Organización del Computador II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico Número 2

Procesamiento de imágenes

Integrante	LU	Correo electrónico
Alcain Pablo	485/07	pabloalcain@gmail.com
Gamarra Ignacio	792/11	gamarra_mi32@yahoo.com.ar
Santos Diego	874/03	diego.h.santos@gmail.com

Grupo

My Generation

Índice

1. Introducción

En este trabajo hemos realizado la implementación de funciones de procesamiento de imágenes en dos lenguajes de programación, *Assembler* y *C*. Además realizamos un análisis de la performance de las mismas a fin de determinar cual es mas eficiente respecto al tiempo de ejecución. Las funciones implementadas son:

- `Diferencia de Imágenes`, a partir de dos imágenes genera una tercera imagen que resalta los pixeles donde las dos imagenes fuentes difieren. Para obtener la tercer imagen se toma la norma infinito la norma infinito de la resta vectorial entre los píxel, ignorando el canal alfa. La fórmula a aplicar para obtener la tercer imágen es;

- `Blur Gaussiano`, tomando una imagen de fuente genera una segunda similar a la fuente pero con un aspecto desenfocado. La manera de lograr este efecto es calculando cada componente de la imagen de salida como un promedio ponderado con una gaussiana 2D de los píxeles que la circundan. Las formulas a aplicar para realizar el filtro son las siguientes:

En el caso de las implementaciones hechas en *Assembler* hemos usado el modelo de programación *SIMD*, a través del set de instrucciones `SSE`, pues el objetivo de este trabajo práctico es estudiar las ventajas y desventajas de usar ese modelo contra uno *SISD*.

En las siguientes secciones se explicará como se implementaron los filtros, se presentarán gráficos mostrando los tiempos de ejecución de cada filtro comparando ambas implementaciones y luego se presentará una conclusión acerca de los resultados obtenidos y el costo de usar un modelo *SIMD* para programar.

2. Desarrollo

2.1. Diferencia de Imagenes

Hicimos el filtro `diff` tanto en C como en ASM. Su implementación es bastante inmediata, al menos sin tener en cuenta los registros `XMM` de `SSE`. De esta manera, el filtro `diff` recorre todos los píxeles de ambas imágenes, de forma secuencial, resta los tres componentes de colores y toma el valor absoluto. Así, en cada píxel (i, j) [de 4 bytes cada uno], obtenemos tres bytes $(\Delta R, \Delta G, \Delta B)$. En este filtro `diff`, convenimos que la imagen diferencia fuera la norma infinito del vector en las tres posiciones (es decir, el máximo entre los tres). Así, una vez obtenidos $(\Delta R, \Delta G, \Delta B)$, calculamos el máximo entre los tres, ΔC, y en el píxel (i, j) de la imagen de salida anotamos los tres bytes $(\Delta C, \Delta C, \Delta C)$ y 0xFF en el canal alpha. Repetimos este procedimiento para todos los píxeles de la imagen.

En la implementación de ASM, aprovechamos los registros `SSE` disponibles en el procesador[1]. En cada registro entran 16 bytes: 4 píxeles que pueden procesarse a la vez. Ahora procesarlos no es tan trivial como antes, pero podemos ver que pensar algunas relaciones básicas de álgebra, ayuda a ver el camino a seguir. Recordemos los dos pasos:

Restar los tres componentes y tomar valor absoluto La clave para reducir este problema es considerando la siguiente igualdad:

$$\text{abs}(a - b) = \text{máx}(a, b) - \text{mín}(a, b) \tag{1}$$

Así, simplemente tenemos que asegurarnos de estar restando el máximo (byte a byte) menos el mínimo. Para eso, supongamos que tenemos las tiras de 16 bytes de ambas imágenes en `XMM0` y `XMM3`. Ahora, copiamos cualquiera de los dos (por caso, `XMM0`) a un registro extra `XMM1`, y guardamos `máx (XMM0, XMM3)` en `XMM0` [`pmaxub`]. Luego, guardamos `mín (XMM1, XMM3)` en `XMM1` [`pminub`]. Ahora, `XMM0` tiene los máximos y `XMM1` los mínimos, byte a byte, de los 4 píxeles. Los restamos byte a byte y guardamos el resultado en `XMM0` [`psubb`]. Hacemos aquí unas cuentas extras, ya que también estamos procesando el canal alpha. Esto no significa un detrimento en performance (ya que la operación está paralelizada), sino que si la información estuviera guardada en RGB sin canal alpha se podrían procesar más píxeles a la vez[2]. Estos pasos se ven esquematizados en la figura **??**

Obtener el máximo Ahora el desafío es encontrar, con operaciones `SSE`, una forma de encontrar el máximo de cada bloque de 4 bytes. Pensemos cada bloque de 4 bytes por separado primero. Tenemos tres valores R, G, B, A y queremos que esto se transforme en C, C, C, X [C es el máx(R, G, B) y X es el canal alpha, que por ahora puede tomar cualquier valor]. Si $\mathbf{t}$ es el vector de salida, queremos

$$\mathbf{t} = (\text{máx}(R, G, B), \text{máx}(R, G, B), \text{máx}(R, G, B)) \tag{2}$$

Para vectorizar esta expresión, tenemos que usar la siguiente identidad:

$$\text{máx}(R, G, B) = \text{máx}(R, \text{máx}(G, B)) = \text{máx}(G, \text{máx}(B, R)) = \text{máx}(B, \text{máx}(R, G)) \tag{3}$$

Definimos al vector $\mathbf{p} = (R, G, B)$ y al operador `rot` tal que:

[1] El procesador utilizado, intel i7, tiene disponible `AVX`, pero no utilizamos estas instrucciones

[2] Claro que ahora no podríamos garantizarnos esta hermosura de que entren *justo* 4 píxeles en un registro `SSE`

$$\text{rot}(R, G, B) = (G, B, R) \tag{4}$$

Las sucesivas aplicaciones de `rot` las definimos también:

$$\mathbf{q} = \text{rot}(R, G, B) = (G, B, R) \tag{5}$$
$$\mathbf{r} = \text{rot}(\text{rot}(R, G, B)) = (B, R, G) \tag{6}$$

Con esta notación, la identidad 3 queda

$$\text{máx}(R, G, B) = \text{máx}(p_1 \text{máx}(q_1, r_1)) = \text{máx}(p_2, \text{máx}(q_2, r_2)) = \text{máx}(p_3, \text{máx}(q_3, r_3)) \tag{7}$$

y, en consecuencia, $\mathbf{t}$ es

$$\mathbf{t} = \text{máx}(\mathbf{p}, \text{máx}(\mathbf{q}, \mathbf{r})) \tag{8}$$

Esta ecuación ahora está vectorizada y podemos usar `pmaxub` para calcular los máximos Sólo queda crear, a partir de $\mathbf{p}$ los vectores rotados $\mathbf{q}$ y $\mathbf{r}$; es decir, implementar el operador `rot`. Para esto, utilizamos una máscara que rota a la izquierda los tres bytes, es decir: $(R, G, B) \to (G, B, R)$, utilizando `pshufb` y una máscara del tipo (1, 2, 0).

Como, en rigor, en cada píxel tenemos también el canal alpha, la máscara para cada byte va a ser (1, 2, 0, 0xFF), poniendo un 0 en el canal alpha [esto también podría ser la posición 3 o cualquier otra, porque el canal alpha *tiene que* ser reescrito].

Esto explica el procedimiento general. Aumentarlo para procesar n píxeles en paralelo es directo, simplemente la máscara de rotación va a ser $\mathbf{m} = (1, 2, 0, 0\text{xFF}, 4+1, 4+2, 4+0, 0\text{xFF}, 8+1, 8+2, 8+0, 0\text{xFF}, \ldots)$. En general,

$$m_i = \begin{cases} 4 * \text{floor}(i/4) + \text{mod}(i+1, 3) & \text{mod}(i, 4) \neq 3 \\ 0\text{xFF} & \text{mod}(i, 4) = 3 \end{cases} \tag{9}$$

La implementación entonces es directamente una traducción de esto: Copiamos el registro `XMM0` ($\mathbf{p}$) a un registro `XMM1` y mantenemos la máscara de rotación en `XMM2`. Ahora rotamos `XMM1` con `pshufb` y obtenemos $\mathbf{q}$. Obtenemos el máximo de `XMM0` ($\mathbf{p}$) y `XMM1` ($\mathbf{q}$) con `pmaxub` y lo guardamos en `XMM0`. Rotamos nuevamente `XMM1` y resulta $\mathbf{r}$. Ahora calculamos el máximo de `XMM0` ($\text{máx}(\mathbf{p}, \mathbf{q})$) y `XMM1` ($\mathbf{r}$) y lo guardamos en `XMM0` (el objetivo final, $\mathbf{t}$).

En la figura **??** se puede observar un esquema de este procedimiento.

2.2. Blur Gaussiano

El desarrollo del filtro de `blur` también lo realizamos en C y en ASM. Este filtro está caracterizado por el radio r y por el parámetro de difusión σ. Vamos a utilizar los siguientes términos

Marco: El recuadro de tamaño r que bordea a la figura, donde los píxeles no se afectan.

Centro: La parte de la figura que no es el marco.

Filtro: La matriz con la que se convoluciona cada píxel.

Vecinos: Los píxeles cercanos que afectan el valor de salida de un píxel en la imagen procesado. Es decir, todos los elementos que están a una distancia (en norma 1) menor a r de un píxel dado.

En este filtro, tenemos que aplicar la siguiente relación que mencionamos en la introducción:

$$O[i,j,k] = \sum_{x=-r}^{y=r} I[i+x, j+y, k] K_{\sigma,r}[r-x, r-y] \tag{10}$$

para todos los píxeles en el *centro*. Así, la implementación en C es, para cada componente (R,G,B) de cada píxel (i,j) del *centro*, hacer un ciclo en (x,y) entre $(-r,-r)$ y (r,r). Un detalle es la identificación del vecino (x,y) del pixel (i,j) con la posición absoluta en la imagen, $(i+x, j+y) = (i+x)*n_{\text{cols}}+(j+y)$. Así, con los valores de (R,G,B) del píxel (i,j) de la imagen salida inicializados a 0, acumulamos en cada uno $I[i+x, j+y]K_{\sigma,r}[r-x, r-y]$. Luego, redondeamos este resultado (que es de punto flotante de 4 bytes) a un char y lo guardamos en la imagen de salida.

Unas consideraciones hay que realizar en el cálculo del filtro:

1. Normalización: a pesar de que la función gaussiana está normalizada a 1, no sucede en este caso (ya que está discretizada y además truncada). De este modo, el coeficiente de normalización del filtro tiene que ser calculado para cada uno, de modo tal que la suma de todos los elementos de la matriz sea 1.

2. Cálculo: para calcular los elementos de una gaussiana hay que realizar muchos cálculos de punto flotante. Para evitarnos repetir estos cálculos una y otra vez, y como el filtro es siempre el mismo, lo implementamos como una *look up table* y lo guardamos en la memoria[3]. La creación de la LUT es en C [de cualquier modo, esta cuenta se realiza *una sola vez por imagen*, así que no es costoso su cálculo].

La implementación en ASM fue realizada nuevamente utilizando las instrucciones `SSE`. Para eso, tenemos que vectorizar el filtro. Llamando $\mathbf{T}$ a la matriz de píxeles de salida (es decir, $\mathbf{T}[i,j] = \mathbf{t}_{i,j} = (R^s_{i,j}, G^s_{i,j}, B^s_{i,j})$) y $\mathbf{P}$ a la matriz de píxeles de entrada ($\mathbf{P}[i,j] = \mathbf{p}_{i,j} = (R^0_{i,j}, G^0_{i,j}, B^0_{i,j})$) la ecuación 10 anterior queda

$$\mathbf{T}[i,j] = \sum_{x=-r}^{y=r} \mathbf{P}[i+x, j+y] K_{\sigma,r}[r-x, r-y] \tag{11}$$

$$\mathbf{t}_{i,j} = \sum_{x=-r}^{y=r} \mathbf{p}_{i+x,j+y} K_{\sigma,r}[r-x, r-y] \tag{12}$$

$$\tag{13}$$

Escribimos cada término de la suma:

$$\hat{\mathbf{p}} = \mathbf{p}_{i+x,j+y} K_{\sigma,r}[r-x, r-y] \hat{\mathbf{p}} = \mathbf{p}_{i+x,j+y}\mathbf{k} \tag{14}$$

Aquí, $\mathbf{k} = (K_{\sigma,r}[r-x, r-y], K_{\sigma,r}[r-x, r-y], K_{\sigma,r}[r-x, r-y], K_{\sigma,r}[r-x, r-y])$ es el filtro expandido hacia un vector, y el último producto es un producto elemento a elemento. Así, llegamos a una ecuación

[3]Como esta matriz es utilizada muy seguido, intuimos que se mantiene siempre en cache, de forma tal que su acceso sería bastante rápido.

vectorizada en (R, G, B, A), elementos contiguos en memoria que, al ser floats de 4 bytes cada uno, es el tamaño de los registros `XMMn`. En la implementación de ASM, precalculamos algunos valores que quedan en los registros:

1. Primer elemento del cuadrado de vecinos: La distancia, en memoria, entre el píxel (i, j) y el píxel $(i - r, j - r)$:

$$\begin{aligned} d_{1v} &= \text{pos}[(i-r),(j-r)] - \text{pos}[i,j] && (15)\\ &= [(i-r)n_{\text{cols}} + (j-r)] - [i\, n_{\text{cols}} + j]\, 4\text{bytes} && (16)\\ &= (-r\, n_{\text{cols}} - r)\, 4\text{bytes} && (17)\\ &= -r\,(n_{\text{cols}} + 1)\, 4\text{bytes} && (18) \end{aligned}$$

2. Nueva fila en el filtro: La distancia, en memoria, entre el píxel $(i+x, j+r)$ y el píxel en la siguiente fila $(i + (x + 1), j - r)$:

$$\begin{aligned} \Delta &= \text{pos}[(i+x),(j+r)] - \text{pos}[i+(x+1),(j-r)] && (19)\\ &= [(i+x)n_{\text{cols}} + (j+r)] - [(i+x+1)n_{\text{cols}} + (j-r)]\, 4\text{bytes} && (20)\\ &= (-n_{\text{cols}} + 2\, r)\, 4\text{bytes} && (21) \end{aligned}$$

Los ciclos ahora también van a ser sobre el *centro* de la imagen, dentro de cada cual calculamos la salida para el primer píxel (i, j). Destinamos un registro, por ejemplo `XMM1`, al acumulador. Dentro de él vamos sumando (con `addps`) cada uno de los términos de acuerdo a la expresión 14. Vamos a explicar con detalle cómo se calcula esta expresión para el primer píxel vecino en $(i - r, j - r)$, que es la parte vectorizada.

Primero, cargamos los 4 bytes del píxel en un registro, llamémoslo `XMM2`. Ahora queremos convertirlos a punto flotante. Para eso, tenemos que considerar que la carga anterior llena la parte baja del registro. Para convertirlo, primero vamos a extender estos 4 bytes a 4 doublewords en un registro temporal `XMMt` (mediante `pmovzxbd`) y luego los convertimos a números de punto flotante de 4 bytes con `cvtdq2ps` y los guardamos de nuevo en `XMM2`. En este punto, `XMM2` tiene guardados los 4 "colores" en una representación de punto flotante ($\mathbf{p}_{i+x,j+y}$). Ahora, en la parte baja del registro `XMM3` cargamos los 4 bytes (con `movd`) correspondientes al valor del filtro para $(-r, -r)$ (usando d_{1v}). Ahora, para crear el vector $\mathbf{k}$ hay que expandirlo (hacer un broadcast del elemento 0, viendo al registro como 4 números de punto flotante de 4 bytes) a todos los elementos, a través de `suhfps XMM3, XMM3, 0x00`. Ya tenemos los dos vectores, y los multiplicamos y lo guardamos en `XMM2` con `mulps`. Ya estamos listos para acumularlo en `XMM1` (`addps`). Ahora pasamos al siguiente elemento en el filtro y en la imagen (agregando 4 bytes). Este procedimiento lo realizamos para toda una fila, y luego, para pasar a la siguiente columna sumamos Δ a la posición de memoria de la imagen. Este procedimiento se puede observar en el esquema **??**.

Terminado esto, ¡ya tenemos procesado el píxel! Hacemos esto para toda la fila [sumando de a 4 bytes para pasar al siguiente elemento] y, para evitar el *marco*, en el último elemento de la fila sumamos $2\, r$ 4bytes para pasar al siguiente elemento del *centro*.

3. Resultados

En esta sección se muestran los tiempos de ejecución de cada algoritmo comparandose las implementaciones hechas en **C** contra las implementaciones hechas en **ASM**.
Dichos resultados se corresponden con la cantidad de ticks del procesador que cada algoritmo tomó, usando para ello el parámetro **-t** del programa principal, cuyo valor devuelto es justamente la cantidad de ticks para ejecutar el filtro de entrada una cantidad de veces equivalente a la pasada como parámetro.

Como medimos los ticks del procesador? Es algo que se hace internamente en el código que la cátedra nos proporcionó pero de todas formas sabemos que esto se lleva a cabo usando la instrucción **rdtsc**, la cual obtiene el Time Stamp Counter (TSC). Dicho registro se incrementa en uno con cada ciclo del procesador, de modo que la cantidad de ciclos total equivale a la diferencia del valor después y antes de ejecutar cada filtro.
Notar que este registro es global y por ende cuenta ticks que todos los procesos del sistema estan consumiendo, no solo el nuestro, de modo que sería incorrecto hacer solo una medición, en vez de eso hacemos 1000 y tomamos el promedio, para suavizar outliers (observación numéricamente muy distante al resto de los valores).

3.1. SpeedUp

En computación paralela el *speedup* refiere a cuánto más rápido es un algoritmo paralelo (en nuestro caso refieren a las implementaciones en Asm que hacen uso de las instrucciones SSE) que el correspondiente algoritmo secuencial (lo que sería cada implementación en C).

Porqué consideramos importante medir que tanto más rápido es la versión paralela que la secuencial? Si bien este tipo de análisis excede originalmente lo pedido por la cátedra, nos pareción razonable dar una idea de la magnitud de que tantas veces es mejor una implementación paralela que secuencial. Si bien este valor se desprende de la cantidad de ticks insumidos por cada implementación, nos pareció adecuado formalizarlo usando el concepto de *speedup* entre las dos implementaciones. Se calcula con la fórmula:

$$S_{ivsj} = \frac{T_i}{T_j} \tag{22}$$

En nuestro caso, será:

- T_i: cantidad de ticks del algoritmo en C
- T_j: cantidad de ticks del algoritmo en ASM

Para el análisis de benchmark generamos un script en `python` que genera imagenes al azar, de distintas dimensiones.

3.2. Comparaciones entre C y ASM: Dependencia en tamaño

Para un *aspect ratio* 1:1, comparamos el tiempo que tarda C con optimización -O3 y ASM en procesar cada imagen.

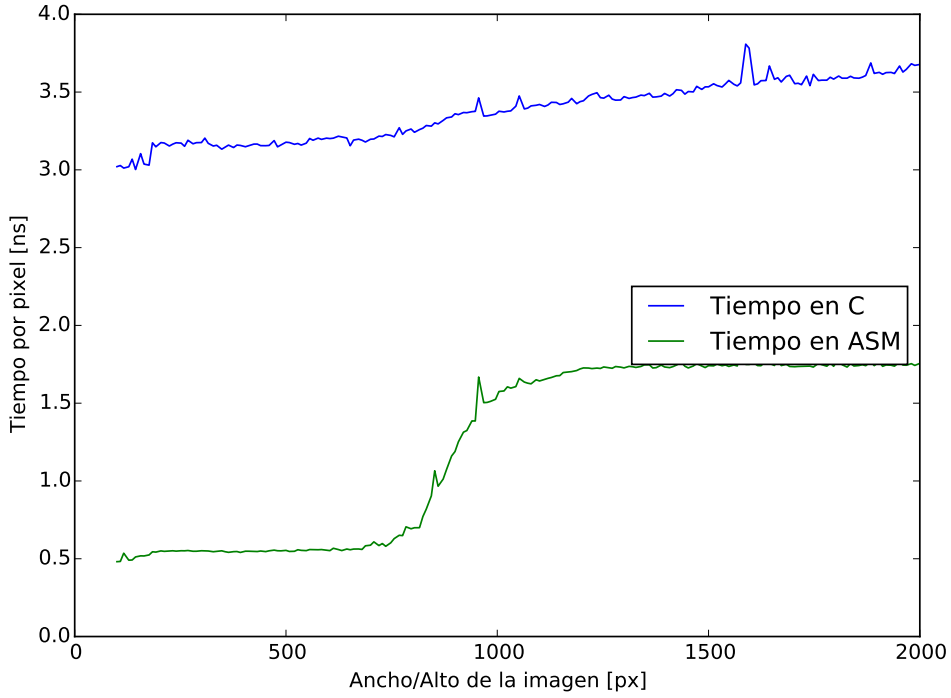

Figura 1: Tiempo por píxel para imágenes cuadradas. Observamos un salto, para ASM, entre 800 y 1000 píxeles.

3.2.1. Diferencia de Imagenes

En la figura 1 observamos el tiempo de procesado, por píxel, para varias imágenes entre 100x100 y 2000x2000 píxels.

Observamos una muy pequeña tendencia creciente en el tiempo de procesado por píxel para la implementación en C (20 % más lento en imágenes de 100 veces la cantidad de píxeles), mientras que en el caso de ASM el tiempo parece ser constante entre 200 y 800 píxeles y entre 1000 y 2000, con una bastante abrupta transición en la que se triplica el tiempo de procesado entre 800 y 1000 píxeles. Para 800x800 píxeles de imagen, tenemos en total 0,64 MP $\cdot$ 4bytes/pixel = 2,5MB. Como son en total tres imágenes, estamos usando 7,5MB de memoria, que es el tamaño de la cache L3 del procesador utilizado. Se vuelva a estabilizar al llegar a 1000x1000, 4MB por imagen (dos imágenes entrarían en cache).

En la implementación de C no se observa ese comportamiento, probablemente debido a que la optimización O3 realiza un algoritmo *cache aware*, mediante el flag `-freorder-blocks-and-partition`. Este mismo salto, y el hecho de que suceda sólo en la implementación en ASM, implica un salto también en el *speedup*, como se ve en 2.

3.2.2. Blur Gaussiano

En la figura 3 observamos el tiempo de procesado, por píxel, para varias imágenes entre 100x100 y 2000x2000 píxels.

En este gráfico vemos que ambos algoritmos arrancan con una tendencia creciente hasta los 300 px, a partir del cual el tiempo por píxel se mantiene constante. Que ambos sean de la misma tendencia se corrobora en el estudio del *speedup*, en el que se ve que se mantiene uniforme para todos los tamaños

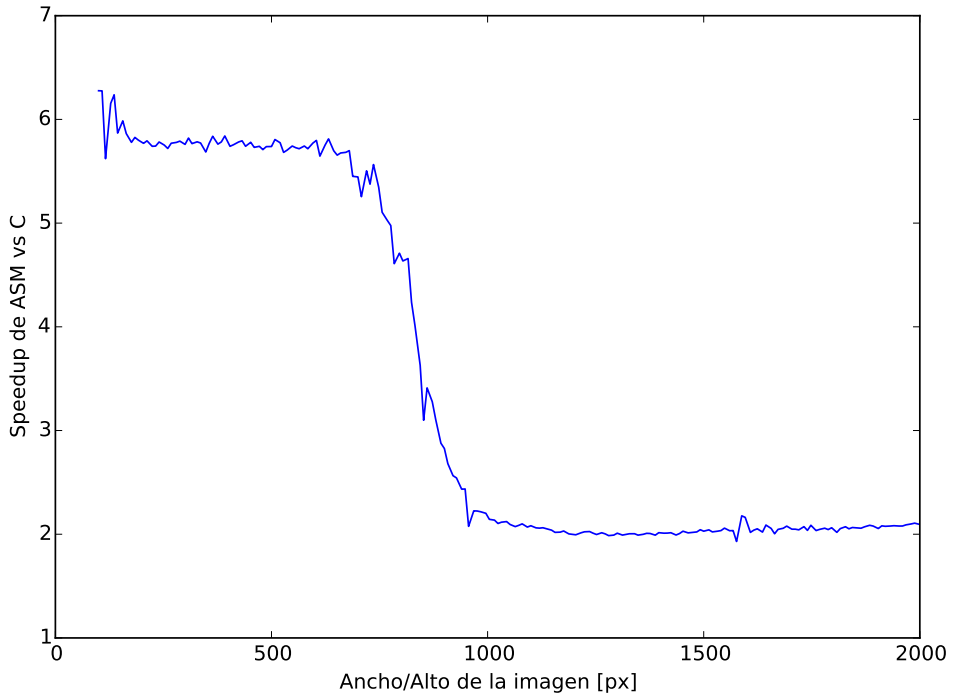

Figura 2: *Speedup* para imágenes cuadradas. Observamos un salto entre 800 y 1000 píxeles, que se traslada de la baja de performance en el código de ASM

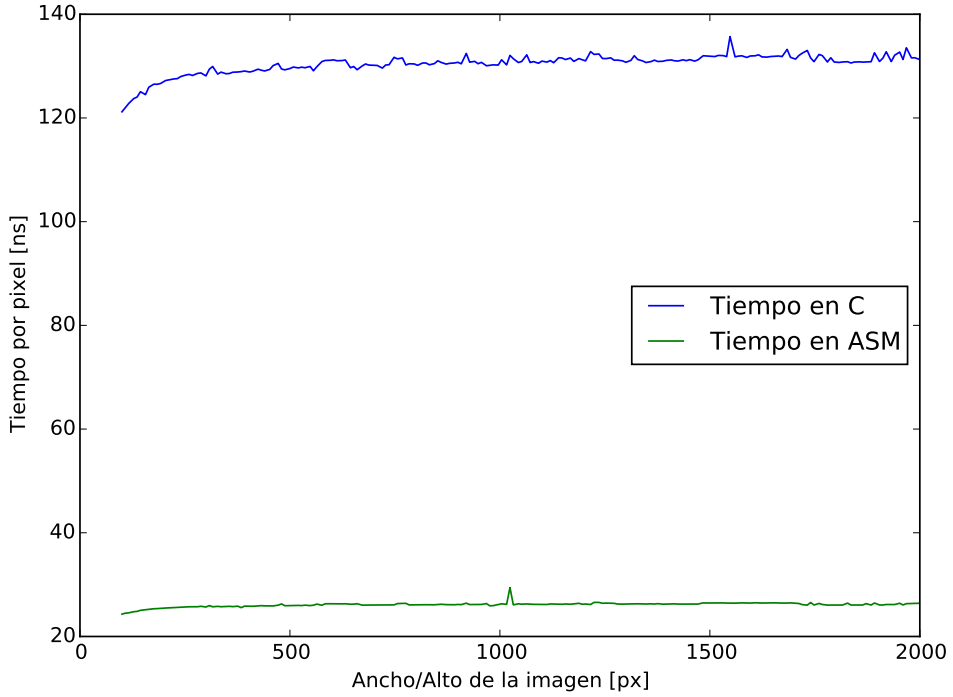

Figura 3: Tiempo por píxel para imágenes cuadradas. Observamos un comportamiento creciente, que se estabiliza a partir de ≈ 300 px.

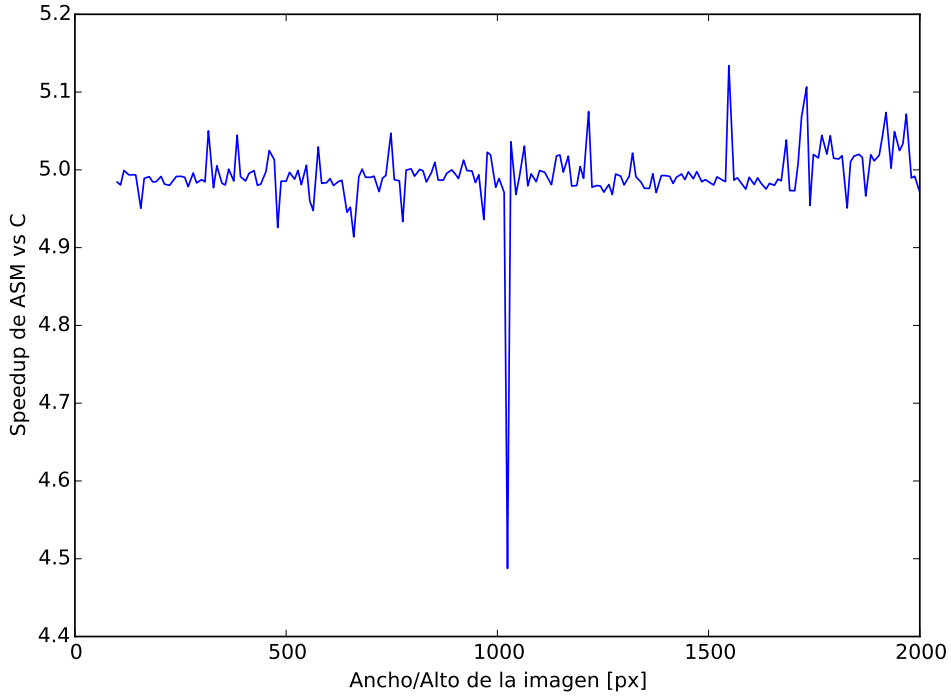

Figura 4: *Speedup* para imágenes cuadradas. Observamos un comportamiento uniforme de 5x.

de la imagen.

3.3. Comparaciones entre C y ASM: Dependencia en aspect ratio

En este caso, mantuvimos el tamaño de la imagen constante y variamos el aspect ratio. No se observó ninguna dependencia en ninguno de los casos, como se puede ver en las figuras 5, 6, 7, 8.

3.4. Comparaciones entre C y ASM: Dependencia con el radio de blur

Finalmente, observamos en la figura **??** el tiempo de procesado por píxel para distintos radios.

3.5. Comparaciones entre C -O0 y C -O3: Filtro de diferencias

3.6. Algunas conclusiones

A partir de los resultados y tablas vistas podemos mencionar las siguientes conclusiones generales que aplican a todos los filtros:

- Las implementaciones hechas en *Assembler* son efectivamente más rápidas que las implementaciones hechas en *C*. Esto era lo esperado pueés las implementaciones en *Assembler* hacen uso de las instrucciones SSE y se procesan mas de un pixel simultaneamente.
- En los algoritmos en los que no hubo conversión a Float de los datos, la velocidad de resolución fue mayor porque al no ser necesario usar floats, cada pixel ocupaba menos bytes y por ende podíamos procesar mayor cantidad simultaneamente.

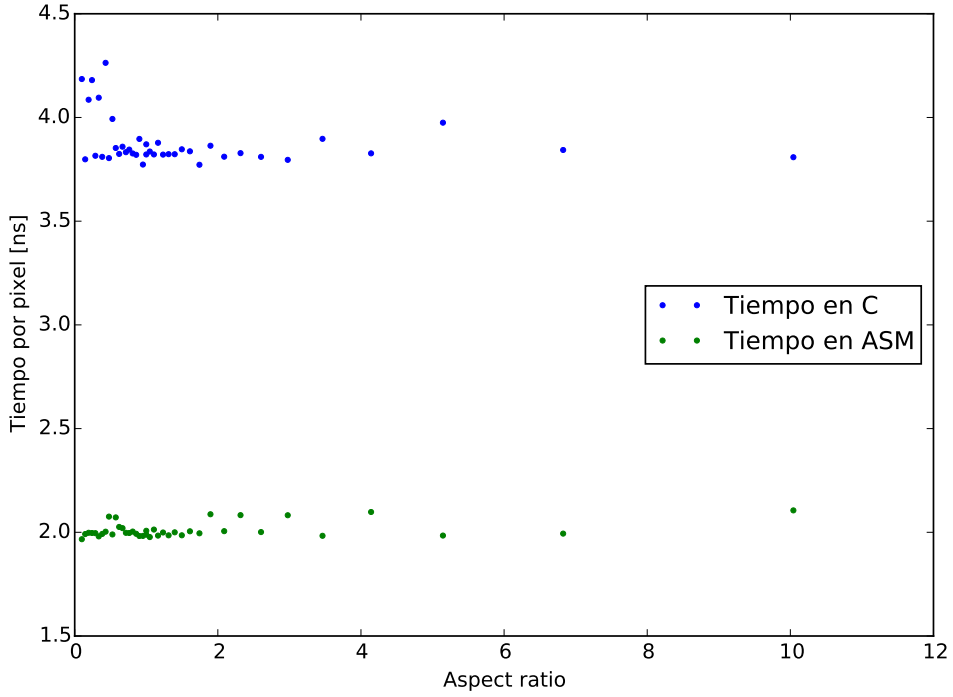

Figura 5: Tiempo de procesado por píxel para distintos aspect ratio con filtro diff.

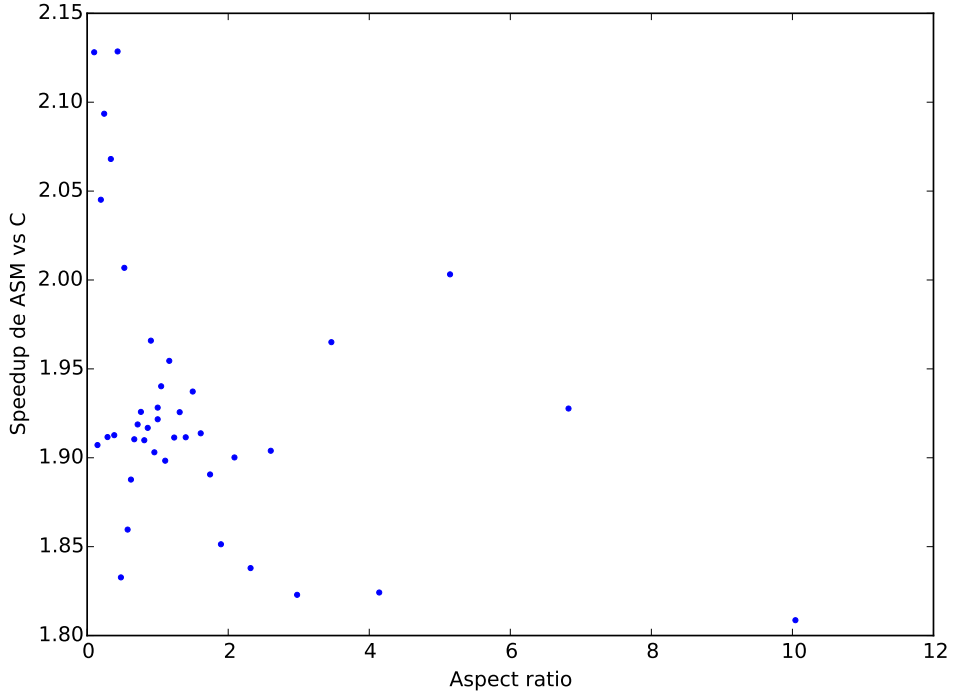

Figura 6: *Speedup* de procesado por píxel para distintos aspect ratio con filtro diff.

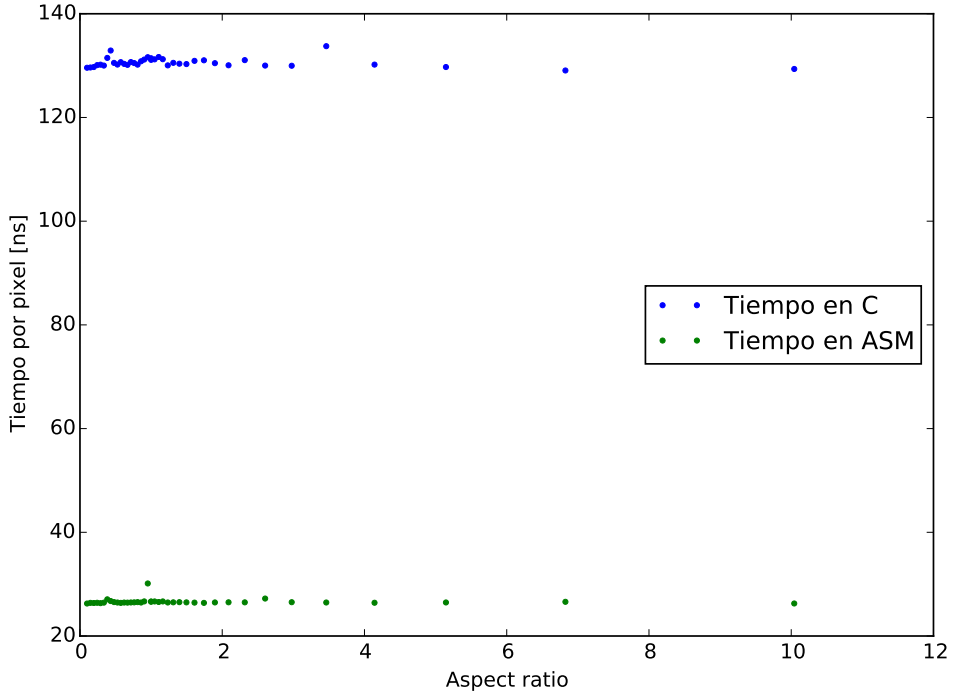

Figura 7: Tiempo de procesado por píxel para distintos aspect ratio con filtro blur.

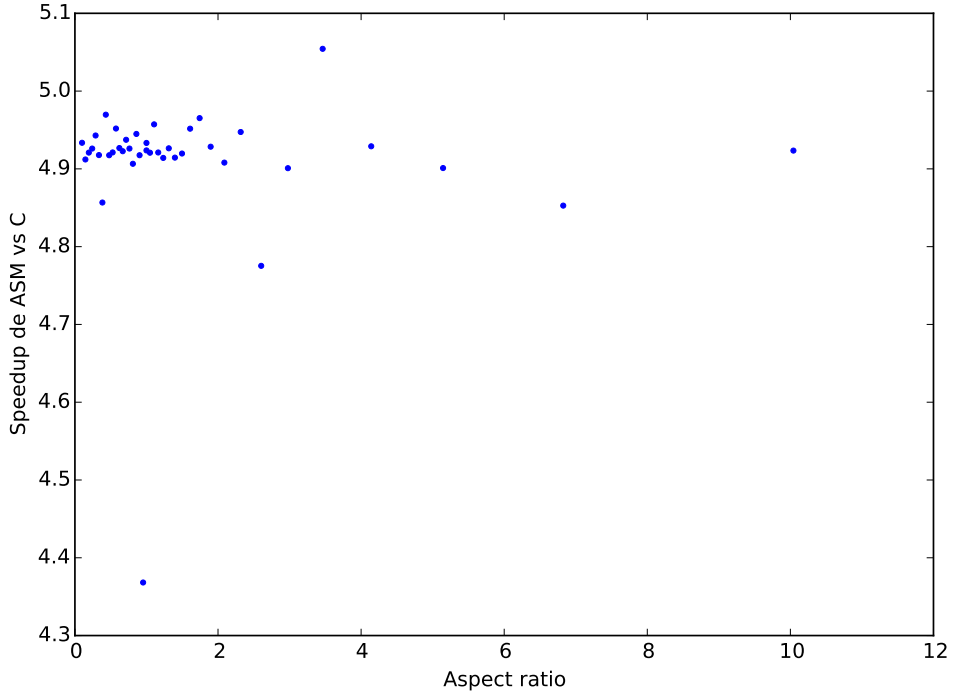

Figura 8: *Speedup* de procesado por píxel para distintos aspect ratio con filtro blur.

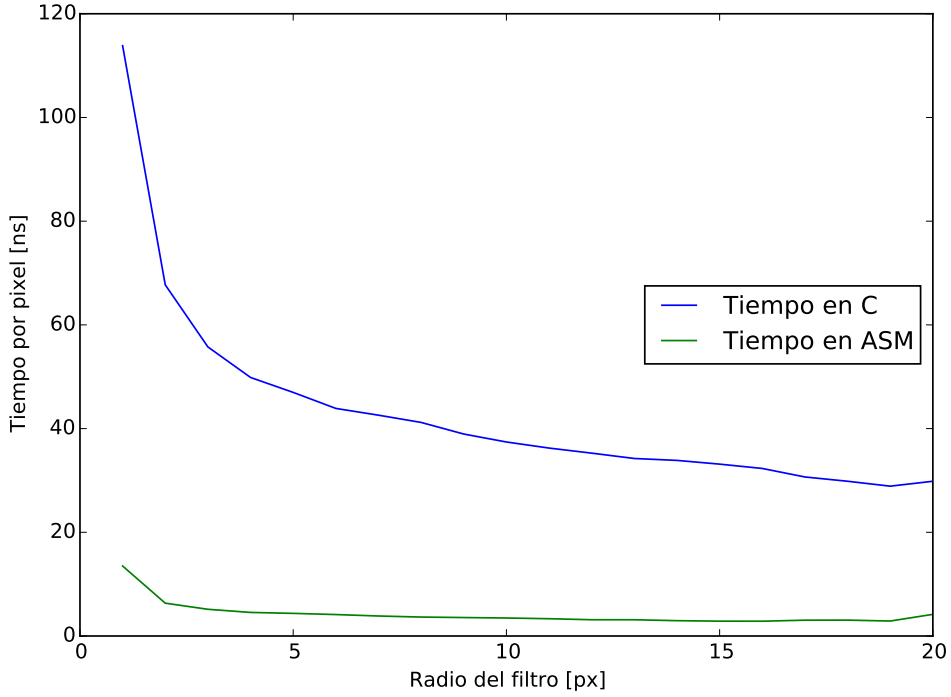

Figura 9: Tiempo de procesado por píxel para distintos radios con filtro blur.

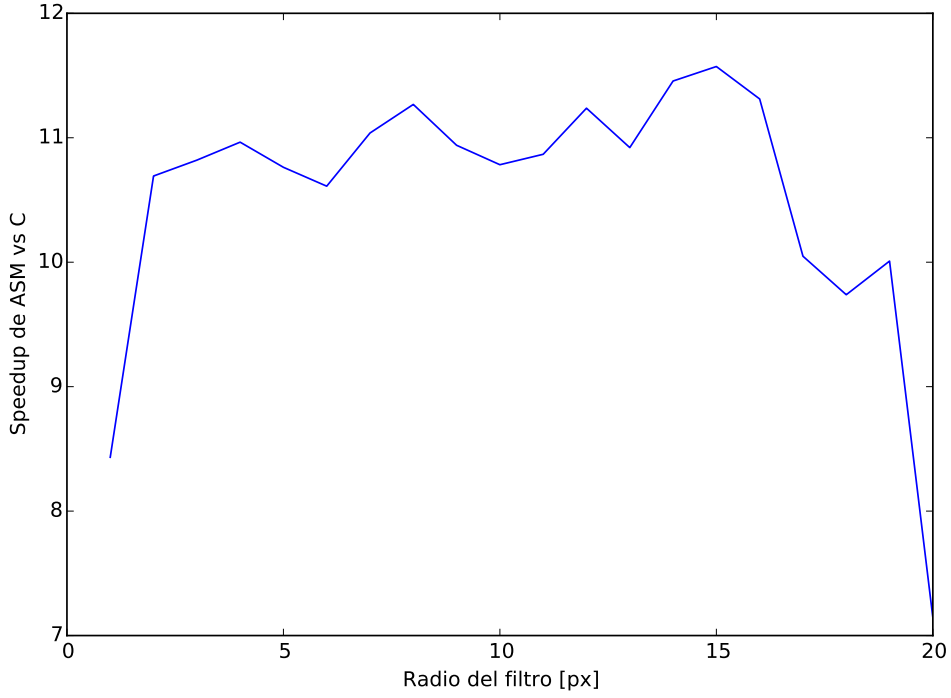

Figura 10: *Speedup* de procesado por píxel para distintos radios con filtro blur.

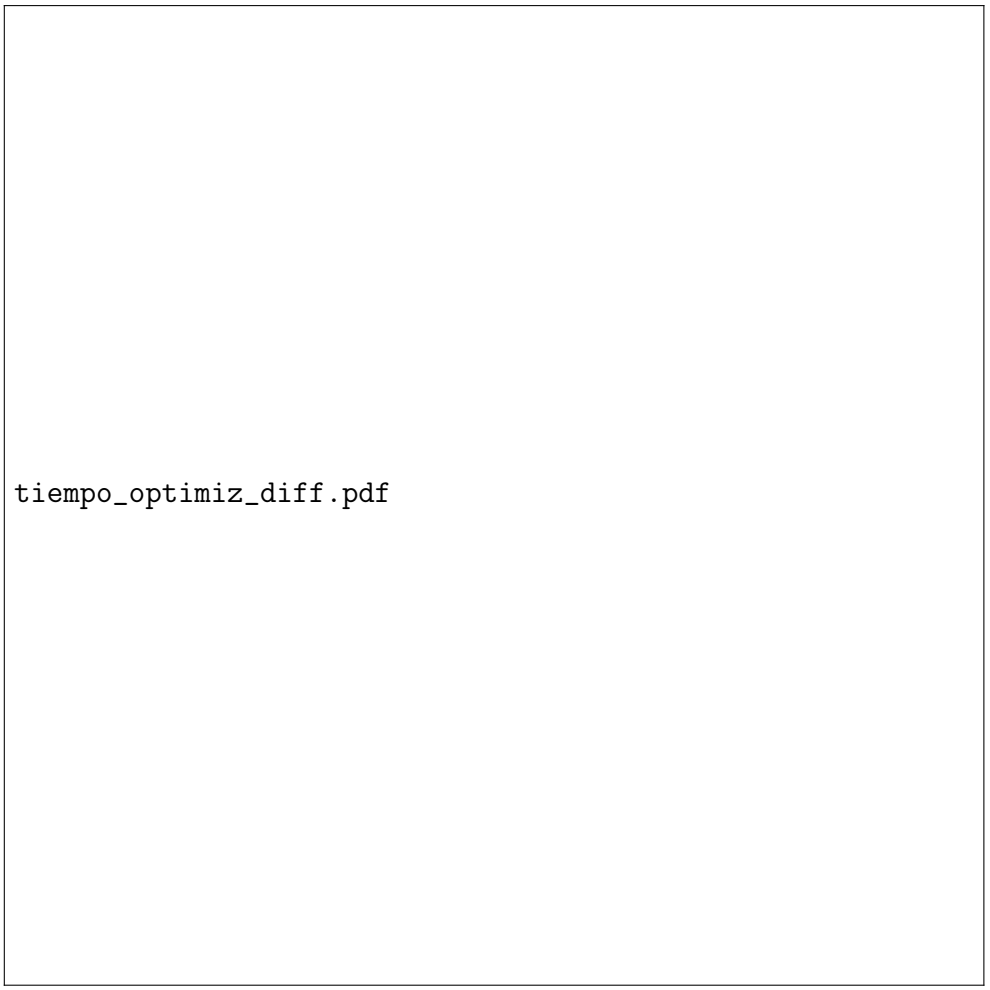

Figura 11: Tiempo de procesado por píxel para distintos tamaños.

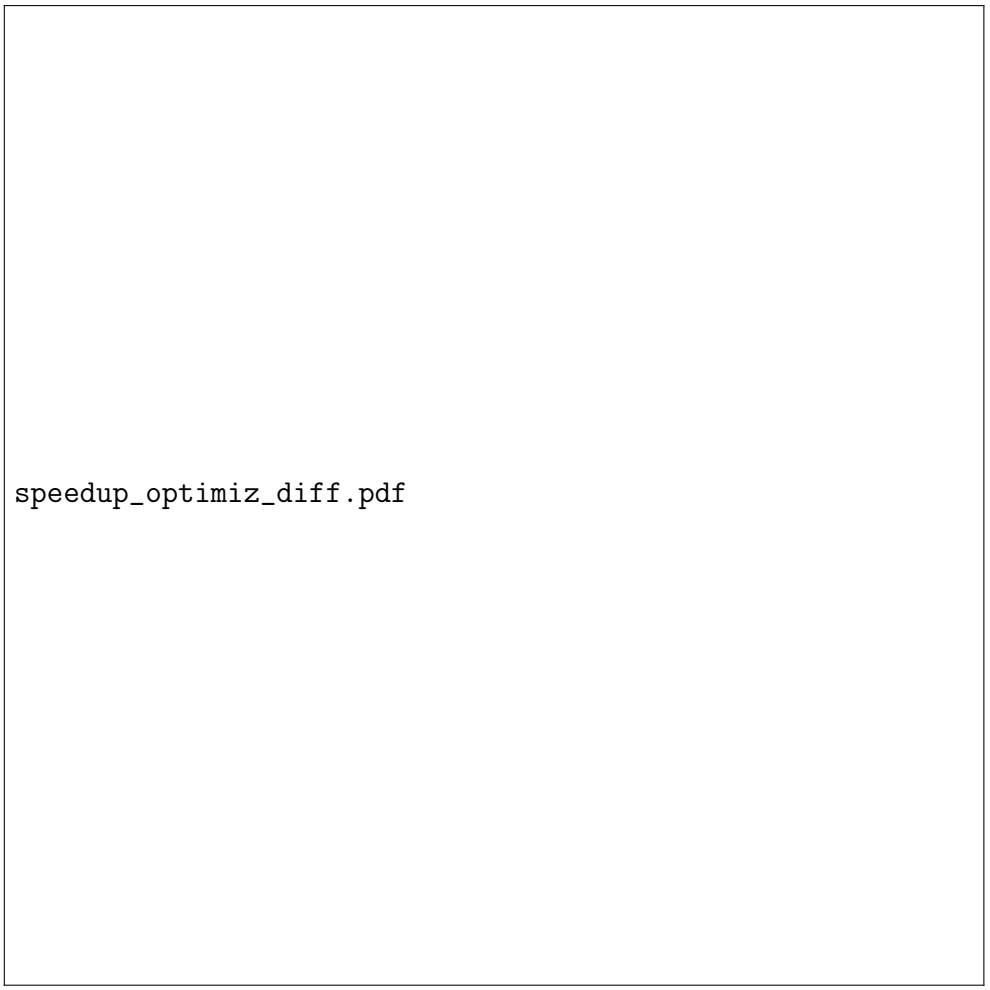

Figura 12: *Speedup* de procesado por píxel para distintos tamaños.

- El SpeedUp de las funciones no cambió demasiado entre las funciones. Y todas estas operaciones se pueden obtener directamente con instrucciones SIMD, mientras que en C hubo que programarlas, realentizando el algoritmo.
- En los gráficos de SpeedUp se pueden ver picos, que si bien no son muy marcados, llaman la atención. Pensamos que esto puede suceder porque medimos los ciclos que tarda el procesador en correr las funciones, y estos se pueden ver afectados por otro uso en simultáneo que se le esté dando al mismo.
- En todos los gráficos de comparacion entre C y ASM las curvas presentan un crecimiento semejante al de una funcion cuadrática. Esto se debe a que la escala tomada en el eje de abscisas hace referencia al tamaño de un lado de las imagenes (todas las imagenes son cuadradas). Como la cantidad total de pixeles en la imagen es el cuadrado de este valor, la cantidad de ciclos de cpu aumenta linealmente respecto a la cantidad de pixeles.

4. Conclusión

Básicamente, este trabajo práctico nos ha hecho pensar como implementar ciertas funciones de una manera distinta a la que estamos acostumbrados. Además, nos ha hecho conocer en profundidad el modelo *SIMD*.

La programación vectorial es muy útil para optimizar funciones. Creemos que nuestras implementaciones muestran claramente la ganancia en performance que hay al programar así. Sin embargo, hay un costo. Los algoritmos son más complejos debido a que el número de instrucciones aumenta significativamente, se debe pensar cuidadosamente como se cargan los datos de memoria y como se guardan para evitar errores de *Violación de segmento*. Muchas veces los datos deben ser transformados y reacomodados para poder usarlos con las operaciones *SIMD*. También hay que analizar bien cuando finalizar los ciclos y si se debe tratar aparte y como se deben tratar los últimos datos, esos que no pudieron ser tratados en el ciclo debido a que no eran suficientes como para poder cargarlos en un registro `xmmx`. La otra desventaja que tiene este modo de operación es que nos ata a una arquitectura específica haciendo nuestro programa no portable. Nos ha pasado tener que repensar ciertos partes del código por no contar, por ejemplo, con la extensión `SSE3`.

Para cerrar, nos resultó interesante ver la aplicación de la programación vectorial en un tema concreto como el tratamiento de imágenes.