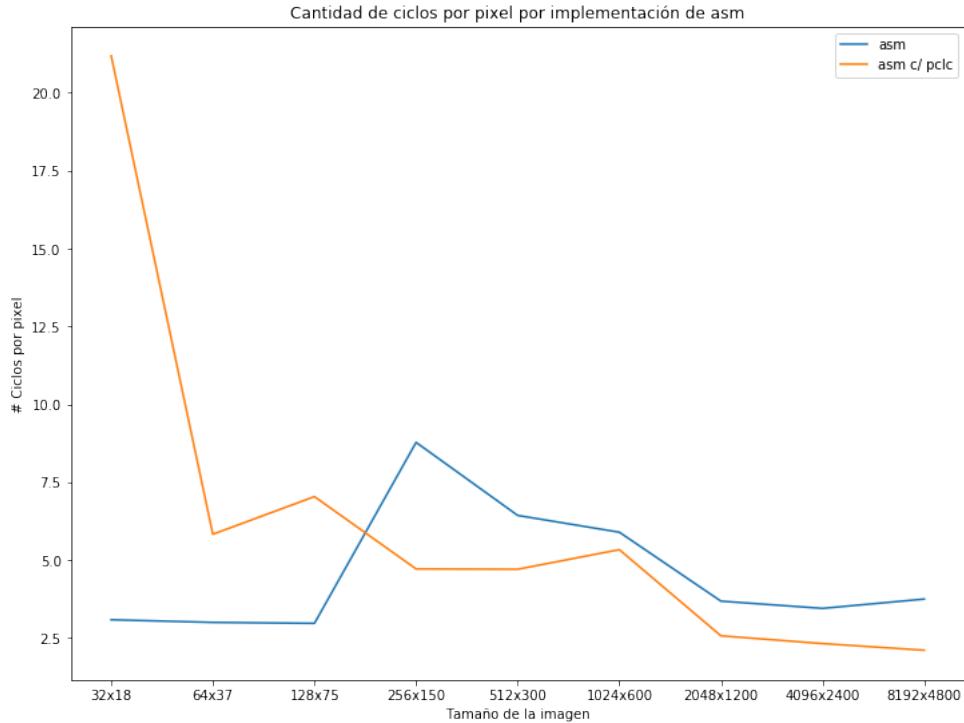


Figura 9: Ciclos de ejecución para cada implementación de ASM

4.4. Posibles mejoras a implementaciones

4.4.1. Nivel

La implementación del filtro en cuestión fue realizada con comparaciones empaquetadas, pero bien podría haberse hecho uso de operaciones como el packed-shift aritmético para obtener el resultado deseado. Simplemente, habría bastado con shiftear a izquierda de manera que en el bit más significativo quede el bit de nivel y luego hacer un shift aritmético a derecha de 7 bits para extender el signo. De esta manera nos quedaría el resultado deseado sin necesidad de comparaciones y negaciones.

4.4.2. Bordes

Para la implementación de este filtro, se planteó procesar 8 píxeles en simultaneo para disminuir la cantidad de instrucciones de desempaquetado. La elección fue motivada por aumentar la legibilidad del código y facilitar su escritura; pero si no tenemos en cuenta esos factores, podríamos procesar 16 píxeles en simultaneo, así aumentando el rendimiento de la implementación.

4.4.3. Rombos

Independientemente de la mejora propuesta durante la experimentación, se puede pensar trivialmente en una oportunidad de mejora levantando 4 píxeles a la vez en lugar de 2.

Ya que se levantan valores como enteros (32 bits) cuando en realidad son shorts (16 bits) se podrían almacenar en la mitad de espacio, y procesar el doble a la vez, lo cual permitiría efectivamente levantar 4 píxeles.

5. Conclusiones

Notamos como conclusiones:

El hecho de que el uso de *SIMD* supone una gran mejora en la performance de un algoritmo al ser utilizado, pudiendo mejorar al menos diez (10) veces la velocidad de ejecución del mismo en comparación a otras tecnologías (como *C*). Además, resulta muy notoria la mejora de rendimiento entre implementaciones en *C* y en *ASM*, incluso sin el uso de *SIMD*, y observamos que la memoria caché es una gran aliada cuando este último conjunto de instrucciones no se aprovecha.

Por otra parte, notamos que loop-unrolling puede ser bastante útil para mejorar el rendimiento de una implementación, siempre y cuando se utilice de forma medida (pues, como pudimos ver, a partir de cierto punto comienza a tornarse contraproducente).

En cuanto a lo que a precálculos y su almacenamiento en memoria se refiere, concluimos que esto puede contribuir a notorias mejoras en el rendimiento del algoritmo, siempre y cuando se aproveche la tecnología de memoria caché, ya sea teniendo un precálculo suficientemente pequeño de forma tal que entre en una sola línea de la misma, o leyéndolo de forma inteligente para que no se vaya desalojando.