DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Programación SIMD

Organización del Computador II
Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Matias Nahuel Castro Russo	203/19	castronahuel14@gmail.com
Juan Manuel Torsello	248/19	juantorsello@gmail.com
Maximo Yazlle	310/19	myazlle99@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

1.1. Sobre el informe

El objetivo de este trabajo es trabajar con el set de instrucciones SIMD - *Single Instruction Multiple Data* - de la arquitectura IA-32 (x86-64) de Intel. Estas instrucciones nos permiten realizar el procesamiento de múltiples datos en paralelo, lo cual nos evita tener que usar una cantidad mayor de instrucciones y operar con un menor contenido a la vez.

Con este objetivo en mente, implementamos distintos filtros gráficos que operan sobre imágenes a color. Para cada filtro contamos con una implementación en lenguaje C, cuyo algoritmo que opera píxel a píxel, y una implementación en lenguaje ASM utilizando instrucciones del modelo SIMD.

Lo que nos planteamos probar es si, dadas estas implementaciones en ASM y C, la implementación en lenguaje C será siempre menos performante que una implementada en ASM con instrucciones SIMD. Dado que este modelo nos permite operar sobre múltiples píxeles en simultáneo, creemos que habría una mejora en la velocidad de ejecución. La pregunta sería, de existir, ¿cuánta diferencia en velocidad hay entre ambas implementaciones?

1.2. Consideraciones y formato de la imagen

El formato de imagen sobre el cual trabajan las implementaciones es BMP. Las imágenes son consideradas como una matriz de píxeles, almacenadas en memoria de izquierda a derecha fila por fila Cada píxel, a su vez, consta de cuatro componentes que son RGBA (Red, Blue, Green, Alpha) las cuales ocupan un byte cada una. El valor de estos componentes está comprendido entre 0 y 255. Si se pasa de dicho margen, el valor será saturado (*hardcodeado* a su máximo o mínimo según corresponda).

El formato BMP implicaría que el mapa de bits de la imagen se lee desde abajo hacia arriba y de izquierda a derecha. Pero en este trabajo, se invirtió el mismo, de modo que las líneas de las imágenes se leen de arriba hacia abajo y de izquierda a derecha.

Adicionalmente, se considera el ancho de todas las imágenes mayor a 16 pixeles y divisible por 8.

1.3. Filtros

Implementamos tres filtros sobre el cual basamos nuestro informe y llevamos a cabo nuestro análisis. A continuación, damos una breve descripción de cada filtro.

Imagen fantasma

Este filtro genera un efecto de imagen fantasma sobre la imagen. Para esto se utiliza la misma imagen original en escala de grises y al doble de tamaño. Se puede pasar como parámetro el offset horizontal y vertical donde debe ubicarse la imagen fantasma siempre y cuando sean menor al tamaño de la imagen.

Color bordes

Este filtro resalta los bordes y siluetas de la imagen mediante una serie de cálculos entre los píxeles que rodean a cada píxel. Esto lo logra haciendo la suma de las diferencias entre las componentes de los tres pares de píxeles que rodean al píxel tanto vertical como horizontalmente. El resultado de esta operatoria corresponde a la detección de bordes.

Reforzar brillo

Este filtro aumenta y disminuye el brillo de una imagen según el brillo que ya tenía. Se pasan dos umbrales por parámetro, si el píxel supera el umbral superior, el brillo se aumenta, si el píxel está debajo del umbral inferior, el brillo se disminuye. El efecto resultante es un refuerzo del brillo diferenciado.

Imagen original

Filtro fantasma

Color bordes

Reforzar brillo

2. Desarrrollo

En todas las implementaciones de los filtros de imagen, nuestra idea general fue realizar un algoritmo similar al presentado en lenguaje C, pero adaptándolo de modo que se obtenga el mayor nivel de paralelismo posible. Esto lo logramos a través del uso del modelo SIMD utilizado el set de instrucciones SSE - *Streaming SIMD Extensions*- y los registros XMM (128 bits) que nos permite operar con mayor cantidad de datos en simultáneo. Esto nos permite el procesamiento paralelo de los componentes que conforman al píxel, reduciendo así, la cantidad de instrucciones finales en cada filtro. Con esto en mente, nuestro principal objetivo e hipótesis, es que el uso de este modelo reduzca significativamente el tiempo de ejecución en comparación con la implementación de C.

A lo largo del informe vamos a referirnos a la resta entre píxeles como: $Px_a - Px_b$. Cuando así sea, nos estamos refiriendo a la resta uno a uno de sus componentes. Es decir, a restar las componentes Red, Green, Blue y Alpha (RGBA).

2.1. Imagen fantasma

2.1.1. Descripción/Pseudocódigo/Presentación del algoritmo

El siguiente pseudocódigo, presentado por la cátedra, describe el algoritmo:

```
Para j de 0 a height - 1:
  Para i de 0 a width - 1:

    ii = i/2 + offsetx;
    jj = j/2 + offsety;

    b = ( src_matrix[jj][ii].r + 2 * src_matrix[jj][ii].g + src_matrix[jj][ii].b ) / 4

    dst_matrix[j][i].r = SATURAR( src_matrix[j][i].r * 0.9 + b/2 )
    dst_matrix[j][i].g = SATURAR( src_matrix[j][i].g * 0.9 + b/2 )
    dst_matrix[j][i].b = SATURAR( src_matrix[j][i].b * 0.9 + b/2 )
```

2.1.2. Implementación en ASM

Este filtro toma como parámetro dos números enteros. Uno representando el offset vertical y otro el horizontal sobre los cuales vamos a aplicar el *ghosting*. A continuación hacemos una descripción del desarrollo del algoritmo implementado en lenguaje ensamblador:

- Para poder trabajar con 4 pixeles al mismo tiempo hay que tener en cuenta los distintos tamaños que utilizan las instrucciones que vamos a utilizar. Es por esto que decidimos hacer lo siguiente:
  - Levantar de memoria con la instrucción correspondiente al tamaño original de los pixeles con la instruccion MOVDQU.
  - Desempaquetar en 4 registros xmm donde cada byte fue extendido a dword (llenando con ceros los espacios).
  - Realizar la conversión de enteros de 32 bits a puntos flotante de 32 bits para, posteriormente, realizar las operaciones aritméticas de punto flotante.
- Lo siguiente fue hacer el calculo de las b que utilizaremos para modificar los valores de cada pixel de acuerdo a lo especificado en el pseudocódigo:

- Levantamos de memoria con la instrucción de desempaquetado, de manera que cada byte sea extendido a word, para facilitar el manejo de las próximas instrucciones.
- Con el uso de máscaras, filtramos los valores de los píxeles que no necesitamos para después hacer dos sumas horizontales que no permitan obtener los valores finales de b_0 y b_1.
- Por la especificación del algoritmo, que podemos observar en el pseudocódigo de arriba, calculamos la b utilizando el primer y segundo píxel. Llamamos entonces b_0 a la b que se utiliza para los primeros dos pixeles, y b_1 al que se utiliza para los otros dos.
- Extendemos el tamaño de los componentes de los píxeles de word a dword para poder hacer la conversión a float, ya que no existen puntos flotantes de tamaño menor a 32 bits.
- Para evitar realizar operaciones de más, preferimos optimizar el algoritmo haciendo directamente, por propiedades matemáticas, $b/8$ en lugar de $(b/4)/2$
- A continuación, convertimos los enteros en float y utilizamos un shuffle para replicar los valores de b en el orden deseado.
- Finalmente, utilizando un registro previamente llenado con una máscara que contiene dwords con el valor 8, dividimos las b

A continuación una representación que explica los pasos descritos:

XMM_2	B	2G	R	0	B	R	2G	0

Aplicando las suma horizontal PHADDSW nos termina quedando:

XMM_2	0	0	0	0	0	0	b_1	b_0

Con la operación PMOVZXWD extendemos los resultados de word a dword:

XMM_7	0	0	b_1	b_0

Luego utilizamos la operación CVTDQ2PS para transformar enteros en floats. Y con la operación SHUFPS ponemos los b_0 y b_1 donde corresponden:

XMM_7	b_1	b_1	b_0	b_0

Utilizando la operación DIVPS, pasando como origen un registro que contiene una mascara con dwords de valor 8, dividimos las 4 dwords:

XMM_7	$b_1/8$	$b_1/8$	$b_0/8$	$b_0/8$

- Ya con todas las b necesarias, pasamos a la etapa final del algoritmo donde realizamos las operaciones aritméticas para obtener el valor final de cada píxel.

  tem Primero creamos registros xmm con las máscaras necesarias para la multiplicación y, a su vez, cargamos otro registro xmm con la b correspondiente a cada píxel para, finalmente, realizar la multiplicación y suma.Después realizamos la conversión de punto flotante a entero, y empaquetamos con operaciones que simultáneamente saturan el número en caso de ser necesario. Por ultimo movemos los datos modificados a la memoria destino con la instrucción MOVUPS.

Representación que explica las cuentas realizadas en los últimos pasos del algoritmo en uno de los cuatro pixeles:

- Con la operación MULPS multiplicamos según la especificación dada en el pseudocódigo:

XMM_{12}	B * 0.9	G * 0.9	R * 0.9	A

Usamos un shuffle para extenders la b correspondiente:

XMM_8	$b_1/8$	$b_1/8$	$b_1/8$	0

Utilizando la operación ADDPS, sumamos los registros XMM9 y XMM8:

XMM_{12}	$b_1/8$ + B * 0.9	$b_1/8$ + G * 0.9	$b_0/8$ + R * 0.9	A

Tras empaquetar los pixeles, dejamos el registro XMM9 listo para mover a destino:

XMM_9	Px_3	Px_2	Px_1	Px_0

2.2. Color bordes

2.2.1. Descripción/Pseudocódigo/Presentación del algoritmo

El siguiente pseudocódigo, presentado por la cátedra, describe el algoritmo:

```
Para j de 1 a height - 1:
  Para i de 1 a width - 1:

    r=0 g=0 b=0

    Para jj de j-1 a j+1:
      r += abs( src_matrix[jj][i-1].r - src_matrix[jj][i+1].r )
      g += abs( src_matrix[jj][i-1].g - src_matrix[jj][i+1].g )
      b += abs( src_matrix[jj][i-1].b - src_matrix[jj][i+1].b )

    Para ii de i-1 a i+1:
      r += abs( src_matrix[j-1][ii].r - src_matrix[j+1][ii].r )
      g += abs( src_matrix[j-1][ii].g - src_matrix[j+1][ii].g )
      b += abs( src_matrix[j-1][ii].b - src_matrix[j+1][ii].b )


    dst_matrix[j][i].r = SATURAR(r)
    dst_matrix[j][i].g = SATURAR(g)
    dst_matrix[j][i].b = SATURAR(b)
```

2.2.2. Implementación en ASM

La idea del algoritmo es iterar la imagen analizando en cada iteración de a 4 pixeles en donde a cada uno se le aplica la operación especificada. Sí uno de los píxeles que estuviésemos analizando sería el Px_7, habría que realizar la siguiente operación al componente que corresponda:

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

Figura 1: Los pixeles resaltados son los trabajados en una iteración

- 1er Ciclo: $Px_7 = |Px_0 - Px_2| + |Px_6 - Px_8| + |Px_{12} - Px_{14}|$
- 2do Ciclo: $Px_7 += |Px_0 - Px_{12}| + |Px_1 - Px_{13}| + |Px_2 - Px_{14}|$

A continuación hacemos una breve descripción de un ciclo de nuestra implementación:

- Arrancamos limpiando los registros xmm que vamos a utilizar como acumuladores para almacenar los resultados parciales de las operaciones hechas sobre los píxeles.
- Dentro de una iteración, tenemos 2 ciclos internos. Uno para sumar las diferencias verticales y otro para las horizontales, tal como se muestra en la ecuación de arriba. Finalmente, dicha acumulación va a ser el píxel final.
- En el primero, se van calculando las sumas especificadas anteriormente. Al tener que sacar diferencias sobre 8 pixeles, notamos que dos pares, son iguales. Tomamos la decisión de almacenarlos en tres registros XMM. A continuación mostramos un ejemplo de cuales se comparten.
  - jj = j-1 $\rightarrow$ Px_7 y Px_9 comparten Px_2 . Px_8 y Px_{10} comparten Px_3
  - jj = j $\rightarrow$ Px_7 y Px_9 comparten Px_8 . Px_8 y Px_{10} comparten Px_9
  - jj = j+1 $\rightarrow$ Px_7 y Px_9 comparten Px_{14} . Px_8 y Px_{10} comparten Px_{15}
  - El próximo paso es realizar las operaciones de resta y el valor absoluto.

`Así quedarían los registros después de levantar de memoria:`

XMM$_2$	Px_1	Px_0
XMM$_3$	Px_3	Px_2
XMM$_4$	Px_5	Px_4

`Aplicando las PSUBW para la resta y los PABSW para el valor absoluto:`

XMM$_2$	$\|Px_1$ - $Px_3\|$	$\|Px_0$ - $Px_2\|$
XMM$_3$	$\|Px_3$ - $Px_5\|$	$\|Px_2$ - $Px_4\|$

  - El ciclo concluye después de realizar las sumas de los registros obtenidos en los acumuladores correspondientes.
  - Una vez finalizado el ciclo, ya tenemos los acumuladores con la sumatoria de las diferencias horizontales.
- Ahora pasamos al segundo ciclo para sumar en los acumuladores las diferencias restantes
  - Al igual que en el anterior, calculamos las diferencias faltantes. Para este ciclo vamos a tener que almacenar traer los 8 de los píxeles que los rodean y analizarlos. Estos almacenamos en 4 registros XMM , por cada iteración no se comparten pixeles como en el ciclo anterior.
  - Ya con los registros actualizados pasamos a realizar la una operatoria similar a la anterior:

Así quedarían los regitros despues de levantar de memoria:

XMM$_2$	Px_1	Px_0
XMM$_3$	Px_{13}	Px_{12}
XMM$_4$	Px_3	Px_2
XMM$_5$	Px_{15}	Px_{14}

Aplicando las PSUBW para la resta y los PABSW para el valor absoluto:

XMM$_2$	$\lvert Px_1 - Px_{13}\rvert$	$\lvert Px_0 - Px_{12}\rvert$
XMM$_4$	$\lvert Px_3 - Px_{15}\rvert$	$\lvert Px_2 - Px_{14}\rvert$

  - Otra vez, el ciclo concluye después de realizar las sumas de los registros obtenidos en los acumuladores correspondientes.

- Como no debemos modificar la transparencia y siempre es 255, seteamos la componente A de los pixeles en el mismo valor. Una vez hecho esto, movemos los datos modificados a la memoria destino con la instrucción MOVUPS.

- Una vez terminado el ciclo principal del algoritmo, armamos el marco blanco de 1 píxel de ancho en la imagen final. Para esto utilizamos dos registros XMM para almacenar una máscara horizontal y otra vertical. La horizontal contiene 4 pixeles con los valores (255,255,255,255) equivalentes al color blanco, mientras que la mascara vertical contiene lo mismo, pero únicamente en el primer píxel. El ciclo horizontal es trivial con la máscara blanca ya hecha, solo hay que recorrer la primera y última fila de la imagen y hacerle una suma saturada de la máscara con la instrucción PADDUSB. Ahora con el vertical , lo único que resta es calcular el salto necesario $((width - 1) * 4)$ para que con el primer píxel de la máscara poder pintar tanto el primer píxel de cada file, como el último.

2.3. Reforzar brillo

2.3.1. Descripción/Pseudocódigo/Presentación del algoritmo

El siguiente pseudocódigo, presentado por la cátedra, describe el algoritmo:

```
Para i de 0 a height - 1:
  Para j de 0 a width - 1:

    b = ( src_matrix[i][j].r + 2 * src_matrix[i][j].g + src_matrix[i][j].b ) / 4

    Si b > umbralSup:
      b = SATURAR(src_matrix[i][j].b+brilloSup)
      g = SATURAR(src_matrix[i][j].g+brilloSup)
      r = SATURAR(src_matrix[i][j].r+brilloSup)
    Sino, Si umbralInf > b:
      b = SATURAR(src_matrix[i][j].b-brilloInf)
      g = SATURAR(src_matrix[i][j].g-brilloInf)
      r = SATURAR(src_matrix[i][j].r-brilloInf)
    Sino:
      b = src_matrix[i][j].b
      g = src_matrix[i][j].g
      r = src_matrix[i][j].r

    dst_matrix[i][j].b = b
    dst_matrix[i][j].g = g
    dst_matrix[i][j].r = r
```

2.3.2. Implementación en ASM

Esta función recibe como parámetro de entrada cuatro números enteros. Dos constantes que funcionan como umbrales: $Umbral_{Sup}$ y $Umbral_{Inf}$. Y dos constantes, $BrilloSup$ y $BrilloInf$, que vamos a utilizar para aumentar o disminuir el brillo.

Para este filtro, el algoritmo implementado procesa 4 pixeles en cada iteración, donde cada uno de ellos puede caer en tres escenarios distintos:

1. Si el píxel supera el umbral superior, el brillo se aumenta.
2. Si está por debajo del umbral inferior, el brillo se disminuye.
3. Caso contrario, conserva el valor de sus componentes originales.

Ahora procedemos a detallar nuestra implementación:

- Para evitar una cantidad innecesaria de accesos a memoria, levantamos de la misma los 16 bytes en su tamaño original, y los desempaquetamos en otros 2 registros XMM para calcular cada b. Usamos la instrucción MOVDQU para levantar y PMOVXZBW (Byte a Word) para el desempaquetado.
- Con una operatoria similar a la de *Filtro fantasma*[1], obtenemos los valores de las b necesarias para la comparación con los umbrales. Con la diferencia que en este caso se trata de una b para cada píxel.

[1] pág 5 y 6

- Armamos un ciclo interno de 4 iteraciones, donde en cada uno compara cada b con los umbrales.
- Utilizamos dos registros para almacenar los valores de las constantes a sumar y restar ($BrilloSup$ y $BrilloInf$) según corresponda. Dando como resultado:
  - Si b_i, correspondiente a Px_i, supera el $umbral_{Sup}$, entonces cargamos con la operación BLENDVPS la Dword que contiene cuatro veces la constante $brillo_{Sup}$ en Bytes dentro del registro XMM9.
  - En caso que b_i sea menor al $umbral_{Inf}$, cargamos en XMM10 la Dword que contiene cuatro veces la constante $brillo_{Inf}$ en Bytes.
  - Sino, no modificamos ningún registro, conservando el valor original.

  Una representación de un ejemplo podría ser:

$$b_0 > umbral_{Sup}\ ,\quad umbral_{Inf} \leq b_1 \leq umbral_{Sup}\ ,\quad b_2 < umbral_{Inf}\ ,\quad b_3 > umbral_{Sup}$$

Registro previo a las operaciones de suma y resta(cada píxel esta dividido en 4 bytes que representan cada color):

XMM_2	Px_3	Px_2	Px_1	Px_0

Posible caso de registro a sumar:

XMM_9	$brilloSup$	0	0	$brilloSup$

Posible caso de registro a restar:

XMM_{10}	0	$brilloInf$	0	0

- Con el paso anterior finalizado, solo queda sumar y restar a los valores originales de cada píxel, los registros que se obtenieron en la etapa previa.

Este sería el resultado final de los 4 pixeles despues de aplicar el filtro:

XMM_2	$Px_3 + brilloSup$	$Px_2 - brilloInf$	Px_1	$Px_0 + brilloSup$

- Como no debemos modificar la transparencia y siempre es 255, seteamos la componente A de los pixeles en el mismo valor . Una vez hecho esto, movemos los datos modificados a la memoria destino con la instrucción MOVUPS.

3. Comparación

Nuestro objetivo a lo largo del trabajo, no solo fue aprender a usar el modelo SIMD. Si no también, verificar si realizar una implementación en lenguaje ensamblador utilizando estas instrucciones, mejoraban la performance en comparación con la implementación en lenguaje C, y en caso de que si, tener una idea en cada caso, de cuanto mejores son.

Para verificar esto, nos dispusimos a realizar las pruebas pertinentes para demostrarlo:

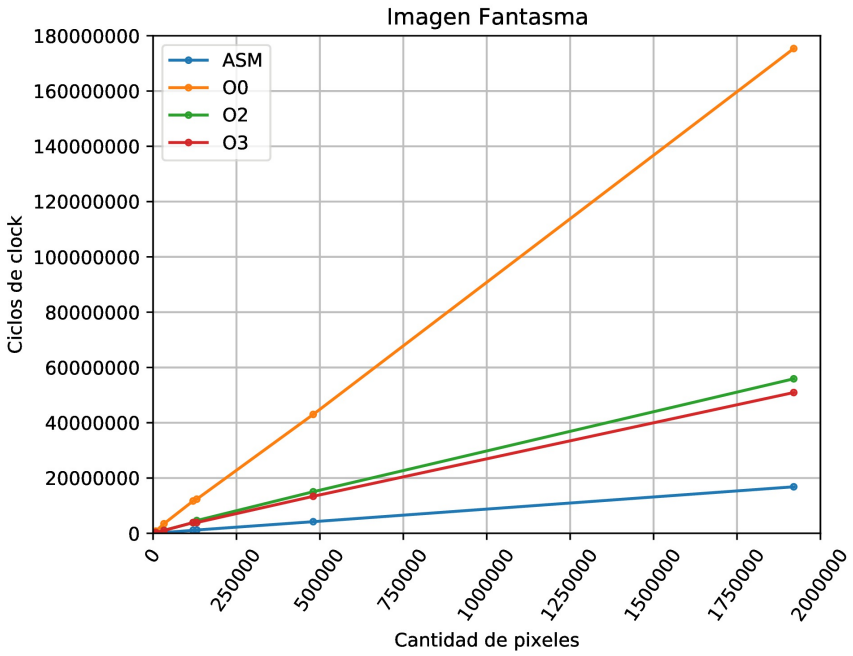
- Corrimos ambas implementaciones con las mismas imágenes reiteradas veces, en distintos tamaños y con las distintas optimizaciones de C.
- Quitando los outliers, calculamos la mediana con la cantidad de corridas para cada tamaño.
- Graficamos los tiempos ,en ciclos de clocks, en base al tamaño en pixeles, para cada implementación

Antes de pasar a ver cada filtro en particular, pasamos a comentar algunas aclaraciones sobre los criterios utilizados para las mediciones:

- El tiempo de ejecución de cada imagen se midió con la instrucción RDTSC, a través del macro de C provisto por la cátedra.
- Se utilizaron los siguientes tamaños de imágenes: 32x16, 64x32, 128x64, 256x128, 400x300, 512x256, 800x600, 1600x1200.
- La imagen utilizada fue SweetNovember.bmp, provista por la cátedra.
- Cada medición se realizó 300 veces, tomando como valor final la mediana, donde previamente se limpiaron los outliers, esto consideramos es la manera más representativa de mostrar los valores.
- Se utilizó siempre la misma computadora equipada con un Procesador AMD® Ryzen 5 2600, 32 GB DDR4 de memoria Ram y Pop!_OS 20.04 LTS para evitar diferencias de hardware y software.
- Los datos obtenidos fueron almacenados en formato csv y los mismos fueron analizados y graficados utilizando las bibliotecas de Python Pandas y Matplotlib.

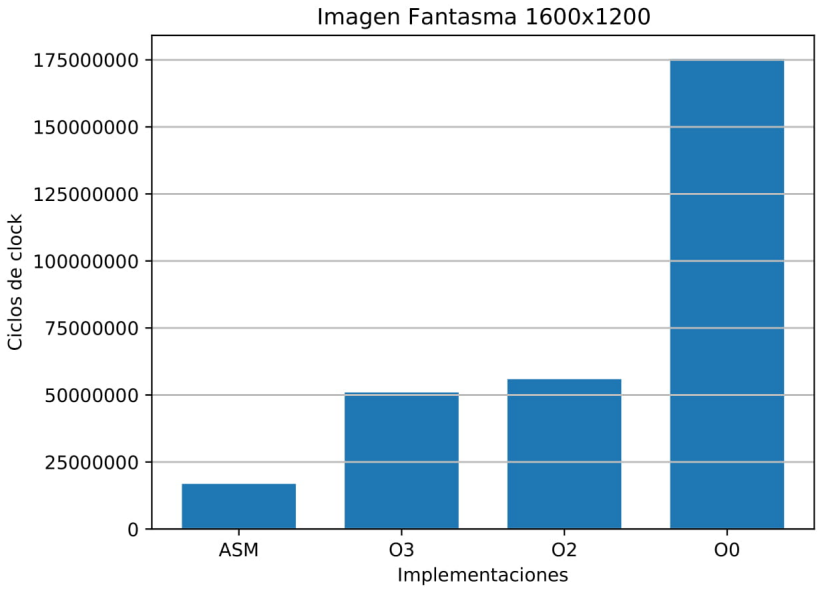
A continuación analizamos los distintos gráficos obtenidos de las mediciones previamente mencionadas, para cada uno de los filtros.

3.1. Imagen fantasma



Lo primero que se aprecia en este gráfico es que, más allá de la implementación utilizada, el tiempo de ejecución del filtro aumenta linealmente con respecto al tamaño de la imagen. Al mirar el mismo, es evidente la diferencia de performance entre el algoritmo escrito en ASM y el algoritmo escrito en C. Al compilar con distintas optimizaciones podemos ver que la brecha de performance se reduce significativamente. Creemos que esto se debe a un uso más exhaustivo de los registros en lugar de acceder constantemente a memoria. Sin embargo, cabe señalar que el algoritmo escrito en ASM con el modelo SIMD sigue siendo el de mayor rendimiento.

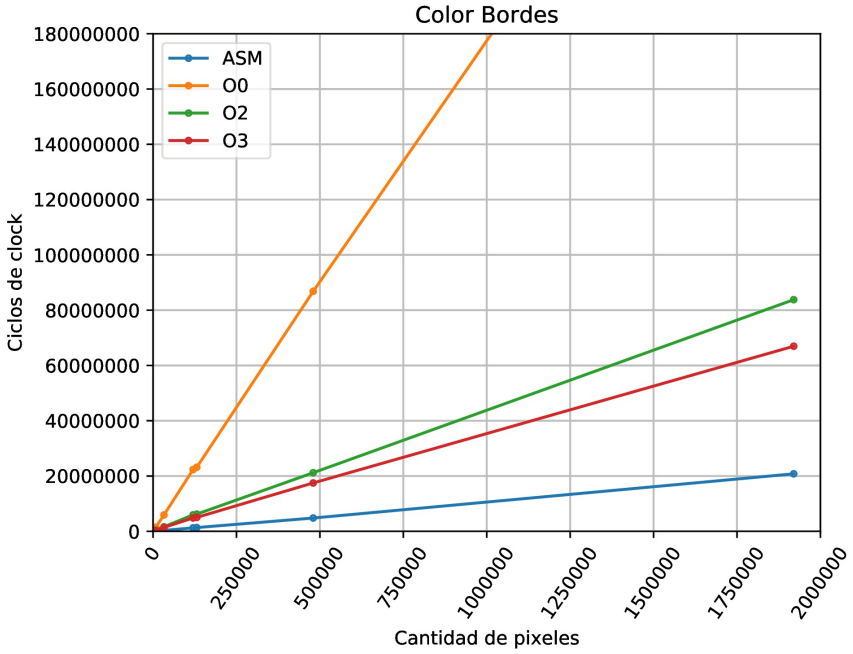
Para entender que tan grande puede ser la diferencia entre la implementación en ASM y las distintas optimizaciones de C, lo que hicimos fue un gráfico donde se vean la cantidad de ciclos de clock contra cada implementación. Elegimos el mayor tamaño, es decir, una imagen de 1600x1200 para de esta manera, observar la diferencia con mayor claridad:



Como podemos observar, la implementación en ASM se ejecuta en un $9\,\%$ del tiempo con respecto a la implementación en C compilado con el flag -O0.

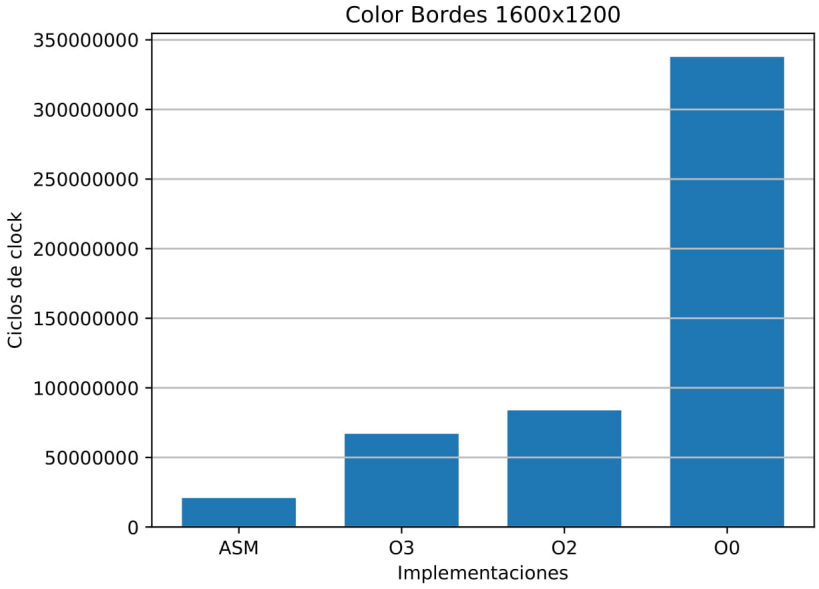
Por otro lado, podemos ver que al ejecutarlo con el flag -O2 u -O3, se reduce hasta un $250\,\%$ la cantidad de ciclos de clock con respecto a su implementación sin optimizar. Sin embargo, con estas optimizaciones, el tiempo de ejecución en ASM es entre $30\,\%$ y $33\,\%$ respecto a las implementaciones optimizadas.

3.2. Color Bordes



En este gráfico, como en el del filtro anterior, vemos que el tiempo de ejecución es lineal y que a mayor tamaño de imagen, mayor será la diferencia de performance entre las implementaciones y optimizaciones.

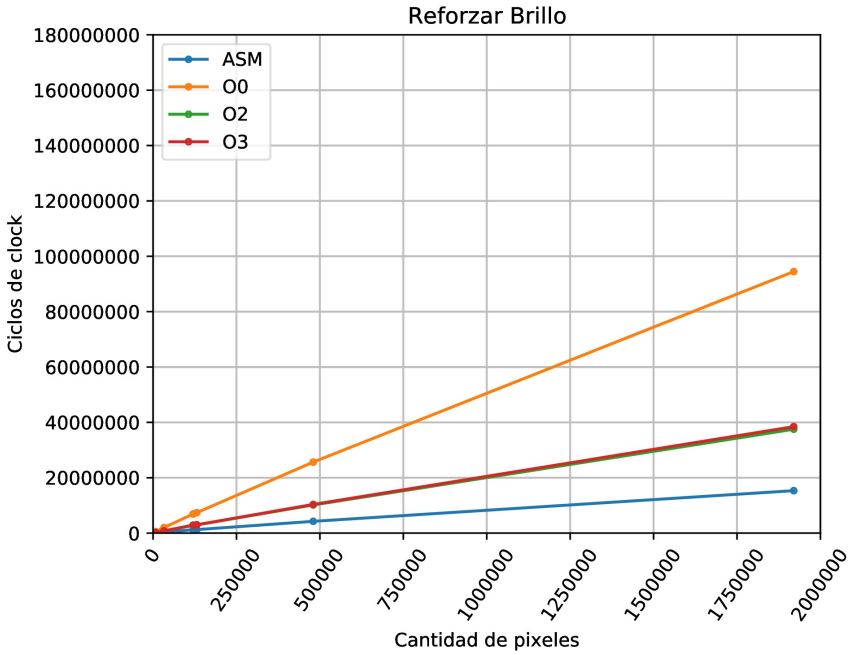
Algo que notamos, es que la diferencia está mucho más marcada en este filtro. Lo cual tiene sentido, teniendo en cuenta que, por lo visto en el pseudocódigo de C y contraponiéndolo con los otros dos pseudocódigos, opera más veces los distintos píxeles de la imagen.



Como podemos observar en nuestras mediciones, la implementación en ASM se ejecuta en un $6\,\%$ del tiempo en el que se ejecuta la implementación en C compilado con el flag -O0. Al ejecutar el código en C con el flag -O2 u -O3, el margen con respecto a la implementación en ASM es considerablemente menor, obteniendo con estas optimizaciones, entre $300\,\%$ y $400\,\%$ veces menos ciclos de clock con respecto a la implementación en C previamente analizada.

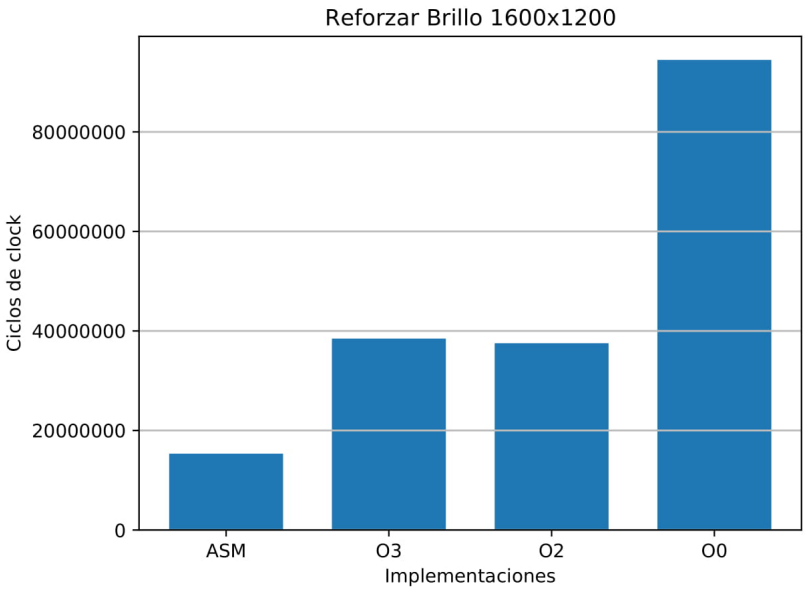
De todas formas, la implementación en lenguaje ensamblador continúa siendo superior a estas. Siendo entre un $24\,\%$ y $31\,\%$ más veloz que las optimizaciones O2 y O3.

3.3. Reforzar brillo



En este gráfico, podemos mantener las mismas primeras conclusiones postuladas en los filtros precedentes. Es decir, los ciclos de clock aumentan linealmente con respecto al tamaño, y la diferencia de ciclos entre las distintas implementaciones es más cuantioso a mayor tamaño de imagen.

Sin embargo, en este filtro, podemos observar que la diferencia entre implementaciones no es tan considerable, sobre todo cuando las implementaciones en C son optimizadas con los flags -O2 y -O3.



Como podemos observar en los gráficos, la implementación en ASM se ejecuta en un $15\,\%$ del tiempo que la implementación en C sin optimizaciones. Otra cosa que podemos notar es que, en este filtro, la implementación que fue compilada con el flag -O2 fue levemente más veloz que la compilada con flag -O3. De todas formas, la implementación en lenguaje ensamblador continúa siendo superior a estas. Siendo entre un $37\,\%$ y $40\,\%$ más veloz que las optimizaciones O2 y O3.

4. Experimentos y resultados

En la etapa de implementación de los algoritmos, fuimos modificando el código de los distintos filtros para conseguir una implementación que creemos más performante. Ahora vamos a realizar experimentos para validar si realmente, con esos cambios, mejoramos o no la performance, y analizar que tanto mejor o peor son con respecto a nuestra implementación final.

Cabe aclarar, que para esta etapa, ajustamos las implementaciones previas, con el fin de asimilarlas lo más posible a nuestra implementación final. Tratando de reducir las diferencias de performance proveniente de otras instrucciones.

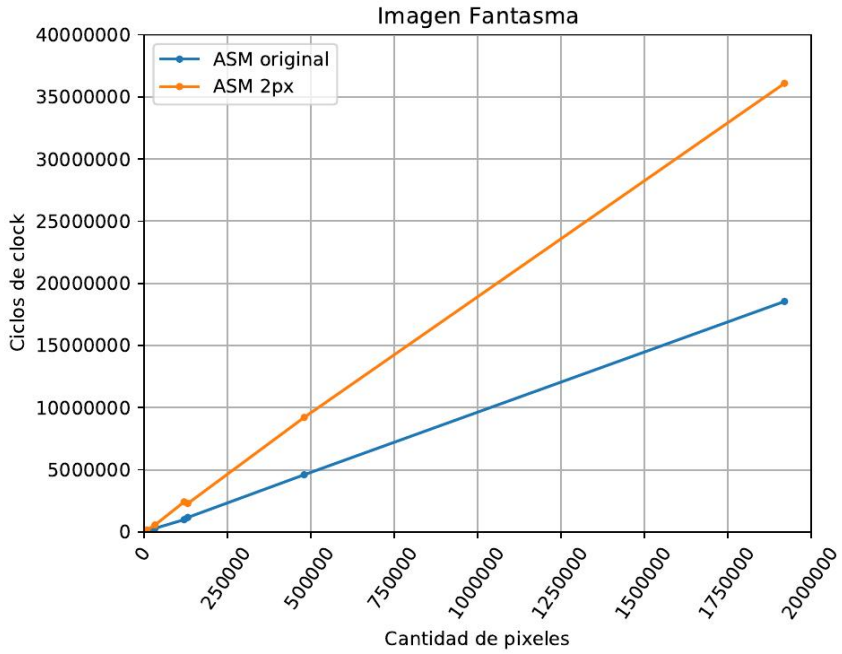
4.1. Operar con 4 pixeles en simultáneo, es más eficiente que operar con 2

Nuestra primera experimentación es el análisis de la implementación de 4 pixeles con respecto a una similar, pero se trabaje con 2 pixeles en simultáneo. En nuestra opinión, usar 4 pixeles tendría que ser considerablemente más performante, ya que nos permite un mejor aprovechamiento del modelo SIMD en varias instrucciones y acceder menos veces a memoria.

Para esto utilizamos nuestra primera implementación del filtro fantasma que recorre la imagen operando de a 2 pixeles y comparamos los tiempos con la implementación vista anteriormente [2], la cual trabaja con 4 pixeles por iteración.

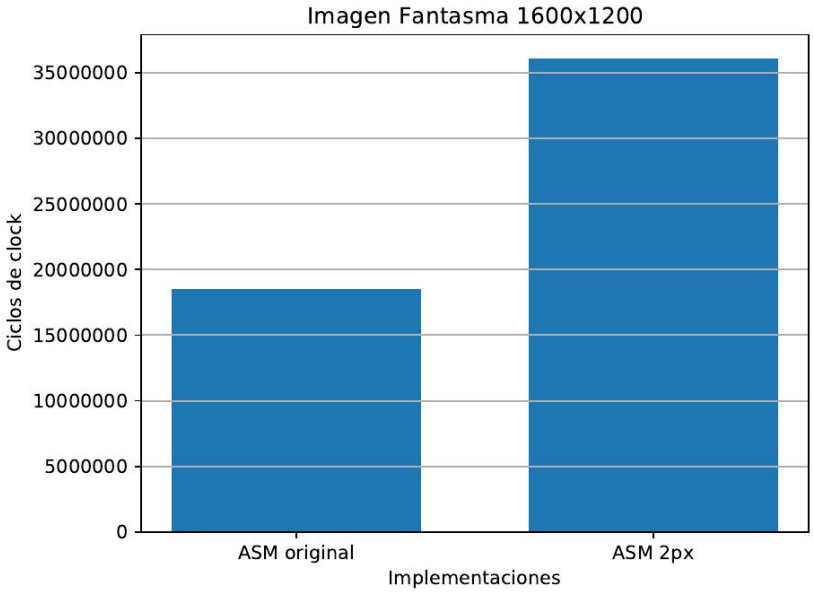
Los resultados del experimento fueron más favorables de lo que esperábamos. Al comparar los resultados vemos que la implementación con 4 pixeles se ejecuta aproximadamente en el $50\,\%$ del tiempo respecto de la implementación que opera de a 2 pixeles.

A continuación los gráficos que ilustran esta diferencia.

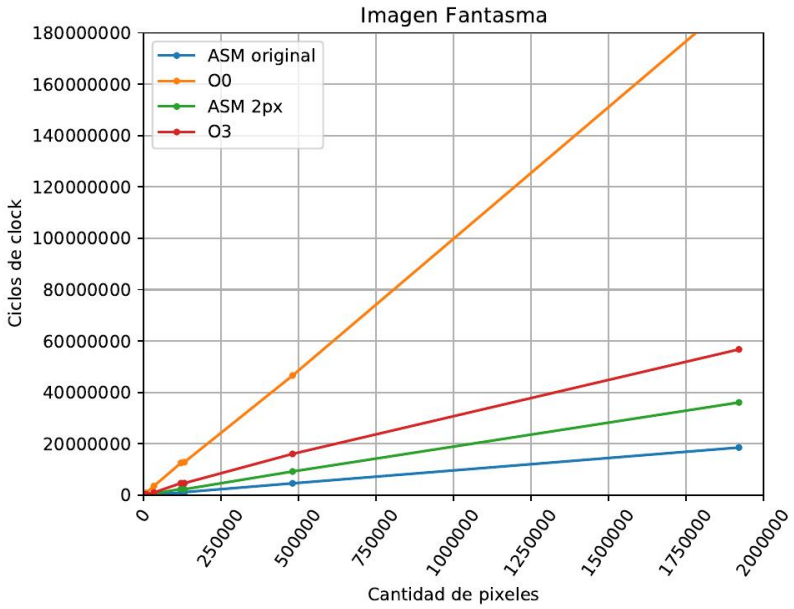


En este segundo gráfico, nos centramos en la diferencia de clocks en la foto de mayor tamaño provista por la cátedra (1600x1200).

[2] Pág. 5



También los comparamos con las implementaciones de C con flag el -O0 y -O3 para ponerlo en contexto.



Vemos que la implementación en lenguaje ensamblador continúa siendo superior con al menos operar con 2 pixeles en simultáneo, sin embargo, su eficiencia medida en cantidad de ticks está en medio de la implementación de 4 píxeles de ASM y la optimización 3 de la implementación en C.

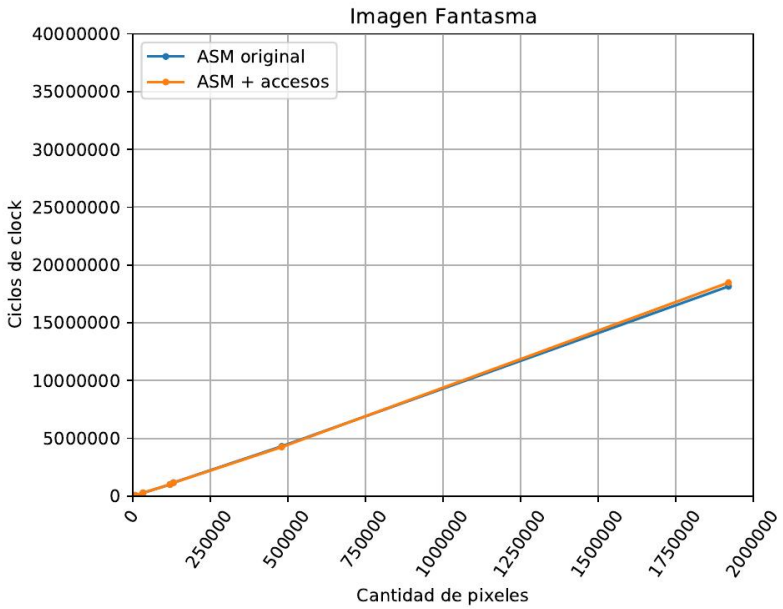
4.2. Reducir cantidad de accesos a memoria tiene impacto considerable en la performance

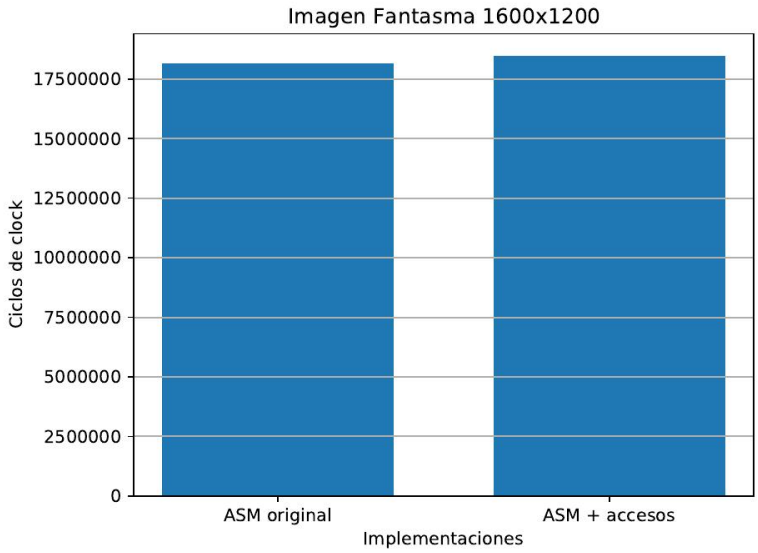
Nuestra hipótesis fue la siguiente: si reducimos la cantidad de accesos a memoria, el algoritmo sería más performante. Para verificar si nuestra idea era correcta, modificamos el código en el filtro Imagen Fantasma.

En la primera implementación cargábamos de memoria, de imagen fuente, 4 pixeles con dos accesos. Movíamos a un registro XMM, 2 pixeles expandiéndolos, y hacíamos lo mismo en otro registro con los otros 2 pixeles. A esta implementación, la vamos a llamar .ASM + accesos”

Al saber que los accesos a memoria son costosos, en términos de tiempo, modificamos el código para traer los 4 pixeles a un único registro XMM, los movemos y extendemos como correspondan. De esta manera, reducimos un acceso a memoria innecesario por cada iteración. Con esto, esperamos que tenga una mejora significativa de performance.

A continuación los resultados:





Por lo que vemos, los resultados son prácticamente iguales, se ve una diferencia a favor de la implementación con menos accesos, pero es prácticamente imperceptible y para nada significativa. Por lo cual, en este caso, no se cumple nuestra hipótesis, lo cual no la descarta que se cumpla para otros casos.

Creemos que esto puede deberse al uso de la memoria cache, la cual no tuvimos en cuenta al momento de plantear la hipótesis. Al recorrer nuestra imagen de manera contigua en memoria, es muy probable, que estas posiciones de memoria solicitadas innecesariamente, ya estén en la memoria cache. Por lo cual el costo el costo de acceder a esa información es bajo.

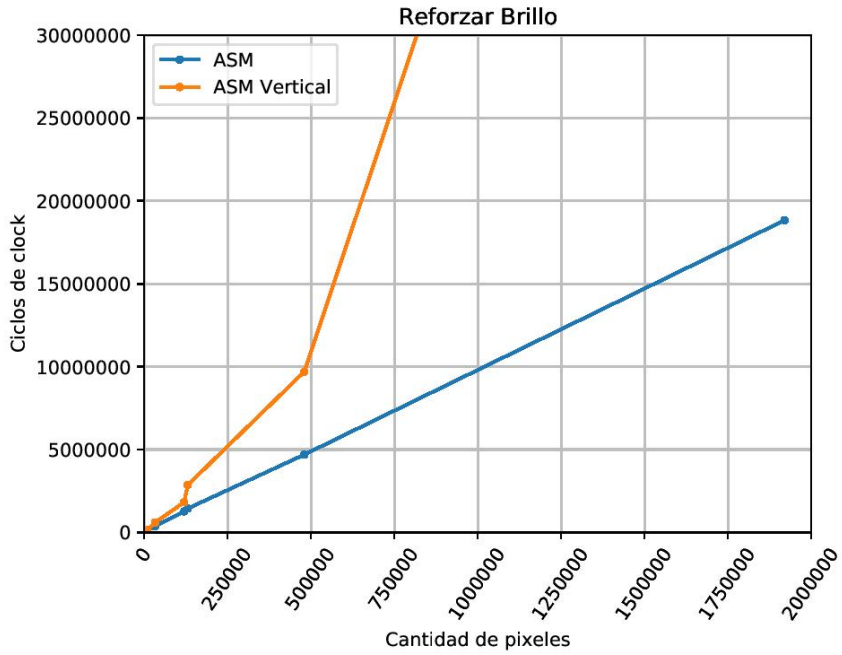
Esto nos llevó a querer realizar otro experimento intentando evitar el uso de la memoria caché. Con el fin de analizarla más en profundidad.

4.3. Recorrer la imagen por fila es más performante que recorrerla por columna

Nuestro tercer y último experimento es el análisis de dos implementaciones iguales con la distinción que una recorre los píxeles de la imagen horizontalmente, mientras que la otra los recorre verticalmente. La que recorre de manera horizontal es nuestra implementación original del filtro reforzar brillo [3], y la vertical es una nueva versión que implementamos, la vamos a llamar "ASM Vertical".

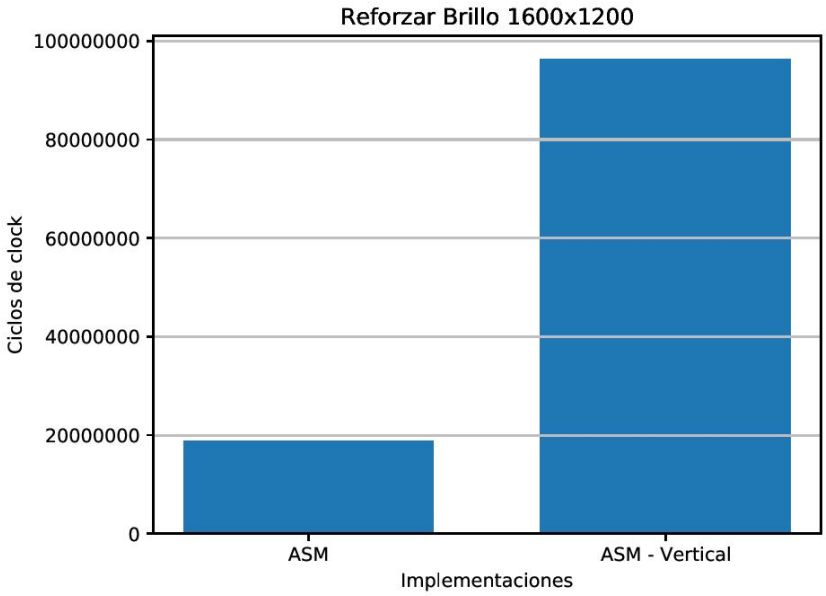
Nuestra hipótesis propone que al operar los píxeles en columnas, dará como resultado una bastante peor implementación, en términos de tiempos, con respecto a la que operamos horizontalmente. Planteamos que esto se debe al uso de la memoria cache, ya que por el principio de vecindad, esta almacena copias de páginas de la memoria principal. Cuando el procesador no encuentra la posición de memoria requerida en la caché, lo solicita de la memoria principal y se guarda una copia de la página que contiene la posición solicitada.
Es por esto que al recorrer horizontalmente, es decir, contiguamente en memoria, se espera un *hit rate* alto, por ende, un óptimo uso de la caché.
Por otro lado, en la versión ASM Vertical, en imágenes de gran tamaño, su *hit rate* debe ser bastante menor, ya que al no recorrer la imagen de manera contigua en memoria, los datos están bastante distanciados de los solicitados anteriormente, por lo que debe haber muchos más *miss* de caché. Esto concluye en tener que recurrir a traer dicha posición de la memoria, lo cual es costoso en tiempo. Por lo tanto, el procesador termina aumentando considerablemente la cantidad de *wait states*.

A continuación los resultados que obtuvimos:



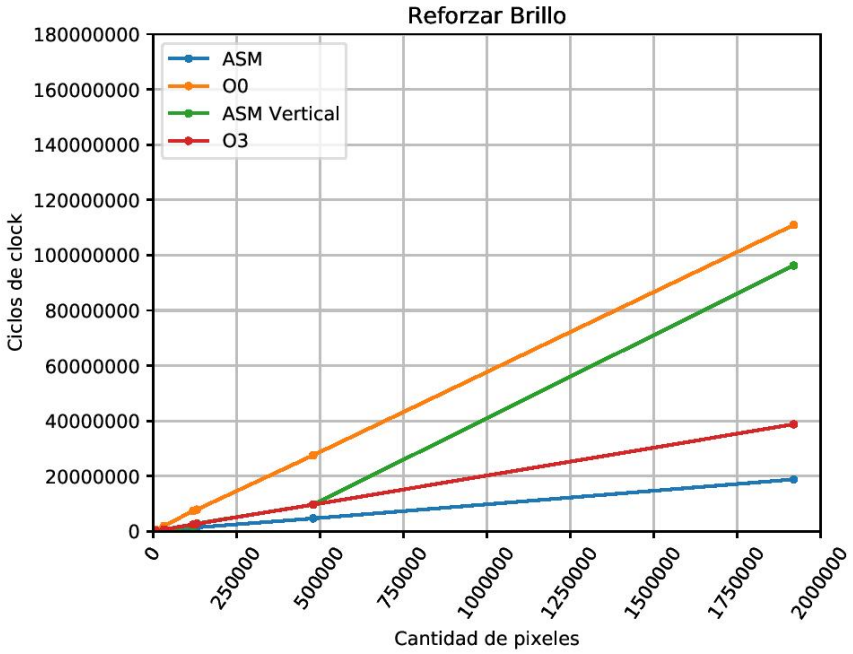
Como nos esperábamos, la diferencia entre las implementaciones es muy grande. Para visualizar que tanta es la diferencia, hicimos un gráfico en el que podemos analizar la performance para un tamaño específico, en este caso el mayor tamaño de las imágenes provista por la cátedra, así de esta manera la diferencia es más notoria.

[3] Pág. 10



Como se puede ver en el gráfico, para este tamaño de imagen, la implementación que recorre horizontalmente, se ejecuta aproximadamente en un $20\,\%$ del tiempo respecto a la que recorre horizontalmente.

Finalmente, para ponerlo en contexto, graficamos estas implementaciones junto con la implementación en lenguaje C con los flags -O0 y -O3.



Viendo esto vemos que la implementación que recorre de a columnas la imagen es aún peor que la implementación en C con optimización -O3. Con esto concluimos que la implementación en lenguaje ensamblador que recorre filas es ampliamente superior a la implementación que recorre por columnas y, además, nos muestra la importancia de la memoria cache y cuanto se optimiza la performance de un filtro con cada *hit*.

5. Conclusión

A lo largo del trabajo nos planteamos si la implementación en ASM de distintos filtros aplicados a imágenes en formato bmp lograría disminuir tanto el uso de los recursos del sistema como los tiempos de ejecución en contraposición con su implementación en C. Podemos concluir, por todo lo expuesto anteriormente, que existe una gran diferencia de tiempo y uso de recursos del sistema entre la implementación en lenguaje ensamblador utilizando el modelo SIMD en contra parte con la implementación en C. Más precisamente, vimos que la ejecución de los filtros implementados en ASM, con un uso adecuado de la memoria cache se ejecutan en entre el $6\,\%$ y $33\,\%$ del tiempo con respecto a las implementaciones en C.

Por todo lo dicho, concluimos que, independientemente del poder de la computadora, SIMD es una herramienta clave cuando se quiere tratar con imágenes y conseguir tiempos de ejecución considerablemente mayores a lo que una implementación tradicional en lenguajes de más alto nivel nos pueden brindar.