Organización del Computador II
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

SIMD: Optimización de un detector Canny

Juan Martin Leiva (LU 329/12)
2017

Contenidos

Abstract

Un detector de bordes es un operador o filtro encargado de encontrar las regiones de una imagen digital en las cuales ésta cambia su brillo drásticamente (o sea, sus bordes). Dado que tiene aplicaciones de tiempo real, es necesario que los algoritmos que lo implementan sean eficientes. Canny Edge Detector es un algoritmo multietapa que implementa tal filtro.

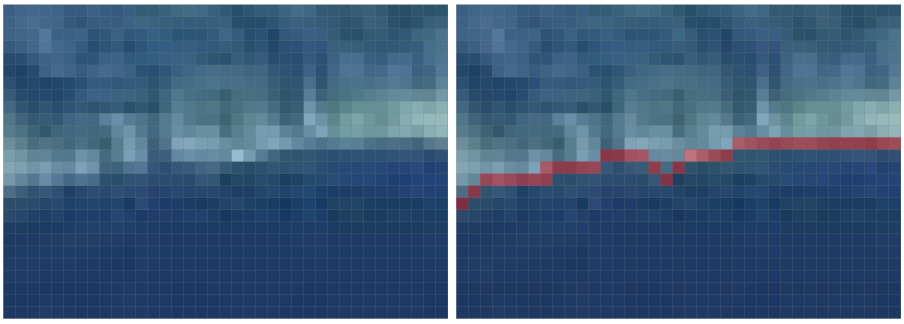
El objetivo de este trabajo es desarrollar una implementación optimizada para aprovechar el poder que ofrecen las extensiones SIMD de las arquitecturas x86/x64 de Intel, y comparar su desempeño contra una implementación en C. Dada la ventaja en cuanto a eficiencia respecta de la tecnología SIMD y de las características algorítmicas del problema propuesto, se espera observar una mejora en el rendimiento de una implementación implementada usando dicha tecnología en relación a su contraparte en C.

1 Introducción

1.1 Bordes en una imagen digital

Antes de poder comenzar siquiera a pensar en una forma de detectar bordes en una imagen mediante un algoritmo, es necesario entender el concepto de borde en una imagen digital.

Podemos decir que un borde (dentro de una imagen digital) es un conjunto de píxeles que forman una recta o curva que separa regiones de dicha imagen con diferentes valores. Además debe darse la propiedad de que ese cambio de valores sea de forma drástica en esos píxeles.



1.2 Motivación

Gran parte de los bordes de una imagen, son producto de objetos y formas dentro de la misma. Casi cualquier tarea que requiera el análisis del contenido de una imagen (Face recognition, OCR, etc) requerirá poder separar los bordes del resto de la imagen.

1.3 Análisis por color vs intensidad lumínica

Existen diferentes conceptos a la hora en analizar una imagen para encontrar bordes. Y algo a considerar es si se quieren analizar cambios en intensidad luminosa (luminancia), cambios en color, o una combinación entre ambos.

En la mayoría de las aplicaciones de un detector de bordes, la información de luminancia de una imagen es suficiente para obtener los resultados deseados. El análisis por color brinda más información y su implementación es más compleja (en el caso más ingenuo, es un análisis similar al de luminancia en cada canal de color)

Dado que en el presente trabajo lo que se está investigando es como se puede mejorar el desempeño de los algoritmos y no los diferentes resultados que pueden dar los diferentes métodos de detección, los algoritmos desarrollados sólo procesarán la luminancia de las imágenes de entrada.

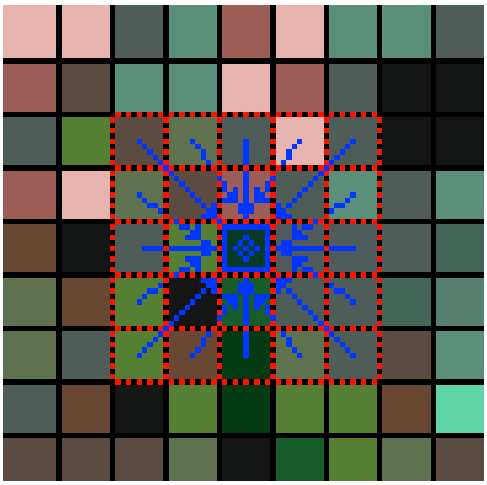
2 Convolución

2.1 Matriz de convolución

Para poder comprender el funcionamiento de algunas etapas, es necesario comprender antes la definición de convolución. En matemáticas y, en particular, análisis funcional, una convolución es un operador matemático que transforma dos funciones f y g en una tercera función.

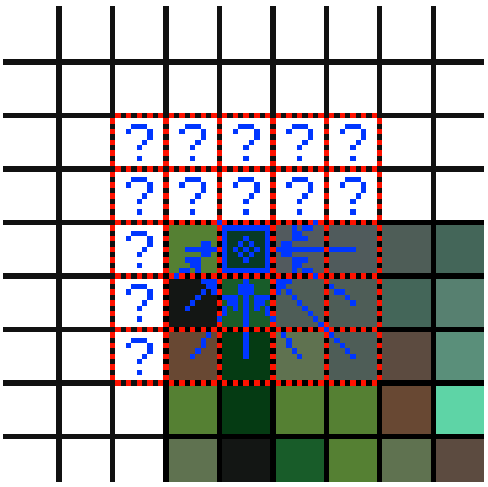
Dentro del dominio digital, y el procesamiento de imágenes, se utilizan convoluciones matriciales. Para esto se parte de dos elementos, la imagen a procesar (que puede considerarse una matriz donde cada píxel representa un elemento en la matriz), y una matriz de convolución o ***kernel***, y se procede, por cada elemento de la primer matriz (imagen) se suman sus vecinos ponderados por el kernel.

$$\left\{ \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{array} \right\} * \left\{ \begin{array}{ccc} \text{a} & \text{b} & \text{c} \\ \text{d} & \text{e} & \text{f} \\ \text{g} & \text{h} & \text{i} \end{array} \right\} = 1a + 2b + 3c + 4d + 5e + 6f + 7g + 8h + 9i$$



2.2 Convolución: bordes

El proceso de convolución de matrices requiere un tratamiento especial en los **bordes**, o mejor dicho, en los píxeles los suficientemente cercanos al comienzo o fin (vertical y/o horizontalmente) de la imagen. Si no se tuviesen en cuenta, al realizar la convolución, algunos elementos a "ponderar" con el kernel quedarían fuera de la imagen.



Existen diferentes formas de solucionar este problema.

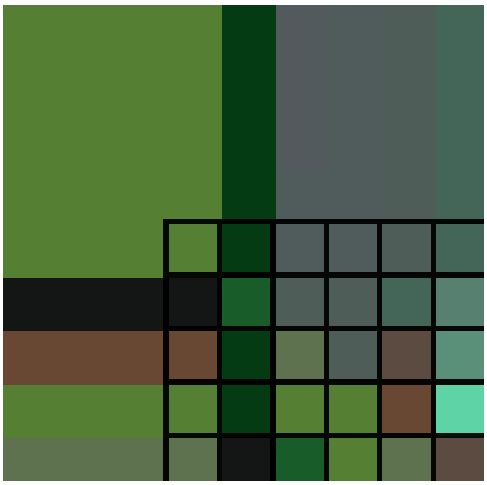
1. Extensión: los píxeles del borde son virtualmente extendidos infinitamente, de esta forma las regiones que se necesitan calcular fuera de la imagen toman el valor de píxel mas cercano.

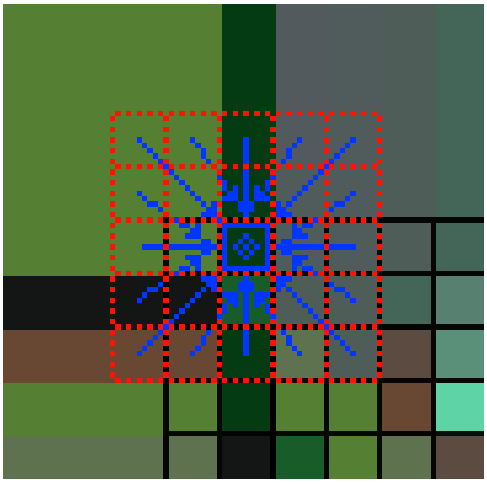
2. Envolvente: la imagen se considera "tiled" o como azulejo (se repite la imagen infinitamente), ie. si la imagen mide 100 píxeles de ancho, la posición (-2, 0) coincide con la (98, 0).

3. Espejado: las posiciones "rebotan". En el ejemplo anterior, (-2, 0) seria igual a (2, 0).

4. Recorte: las posiciones fuera de la imagen no se calculan (es decir el filtro no se aplica a los bordes de la imagen.

En este trabajo se utilizara el método de extensión:





3 Detector Canny

3.1 Origen

El detector o algoritmo de Canny es un operador que usa un algoritmo multi-etapa para detectar bordes en una imagen, desarrollado por John F. Canny en 1986. También fue el fundador de la teoría computacional de la detección de bordes para explicar por que su algoritmo funciona.

Según esta teoría, los puntos más importantes que debe tener un detector de bordes para ser correcto son:

1. Detección con bajos niveles de errores. Es decir, el algoritmo debe detectar con precisión tantos bordes en la imagen como sea posible.
2. Los puntos de borde detectados deben estar localizado en el centro del borde.
3. Cada borde debe ser detectado una y solo una vez, y de ser posible, el ruido no debe crear falsos bordes.

3.2 Proceso

El algoritmo de Canny posee 5 etapas, donde la primera toma la imagen original, y se van pasando los resultados de cada etapa como entrada de la siguiente, hasta la última que genera el resultado final.

1. Noise reduction (Reducción de ruido)
2. Intensity Gradient (Gradiente de Intensidad)
3. Non-Maximum Suppression (Supresión de No-Máximos)
4. Double Threshold (Doble Umbral)
5. Hysteresis Threshold (Umbral por histéresis)

Además, como se mencionó anteriormente, solo se analizará la luminancia de las imágenes de entrada, por lo cual se agrega una etapa extra al inicio de conversión a escala de grises.

3.3 Escala de grises

Esta etapa se encarga de transformar la imagen de entrada que es leída como una imagen en colores, a escala de grises para conservar solamente su luminancia. Esto hace también que cambie la cantidad de BPP (bits per pixel) de 3 o 4 (según como sea leída, se hablara de esto más adelante), a 1.

La forma elegida para este proceso es una función ingenua que para cada píxel devuelve el promedio de la suma de sus componentes de color.

$$grayscale(r, g, b) = \frac{r + g + b}{3}$$

Donde r, g y b son cada componente de color respectivamente.

3.4 Reducción de ruido

El ruido en una imagen puede ocasionar la detección de falsos bordes, por eso, es necesario aplicar un filtro para reducir el ruido como primera etapa. En este trabajo se utilizó un filtro de desenfoque (Blur) Gaussiano. Esté funciona aplicando una matriz de convolución gaussiana a la imagen.

La matriz de convolución es cuadrada, de tamaño variable, y se genera mediante la siguiente fórmula:

$$gauss(x, y, \sigma) = \frac{1}{2}\pi\sigma^2 e^{-\frac{x^2+y^2}{2\sigma^2}}$$

Donde x e y son la distancia horizontal y vertical respectivamente al centro de la matriz y σ es la desviación estándar de la distribución gaussiana.

Este proceso asegura no modificar el brillo total de la imagen, ya que el ***kernel***, al generarse a partir de una distribución probabilística (distribución de gauss), hace que todas sus componentes sumen 1.

Un ejemplo de un kernel gaussiano de 7 x 7 y un σ de 1.4 :

$$\left\{\begin{array}{ccccccc} 0.000977 & 0.003320 & 0.006914 & 0.008829 & 0.006914 & 0.003320 & 0.000977 \\ 0.003320 & 0.011286 & 0.023505 & 0.030014 & 0.023505 & 0.011286 & 0.003320 \\ 0.006914 & 0.023505 & 0.048952 & 0.062509 & 0.048952 & 0.023505 & 0.006914 \\ 0.008829 & 0.030014 & 0.062509 & 0.079820 & 0.062509 & 0.030014 & 0.008829 \\ 0.006914 & 0.023505 & 0.048952 & 0.062509 & 0.048952 & 0.023505 & 0.006914 \\ 0.003320 & 0.011286 & 0.023505 & 0.030014 & 0.023505 & 0.011286 & 0.00332 \\ 0.000977 & 0.003320 & 0.006914 & 0.008829 & 0.006914 & 0.003320 & 0.000977 \end{array}\right\}$$

3.5 Gradiente de Intensidad

Como primer paso para poder encontrar los bordes en una imagen, se debe realizar una comparación lineal entre la diferencia de intensidad entre un píxel y sus vecinos. Dado que el borde puede tener diferentes ángulos, es necesario hacer más de una "pasada" en diferentes direcciones. En este trabajo esto se resuelve usando el Operador Sobel, que integra dos "pasadas", una vertical

y otra horizontal. Este operador tiene dos pasos; en el primero, se aplican dos matrices de convolución, una para detectar los gradientes horizontalmente, y la otra verticalmente. Este paso nos da como resultado dos matrices que son tomadas como entrada para el segundo paso, donde se generan otros dos resultados, el gradiente de intensidad lumínica, y una matriz que indica los ángulos de dichos gradientes.

Las matrices para el primer paso son:

$$M_x = \begin{Bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{Bmatrix}$$

$$M_y = \begin{Bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{Bmatrix}$$

Esto da como resultado dos matrices que llamaremos G_x y G_y respectivamente.

Para el segudo paso, calcularemos la intensidad del gradiente (S) y sus angulos (θ).

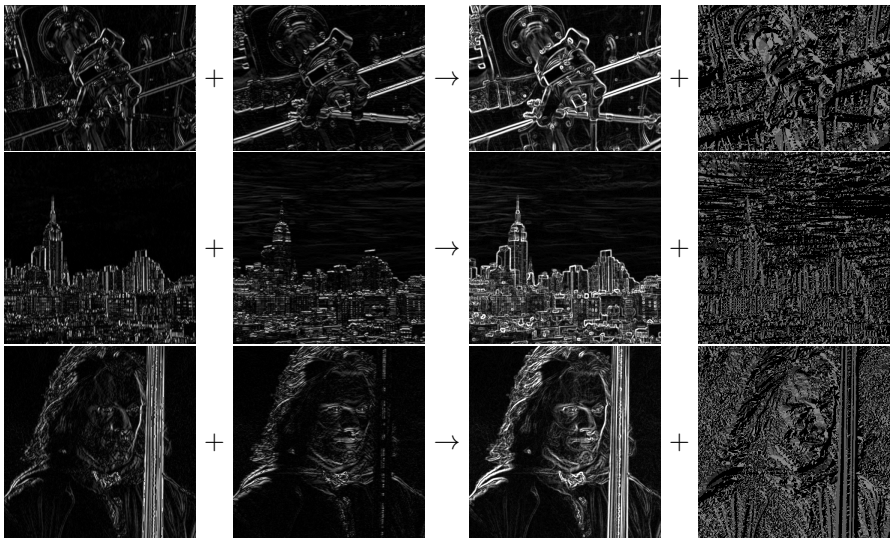
$$S(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2}$$

$$\theta(x,y) = atan\left(\frac{G_x(x,y)}{G_y(x,y)}\right)$$

Cabe aclarar que el resultado de θ es post procesado para su normalización, es decir, una transformación al valor múltiplo de $\frac{\pi}{4}$ mas cercano.

$$\theta_n(x,y) = \begin{cases} 0 & 0 \leq \theta(x,y) \leq \frac{\pi}{8} \vee -\frac{\pi}{8} \leq \theta(x,y) \leq 0 \\ 45 & \frac{\pi}{8} \leq \theta(x,y) \leq \frac{3\pi}{8} \vee -\frac{7\pi}{8} \leq \theta(x,y) \leq -\frac{5\pi}{8} \\ 90 & \frac{3\pi}{8} \leq \theta(x,y) \leq \frac{5\pi}{8} \vee -\frac{5\pi}{8} \leq \theta(x,y) \leq -\frac{3\pi}{8} \\ 135 & \frac{5\pi}{8} \leq \theta(x,y) \leq \frac{7\pi}{8} \vee -\frac{3\pi}{8} \leq \theta(x,y) \leq -\frac{\pi}{8} \end{cases}$$

Estos valores (0, 45, 90, 135) carecen de valor en si mismos, siempre que se puedan distinguir unos de otros. Se eligieron esos valores pues se puede representar perfectamente con un ***byte***, es sencillo para debuggear, y si se guarda el resultante como una imagen, se fácilmente reconocible su patrón.

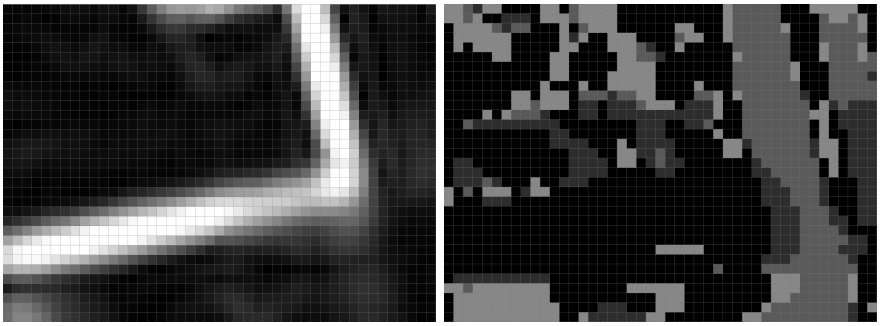


3.6 Supresión de no-máximos

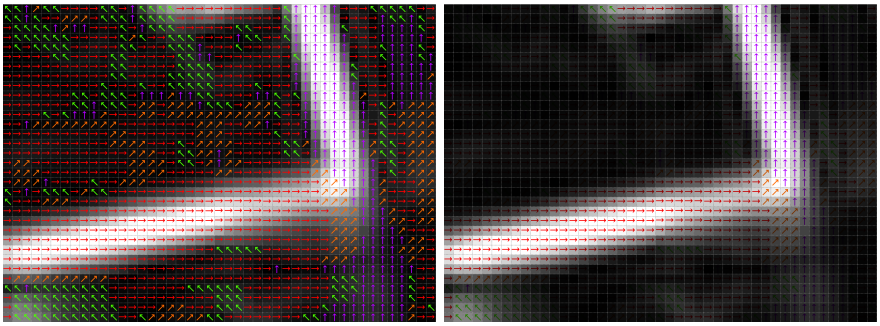
Si bien el resultado de intensidad de sobel nos da una buena aproximación de los bordes de la imagen, no son precisos en absoluto. El primer paso para afinar dichos bordes, es tomar la región de más intensidad de cada borde, y descartar el resto. Para esto, conceptualmente, lo que debemos hacer es recorrer cada borde longitudinalmente y quedarnos solo con el píxel mas brillante a lo largo de su sección. Esta etapa hace esto, recorriendo cada píxel, y se verifican sus vecinos según su orientación (normalizada), buscando al mayor. Solamente se

compara un vecino a cada lado transversalmente a su ángulo, de esta forma se asegura que cada borde sera filtrado para dejar solamente los puntos de mayor intensidad.

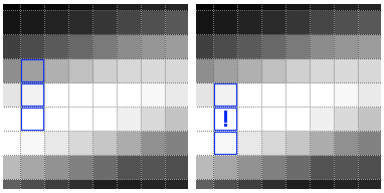
Consideremos la siguiente imagen (escalada), que obtenemos como resultado de la etapa anterior:



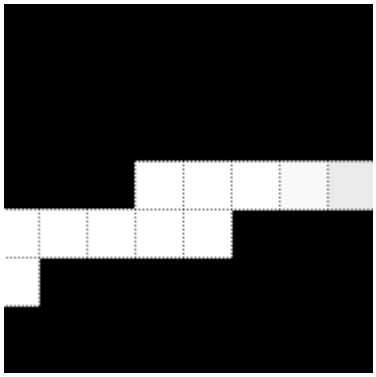
Ambas imágenes significan lo siguiente:



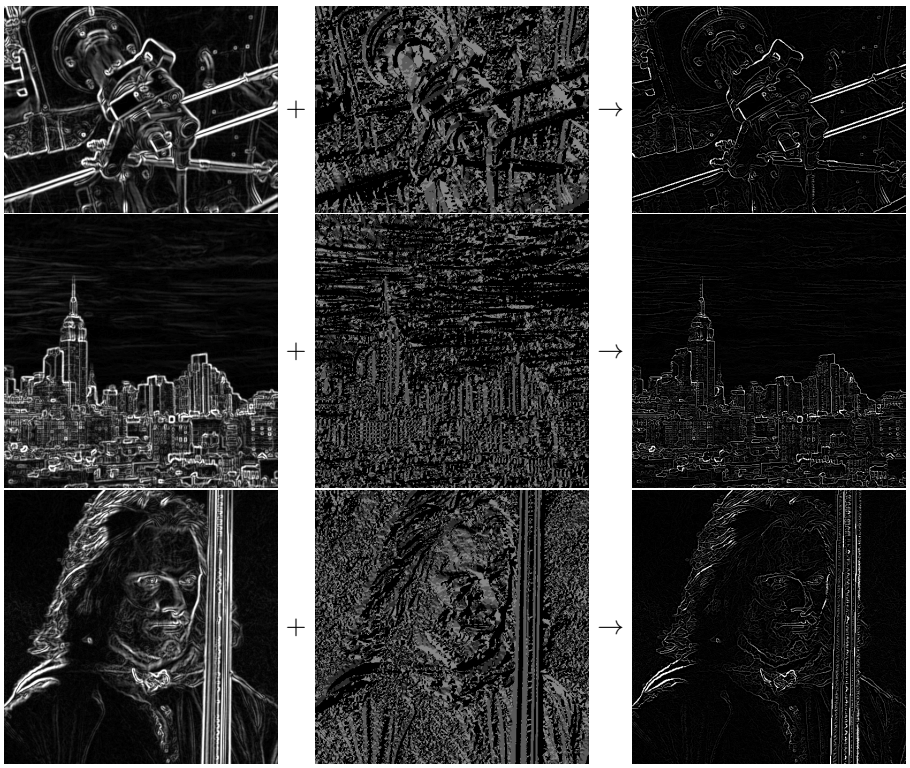
Es decir además de detectar las zonas donde la intensidad varió mas, encontró en qué ángulo lo hizo. Con esto ya tenemos lo necesario para aplicar la Supresión de No-Máximos.



Aplicado a cada punto nos queda lo siguiente:



Con este paso ya tenemos todos los bordes candidatos filtrados de forma precisa.



3.7 Doble umbral

Esta etapa es bastante sencilla, y se encarga separar los candidatos en tres categorías:

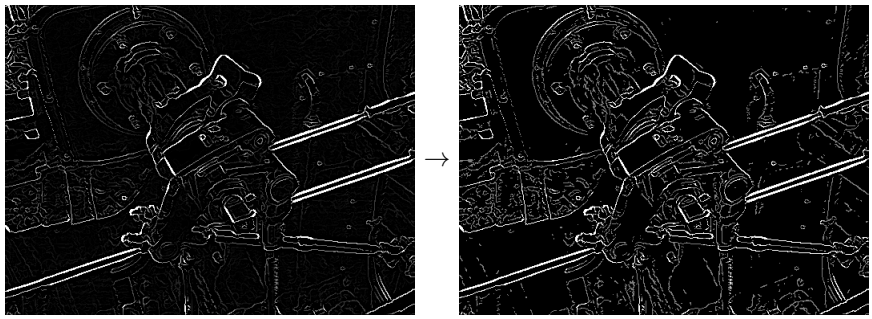
- Ruido: aquellos demasiado tenues como para ser considerados bordes reales.
- Bordes débiles: aquellos con un nivel de brillo bajo o moderados, podrían o no ser bordes reales.
- Bordes fuertes: aquellos con brillo fuerte. Estos ya pertenecen al grupo de los bordes reales.

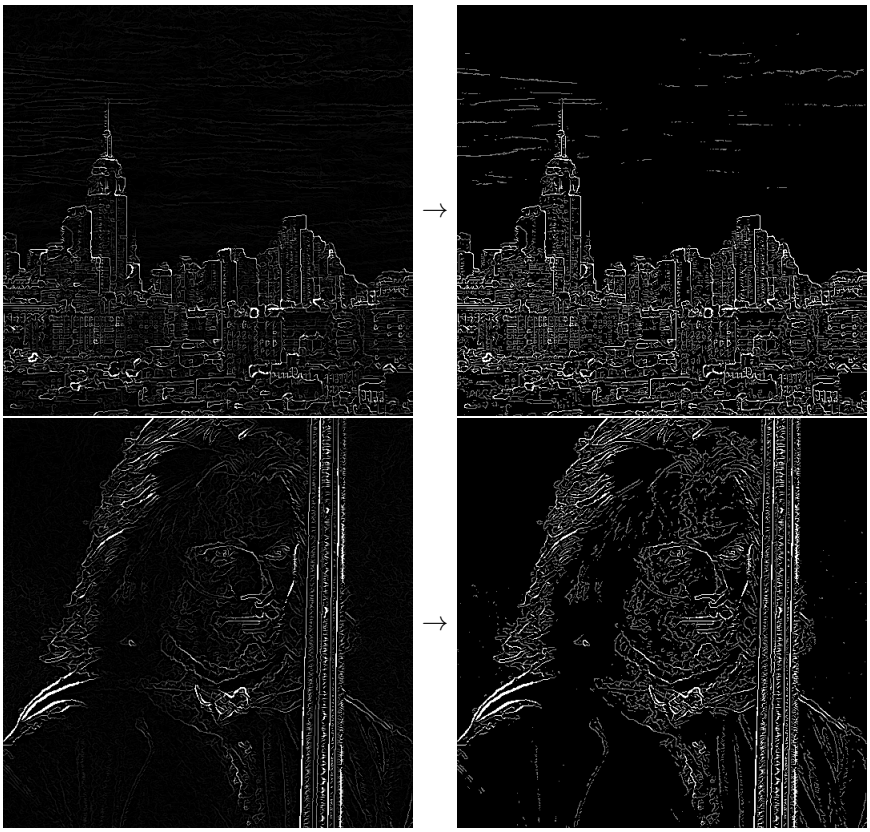
Por cada píxel p el algoritmo hace lo siguiente:

$$nms(p, lT, hT, lV, hV) = \begin{cases} 0 & 0 \leq p \leq lT \\ lV & lT \leq p \leq hT \\ hV & hT \leq p \leq 255 \end{cases}$$

Donde:

- p es el valor del píxel actual con $0 \leq p \leq 256$
- lT es el umbral inferior
- hT es el umbral superior
- lV es el valor bajo
- hV es el valor alto

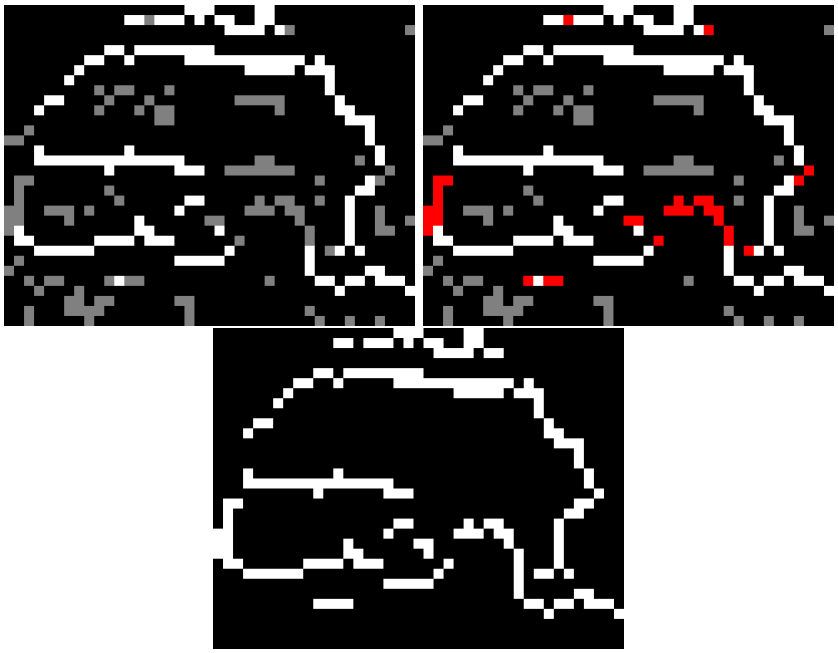




3.8 Umbral por histéresis

Como dijimos el punto anterior, los bordes débiles son potenciales candidatos a bordes reales. El significado de esto es que según su entorno serán o no bordes reales.

El criterio utilizado para decidir si son o no bordes reales es si están en contacto con un borde fuerte. Es decir, todos los puntos que estén conectados (directa o indirectamente) con un punto que sea borde fuerte, serán considerados bordes fuertes.

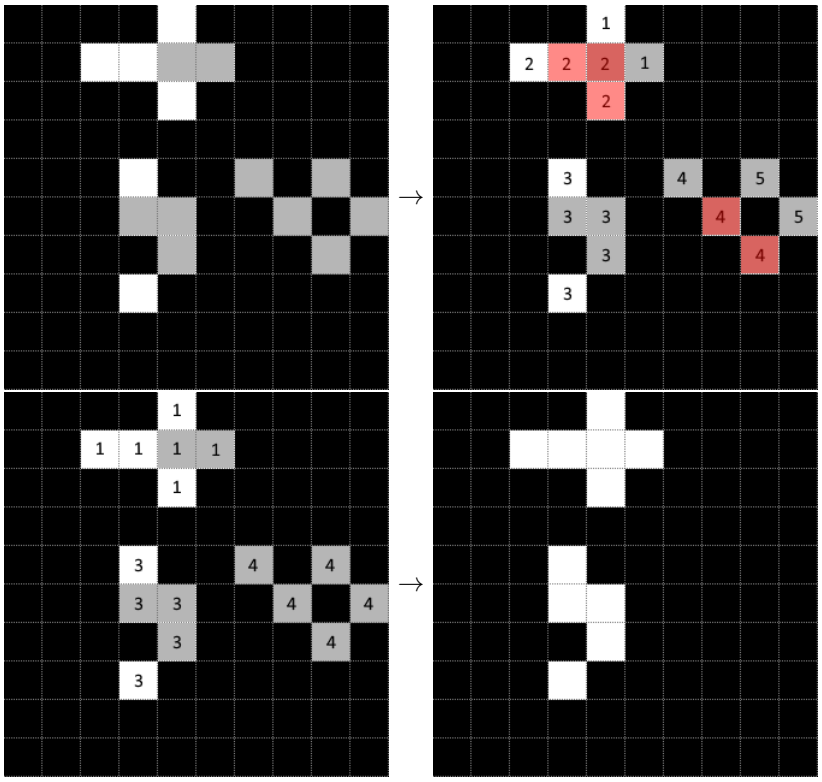


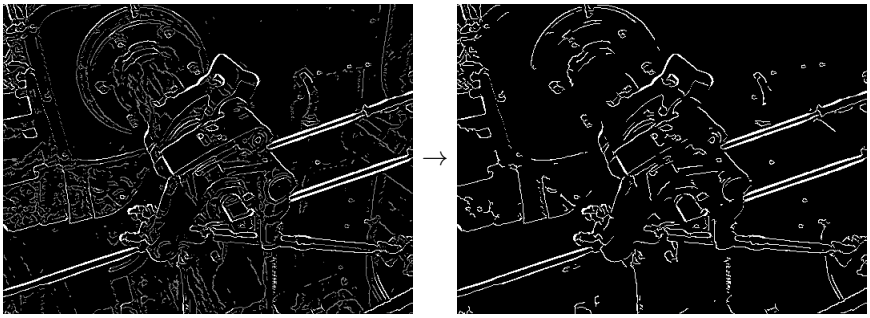
En este trabajo se usó un algoritmo de "Connected Component Labelling" por barrido horizontal. Este algoritmo funciona recorriendo de izquierda a derecha y de arriba a abajo píxel por píxel, y creando grupo usando la siguiente regla:

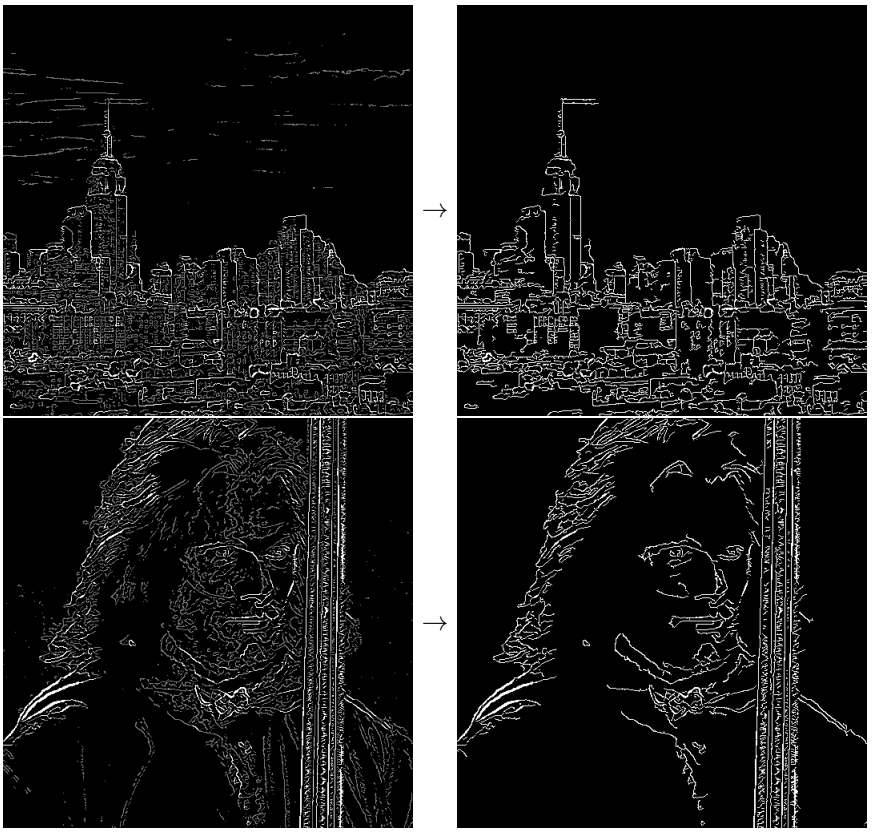
- Si el píxel tiene intensidad 0 (negro), no pertenece a ningún grupo pues no es un componente a etiquetar.
- Si tiene un valor distinto de 0: Si el píxel superior, izquierdo, superior izquierdo o superior derecho tienen intensidad cero, este píxel es marcado como perteneciente a nuevo grupo.
- Si tiene un valor distinto de 0: Si el píxel superior, izquierdo, superior izquierdo o superior derecho tienen intensidad mayor a cero, entonces aquellos con intensidad mayor a cero ya fueron recorridos y catalogados con un grupo.
  - Si todos estos "vecinos" con intensidad mayor a cero son del mismo grupo N, este píxel en catalogado como grupo N.
  - Si existen conflictos, se lo cataloga como grupo de alguno de los vecinos, y los demás grupos son marcados como "en conflicto".

Una vez terminado este paso, se procede a renombrar a todos los grupos en conflicto con el mismo identificador de grupo, con el fin de normalizarlos.

Por último, se obtiene el píxel de mayor intensidad de cada grupo, y se escribe esa intensidad a todos los píxeles del mismo grupo.







4 Notas sobre la implementación

4.1 Compilación

Este trabajo fue desarrollado utilizado los lenguajes C y Ensamblador (Intel x86/x64) bajo el sistema operativo Linux (distribución Ubuntu 16.04.2 LTS) y utilizado las siguientes aplicaciones y bibliotecas

- **GCC**
- **NASM**
- **Make**
- **C standard library**
- **Lodepng**

4.2 Fuentes

El código fuente del trabajo esta disponible en el siguiente repositorio public GIT: `https://github.com/JMLeiva/CannyEdge`

4.3 Lodepng

Dado que el objetivo del trabajo se limita a la implementación y análisis del filtro CannyEdge, las operaciones de decoding / encoding multimedia son llevadas a cabo por una biblioteca opensource y de libre uso llamada Lodepng, creada por Lode Vandevenne.

Para mas información referirse a la página oficial: `http://lodev.org/lodepng/`

4.4 Profiling

Todas las mediciones en este trabajo utilizan el **Time Stamp Counter**. Éste cuenta la cantidad de ciclos del procesador desde el ultimo reset, y está expuesto a través de la instrucción *rdtsc*.

A la hora de realizar las mediciones, es necesario tener en cuenta que los sistema operativos realizan cambios de tareas todo el tiempo para poder ofrecerle al usuario un entorno "multitarea". Esto produce que en cualquier momento de la ejecución de un programa, el sistema operativo puede dejar de ejecutarlo para darle espacio a otro programa, y el **Time Stamp Counter** es independiente de todo este proceso, haciendo que la ejecución de un mismo programa arroje una cantidad diferente de ciclos utilizados para cada ejecución. Es decir, la cantidad de ciclos utilizados en un algoritmo no es determinista (por lo menos no lo es a nivel usuario) en este entorno. Para resolver este problema, es necesario recurrir a las estadísticas, y realizar una cantidad suficiente de ejecuciones para tener

una desviación estándar lo suficientemente pequeña como para poder confiar en los resultados arrojados.
Además, dado que en un pequeño porcentaje de la ejecuciones se observaron variaciones muy grandes (outliers), capaces de modificar los resultados de la medición, se optó por eliminar dichos resultados del análisis final. Esto se hace calculando primero la desviación estándar de la muestra completa, y luego eliminando todos los resultados que se encontrasen a mas de 2 desviaciones estándar de la media.
Más formalmente:

Sea una muestra M = $\{m_1, m_2, m_3, ..., m_n\}$ de tamaño n

$$\mu(M) = \frac{\sum_{i=1}^{n} m_i}{n}$$
$$\sigma(M) = \sqrt{\frac{\sum_{i=1}^{n} (m_n - \mu)^2}{n}}$$

Donde μ es la media, y σ es la desviación estándar.

4.5 Hardware

Todas las mediciones de este trabajo fueron realizadas sobre un Procesador Intel® Core™ i7-7500U 7th generación (2700MHz)

4.6 SIMD

SIMD significa Single Instruction Multiple Data, y representa el conjunto de operaciones que permite a un procesador realizar una misma operación sobre un conjunto de datos de forma paralela. Esto nos permite acelerar considerablemente ciertas tareas. Los filtros multimedia suelen necesitar aplicar una (o más) misma operación a cada píxel, y por lo tanto es un candidato a verse beneficiado por las ventajas del uso de SIMD.
Por las características algorítmicas de las etapas, sólo las siguientes fueron implementadas (parcial o totalmente) en ambos lenguajes (C y Ensamblador)

- Grayscale
- Gauss
- Sobel
- Double Threshold

4.7 Alineación

Las operación de Load y Store de memoria en conjunto con SIMD se ven beneficiadas si son realizadas de forma alineada a al tamaño de los registros utilizados, es decir que la posición de memoria de donde se lee o escribe es múltiplo de dicho tamaño. Este trabajo fue realizado utilizando tecnología xmm, que utiliza registros de 16 bytes, por lo tanto, la mayoría de las estructuras de datos y sus elementos fueron alineados a 16 bytes.

4.8 Casos borde en Ensamblador

Existen dos tipos de casos bordes

- **Algorítmicos**: casos donde, por propiedades del algoritmo, se deben tomar medidas especiales. En nuestro caso, serán los ya mencionados casos bordes de las convoluciones.
- **Propios de SIMD**: Dado que al trabajar con SIMD, se debe leer y escribir de a un numero determinado de bytes (16 en este trabajo), existen situaciones en donde no es posible leer o escribir pues se estaría violando la región de memoria donde se esta trabajando.

Dado que los casos bordes para este trabajo ocupan un pequeño porcentaje de la tarea a realizar (solos los bordes y final de imagen), y que su implementación puede llegar a ser mucho mas compleja que los casos normales, se decidió implementarlos en C, dejando Ensamblador y SIMD para los casos normales que ocupan el grueso de las operaciones.

4.9 Optimizaciones del compilador

Todas las pruebas de rendimiento fueron compiladas utilizando el nivel 3 de optimización de C (O3).

5 Detalles de implementación

5.1 Introducción

A continuación se conocerán los detalles de implementación etapa por etapa, haciendo foco en las diferencias entre C y Ensamblador, y el uso de SIMD. Básicamente, la idea fue aprovechar SIMD para poder realizar mas operaciones con menos instrucciones, pero en algunos casos la implementación no es trivial.

5.2 Modo de uso: Parámetros

La aplicación soporta las siguientes opciones de entrada:

- **-i** [**String**]: define el nombre de la imagen a procesar, dentro de la carpeta del ejecutable.
- **-t** [**int**]: cantidad de veces a procesar la imagen.
- **-impl** [**c** — **asm**]: tipo de implementación (C o Ensamblador).
- **-gr** [**int**]: radio del kernel de gauss.
- **-gs** [**float**]: *sigma* de gauss.
- **-mint** [**int**]: Umbral inferior del Doble Threshold.
- **-maxt** [**int**]: Umbral superior del Doble Threshold.
- **-bpp** [**3** — **4**]: Cantidad de Bits per Pixel con que cargar la imagen.
- **-oe** Output enabled (Genera las imagenes de resultado de cada etapa). No usar en conjunto con -be.
- **-be**: Benchmark enabled. Habilita el profiling. No usar en conjunto a -oe.

5.3 Entry Point

La función de entrada $int\ main(argc, char*argv[])$ se encuentra en el archivo CannyEdge.c y se encarga de procesar los parámetros de entrada, cargar la imagen usando Lodepng, e ir pasando dicha imagen por cada etapa tantas veces como se haya definido en los parámetros de entrada. Durante cada etapa, si fue requerido por los parámetros se realiza el profiling, y como ultimo paso (también, si fue definido en los parámetros), se escribe el resultado de cada etapa como una imagen de salida.

5.4 Estructuras

Las estructuras usadas en este trabajo están definidas en Structs.h y son las siguientes.

```
typedef struct{
        unsigned int width;
        unsigned int height;
        unsigned char bpp;
        __attribute__((__aligned__(16)))
        unsigned char* data;
} Image;

typedef struct {
        unsigned char size;
        __attribute__((__aligned__(16)))
        float* data;
} SquareMatrix;

typedef struct{
        unsigned short width;
        unsigned short height;
        unsigned int* data;
} UI_Matrix;

typedef struct{
        uint64_t grayscale_t, gauss_t, sobel_t, nonMax_t, lowHigh_t, hysteresis_t;
        bool valid;
} Benchmark;
```

- **Image**: imagen, con su alto, ancho, bpp y datos descomprimidos (array de bytes).
- **SquareMatrix**: Matriz cuadrada de floats que representa un kernel de convolución.
- **UI_Matrix**: Matriz utilizada para trackear los grupos en la etapa de histéresis.
- **Benchmark**: Estructura usada para el profiling.

5.5 Grayscale

La implementación en C (Grayscale.c) es muy sencilla y el código principal del algoritmo es el siguiente:

```
...

unsigned int grayIndex = 0;

for (unsigned int i = 0; i < src->width * src->height * src->bpp; i += src->bpp) {
        val = 0;

        for (unsigned char b = 0; b < colorChannels; b++) {
                val += src->data[i + b];
        }

        newData[grayIndex] = val / colorChannels;
        grayIndex++;
}

...
```

En Ensamblador, luego de preparar los registros para el loop principal, el código de divide en 2 branches, dependiendo si la imagen fue ingresada con 3 o 4 bpp.

Ambos códigos funcionan de forma similar, pero la diferencia reside en que para 3 BPP, se cargan de a 5 píxeles por vez, pero esta carga es desalineada. Primero se cargan 16 bytes, y son copiados a xmm1, xmm2, xmm3, xmm4 y xmm5

```
movdqu  xmm1,   [r10]
movdqa  xmm2,   xmm1
movdqa  xmm3,   xmm1
movdqa  xmm4,   xmm1
movdqa  xmm5,   xmm1
```

Luego, se toma un solo píxel (3 bytes) en cada registro xmm extendido a dword mediante *pshufb*

```
pshufb  xmm1,   [shuffle_packed_bbp3_byte_0]
pshufb  xmm2,   [shuffle_packed_bbp3_byte_1]
pshufb  xmm3,   [shuffle_packed_bbp3_byte_2]
pshufb  xmm4,   [shuffle_packed_bbp3_byte_3]
pshufb  xmm5,   [shuffle_packed_bbp3_byte_4]
```

Se transforma a float

```
cvtdq2ps    xmm1,   xmm1
cvtdq2ps    xmm2,   xmm2
cvtdq2ps    xmm3,   xmm3
cvtdq2ps    xmm4,   xmm4
cvtdq2ps    xmm5,   xmm5
```

Cada píxel se divide por 3 (guardado en xmm0)

```
divps   xmm1,   xmm0
divps   xmm2,   xmm0
divps   xmm3,   xmm0
divps   xmm4,   xmm0
divps   xmm5,   xmm0
```

Por ultimo se reconvierten a bytes y se escriben en memoria

```
pshufb  xmm1,   [shufle_repack_1] ;REPACK 4 pixels
pshufb  xmm5,   [shufle_repack_2] ;REPACK 1 Pixel

movd    r15d,   xmm1
mov     [r12],  r15d
add     r12,    4
movd    r15d,   xmm5
mov     [r12],  r15b
add     r12,    1
```

En la versión de 4 BPP, se cargan 4 píxeles por vez, pero la cara es alineada.

```
movdqa  xmm1,   [r10]
movdqa  xmm2,   xmm1
movdqa  xmm3,   xmm1
movdqa  xmm4,   xmm1

pshufb  xmm1,   [shuffle_packed_bbp4_byte_0]
pshufb  xmm2,   [shuffle_packed_bbp4_byte_1]
pshufb  xmm3,   [shuffle_packed_bbp4_byte_2]
pshufb  xmm4,   [shuffle_packed_bbp4_byte_3]

; CVT to float
cvtdq2ps    xmm1,   xmm1
cvtdq2ps    xmm2,   xmm2
cvtdq2ps    xmm3,   xmm3
cvtdq2ps    xmm4,   xmm4

; Divide by bpp
divps   xmm1,   xmm0
divps   xmm2,   xmm0
divps   xmm3,   xmm0
divps   xmm4,   xmm0

pshufb  xmm1,   [shufle_repack_1] ;REPACK 4 pixels
movd    r15d,   xmm1
mov     [r12],  r15d
add     r12,    4
```

Como puede observarse, usando SIMD se procesan 4 y 5 píxeles simultáneamente, mientras que en C las operaciones deben ser componente a componente.

5.6 Gauss

En C, las 2 partes mas importantes son:

```
SquareMatrix getMatrix(const unsigned char mSize, const float sigma) {

        SquareMatrix mat;
        mat.size = mSize;
        mat.data = (float*)malloc(mSize*mSize * sizeof(float));

        int i = 0;

        for (int y = -mSize / 2; y <= mSize / 2; y++) {
                for (int x = -mSize / 2; x <= mSize / 2; x++) {
                        mat.data[i] = getGaussValue(x, y, sigma);
                        i++;
                }
        }

        return mat;
}
```

Que construye el Kernel de convolución, y:

```
for (y = 0; y < src->height; y++) {
        for (x = 0; x < src->width; x++) {

            performConvolutionStep(src, mat, x, y, newData + srcIndex);
                srcIndex += src->bpp;
        }
}

.....

void performConvolutionStep(const Image* image, const SquareMatrix* mat,
                            const unsigned int x, const unsigned int y, short* dst){
        int srcIndex;
        int matIndex = 0;

        int xOffset, yOffset;

        float fDst = 0;
```

```
        for (int my = -mat->size / 2; my <= mat->size / 2; my++) {
                for (int mx = -mat->size / 2; mx <= mat->size / 2; mx++) {

                        if ((int)y + my < 0 || y + my >= image->height) {
                                yOffset = 0;
                        } else {
                                yOffset = my;
                        }

                        if ((int)x + mx < 0 || x + mx >= image->width) {
                                xOffset = 0;
                        } else {
                                xOffset = mx;
                        }

                        srcIndex = ((y + yOffset) * image->width + (x + xOffset));

                        float matValue = mat->data[matIndex];

                        float delta = image->data[srcIndex] * matValue;
                        fDst += delta;

                        matIndex++;
                }
        }

        dst[0] = (short)fDst;
}
```

Que realiza la convolución. Notar que en esta última función se tienen en cuenta los casos bordes.

En Ensamblador, una implementación eficiente que aproveche SIMD no fue trivial dado que:

- Cada elemento debe ser convertido a float (4 Bytes) antes de operar.
- El kernel tiene tamaño impar y variable.
- En la imagen, cada linea usada para la convolución no se encuentra en una zona contigua de memoria.

Por lo tanto, esta implementación fue pensada de la siguiente manera.

- Cada linea a procesar sera realizada de forma independiente.
- Dependiendo del tamaño del kernel, cada linea a procesar podría requerir múltiples cargas de memoria.
- Para evitar lecturas desalineadas del kernel, se uso una estructura diferente a C, donde cada linea del mismo esta alineada a 16 bytes.

En pseudo código seria algo como:

```
accumulator = 0

for line in lineas(kernel)
    lineSections = (kernel.size - 1) / 4 + 1

    for i in 0...lineSection

        kernelSection = readSection(line, i)
        truncateSectionIfNeeded(kernelSection)

        imageSection = read(imageLine + offset)

        convertToFloat(imageSection)

        accumulator += sumHorizonta( multiplySIMD (kernelSection, imageSection) )

        advance(offset)
    end for
end for
```

Un ejemplo de un kernel alineado de 5 x 5

$$\begin{Bmatrix} 0.003765 & 0.015019 & 0.023792 & 0.015019 & 0.003765 & 0 & 0 & 0 \\ 0.015019 & 0.059912 & 0.094907 & 0.059912 & 0.015019 & 0 & 0 & 0 \\ 0.023792 & 0.094907 & 0.150342 & 0.094907 & 0.023792 & 0 & 0 & 0 \\ 0.015019 & 0.059912 & 0.094907 & 0.059912 & 0.015019 & 0 & 0 & 0 \\ 0.003765 & 0.015019 & 0.023792 & 0.015019 & 0.003765 & 0 & 0 & 0 \end{Bmatrix}$$

En ensamblador, una version resumida es la siguiente:

```
loop_y:                 ;for (y = 0; y < src->height; y++)
        ....
loop_x:                 ;for (x = 0; x < src->width; x++)
        ....
        ;performConvolutionStep(src, mat, x, y, newData + srcIndex);
        call    performConvolutionStep_1bpp
    ....
end_loop_x:
        ....
end_loop_y:


;;;;;;;;;;;;;


performConvolutionStep_1bpp: ;const Image* image, const SquareMatrix* mat,
                             ;const unsigned int x, const unsigned int y, short* dst

....
; result is stored in xmm1
pxor        xmm8,   xmm8
convolution_line_loop:
....
call        performConvolutionLine_1bpp ;(const char* line, float* matLine,
                                        ; const unsigned int x, int matSize)

addss       xmm8,   xmm0
....
pshufb      xmm2,   [shuffle_packed_4_byte_to_float]
cvtdq2ps    xmm2,   xmm2
....


end_convolution_line_loop:
....
ret



;;;;;;;;;;;;;


performConvolutionLine_1bpp: ;float (const char* line, float* matLine,
                             ; const unsigned int x, int matSize)
```

```
....
loop_sections:
....
movdqa  xmm1,   [rsi]                ; load matrix line section
movdqu  xmm2,   [rdi]                ; load image line section

....
; Check if last section and trunkate
....

....
end_loop_sections:
```

5.7 Sobel

La primera parte de la implementación de sobel, utiliza el mismo código de convolución en Gauss.
La segunda parte, está implementada con el siguiente código:

```
for (unsigned int i = 0; i < src->width * src->height * src->bpp; i++) {
                float v = sqrt(xResult[i] * xResult[i] + yResult[i] * yResult[i]);

                if(v > 255) {
                        dstLum->data[i] = 255;
                } else {
                        dstLum->data[i] = (unsigned char)v;
                }

                short angle = 0;

                angle = normalizeAngle(atan2f(xResult[i], yResult[i]));

                dstAngle->data[i] = angle;
}
```

Donde *normalizeAngle* es una simple seguidilla de if-else anidados que normaliza el ángulo como se explicó anteriormente.

Vale la pena aclarar, que aparte de la convolución, el resto de la implementación en ensamblador es igual al de C. Esto se debe a la imposibilidad de utilizar la función *atan*2 de forma eficiente junto a SIMD. Se implementó una versión de todo este código en Ensamblador, pero resulto mas ineficiente que C, por lo cual fue removida.

5.8 Non-Maximus Supression

Ésta etapa recorre píxel a píxel la imagen de entrada de intensidad, y según el ángulo de dicho píxel en la imagen de entrada de ángulo, lee el valor de los píxeles vecinos. Estos píxeles, excepto para el caso de ángulo = 0, no son contiguos, y por lo tanto no es un algoritmo elegible para ser optimizado mediante SIMD, y por ese motivo su implementación fue hecha únicamente en C.

....

```
for (unsigned short y = 0; y < lum->height; y++) {
        for (unsigned short x = 0; x < lum->width; x++) {

                unsigned int index = (y * lum->width + x);

                if (isMax(lum, x, y, angle->data[index], 0)) {
                        dst->data[index] = lum->data[index];
                } else {
                        dst->data[index] = 0;
                }
        }
}
```

....

```
char isMax(const Image* image, const unsigned int x, const unsigned int y,
            const unsigned char angle, const unsigned char bOffset){

        unsigned int xA, yA, xC, yC;

        int iB = ((y * image->width + x) * image->bpp) + bOffset;


        switch (angle) {
        case 0:
        case 45:
                if (y == 0){
                        yA = y;
                } else {
                        yA = y-1;
                }

                if (y >= image->height-1) {
                        yC = y;
                } else {
                        yC = y + 1;
                }
```

```
        break;
case 90:
        yA = y;
        yC = y;
        break;
case 135:
        if (y == 0) {
                yC = y;
        } else {
                yC = y - 1;
        }

        if (y >= image->height-1) {
                yA = y;
        } else {
                yA = y + 1;
        }
        break;
}

switch (angle) {
case 0:
        xA = x;
        xC = x;
        break;
case 45:
case 90:
case 135:
        if (x == 0) {
                xA = x;
        } else {
                xA = x - 1;
        }

        if (x >= image->width-1) {
                xC = x;
        } else {
                xC = x + 1;
        }
        break;
}

int iA = ((yA * image->width + xA) * image->bpp) + bOffset;
int iC = ((yC * image->width + xC) * image->bpp) + bOffset;
```

```
        unsigned char pA = image->data[iA];
        unsigned char pB = image->data[iB];
        unsigned char pC = image->data[iC];


        return pB >= pA && pB >= pC;
}
```

5.9 Souble Threshold

Esta etapa es bastante trivial, y está implementada en C y en Ensamblador.

```
for (unsigned int i = 0; i < src->width * src->height; i += 1) {
        unsigned short lum = 0;

        lum += src->data[i];

        if (lum < thresholdlow) {
                dst->data[i] = 0;

        } else if (lum < thresholdHigh) {
                dst->data[i] = vlow;
        } else {
                dst->data[i] = vHigh;
        }
}
```

En ensamblador la implementación es bastante sencilla. Primero se setean registros que se usaran de comparadores con los valores de umbral y sus limites.

```
;; Prepare xmm comparers
        and     esi,    0x000000FF
        and     edx,    0x000000FF
        and     ecx,    0x000000FF
        and     r8d,    0x000000FF

        pxor    xmm0,   xmm0
        pxor    xmm1,   xmm1    ; < thres high
        pxor    xmm2,   xmm2    ; < val low
        pxor    xmm3,   xmm3    ; < val high
        movd    xmm0,   esi     ; < thres low
        movd    xmm1,   edx
        movd    xmm2,   ecx
        movd    xmm3,   r8d
```

Y luego se hace el loop que compara y setea según el umbral

```
loop:
        ....

        ; Read first 8 bytes
        movdqa  xmm5,   [rdi]
        pshufb  xmm5,   [shuffle_l_byte_to_word]

        movdqa  xmm6,   xmm5

        pcmpgtw xmm5,   xmm0    ; thres low flags
        pcmpgtw xmm6,   xmm1    ; thress high flags
        movdqa  xmm7,   xmm6
        pxor    xmm7,   xmm4    ; thres high inverse --- logical not xmm3 (xmm4 is all ones)
        pand    xmm5,   xmm7    ; abs thres low flag (without thres high flags)

        pand    xmm5,   xmm2
        pand    xmm6,   xmm3

        por     xmm5,   xmm6

        ;Read second 8 bytes
        movdqa  xmm8,   [rdi]
        pshufb  xmm8,   [shuffle_h_byte_to_word]

        movdqa  xmm6,   xmm8

        pcmpgtw xmm8,   xmm0    ; thres low flags
        pcmpgtw xmm6,   xmm1    ; thress high flags
        movdqa  xmm7,   xmm6
        pxor    xmm7,   xmm4    ; thres high inverse --- logical not xmm3 (xmm4 is all ones)
        pand    xmm8,   xmm7    ; abs thres low flag (without thres high flags)

        pand    xmm8,   xmm2
        pand    xmm6,   xmm3

        por xmm8,   xmm6

        ; Write result
        pshufb  xmm5,   [shuffle_l_word_to_byte]
        pshufb  xmm8,   [shuffle_h_word_to_byte]

        por     xmm5,   xmm8
        movdqa  [rax],  xmm5

        ....
        jmp loop
```

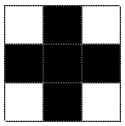
```
endLoop:
```

5.10 Hysteresis Threshold

Esta etapa es la más compleja a nivel algorítmico ya que es necesario agrupar los píxeles por grupos, para luego asignarles a todo su valor máximo (separando así bordes bordes reales de bordes "fantasmas").

Sobre las estructuras:

- La estructura que guarda el grupo de cada píxel, es una matriz de unsigned int UI_Matrix
- Para los conflictos, y el valor máximo del grupo, por una cuestión de eficiencia, se utilizaron arrays. Ocupan mucho mas espacio que un mapa, pero mejora el rendimiento en varios ordenes de magnitud. Dado que es un array, se le debe definir un tamaño máximo, y este coincide con la máxima cantidad de grupos que podría existir en una imagen. Este valor es igual a $(alto/2 + 1) * (ancho/2 + 1)$



Primero se hace la pasada "row by row" para agrupar los pixeles y marcar los conflictos.

```
for (unsigned int index = 0; index < src->width * src->height; index++) {

    // If zero, no need to group
        if (src->data[index] == 0) {
                hystMatrix.data[index] = 0;
                continue;
        }

        ....

        int parentVals[] = { -1, -1, -1, -1 };

        if (uIndex > 0) {
                parentVals[0] = hystMatrix.data[uIndex];
        }
        if (lIndex > 0) {
                parentVals[1] = hystMatrix.data[lIndex];
        }
        if (ulIndex > 0) {
                parentVals[2] = hystMatrix.data[ulIndex];
```

```
}
if (urIndex > 0) {
        parentVals[3] = hystMatrix.data[urIndex];
}

int firstValue = -1;
int fisrtValueIndex = -1;
bool conflicted = 0;

for (int v = 0; v < 4; v++) {
        if (parentVals[v] > 0) {
                if (firstValue < 0) {
                        firstValue = parentVals[v];
                        fisrtValueIndex = v;
                } else {
                        if (parentVals[v] > 0 && parentVals[v] != firstValue) {
                                conflicted = 1;
                                break;
                        }
                }
        }
}

if (firstValue < 0) {
        // New Group (may have conflicts later on)
        lastNewGroup++;
        hystMatrix.data[index] = lastNewGroup;

        continue;
} else {
        hystMatrix.data[index] = firstValue;

        if (!conflicted) {
                continue;
        } else {
                for (int v = fisrtValueIndex + 1; v < 4; v++) {
                        if (parentVals[v] > 0) {
                                // Check Not Equals
                                if (parentVals[v] == firstValue) {
                                        continue;
                                }

                                // Avoid circular reference
                                if (conflictMap[firstValue] != 0
                                                && conflictMap[firstValue] == parent
                                        continue;
```

```
                                        }

                                        if (parentVals[v] == 1) {
                                                //printf("asdasd");
                                        }

                                        conflictMap[parentVals[v]] = firstValue;
                                }
                        }
                }
        }
}
```

Se resuleven los conflictos:

```
....
flattenConfilctMap(conflictMap, maxGroupSize);
...

void flattenConfilctMap(unsigned int* map, unsigned int size)
{
        for(unsigned int index = 0; index < size; index++)
        {
                map[index] = maxValueForConflictGroup(map, size, index);
        }
}

int maxValueForConflictGroup(unsigned int* map,
                             unsigned int size, unsigned int index)
{
        unsigned int maxValue = map[index];

        if(index > maxValue)
        {
                return index;
        }

        while(TRUE)
        {
                if(maxValue == 0) break;
                if(maxValue >= size) break;
                if(map[maxValue] == 0) break;

                return maxValueForConflictGroup(map, size, maxValue);
        }
```

```
        return maxValue;
}
```

Luego, se encuentra el máximo de cada grupo

```
....
// Check all pixels with same group, and select the maximun value for that group
for (unsigned int index = 0; index < hystMatrix.width * hystMatrix.height;        index++) {
        unsigned int group = hystMatrix.data[index];

        if (valueMap[group] != 0) {
                valueMap[group] = valueMap[group] > src->data[index] ? valueMap[group] : src
        } else {
                valueMap[group] = src->data[index];
        }
}
....
```

Y por ultimo se "pintan" los pixeles de cada grupo con su valor máximo

```
// PAINT
for (unsigned int index = 0; index < hystMatrix.width * hystMatrix.height;
                index++) {
        unsigned int group = hystMatrix.data[index];

        if (group == 0) {
                dst->data[index] = 0;
        } else {
                dst->data[index] = valueMap[group] == 255 ? 255 : 0;
        }
}
```

6 Optimizaciones

6.1 Sobre las mediciones

Todas las mediciones de las optimizaciones se hicieron utilizado imágenes de los siguientes tamaños:

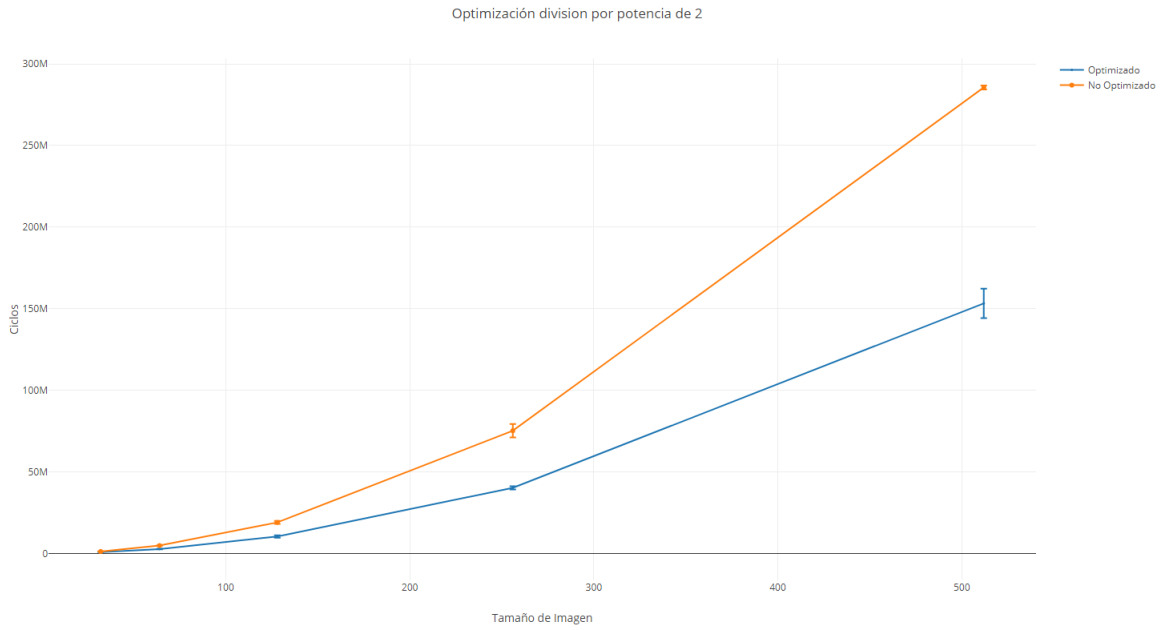
- 32 x 24
- 64 x 48
- 128 x 97
- 256 x 193
- 512 x 386

Y con los siguientes parámetros:

- $\sigma = 1.4$
- Matriz de gauss = 5 x 5 (Radio 2)
- Min threshold = 30
- Max threshold = 120

6.2 Implementación original

La implementación original de Ensamblador, aunque perfectamente funcional, resultó menos eficiente que su contraparte en C optimizado. Esto llevó a la necesidad de investigar si era posible realizar una o mas optimizaciones que permitieran validar la hipótesis de este trabajo en lugar de falsificarla.



En promedio, C era un 274% mas eficiente que la implementación original de ensamblador.

6.3 División por potencias de 2

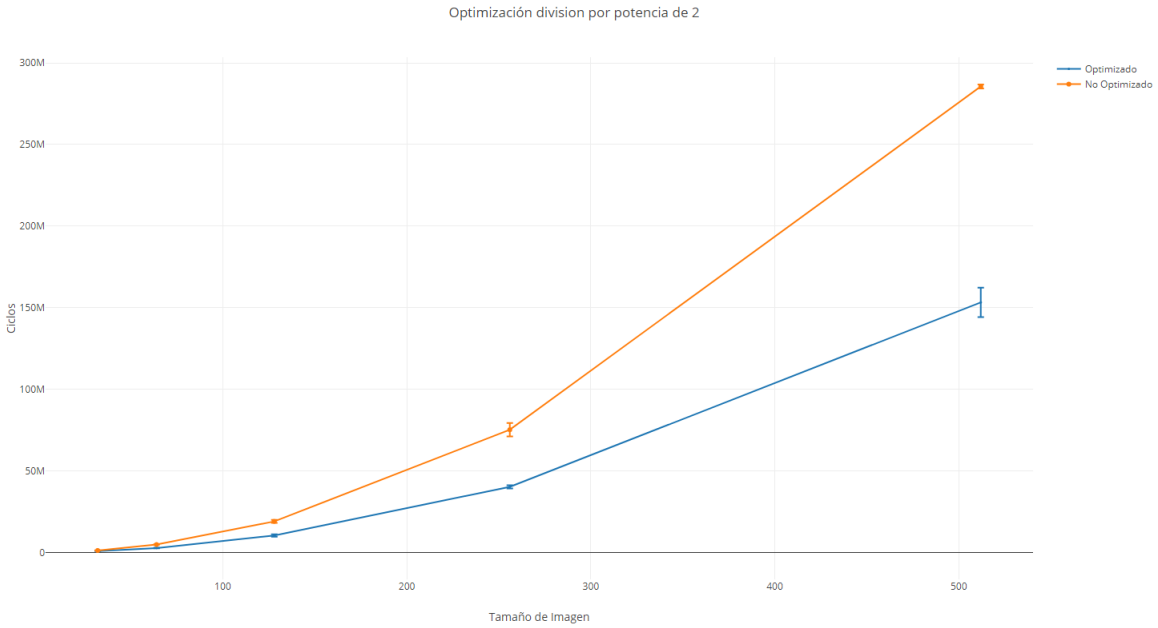
En varias secciones del código, fue necesario dividir enteros por números potencia de 2. En estos casos, originalmente resueltos con la instrucción idiv, fueron optimizados aprovechando las características de los números binarios y las operaciones a nivel bit. Ej.

```
mov     eax,    ecx
mov     edx,    0
mov     r8,     4
idiv    r8          ;   eax = ecx / 4 |||| edx = ecx % 4
```

vs.

```
mov eax,    ecx
sar eax,    2       ;   eax = ecx / 4
mov edx,    ecx
and edx,    0x03    ;   edx = ecx % 4
```

Este cambio arrojo los siguientes resultados:



En promedio, se mejoro el rendimiento en un 76.11%

6.4 Eliminación saltos mediante redundancia

En muchas situaciones, al momento de "branchear" entre 2 opciones, originalmente estaba desarrollado de la siguiente manera:

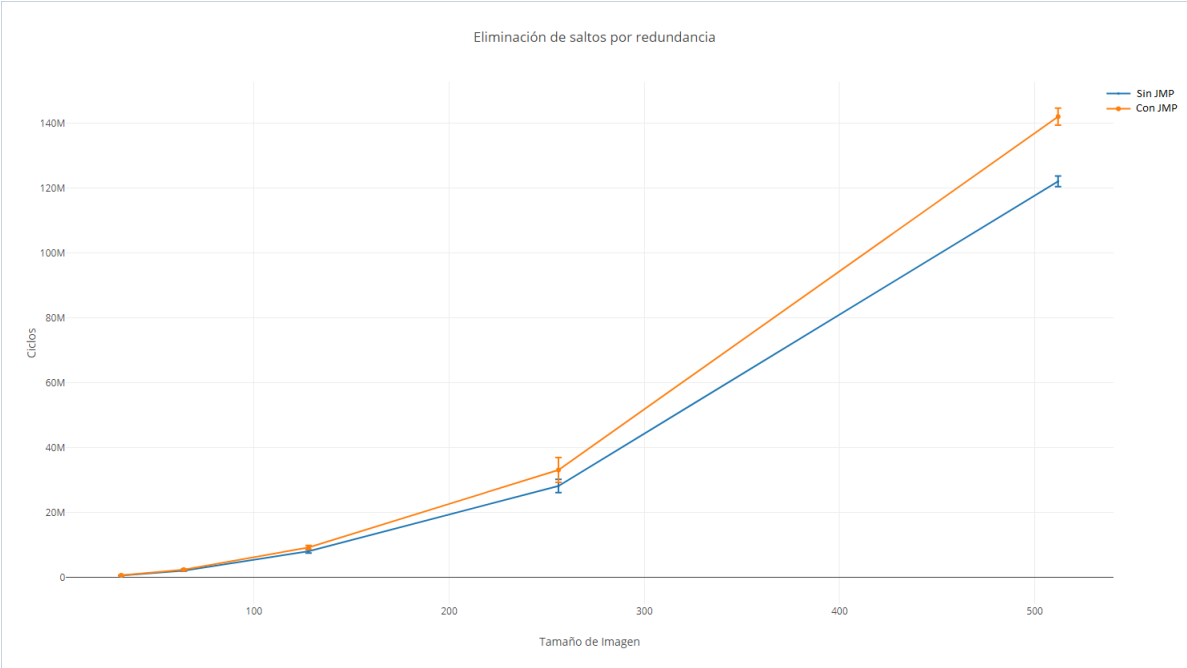
```
loop_start:

cmp r14d, 1
je if_statement
jmp else_statment

if_statement:
        mov r8, 1
        jmp end_if
else_statment:
        mov r8, 2
end_if:
```

```
A
B
C

jmp loop_start
```

Si bien esta implementación es mucho mas clara, el ultimo jmp puede ser eliminado, agregando una copia del "desenlace":

```
loop_start:

cmp r14d, 1
je if_statement
mov r8, 1
A
B
C
jmp loop_start

if_statement:
        mov r8, 1
A
B
C
jmp loop_start
```

Esto, implementado en varias secciones del código, arrojo el siguiente resultado:



Mejoran el rendimiento en un 14.98%

6.5 Carga alineada y desalineada

En la implementación original, asumiendo que cargar datos en registros SIMD de forma alineada seria mucho mas eficiente, aunque hubiese que agregar mas lógica para poder leer los datos correctamente, fue implementado de esta forma.

Primero:

- Antes de leer un dato, se calculada su posición anterior alineada y el offset desde este punto.

Es decir, suponiendo que la posición a leer fuere 0x10A77004
Se obtiene por un lado la posición alineada = 0x10A77000 y el offset = 0x4
Si la posición estaba en r9:

```
mov     edx,    r9d
mov     eax,    r9d
and     edx,    0x0F          ; edx = x % 16
and     eax,    0xFFFFFFF0    ; eax / 16  then eax * 16
```

- Luego de leer la posición alineada con movdqa, se realizaba un shift del registro igual al offset.

Como la única instrucción para shiftear registros xmm es con un valor inmediato, originalmente, se utilizo un loop de shifts de "a uno".

Teniendo el offset en ecx

```
line_unaligned_loop:
    psrldq  xmm2,   1
    loop    line_unaligned_loop
```

Esta operación era muy lenta, y se mejoro usando una tabla de saltos (versión Ensamblador de un switch case).

También, con el offset en ecx:

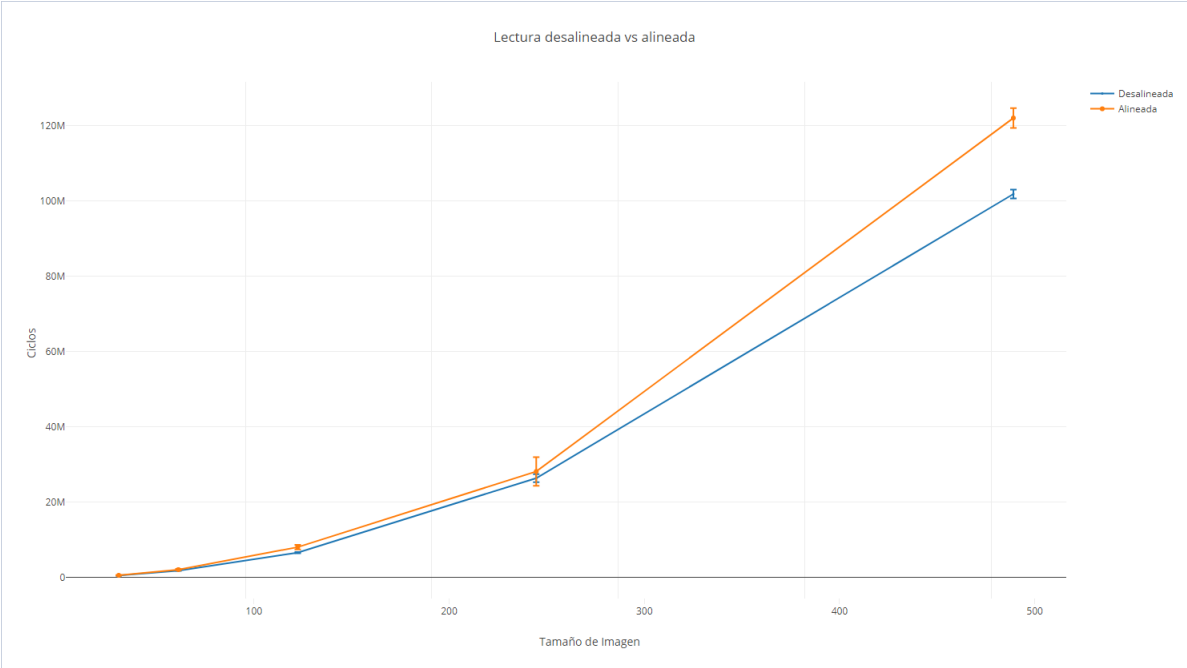
```
call shift_xmm2_right

....


section .rodata
shift_xmm2_right_table:
        dq        .0, .1, .2, .3, .4, .5, .6, .7, .8, .9, .10, .11, .12, .13, .14, .15

section .text

shift_xmm2_right:        ; (ecx times)
        cmp ecx, 16
        jge        default_shift_xmm2_right
        mov rcx, [shift_xmm2_right_table + ecx * 8]
        jmp rcx
        default_shift_xmm2_right:
        pxor xmm2, xmm2
        ret


shift_xmm2_right_table.0:
        ret
shift_xmm2_right_table.1:
        psrldq xmm2, 1
        ret
shift_xmm2_right_table.2:
        psrldq xmm2, 2
        ret

....
```

- Por último, se podía dar el caso de que al tratar de leer una posición, que luego de calcular su versión alineada, el offset quedara superior a 12 (los datos son leídos de a 4, pues hay que pasarlos a float), era necesario

realizar una segunda lectura alineada con los siguientes 16 bytes, y juntar los resultados.

```
check_double_line:
; If data is really separated by 16bytes alignment, must do another read, algin and join
        mov     ecx,    r9d             ; x
        add     ecx,    r13d            ; x + offset
        add     ecx,    r11d            ; x + offset + matSize

        cmp     ecx,    16              ; if(x + matSize > 16) then it is separated
        jle     continue_line

        movdqa  xmm3,   [rdi+16]        ; <- Second part of line

        mov     ecx,    16
        sub     ecx,    r12d            ; Inverse Alignment
        sub     ecx,    r13d

        cmp     ecx,    0               ; if ecx is zero, loop never ends
        je      line_unaligned_loop_2_end
        jg      line_unaligned_loop_2_left
        ; ecx is neither zero nor positive
        ; ecx = -ecx
        neg     ecx

        jmp     line_unaligned_loop_2_right


line_unaligned_loop_2_left:
        call    shift_xmm3_left
        jmp     line_unaligned_loop_2_end

line_unaligned_loop_2_right:
        call    shift_xmm3_right

line_unaligned_loop_2_end:
        por     xmm2,   xmm3
```

Todo este proceso agregaba un overhead tan grande que terminó siendo menos eficiente que usar una lectura desalineada con movdqu.



Siendo esto un 16.55% mas rápido.

7 Resultados

7.1 Sobre las mediciones

Todas las mediciones finales se hicieron utilizado imágenes de los siguientes tamaños:

- 32 x 24
- 50 x 39
- 64 x 48
- 128 x 97
- 256 x 193
- 512 x 386
- 1000 x 450

Y con los siguientes parámetros:

- $\sigma = 1.4$
- Matriz de gauss
  - 3 x 3 (radio 1)
  - 5 x 5 (radio 2)
  - 7 x 7 (radio 3)
- Min threshold = 30
- Max threshold = 120

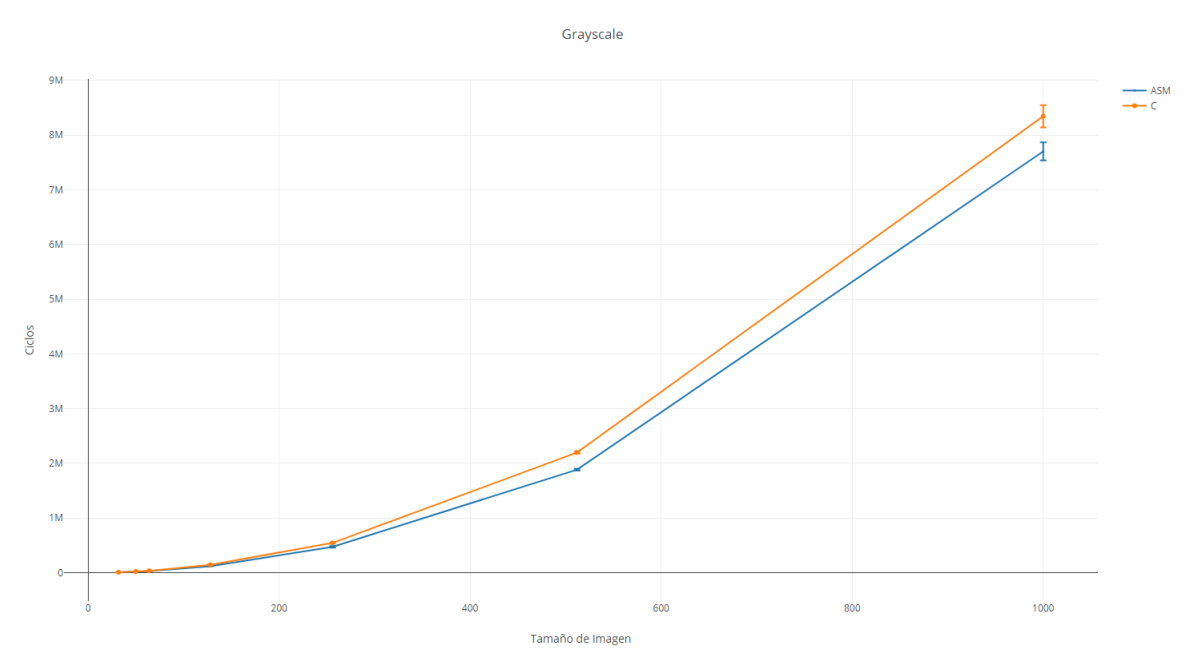
Todas las mediciones incluyen:

- Cantidad de ciclos usando la instrucción *rdtsc*.
- Cache misses usando la herramienta *perf* de linux (solo para proceso completo y no por etapas).
- Tiempo real usando la herramienta *perf* de linux (solo para proceso completo y no por etapas).

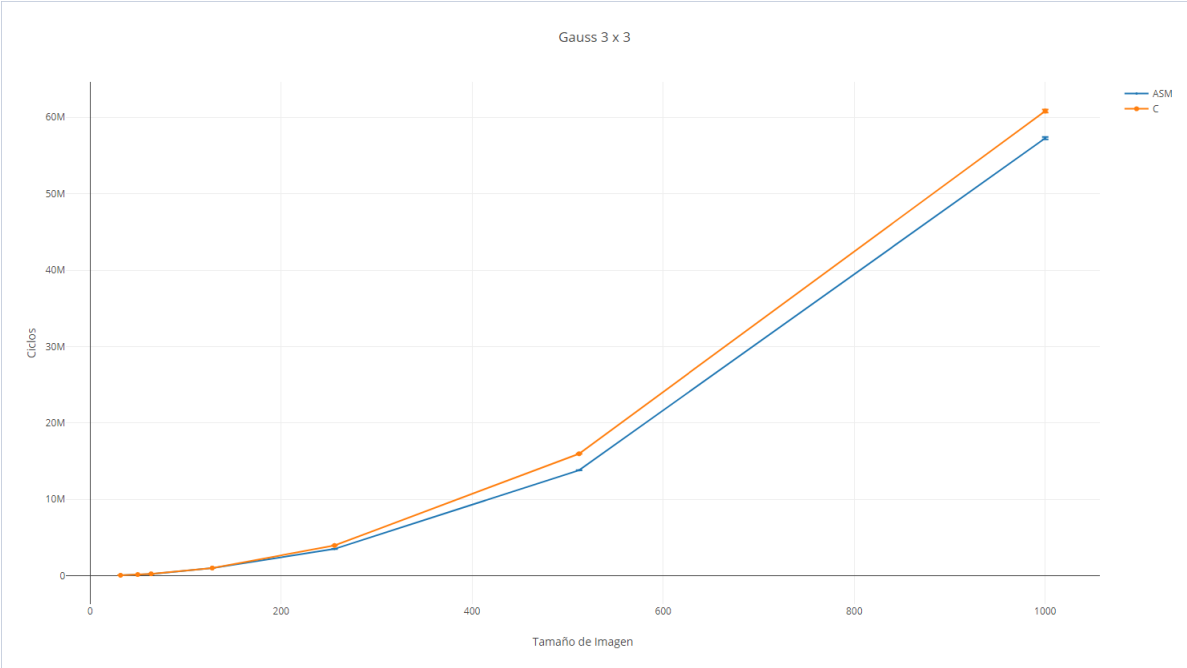
En cada medición, se utilizaron 10000 iteraciones.

7.2 Resultados por etapa

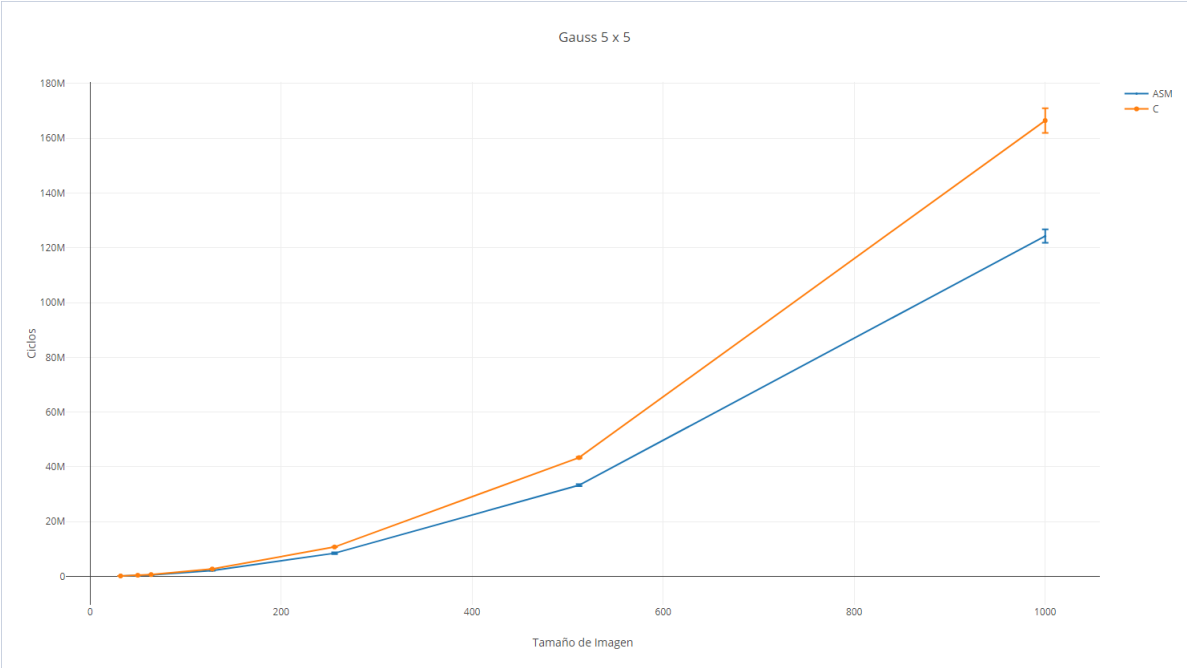
A continuación veremos una comparación de rendimiento en ciclos por etapa. No-Maximum Supression e Hysteresis no fueron comparadas ya que no tienen implementación en Ensamblador.



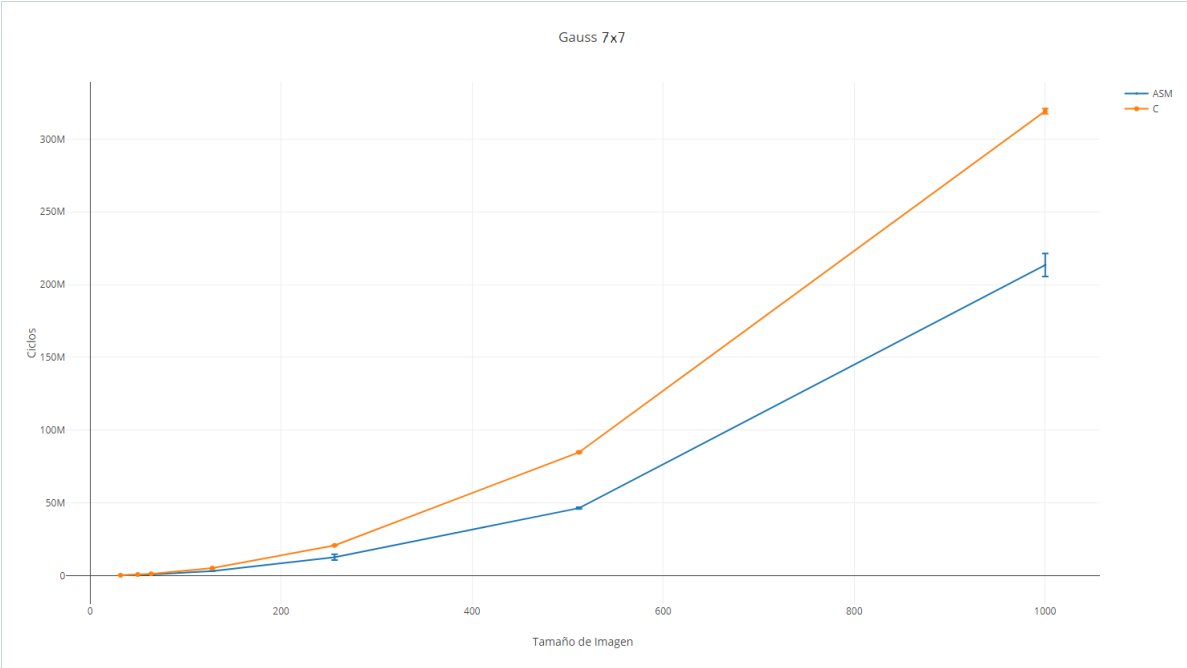
Mejora de rendimiento: 18.57%



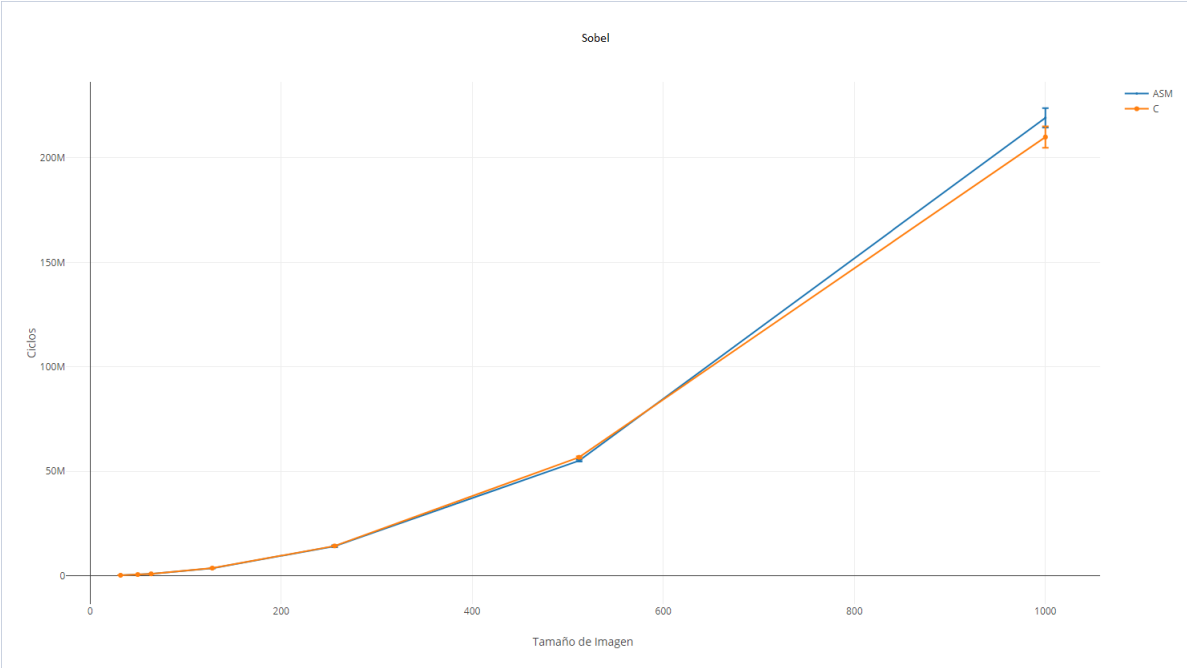
Mejora de rendimiento: 6.20%



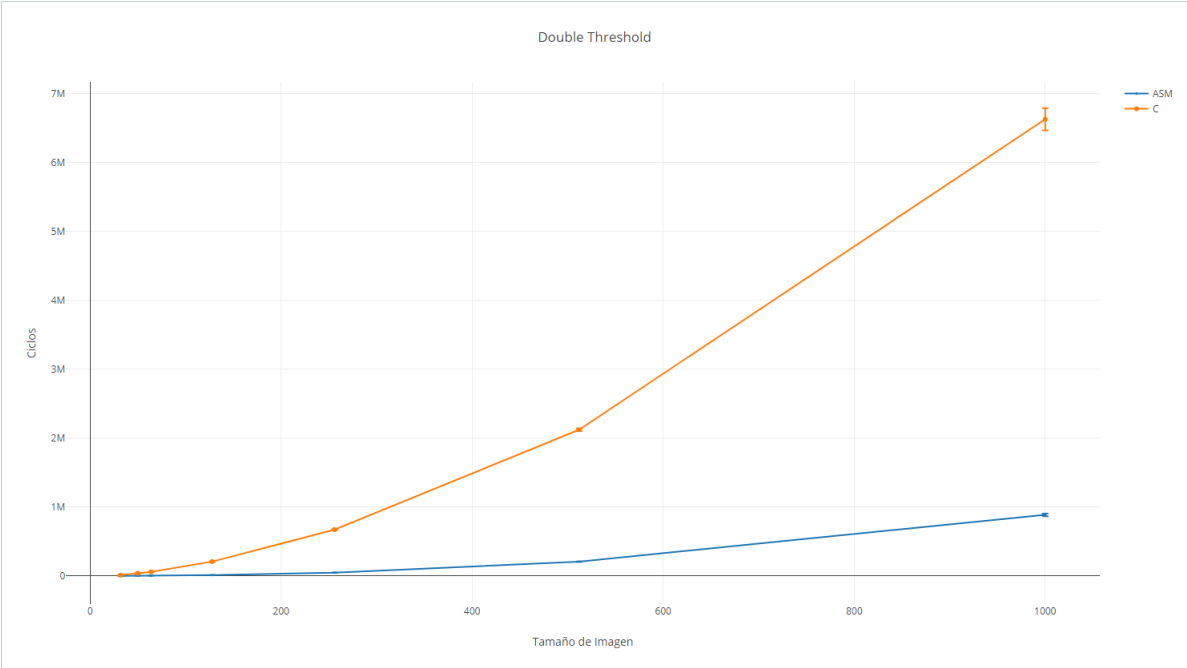
Mejora de rendimiento: 23.14%



Mejora de rendimiento: 52.63%

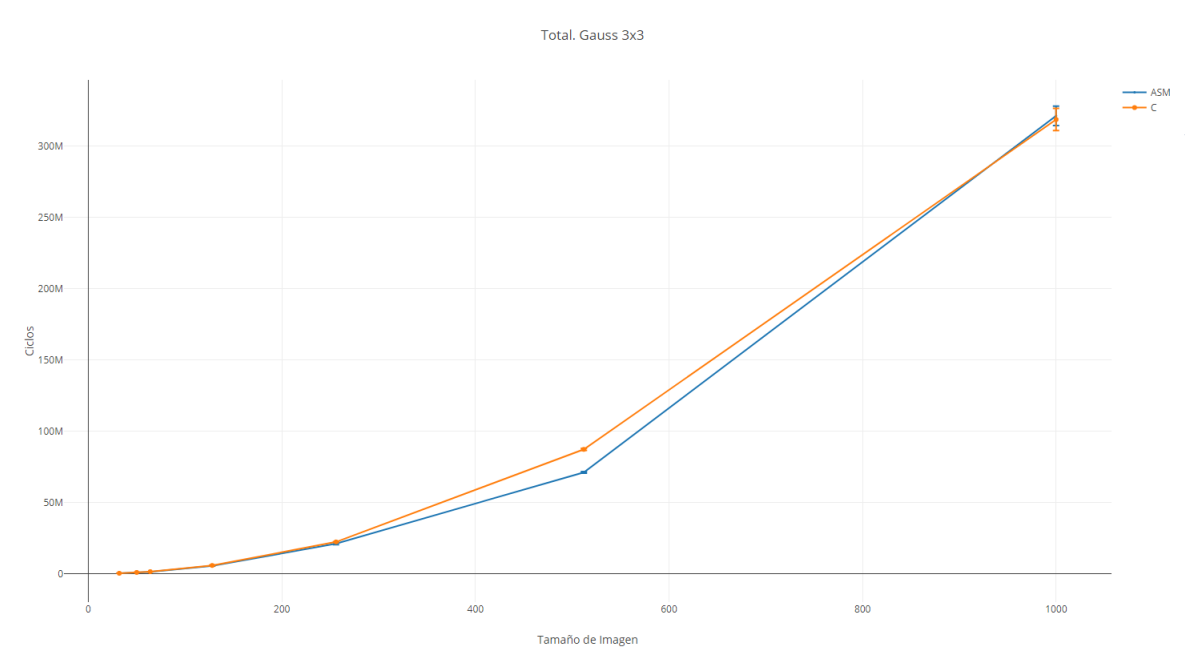


Mejora de rendimiento: 2.71%

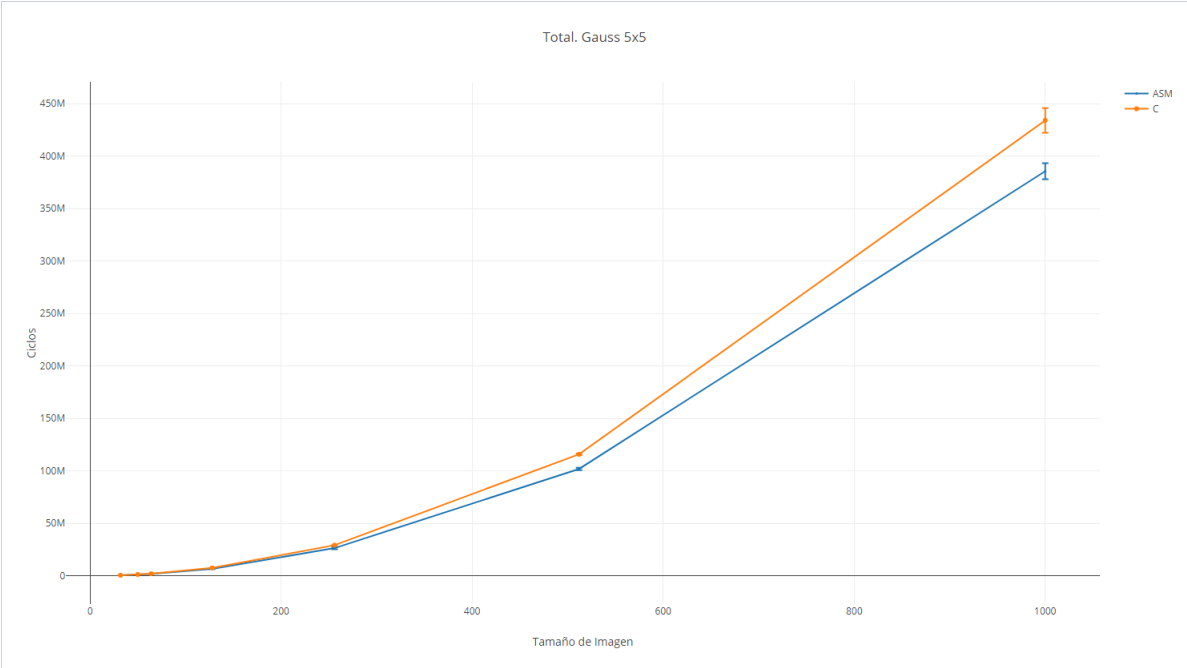


Mejora de rendimiento: 1301.35%

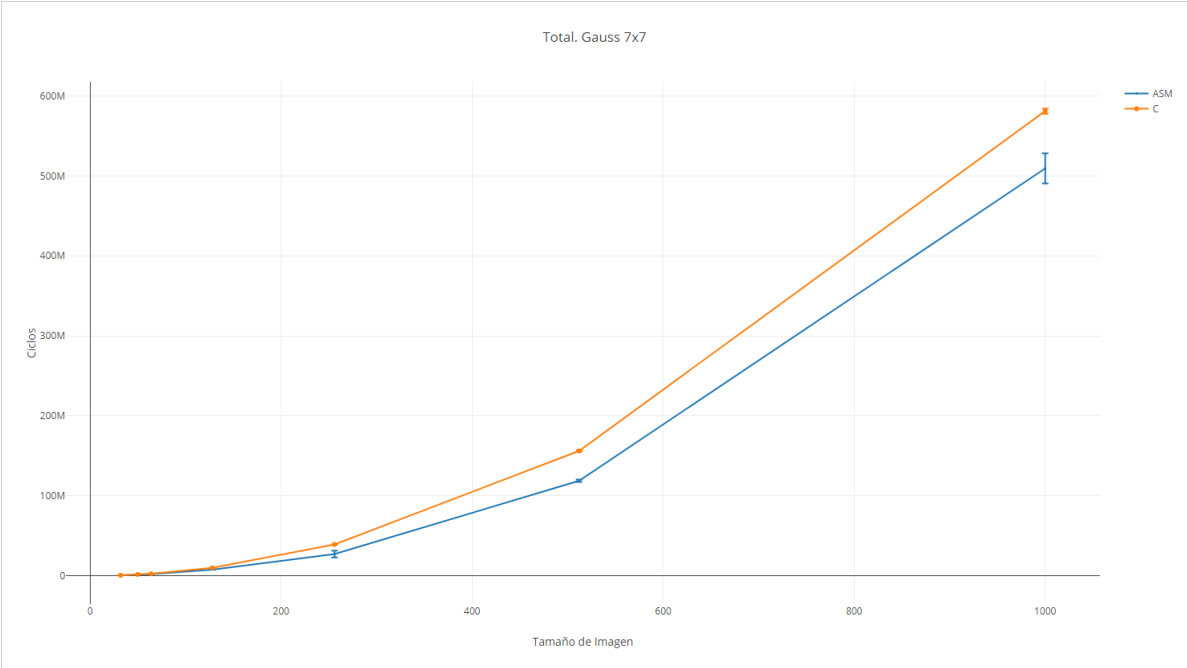
7.3 Resultados totales en ciclos



Mejora de rendimiento: 2.75%

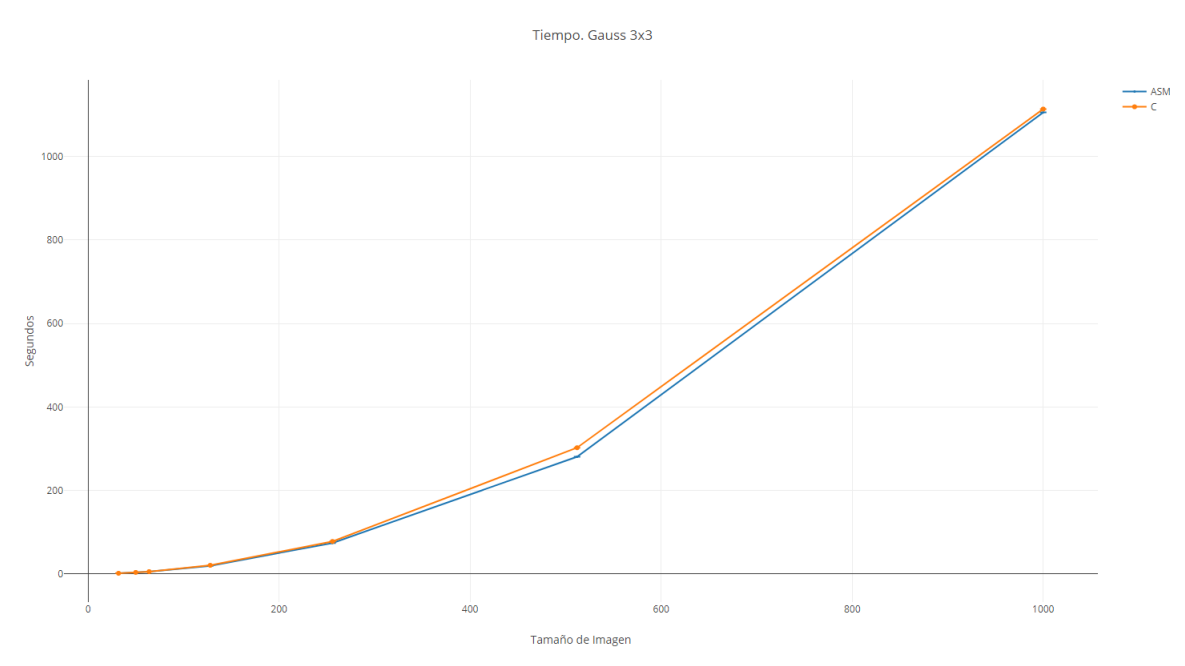


Mejora de rendimiento: 10.25%

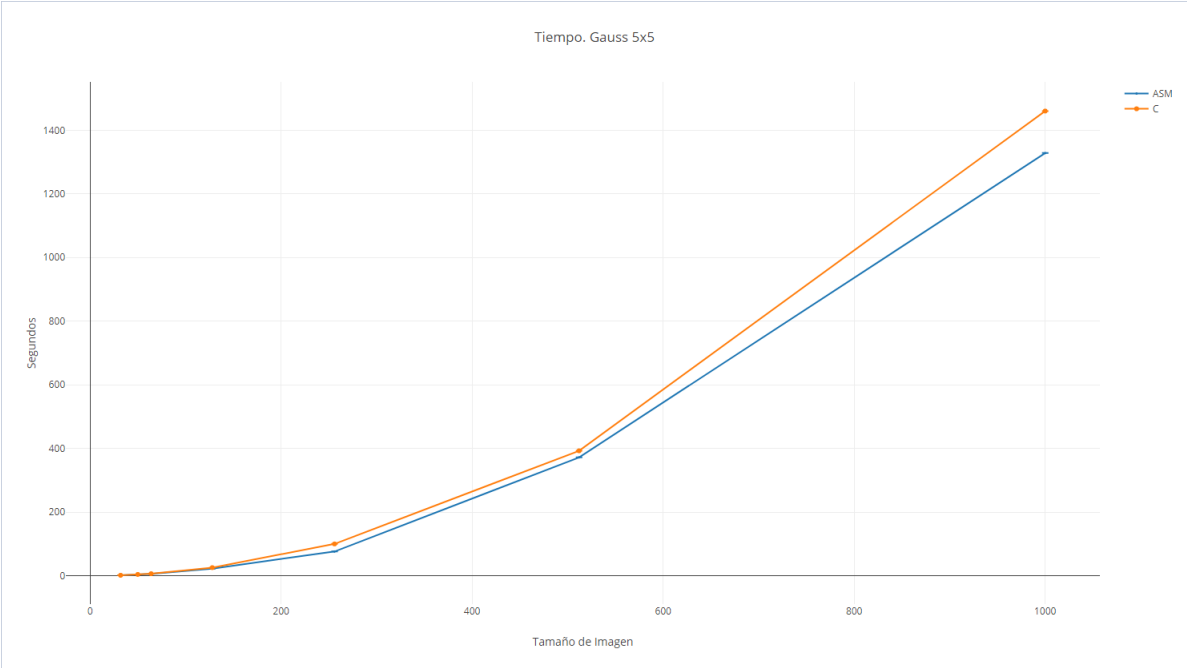


Mejora de rendimiento: 26.90%

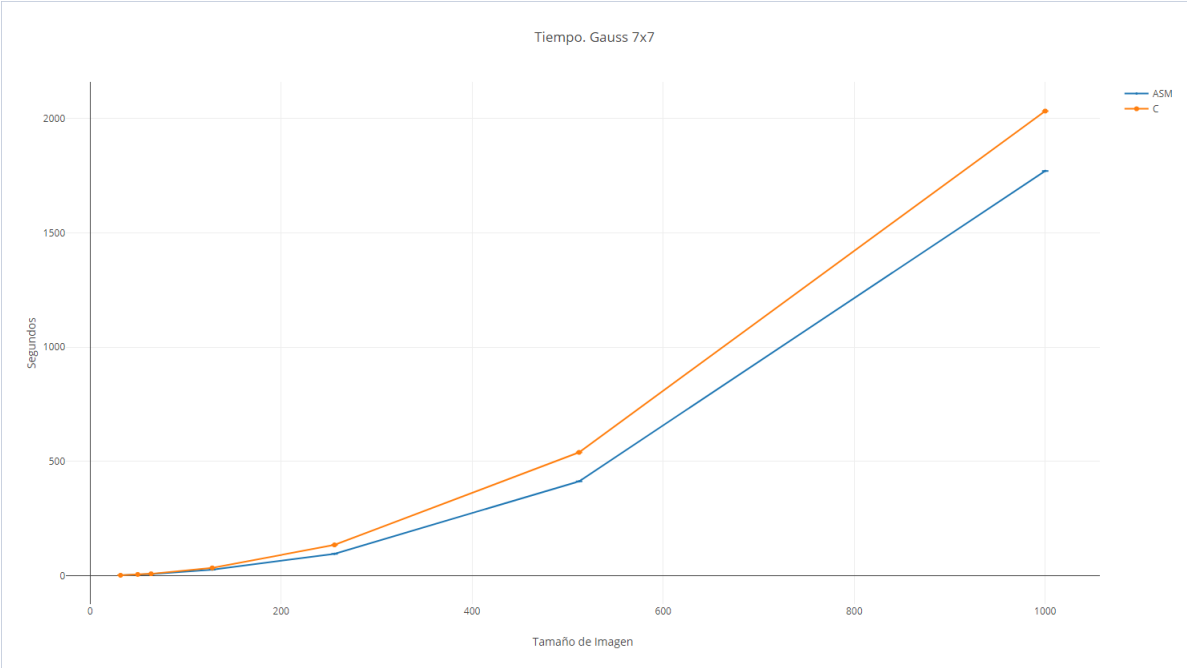
7.4 Resultados totales en segundos



Mejora de rendimiento:6.48%

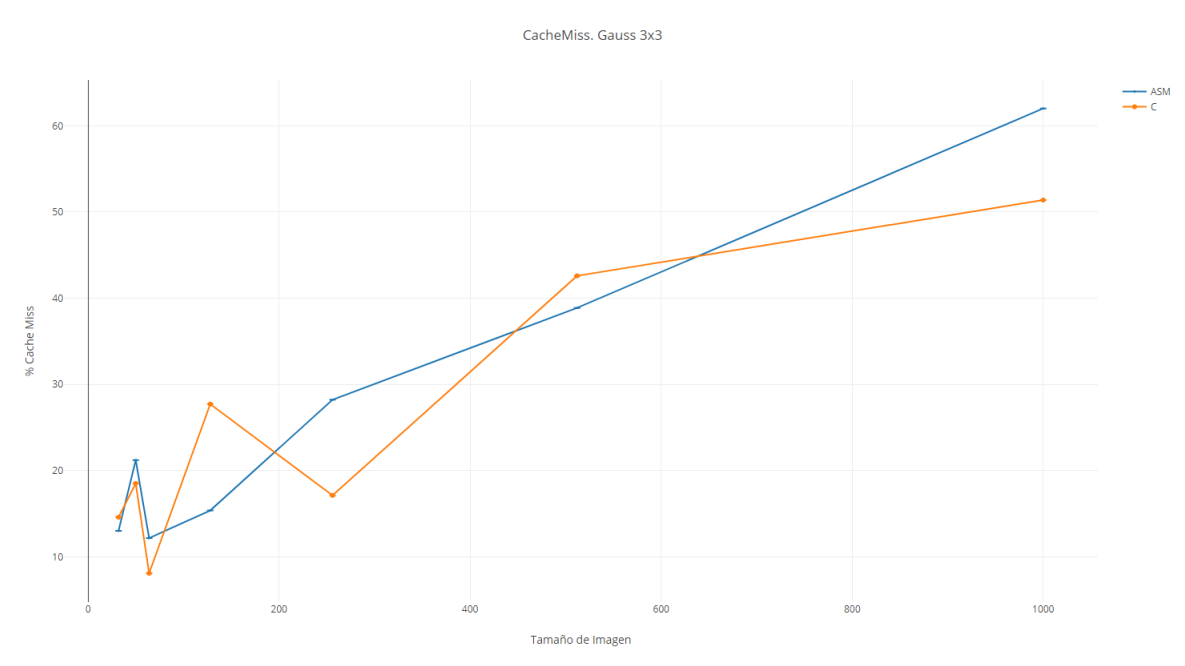


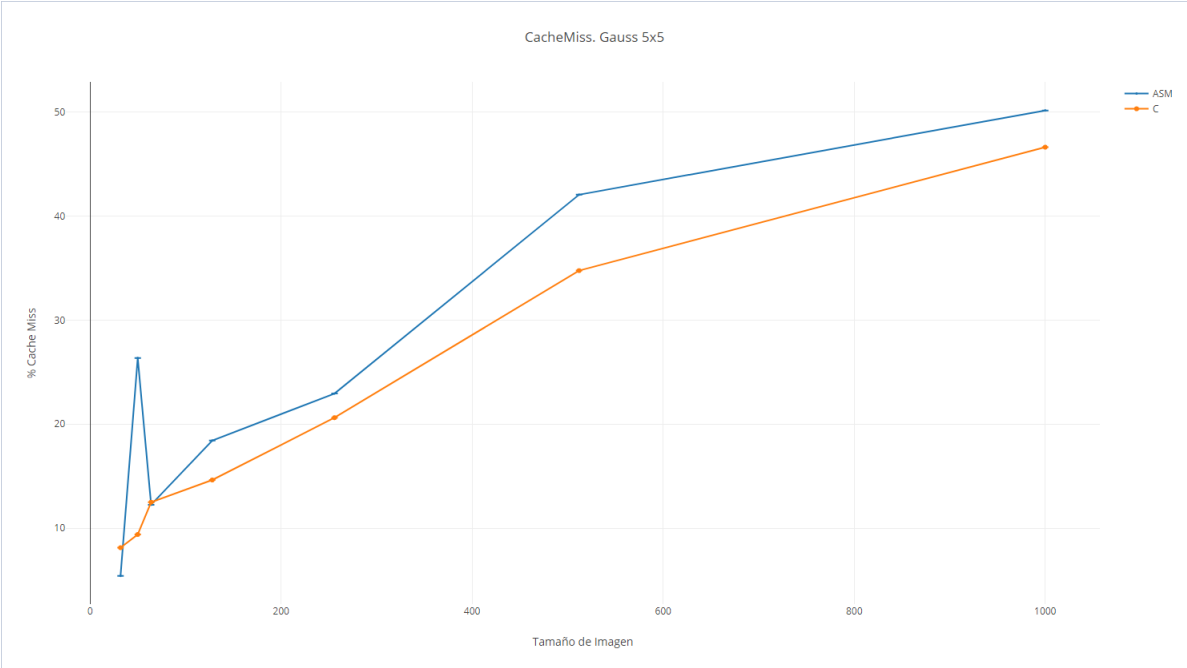
Mejora de rendimiento: 11.66%

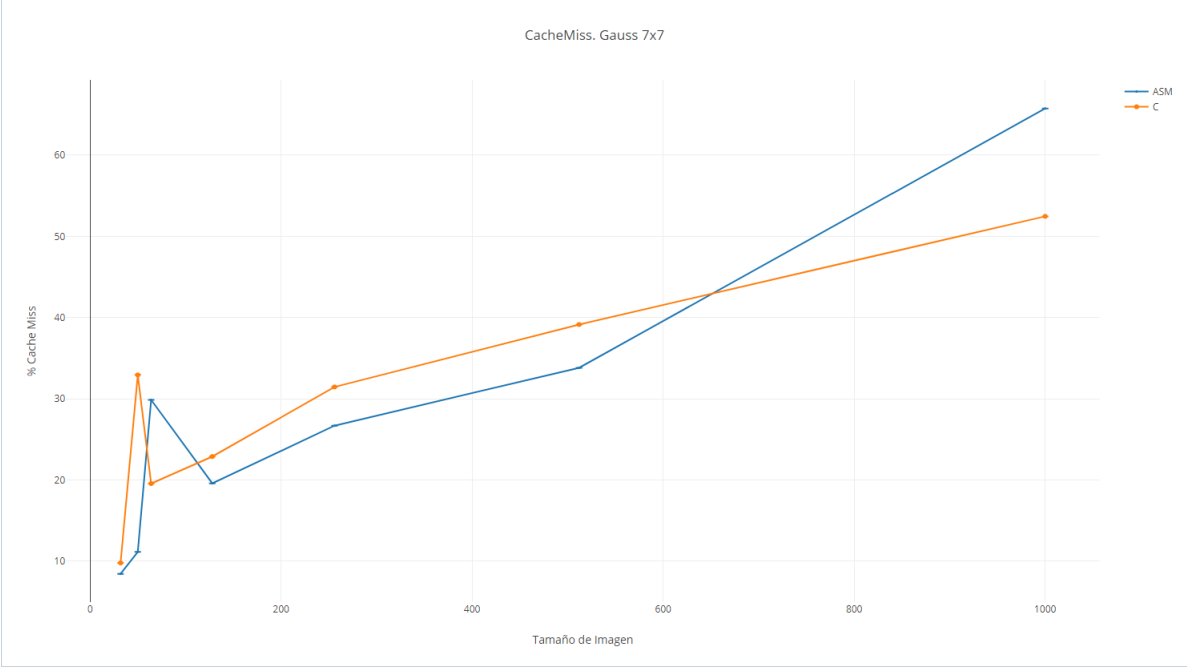


Mejora de rendimiento: 25.86%

7.5 Resultados totales. Cache Miss




CacheMiss. Gauss 5x5
ASM
C
% Cache Miss
50
40
30
20
10
0
200
400
600
800
1000
Tamaño de Imagen


CacheMiss. Gauss 7x7
ASM
C
% Cache Miss
60
50
40
30
20
10
0
200
400
600
800
1000
Tamaño de Imagen

8 Conclusión

8.1 Sobre la hipótesis

La hipótesis fue comprobada. La implementación en Ensamblador es mas eficiente que su contraparte en C.

8.2 Observaciones

- Observando las mediciones, se puede ver que el algoritmo más sencillo (double threshold), fue el más beneficiado, ya que su simplicidad permite sacar mayor provecho al paralelismo de SIMD.
- Para la convolución, mientras más grande fue la matriz, mayor fue la mejora de rendimiento. Esto se debe a que mientras más grande es la matriz, más grandes son sus filas, y por esto, más datos pueden ser procesados en paralelo. Esto nos permite decir que (por lo menos con esta implementación) es posible sacarle provecho a cualquier otro filtro que utilice una o más convoluciones como parte de su procedimiento.
- El proceso de la imagen no potencia de 2 (50x39) no presento cambios radicales en cuanto a su eficiencia más allá del incremento en su tamaño, pero si se presentaron picos de Cache Misses. Esto podría deberse a los saltos extras necesarios para considerar los casos bordes.

9 Referencias

- Canny, J., A Computational Approach To Edge Detection
- https://en.wikipedia.org/wiki/Canny_edge_detector
- http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm
- https://www.tutorialspoint.com/dip/sobel_operator.htm
- https://en.wikipedia.org/wiki/Connected-component_labeling
- https://www.codeproject.com/Articles/336915/Connected-Component-Labeling-Algorithm