DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Informe

Organización del Computador II

Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Tomas Curti	327/19	tomasacurti@gmail.com
Guido Rodriguez Celma	374/19	guido.rc98@gmail.com
Fermin Schlottmann	160/19	fschlottmann@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

El presente informe se enmarca dentro del segundo trabajo práctico de la materia Organización del Computador II. La consigna de este último consistió en el estudio del modelo de procesamiento SIMD, la implementación de filtros visuales haciendo uso de este modelo y el análisis empírico del funcionamiento de éstos.

El trabajo se encuentra dividido en tres partes: en la primera sección presentamos los distintos filtros gráficos con los que trabajamos. Discutimos sus algoritmos en líneas generales para luego adentrarnos en su implementación a bajo nivel; presentamos variantes y alternativas a estos que retomamos en secciones posteriores. La caracterización de algunas instrucciones de procesamiento SIMD y el análisis de posibles aplicaciones en una series de algoritmos, puede ser de especial interés para aquellos interesados en el estudio del uso de operaciones vectoriales a bajo nivel. Adicionalmente, personas interesadas en el uso y experimentación de filtros gráficos pueden encontrar particular valor en este informe; anexo a este documento se incluyen archivos fuente con distintas implementaciones de los filtros presentados.

Esto nos lleva a la segunda parte del trabajo, donde discutimos algunas características de las implementaciones de los filtros en dos lenguajes de programación: lenguaje ensamblador y C. Se los compara de manera empírica, teniendo en cuenta diversas versiones de optimización provistas por el compilador. Haciendo uso de estas implementaciones, en la tercera sección llevamos a cabo una serie de experimentos centrados en el análisis empírico del funcionamiento de los filtros. Identificamos dos dimensiones principales, cantidad de accesos a memoria y cantidad de saltos condicionales; y experimentamos con cada una en secciones separadas, observando qué efecto tienen sobre el desempeño de los programas. Con estos resultados nos acercamos a la versión más óptima de cada filtro y establecemos una jerarquía entre las diversas alternativas de sus implementaciones. Esta última parte puede ser de utilidad tanto para usuarios como desarrolladores de filtros gráficos, ya sea por la disponibilidad de una implementación más eficiente y estudiada, como por el análisis de algunos factores que muestran un impacto observable en la performance de éstas. Finalmente, se concluye en la sección 5.

2. Desarrollo

Imagen Fantasma:
Funcionamiento general:

La idea de este filtro es superponerle a la imagen original una versión en escala de grises y del doble de tamaño de la misma, opcionalmente desplazada en el eje X o Y. Para esto se multiplica cada píxel por un escalar y se le suma un coeficiente "b".
Desarrollo de un ciclo del programa:

Primero vamos a calcular el coeficiente "b" para cada píxel, para esto traemos de memoria al registro XMM0 4 píxeles de la imagen original, según los índices "ii" y "jj" que calculamos previamente. Aquí reemplazamos la componente de transparencia "A" por la componente "G" en cada píxel usando inserciones (ver figura 1).

Luego, convertimos las componentes de cada píxel de Byte a Word, dejando los primeros 2 píxeles en el registro XMM1 y los últimos 2 en XMM2, esto lo hacemos para realizar sumas horizontales sin riesgo de tener overflow. Después de dichas sumas, obtenemos en la parte baja de XMM1 los resultados: Ri + 2*Gi + Bi (con i $\in$ { 0,1,2,3 }) (ver figura 2).

Convertimos estos 4 resultados de Words a Double Words y de Double Words a Packed Singles tal que los resultados anteriores ocupen todo el registro XMM1, finalmente dividimos empaquetadamente XMM1 por 8 para obtener el coeficiente "b" correspondiente a cada píxel (la conversión a Singles la hacemos para no perder precisión en esta división).

Ahora buscamos de memoria los píxeles que vamos a mover a la imagen destino. Notar que vamos a recorrer la imagen fuente de a 8 píxeles por iteración pues los coeficientes "b" son los mismos para cada par de píxeles contiguos (considerando la imagen como una tira de píxeles).

Así traemos de memoria al registro XMM12 4 píxeles de la imagen original según el número de iteración, guardamos cada píxel en un registro XMM auxiliar, convirtiendo sus componentes de Bytes a Double Words y de Double Words a Singles. Luego de cargar cada píxel shifteamos el registro XMM12 4

bytes hacia la derecha para tener en la parte baja el próximo píxel a convertir (ver figura 3).

Luego, empaquetadamente, multiplicamos por el escalar $0,9$ (el cual previamente cargamos empaquetadamente como Singles en un registro XMM auxiliar) y sumamos la correspondiente constante "b", a la que previamente hicimos broadcasting en un registro XMM auxiliar usando un shuffle empaquetado tal que se sume a todas las componentes del píxel. (ver figura 4)

Finalmente para cada píxel convertimos sus componentes de Singles a Double Words y empaquetamos saturadamente los 4 píxeles resultado (de DWords a Words y de Words a Bytes) en un registro XMM, el cual movemos a memoria según corresponda su lugar en la imagen destino. Repetimos el proceso para los 4 píxeles siguientes en la imagen fuente y concluimos esta iteración del ciclo principal.

Imagen Fantasma Figura 1:

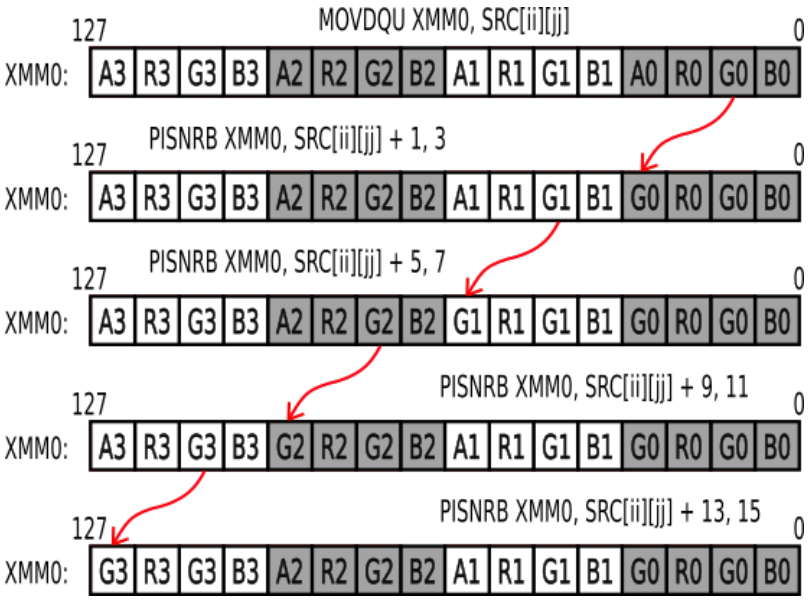


Imagen Fantasma Figura 2:

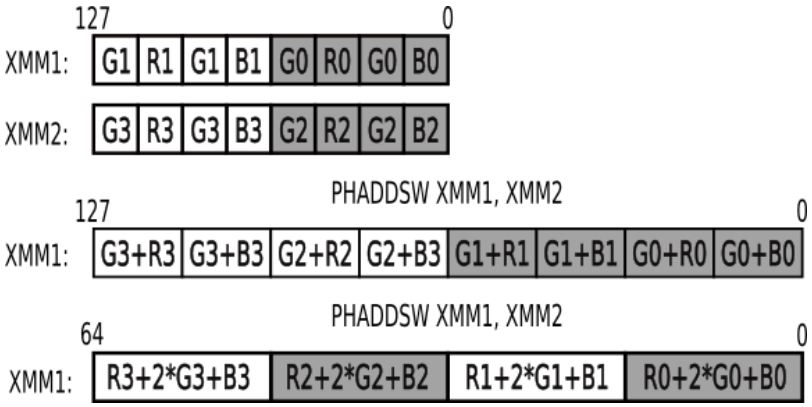


Imagen Fantasma Figura 3:

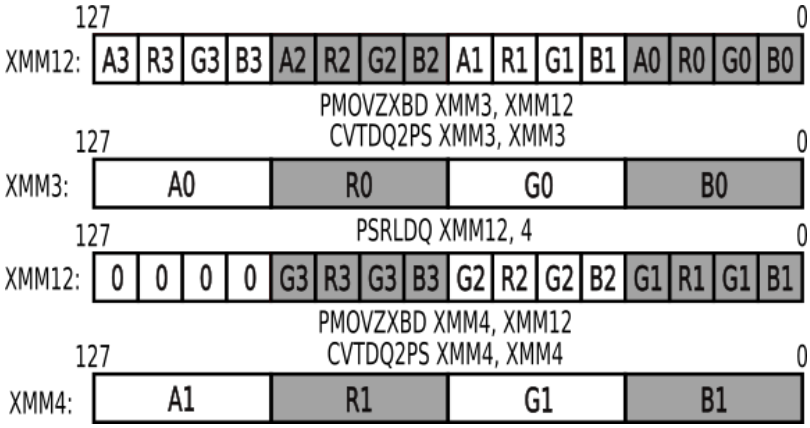
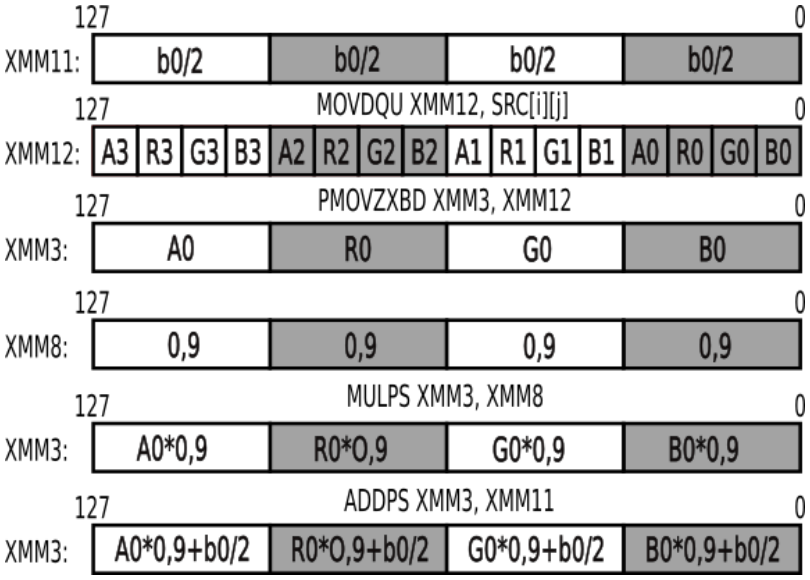


Imagen Fantasma Figura 4:



Color Bordes:

Funcionamiento general:

La idea de este filtro es calcular, para cada píxel de la imagen, la diferencia absoluta de sus 8 píxeles lindantes. Luego se suman estas diferencias y se reemplaza el píxel por el resultado obtenido. El efecto de esta operatoria es que en la imagen obtenida, los píxeles que representan un borde resaltan sobre el resto de píxeles, que quedan más apagados. Además, los márgenes de la imagen deben quedar en blanco.
Desarrollo de un ciclo del programa:

Primero recorremos los márgenes superior e inferior de la imagen con un ciclo propio, seteando en blanco en la imagen destino, todos los píxeles que pertenezcan a los mismos. No incluimos los márgenes izquierdo y derecho porque los trataremos en el ciclo principal (ver figura 1).

Comenzamos el ciclo principal: Vamos a recorrer la imagen de a dos píxeles por iteración, convirtiendo sus componentes de Bytes a Words. Para estos dos, conseguimos sus píxeles lindantes y los almacenamos en registros XMM. Primero conseguimos los pixeles de arriba a la izquierda y arriba a la derecha y guardamos su diferencia absoluta en XMM0, a partir de ahora, XMM0 va a ser el registro donde almacenemos las sucesivas sumas (ver figura 2).

No tendremos que hacer consideraciones por posibles overflows ya que convertimos cada componente de los píxeles de Byte a Word, añadiendo 0s adelante. Como únicamente realizamos sumas de Bytes, necesitaremos a lo sumo un bit más para guardar el resultado.

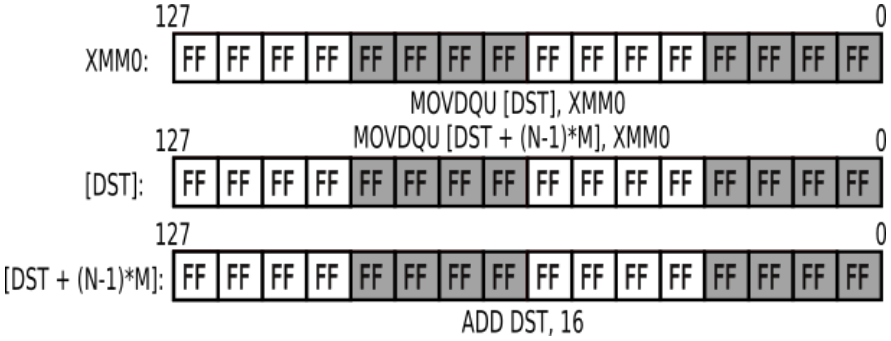
Continuamos calculando las diferencias de los píxeles restantes en registros XMM auxiliares y sumándoselas a XMM0. Cuando terminamos las operaciones con los píxeles lindantes, empaquetamos el registro XMM0 de Words a Bytes, seteamos su componente transparencia en 255 y movemos los 2 píxeles alma-

cenados en su mitad menos significativa a la imagen destino (ver figura 3).

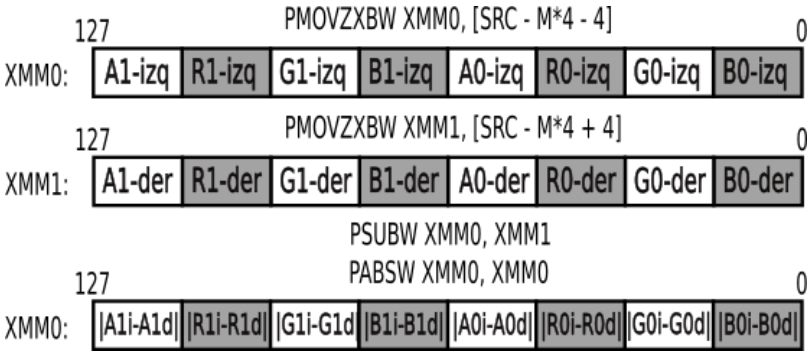
Al llegar al final de cada fila de la imagen (considerada como una matriz de píxeles), seteamos en blanco los pixeles correspondientes a los márgenes izquierdo y derecho de la imagen destino.

Finalizamos la iteración avanzando en 2 píxeles (8 Bytes) los punteros a las imágenes destino y fuente.

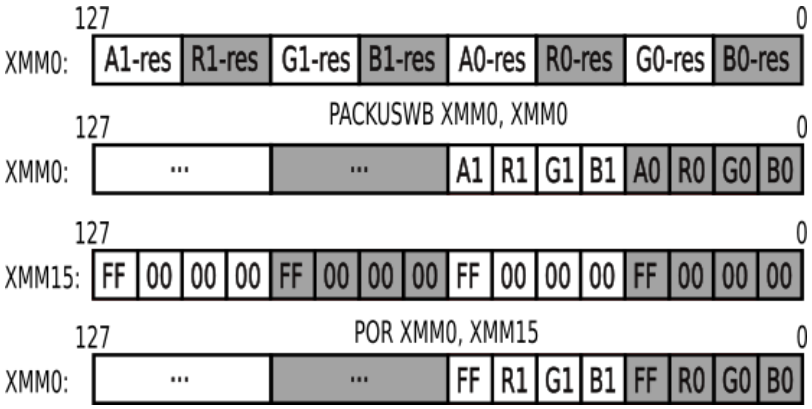
Color Bordes Figura 1:



Color Bordes Figura 2:



Color Bordes Figura 3:



Reforzar Brillo:

Funcionamiento general:

La idea de este filtro es aumentar o disminuir el brillo de cada píxel de la imagen. El criterio para determinar si un píxel debe ser modificado está dado por un coeficiente "b" calculado para cada píxel individualmente. Si el coeficiente supera un umbral superior, se aumenta el brillo del píxel en un valor dado. A su vez, si está por debajo de un umbral inferior, se disminuye el mismo.

Desarrollo de un ciclo principal:

Vamos a considerar la imagen como un vector de píxeles, y la recorreremos de a 4 posiciones por iteración.

Primero calculamos los coeficientes "b" para cada uno de los 4 píxeles que vamos a procesar. Para esto, conseguimos de memoria 4 píxeles según la posición actual en la imagen y los guardamos en el registro XMM0, aplicando una máscara para anular las transparencias. Luego extendemos las componentes de estos píxeles de Bytes a Words, almacenando los primeros dos píxeles en XMM1 y los últimos dos en XMM2 (ver figura 1).

En un registro auxiliar XMM3 guardamos únicamente la componente "G" de los 4 pixeles extendidos a Words (de a 2 por vez), y lo sumamos empaquetadamente con los registros XMM1 y XMM2 respectivamente.

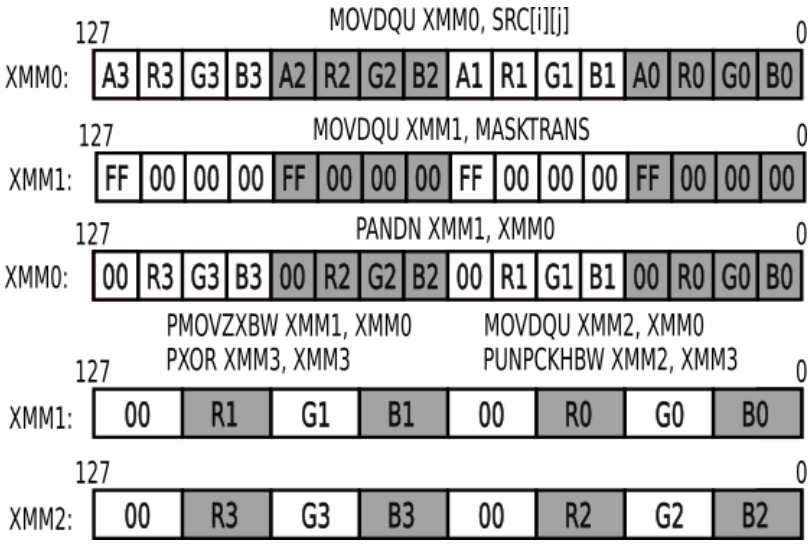
Finalmente juntamos los 4 píxeles en un único registro XMM2 mediante dos sumas horizontales y dividimos empaquetadamente por 4 para obtener los coeficientes "b" en la parte baja del registro. Extendemos estos coeficientes de Words a Double Words pues los úmbrales son enteros de 32bits (ver figura 2).

Ahora comparamos los coeficientes con los umbrales para obtener dos máscaras que nos indiquen qué píxeles superan o están por debajo de los mismos, y por lo tanto cuales debemos modificar.

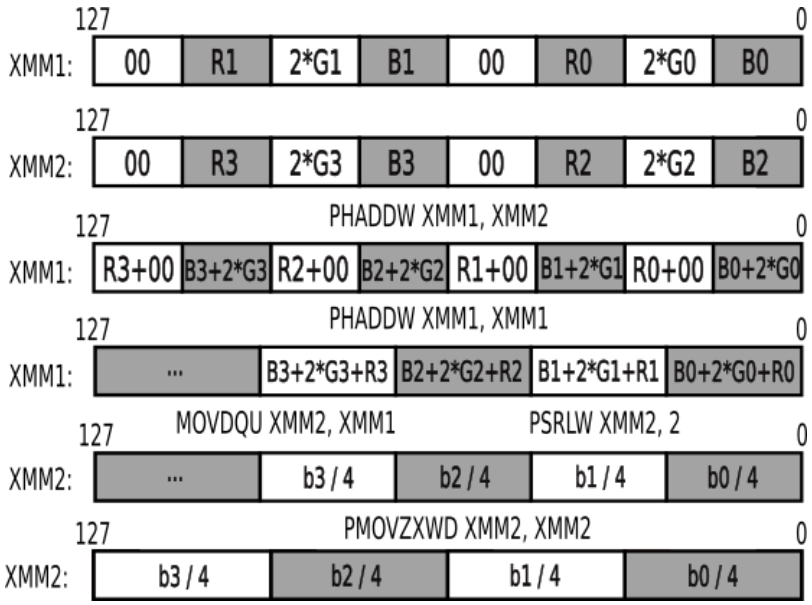
Movemos a un registro auxiliares XMM1 los valores de incremento y decremento que nos dan por parámetro. Les aplicamos las respectivas máscaras que obtuvimos en el paso anterior y sumamos o restamos el resultado, según corresponda, al registro XMM0 que contiene los píxeles originales (ver figura 3).

Para concluir la iteración restauramos las transparencias, movemos los 4 píxeles resultado de XMM0 a la imagen destino y avanzamos 4 posiciones en los vectores destino y fuente.

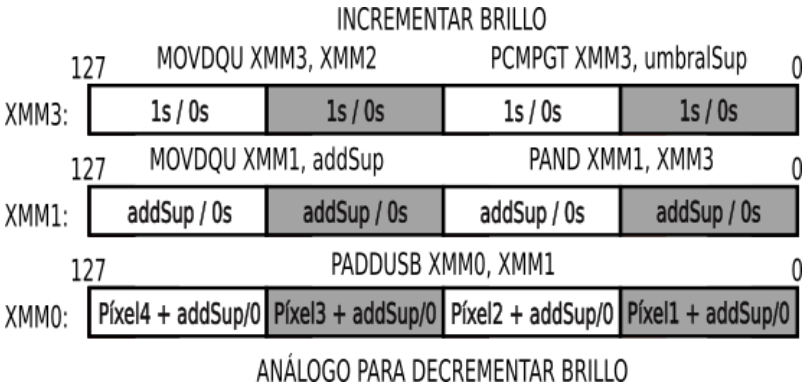
Reforzar Brillo Figura 1:



Reforzar Brillo Figura 2:



Reforzar Brillo Figura 3:



3. Comparación

Como mencionamos previamente, contamos con versiones de los filtros presentados en dos lenguajes de programación: assembler y C. En esta sección comparamos las implementaciones en estos dos lenguajes prestando atención a sus respectivos códigos, a su rendimiento y a las imágenes resultantes.

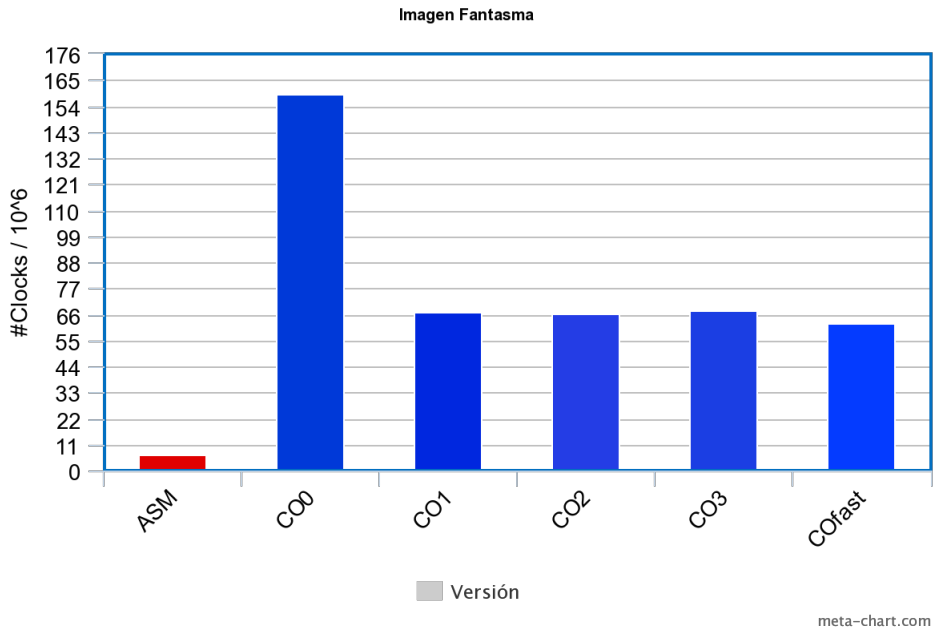
En primer lugar, observamos algunas particularidades de cada lenguaje y como estas se presentan en los programas adjuntos. Algo apreciable del código en C es la semejanza que posee con la idea intuitiva que uno tiene de los algoritmos asociados a cada filtro. Si tomamos -por ejemplo- el filtro Reforzar Brillo, vemos que su implementación es casi idéntica al pseudocódigo correspondiente: para cada píxel, calculamos el valor b asociado a éste y luego lo pasamos por una estructura condicional para decidir qué escribir en la imagen destino. Además, la posibilidad de juntar varias instrucciones en una misma línea de código tiene como consecuencia que la implementación resulta más clara y fácil de leer. Notamos que para calcular este valor b hicieron falta veinte líneas de código de assembler, mientras que en el caso de la implementación en C, solo una. Es verdad que la relación entre claridad y cantidad de lineas de código empleadas no necesariamente es una relación directamente proporcional, pero creemos que esto sí ocurre en el caso de las implementaciones de los filtros con los que trabajamos.

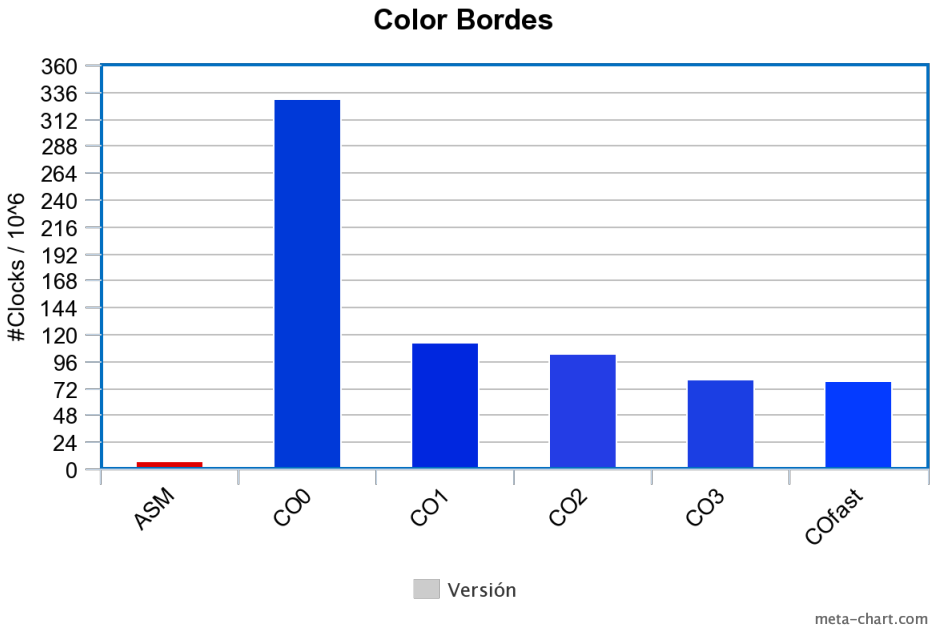
Ahora bien, a pesar de ser un código más extenso, la versión en lenguaje ensamblador posee algunas características que creemos pueden afectar el rendimiento de manera positiva. Por un lado, observamos que en todos los filtros se trabaja con varios píxeles en simultáneo por cada iteración; a diferencia de la implementación en C, donde cada iteración se concentra exclusivamente en uno solo. Esto se logra

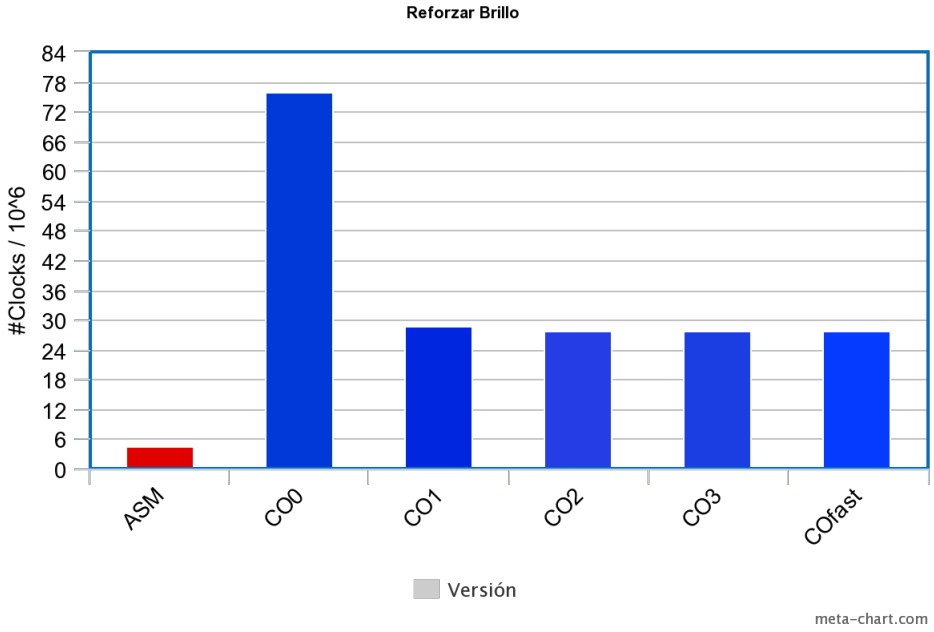
haciendo uso de el set de instrucciones vectoriales provistas por el procesador. Particularmente, esto permite aprovechar los accesos a memoria, buscando o escribiendo más información (en nuestro caso, más píxeles) cada vez que se interactúa con esta. Sabemos que las instrucciones que interactúan con la memoria suelen ser bastante más costosas que las operaciones entre registros. Entonces, retomando el ejemplo del filtro Reforzar Brillo, notamos que para tratar cuatro píxeles, la versión en assembler requiere interactuar dos veces con la memoria (una lectura y una escritura) y la versión en C, 8 (cuatro lecturas y cuatro escrituras). Por otro lado, vemos que en la versión de assembler hay menos instancias del programa en las que se rompe el flujo de control lineal. Esto implica menos instancias en las que se interrumpe el pipeline de ejecución. Identificamos dos razones principales por la cuales esto se da: si se tratan más píxeles por iteración, entonces las iteraciones serán más extensas pero habrá menos de éstas. Es decir, si se trabaja únicamente con un píxel en cada iteración, se tendrán tantas iteraciones como píxeles; luego, si se aumenta la cantidad de píxeles trabajados por iteración, la cantidad de iteraciones se reduce de manera proporcional. Esto es relevante ya que cada iteración significa una interrupción del flujo lineal. Adicionalmente, la posibilidad de usar máscaras para simular la toma de algunas decisiones, desplaza la necesidad de utilizar estructuras condicionales. Notamos que para trabajar con máscaras sólo se requieren operaciones lógicas y aritméticas, las cuales no interrumpen el flujo del programa. Habiendo hecho este análisis preliminar de las implementaciones, nos encontramos inclinados a creer -en una primera instancia- que las implementaciones de los filtros en lenguaje ensamblador serán más veloces que sus contrapartes en C.

Con el objetivo de corroborar este supuesto planteamos el siguiente experimento. Tomamos varias imágenes, les aplicamos las versiones de los filtros en ambos lenguajes de programación y comparamos los tiempos de ejecución correspondientes a cada una. Para cada caso, tomamos 100 mediciones. En el caso de las implementaciones en C, probamos con distintas versiones de optimización utilizando el compilador gcc. Particularmente, las opciones O0, O1, O2, O3 y Ofast; escogimos este subconjunto ya que son todas las opciones que buscan optimizar nuestra métrica de interés: el tiempo de ejecución. Como mencionamos previamente, creemos que la versión en assembler será, para todos los filtros y entradas, la más veloz de todas. Sin embargo, al no conocer del todo bien la manera en la que el compilador optimiza el código de C, no podemos descartar la posibilidad de que alguna de las optimizaciones muestre un rendimiento aún mejor.

A continuación presentamos los resultados del experimento.







En los gráficos de barras se describen los tiempos de ejecución en millones de clocks de procesador; correspondientes a las implementaciones de los tres filtros en C y en assembler. Las barras *CO0*, *CO1*, *CO2*, *CO3*, *COfast* contienen las mediciones asociadas a la implementación en C con su respectiva flag de optimización. Los promedios y desvíos calculados provienen de conjuntos de cien mediciones en todos los casos. En primer lugar, notamos un gran salto entre las opciones de optimización O0 y O1; curiosamente, la optimización lograda por las demás opciones no es del todo apreciable. Esto podría deberse a la escala del programa. Quizás, si estuviéramos trabajando con códigos de dimensión significativamente mayor, la diferencia entre las demás opciones de optimización también lo fuera. Por otro lado, a partir de estos gráficos podemos observar que el código en assembler, a pesar de la longitud, es notablemente mas rápido que el código en C; aún en su versión más eficiente. Creemos que esto se puede deber a que la versión en lenguaje ensamblador trabaja a nivel más bajo. Además, suponemos las ventajas anteriormente nombradas como el hecho de que trabaja de hasta cuatro píxeles por iteración a diferencia del píxel que C hace por iteración que hace el código en C, la utilización de las mascaras y demás ítems explicados en los párrafos anteriores. En la siguiente sección buscamos corroborar estos supuestos de manera empírica.

4. Experimentos y Resultados

4.1. Saltos condicionales

Nuestra idea es analizar el efecto que tienen los saltos condicionales sobre la performance de nuestros programas.

En la materia vimos que los saltos son disruptivos para el pipeline de ejecución del procesador. Esto es, generan que deba volver a armarse desde 0 cada vez que se ejecuta un salto en el programa.

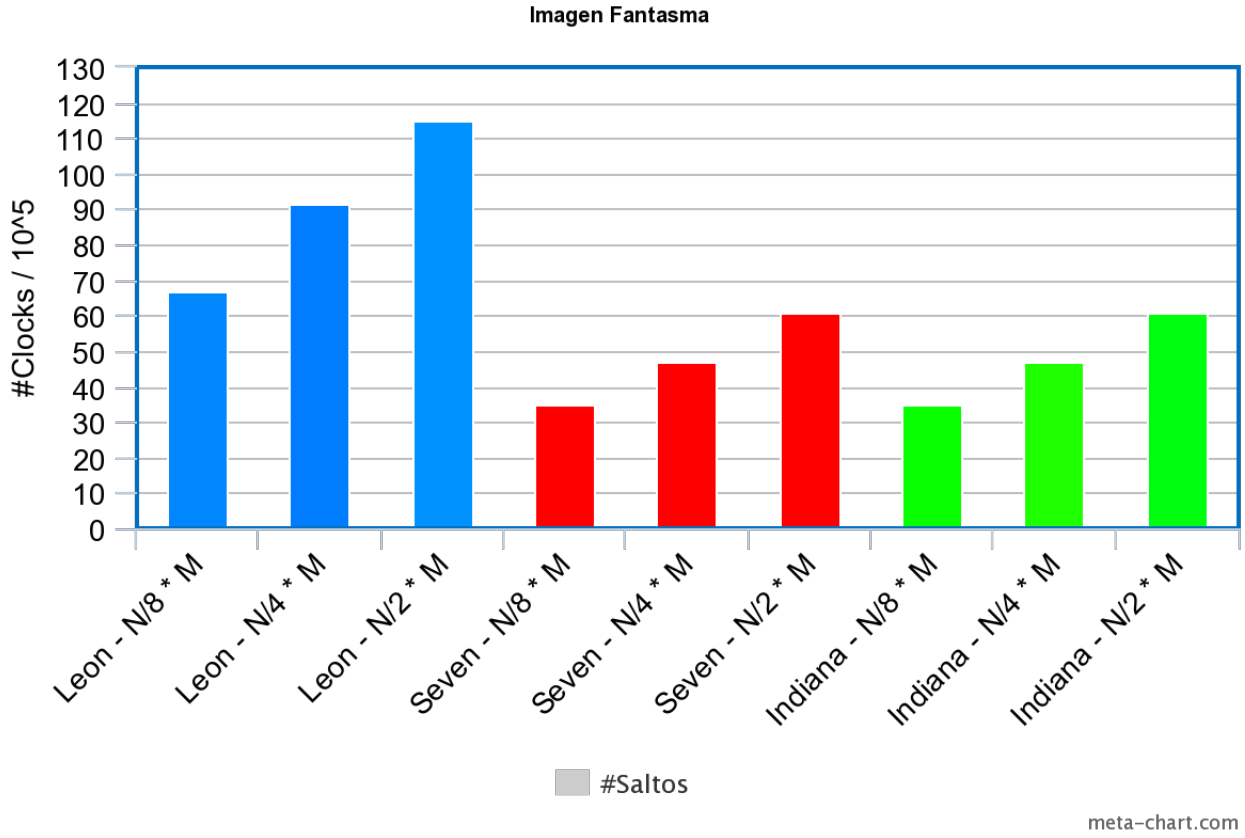
A raíz de esto, la pregunta que nos planteamos fue: ¿Que tanto influye el instruction pipelining en el tiempo de ejecución de un programa?

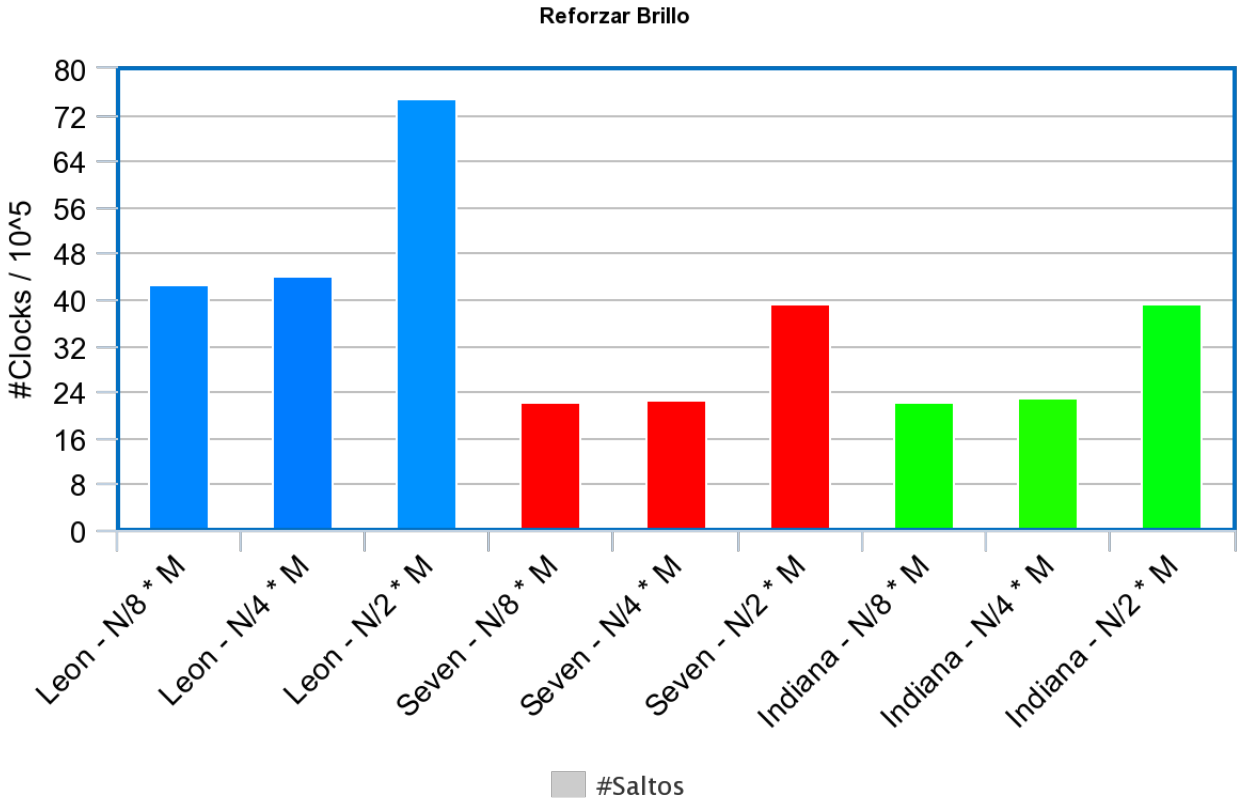
Hipótesis: Al disminuir la cantidad de saltos de los programas que implementan nuestros filtros, se verá una mejora no despreciable en el número de clocks que toman, y por lo tanto, en el tiempo de ejecución.

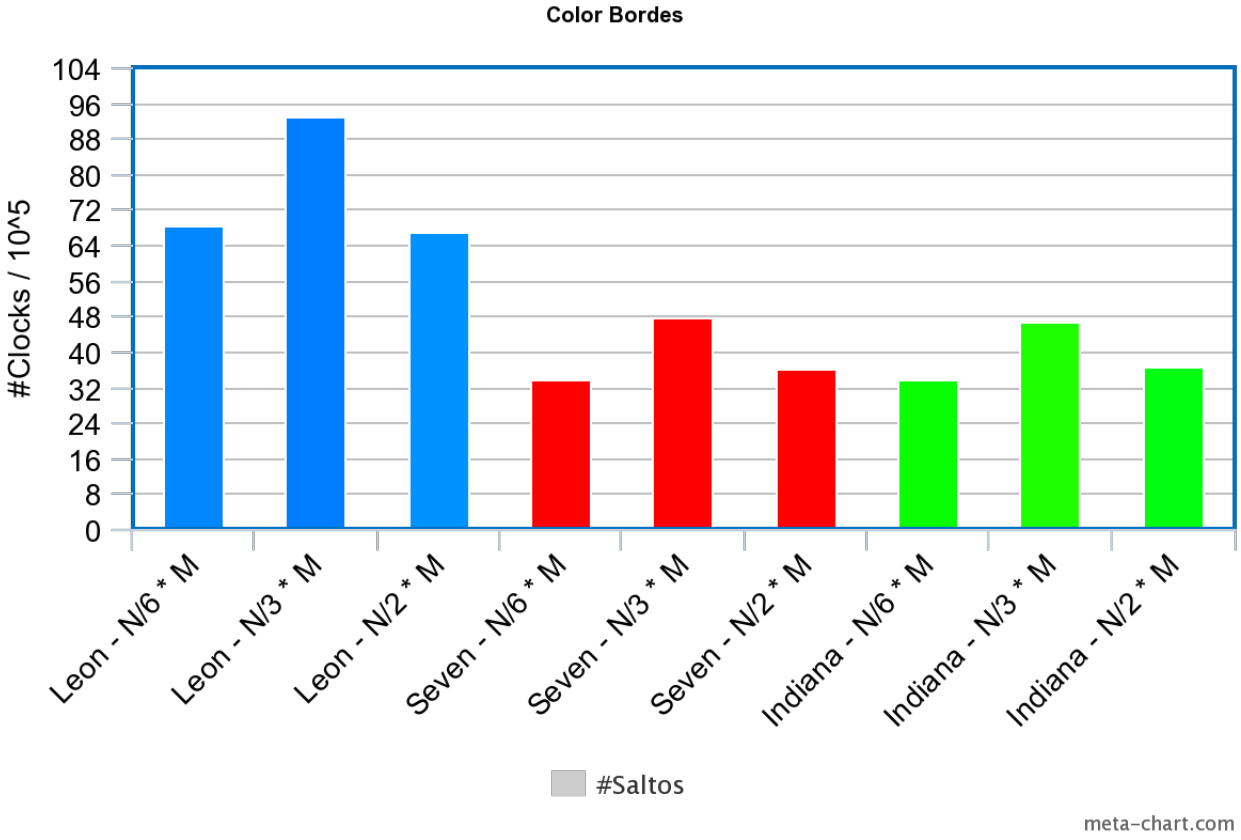
Experimento: Reducir el número de veces que se ejecuta el ciclo principal al aplicar un filtro a una imagen, procesando más píxeles por iteración. Así nuestro programa tendrá un número total de saltos menor.

Resultados: Para cada filtro implementamos 3 versiones en las que procesamos una cantidad variable de píxeles por iteración. A su vez, para cada versión realizamos 100 corridas, guardando el número de clocks que tomo cada una. Luego calculamos la mediana (que usamos en los gráficos) y el desvío estándar de esta muestra.

Tomamos N = Número de columnas y M = Número de Filas de la imagen, considerada como matriz.




Reforzar Brillo
#Clocks / 10^5
80
72
64
56
48
40
32
24
16
8
0
Leon - N/8 * M
Leon - N/4 * M
Leon - N/2 * M
Seven - N/8 * M
Seven - N/4 * M
Seven - N/2 * M
Indiana - N/8 * M
Indiana - N/4 * M
Indiana - N/2 * M
#Saltos
meta-chart.com


Color Bordes
#Clocks / 10^5
104
96
88
80
72
64
56
48
40
32
24
16
8
0
Leon - N/6 * M
Leon - N/3 * M
Leon - N/2 * M
Seven - N/6 * M
Seven - N/3 * M
Seven - N/2 * M
Indiana - N/6 * M
Indiana - N/3 * M
Indiana - N/2 * M
#Saltos
meta-chart.com

Análisis de resultados: Vemos qué para el filtro Imagen Fantasma parece haber una relación directa entre el número de saltos por iteración y el tiempo de ejecución.

Sin embargo, en el filtro Reforzar brillo, esta relación se pierde un poco. Es decir, vemos una notable mejora entre hacer N/2 * M y N/4 * M saltos por iteración, pero prácticamente no se ve diferencia entre hacer N/4 * M y N/8 * M.

Más aún, en el filtro Color bordes, no solo no se observa una mejora respecto a realizar menos saltos por iteración, si no que realizando N/3 * M saltos, el programa tarda más que realizando N/2 * M saltos. Pensamos que esta diferencia se debe a que, al recorrer la imagen como una

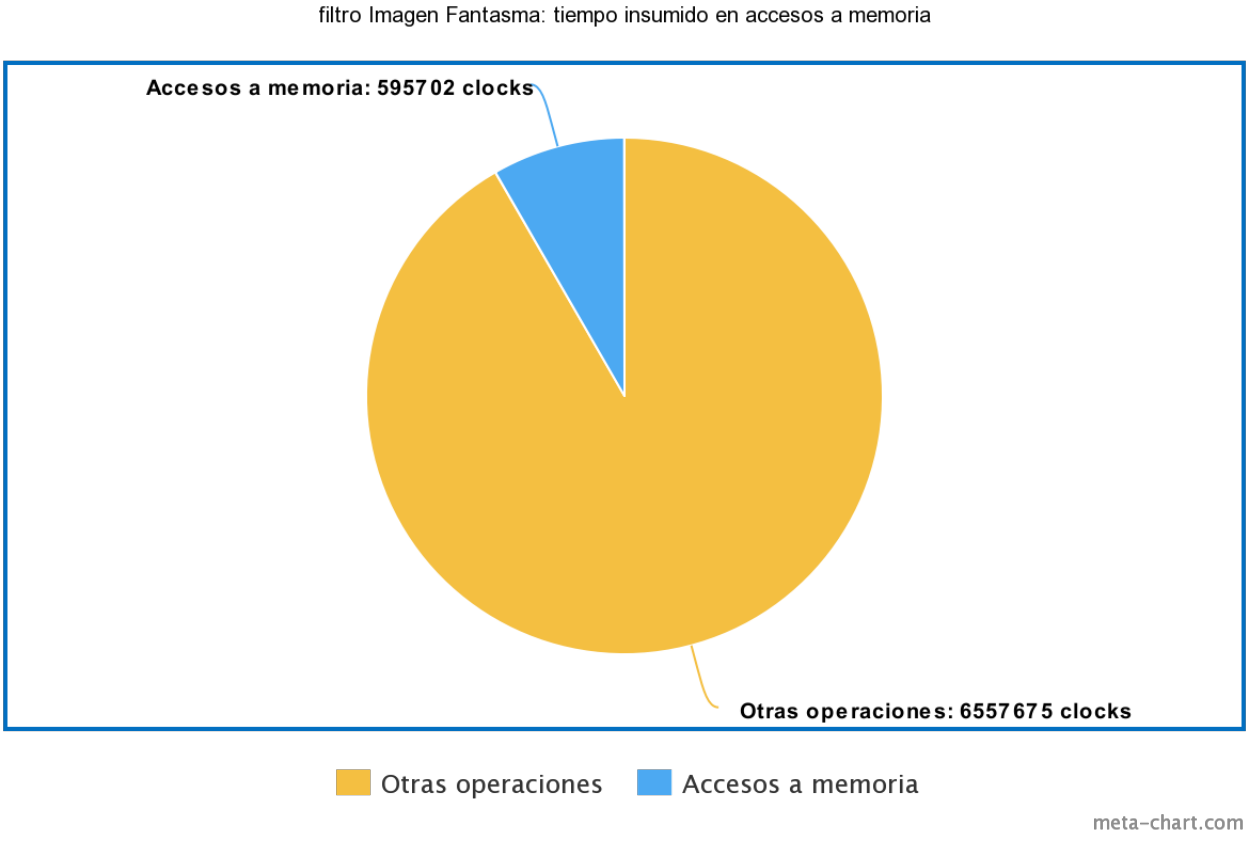
4.2. Interacción con la memoria

En la sección 3 suponemos una relación proporcional entre el tiempo de ejecución de un programa y la cantidad de veces que éste interactúa con la memoria. Creemos que los accesos a memoria significan una penalización no trivial en el rendimiento de un programa. Más aún, fundamentamos nuestra hipótesis en la sección 3 y los resultados obtenidos del experimento planteado, parcialmente apoyándonos en esta idea. Por lo tanto, en esta parte del trabajo nos interesa retomar este supuesto con la intención de corroborarlo empíricamente.

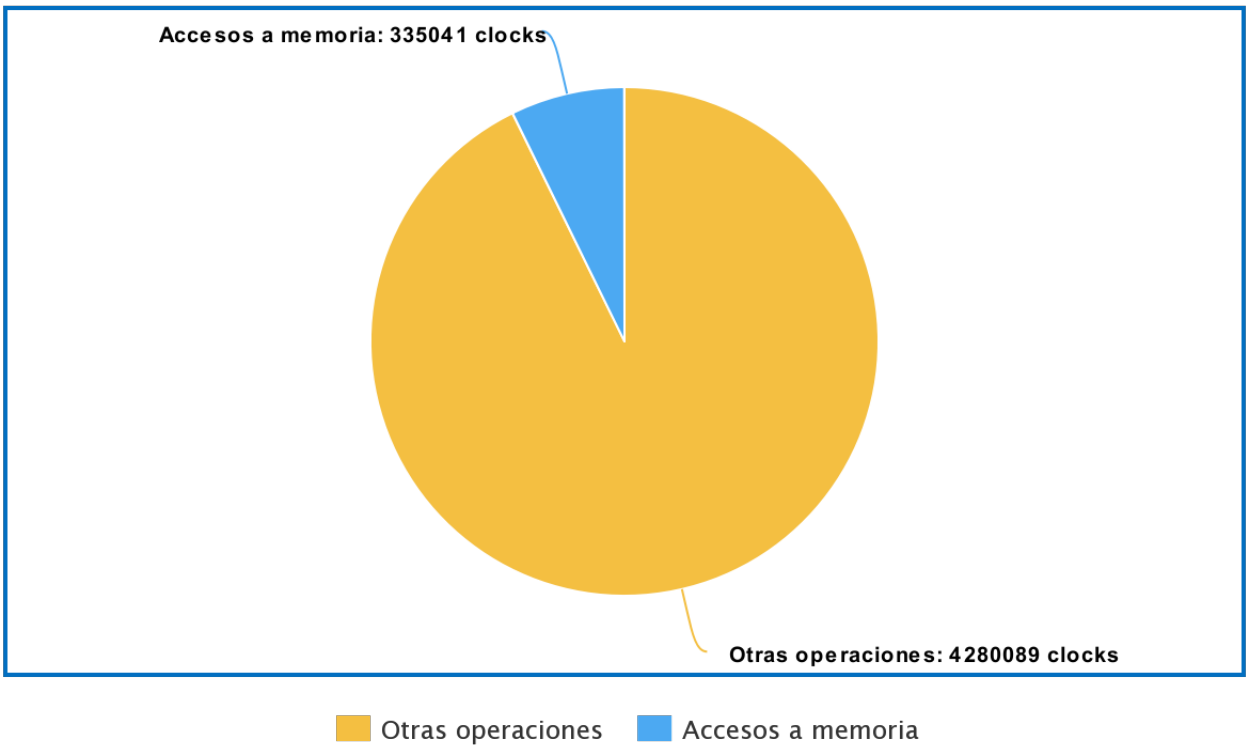
4.2.1.

Ahora bien, en primer lugar necesitamos tener un mejor entendimiento de la relevancia que tienen los accesos a memoria desde el punto de vista del tiempo de ejecución. Para ello, planteamos un experimento con el fin de obtener un porcentaje estimado del tiempo que va dedicado a las interacciones con la memoria. En una primera instancia, instrumentamos el código de los filtros en lenguaje de ensamblador duplicando todas las instrucciones que efectúan lecturas o escrituras a memoria. Luego, para obtener el estimado del tiempo insumido en los accesos, bastaría con calcular la diferencia entre el tiempo de ejecución del programa instrumentado y el del programa original.

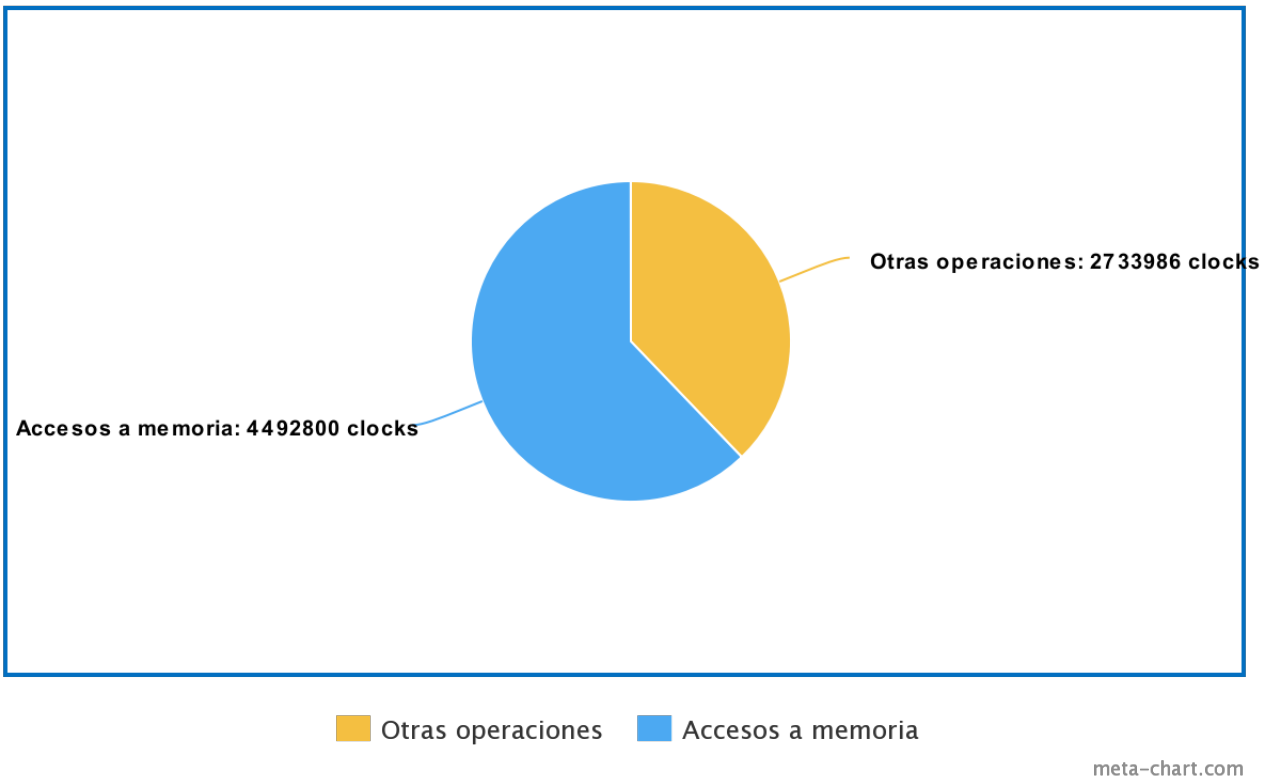
Con los resultados obtenidos a partir de este primer experimento (ver Anexo, Cuadro 3) generamos los siguientes gráficos:



filtro Reforzar Brillo: tiempo insumido en accesos a memoria



filtro Color Bordes: tiempo insumido en accesos a memoria



A raíz de estos resultados, podemos observar que en el caso de los filtros Imagen Fantasma y Reforzar Brillo, los accesos a memoria representan una fracción bastante insignificante del tiempo de ejecución total. Notamos que no llegan a constituir si quiera un octavo del total. Por otro lado, vemos que no este es el caso para el filtro Color Bordes. En contraposición a los otros dos filtros, aquí podemos ver que caso

dos tercios del tiempo fueron destinados a la interacción con la memoria. Una posible razón por la cual los resultados dieron de esta forma inesperada, puede radicar en las implementaciones particulares de cada uno de los filtros. Observamos que la cantidad de transferencias a o desde memoria presentes en cada uno varia significativamente. Dicha cantidad es bastante más chica en las implementación de los filtros Reforzar Brillo e Imagen Fantasma. En una iteración cualquiera del primero se efectúan únicamente dos accesos a memoria, mientras que en una iteración del ciclo principal de ColorBordes, trece. A pesar de que este último se encuentra dentro de lo que nosotros esperábamos y que ayuda a corroborar nuestro supuesto, no podemos ignorar la gran diferencia respecto de los otros dos filtros.

Más aún, notamos un problema con la propuesta de este experimento. Si se duplican las instrucciones que acceden a memoria de la siguiente manera:

```
; código original

movdqu xmm0, [rdi]

movdqu  xmm1, [rsi]
```

y entonces,

```
; código instrumentado

movdqu xmm0, [rdi]
movdqu xmm0, [rdi]

movdqu  xmm1, [rsi]
movdqu  xmm1, [rsi]
```

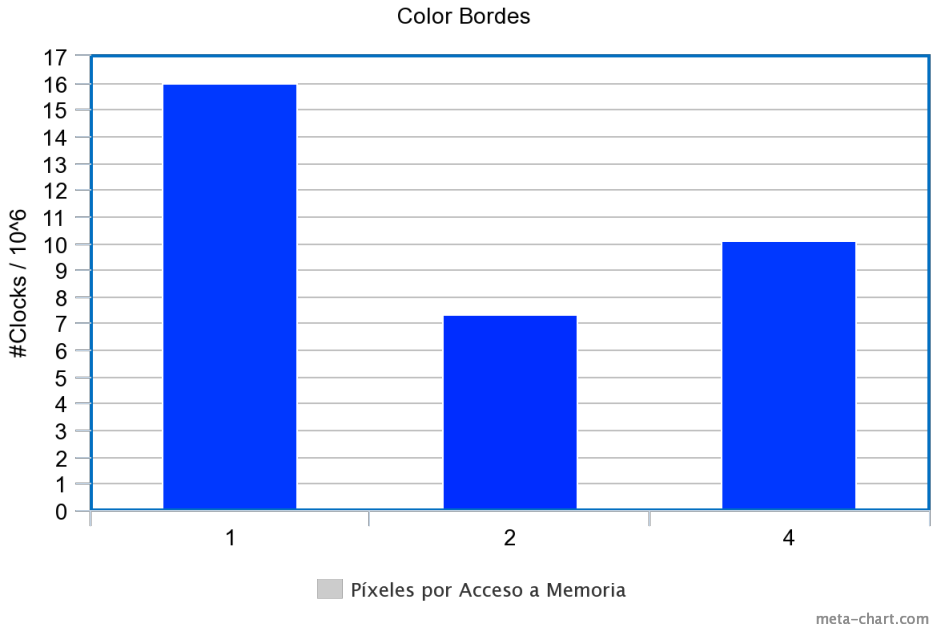
Observamos que en todos los casos, la instrucción duplicada efectúa un acceso a memoria que tiene la garantía de producir un *hit* en la cache; teniendo el cuenta el principio de vecindad temporal. Por lo tanto, no podemos afirmar que las cifras obtenidas en este experimento verdaderamente representen el tiempo insumido en los accesos a memoria. Particularmente, estarían representando el costo de los accesos memoria en el escenario donde tuviéramos todas las direcciones de interés almacenadas en la memoria cache. Desafortunadamente, no es un escenario que siempre podemos garantizar.

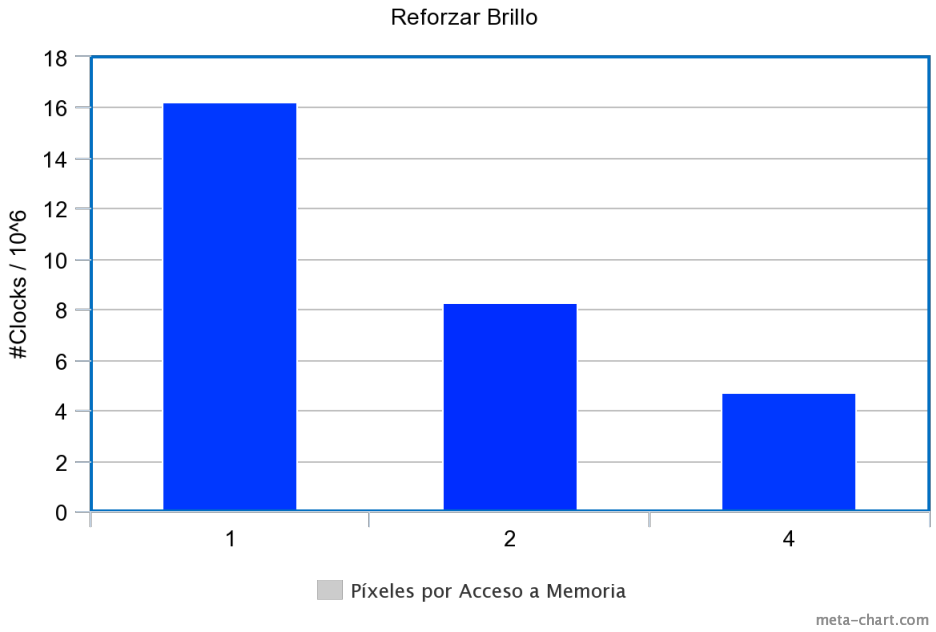
A raíz de esto, creemos que se podría llegar a hacer una segunda instancia de experimentación con el fin de obtener una medición más representativa. Este segundo experimento consistiría en modificar el código de los filtros de la misma forma, pero en sentido opuesto. Es decir, ahora quitaríamos todas las instrucciones que interactúan con la memoria. Entonces, para obtener la medición de interés, bastaría con calcular la diferencia entre el tiempo de ejecución del programa original y el del programa sin accesos a memoria. Notamos que en este caso, la imagen generada por la versión modificada no puede ser tomada como una salida válida; si quitamos los accesos a memoria, jamas se escribirá en la imagen destino. Sin embargo, esto no debería representar un problema siempre y cuando nos concentremos únicamente en el tiempo de ejecución de éste.

4.2.2.

No habiendo podido observar que una gran porción del tiempo de ejecución de todos los filtros se encuentra dedicado a la operaciones que interactúan con la memoria, sentimos interés en variar la cantidad de accesos a memoria que se hacen en las iteraciones y observar qué impacto tienen en el rendimiento de los programas. No necesariamente buscamos reducir esta cantidad; ya que, quizás disminuir la cantidad interacciones con la memoria intentando mantener el mismo comportamiento de las iteraciones, puede implicar que en cada transferencia se mueva más información, lo cual -desconocemos- podría ser más caro. Aún así, si tenemos en cuenta los resultados obtenidos en el experimento anterior, pareciera natural encontrarse inclinado a creer que conforme se disminuya esta cantidad, se observará un impacto positivo en la performance de los filtros.

Con el fin de verificar esta idea, proponemos un experimento. El mismo consiste en generar distintas versiones de los filtros en lenguaje ensamblador, variando la cantidad de píxeles que se transfieren en cada interacción con la memoria. A su vez, tomamos las versiones de estos que trabajan con cuatro píxeles en cada iteración. Este numero nos permite probar con todas las variantes posibles: transferencias de un píxel, dos píxeles o cuatro píxeles. Como queremos observar el impacto que tiene variar la cantidad de accesos a memoria por iteración, necesitamos que el cuerpo de las iteraciones se mantenga lo más consistente posible; caso contrario, podríamos estar introduciendo otros factores que terminen afectando el rendimiento. Para esto, haremos variaciones del siguiente estilo: supongamos alguno de los filtros para el cual necesitamos extender los componenetes de los píxeles a words y que estamos trabajando con cuatro píxeles en cada iteración. Entonces, por un lado podemos leer y extender los primeros dos píxeles en un registro (utilizando la instrucción pmovxz) y hacer lo mismo para los otros dos; por otro lado, podemos levantar los cuatro pixeles en un único acceso y luego repartirlos en dos registros extendiendo sus componentes a words. A pesar de que no creamos que vaya a mostrar un rendimiento óptimo, decidimos incluir una versión que moviera un solo píxel en cada transferencia. Creemos que puede ser interesante comparar el los tiempos de esta versión con las mediciones obtenidas a partir de las implementaciones en C de la sección anterior. Los resultados de este experimento se encuentran en el Cuadro 4 del Anexo. Con estas mediciones construimos los siguientes gráficos.





En los gráficos de barras se describen los tiempos de ejecución en millones de clocks de procesador; correspondientes a las distintas versiones filtros propuestas en el párrafo anterior. Las barras 1, 2 y 4 contienen las mediciones asociadas a la implementación que mueve esa misma cantidad de píxeles en cada transferencia. Escogimos graficar únicamente los resultados obtenidos para la imagen *Leon*, ya que la la relación presentada se daba de manera similar para la tres imágenes. Los promedios calculados provienen de conjuntos de cien mediciones en todos los casos.

En ambos casos observamos que hay una gran diferencia en el rendimiento del programa cuando se pasa a tratar múltiples píxeles por iteración. Por un lado, creemos que esto se debe a la gran cantidad de saltos condicionales que se terminan generando. Por otro lado, observamos que tratar un píxel por iteración implica no poder aprovechar las instrucciones del modelo de procesamiento SIMD; lo cual creemos tiene un gran impacto en la performance. En el caso del filtro *Reforzar Brillo* vemos una relación alineada con nuestra hipótesis: conforme aumentamos la cantidad de píxeles tratados por iteración, el tiempo de ejecución disminuye notablemente.

Sin Embargo, en el filtro Çolor Bordes", esta mejora solo se ve cuando se procesan 2 píxeles por acceso a memoria. Pensamos que esta inconsistencia se debe a un error de nuestra parte en la implementación de la versión que procesa la cantidad máxima de píxeles por registro XMM.

Curiosamente, si comparamos los tiempos obtenidos con la versión que trata un pixel por iteración con la versión más eficiente de la implementación en C, seguimos notando una diferencia no trivial; del orden de las decenas de millones de clocks de procesador. Esto siendo, una clara señal de la ventaja que provee las operaciones vectoriales del modelo de cómputo SIMD.

5. Conclusión

En este trabajo hemos explorado el modelo de procesamiento SIMD en el contexto de la implementación de filtros gráficos. Hemos programado distintas versiones de cada uno en dos lenguajes de programación particularmente distintos. A su vez, utilizamos esta variedad de implementaciones para en una primera instancia, discutir y comparar características propias de estos dos lenguajes; y en una segunda mitad, para estudiar el peso de algunos elementos de la microarquitectura del procesador.

Esta no fue una tarea del todo sencilla: entre todas las mediciones que hicimos, hemos visto que la búsqueda de un experimento adecuado no siempre es trivial. Cuando quisimos medir el porcentaje estimado de tiempo que va dedicado a las interacciones con la memoria, fue evidente la necesidad de comprender algunos conceptos de la microarquitectura.

Particularmente, hemos analizado como se relacionan ciertos aspectos de la interacción con la memoria y de la preservación del pipeline con el tiempo de ejecución de un programa. En ambos casos, observamos que su presencia no es trivial y que prestarles atención a éstos durante la etapa de desarrollo produce resultados más óptimos. Finalmente, nosotros mismos aprovechamos el trabajo empírico que se llevó a cabo para acercarnos a una versión más eficiente y estudiada de cada uno de los filtros gráficos presentados.

6. Anexo

Filtro	Datos	ASM	CO0	CO1	CO2	CO3	COFAST
Imagen Fantasma	Media:	6,95	159,01	67,17	66,38	67,50	62,18
	D.std:	0,38	3,72	0,99	1,05	1,50	0,95
Reforzar Brillo	Media:	4,50	75,85	28,76	27,93	27,89	27,91
	D.std:	0,42	2,57	0,34	0,23	0,19	0,23
Color Bordes	Media:	6,99	329,93	112,15	102,84	79,42	79,11
	D.std:	0,10	7,71	3,62	6,32	2,04	1,66

Cuadro 1: Tiempos de ejecución en millones de clocks de procesador; correspondientes a las implementaciones de los tres filtros en C y en assembler. Las columnas *CO0*, *CO1*, *CO2*, *CO3*, *COfast* contienen las mediciones asociadas a la implementacion en C con su respectiva flag de optimizicación. Los promedios y desvíos calculados provienen de conjuntos de cien mediciones en todos los casos.

Filtro	Saltos por iteración		Leon	Seven	Indiana Jones
Imagen Fantasma	N/2 * M	Media:	114,90	60,90	60,58
		D. std:	0,43	0,93	0,94
	N/4 * M	Media:	91,49	46,99	46,85
		D. std:	0,81	0,74	0,57
	N/8 * M	Media:	67,09	34,79	43,79
		D. std:	0,61	0,49	0,81
Reforzar Brillo	N/2 * M	Media:	75,01	39,24	39,16
		D. std:	0,83	0,62	0,69
	N/4 * M	Media:	43,92	22,69	22,98
		D. std:	0,80	0,52	0,64
	N/8 * M	Media:	42,69	22,21	22,32
		D. std:	1,00	0,54	0,60
Color Bordes	N/2 * M	Media:	66,74	36,27	36,76
		D. std:	1,35	1,04	1,27
	N/3 * M	Media:	92,273	47,68	46,70
		D. std:	1,98	3,05	0,68
	N/6 * M	Media:	68,50	33,48	33,58
		D. std:	1,88	0,61	0,64

Cuadro 2: Tiempos de ejecución en 10^5 clocks de procesador; correspondientes a los tres filtros, variando la cantidad de iteraciones (manteniendo constante el tamaño de la entrada). Las columnas *Leon*, *Seven* e *Indiana Jones* contienen las mediciones asociadas a cada una de estas imagenes de entrada. Los promedios y desvíos calculados provienen de conjuntos de cien mediciones en todos los casos.

Filtro	Version del Filtro		Leon
Imagen Fantasma	Original	Media:	7,15
		D.std:	0,35
	Instrumentada	Media:	7,75
		D.std:	0,18
Reforzar Brillo	Original	Media:	4,61
		D.std:	0,80
	Instrumentada	Media:	4,95
		D:std:	0,77
Color Bordes	Original	Media:	7,22
		D.std:	0,18
	Instrumentada	Media:	11,71
		D.std:	0,34

Cuadro 3: Tiempos de ejecución -medidos en clocks de procesador- de las versiones original e instrumentada de cada filtro. La versión instrumentada consiste en el mismo código que el original, solo que aquellas instrucciones que efectuan accesos a memoria se encuentran duplicadas. Los promedios y desvíos calculados provienen de conjuntos de cien mediciones en todos los casos.

Filtro	Pixeles por Acceso		Leon	Seven	Indiana Jones
Reforzar Brillo	1	Media:	16,20	8,53	8,90
		D. std:	0,78	0,65	0,76
	2	Media:	8,31	4,26	4,43
		D. std:	0,81	0,13	0,22
	4	Media:	4,66	2,41	2,38
		D. std:	0,29	0,12	0,07
Color Bordes	1	Media:	16,03	7,73	8,08
		D. std:	1,55	0,37	0,58
	2	Media:	7,04	3,51	3,55
		D. std:	0,29	0,09	0,15
	4	Media:	10,11	4,88	4,88
		D. std:	0,81	0,09	0,08

Cuadro 4: Tiempos de ejecución en millones de clocks de procesador; correspondientes a los tres filtros, variando la cantidad de píxeles que se cargan por acceso a memoria (manteniendo constante el número procesado por iteración). Las columnas *Leon*, *Seven* e *Indiana Jones* contienen las mediciones asociadas a cada una de estas imagenes de entrada. Los promedios y desvíos calculados provienen de conjuntos de cien mediciones en todos los casos.