DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Filtros de imágenes

Organización del Computador II

Integrante	LU	Correo electrónico
Bruno Gomez	428/18	brunolml99@outlook.es

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 **3**
1.1. Objetivo . . . 3
1.2. Razonamiento que nos llevo a abordar esas preguntas . . . 3

2. Desarrollo **4**
2.1. Filtros . . . 4
2.1.1. Nivel . . . 4
2.1.2. Bordes . . . 4
2.1.3. Rombos . . . 5
2.2. Procedimiento general de experimentación . . . 5
2.3. Procedimiento del Objetivo I . . . 6
2.4. Procedimiento del Objetivo II . . . 6
2.5. Procedimiento del Objetivo III . . . 7

3. Resultados **11**
3.1. Objetivo I . . . 11
3.2. Objetivo II . . . 15
3.3. Objetivo III . . . 16

4. Conclusión **18**

1. Introducción

1.1. Objetivo

El objetivo general de este trabajo práctico es comparar el rendimiento de distintas implementaciones de tres filtros de imágenes. Los objetivos particulares son:

I Comparar el desempeño de código en C compilado con distinto grado de optimización con el de código escrito en assembler (ASM) empleando el modelo Single Instruction Multiple Data (SIMD).

II Evaluar el impacto de la caché en la aplicación de los filtros.

III Contrastar distintas estrategias de procesamiento y la elección de diferentes instrucciones, que cumplen el mismo propósito, sobre la performance de código ASM empleando instrucciones SSE.

1.2. Razonamiento que nos llevo a abordar esas preguntas

El desarrollo de compiladores y lenguajes de alto nivel permite un mayor grado de abstracción y facilita el diseño de algoritmos para problemas complejos. Muchas veces, un mayor nivel de abstracción implica ignorar el funcionamiento detallado del sistema. Este conocimiento del sistema permite en algunos dominios de problemas implementar soluciones más eficientes. Por ejemplo, para el procesamiento de datos multimedia el empleo del modelo SIMD, a través del set de instrucciones SSE, permite paralelizar el cómputo de múltiples datos. Sin embargo el compilador permite especificar distintos niveles de optimización del código generado y, en algunos casos, puede generar código que emplea instrucciones SSE. Por este motivo, en el objetivo I, se busca contrastar el desempeño de código C compilado con distinto grado de optimización con código escrito en ASM.

El procesamiento multimedia implica acceder a una serie de datos que se encuentran en memoria para luego procesarlos. El acceso a memoria principal, tiene un costo significativo en comparación a la ejecución de las instrucciones requeridas para procesar dichos datos. En consecuencia los procesadores modernos cuentan con una memoria de rápido acceso pero menor tamaño, llamada caché, la cual se utiliza para minimizar el costo de los accesos a memoria principal. Por este motivo el objetivo II consiste en evaluar el impacto de la utilización de la caché por parte del procesador sobre el desempeño de los filtros implementados.

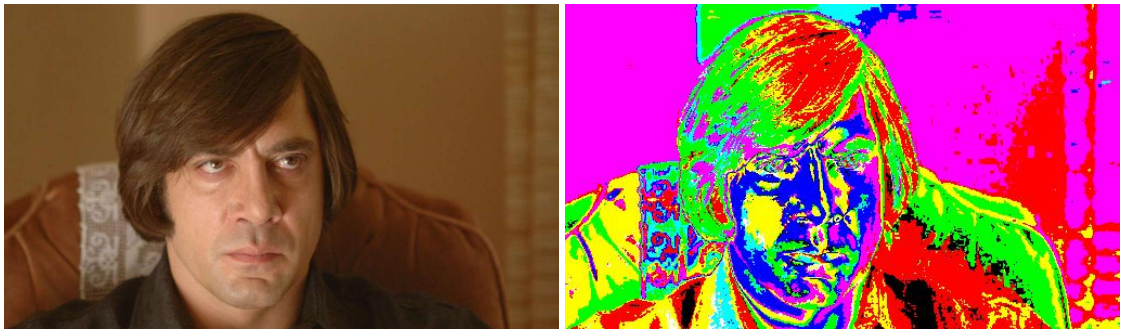
Una linea de código en un lenguaje de alto nivel puede implicar una serie de instrucciones de lenguaje ensamblador. Por este motivo, al trabajar a bajo nivel, se tiene un mayor control de cada paso del algoritmo sobre el que se esta trabajando. Este grado de control permite considerar distintas estrategias, que muchas veces implican el uso de distintos circuitos del procesador. En consecuencia, la elección de distintas instrucciones puede tener un gran impacto en la performance general del código. Teniendo esto en consideración, en el objetivo III se medirá el impacto de distintas implementaciones en lenguaje ensamblador.

2. Desarrollo

2.1. Filtros

A continuación se detallan los filtros sobre los que se van a realizar los experimentos.

2.1.1. Nivel



Este filtro recibe como parámetro un índice que varia entre 0 y 7. Para cada canal, si el bit indicado por dicho índice se encuentra en 1, el canal es saturado, en caso contrario se lo deja en 0.

La implementación en C consta de un ciclo que se encarga de recorrer cada píxel de la imagen realizando el procedimiento mencionado anteriormente.

Por otro lado la implementación en assembler, utilizando el set de instrucciones SSE, procesa de a 4 píxeles en simultaneo. Utiliza una máscara que se construye empleando el índice para seleccionar el bit adecuado y posteriormente saturar los canales correspondientes de cada píxel.

2.1.2. Bordes

En este filtro, se utilizan los coeficientes del operador de Sobel para calcular el valor de cada píxel. Para calcular dicho valor, los coeficientes se multiplican por los valores de los 8 píxeles que lo rodean y por el píxel a procesar. En consecuencia, este filtro no se puede aplicar a los bordes de la imagen porque siempre faltarán valores de tres o más píxeles adyacentes al píxel central. Por este motivo, los bordes de la imagen se pintan de color blanco. Nuevamente, en la implementación en C se recorre la imagen píxel a píxel y se calcula su valor utilizando los píxeles que lo rodean. Finalmente, se recorren los bordes y se los pinta de blanco. Para aplicar el filtro, las imágenes se convierten a escala de grises, es decir que tienen un sólo canal de 8 bits.

En la implementación en ASM, se procesan 16 píxeles a la vez utilizando instrucciones SSE. Se levantan en simultáneo, empleando ocho registros xmm, los píxeles adyacentes necesarios para el cálculo de cada uno de los 16 píxeles que se están procesando y se utiliza un registro más para los 16 píxeles centrales. Se desempaquetan los datos, ampliando su tamaño de byte a word, para evitar que los datos

excedan la representación durante los cálculos. Finalmente, una vez realizados los mismos, se empaquetan nuevamente los datos a byte para conservar la representación original. Posteriormente, se colorean en blanco los bordes superior e inferior procesando de a 16 píxeles a la vez y los laterales de a un píxel por vez.

2.1.3. Rombos

Este filtro modifica el valor de cada píxel para superponer una serie de rombos consecutivos de tamaño 64x64 píxeles. Dado que el tamaño de la imagen es mayor al tamaño del rombo, y que los mismos se repiten, el algoritmo utiliza el modulo de las coordenadas de cada píxel para determinar su posición relativa respecto del rombo correspondiente. A su vez, dicha posición determina el valor que se le va a aplicar a cada uno de los canales del píxel.

El código en C procesa píxel por píxel, realizando el calculo mencionado anteriormente para cada uno de ellos, mientras que el código de assembler procesa 8 píxeles por iteración paralelizándolos en dos cálculos de 4 píxeles cada uno. Para realizar este cálculo se utiliza una mascara donde se tiene la secuencia inicial de los índices, del 0 al 7, y se la va incrementando de a 8 haciendo modulo 64 en cada iteración. Para implementar el módulo se utiliza un AND lógico con 63. También cuando se leen los píxeles de memoria, se los desempaqueta de byte a word para ampliar el rango de representación en el que vienen los canales de los mismos y evitar desbordamientos durante las operaciones aritméticas. Posteriormente se empaquetan los datos para mantener la representación original.

2.2. Procedimiento general de experimentación

La determinación del desempeño de cada implementación se llevó a cabo utilizando la función MEDIR_TIEMPO aportada por la cátedra. Esta función emplea la instrucción de assembler rdtsc, que permite obtener el valor del Time Stamp Counter del procesador. Se llamó a la función para obtener dicho valor antes y después de la aplicación de cada filtro y se calculó la diferencia. Sin embargo, dado que la experimentación se realiza en un sistema operativo multitarea, las mediciones van a verse afectadas por los demás procesos que están siendo ejecutados en un momento dado. Es decir que dos ejecuciones de un mismo código pueden pueden resultar en una cantidad de ciclos de ejecución distinta si el scheduler del sistema operativo interrumpió la ejecución del filtro para correr otro proceso.

Teniendo esto en cuenta, para minimizar el ruido que genera nuestro sistema operativo, cada medición experimental de la cantidad de ciclos que utiliza un filtro dado fue considerada como el mínimo de 10 ejecuciones consecutivas, minimizando así los posibles outliers. Este proceso fue luego realizado 25 veces en distintos momentos. Es decir que, para cada experimento y para cada filtro evaluado, el número de observaciones utilizadas para calcular las medianas y los demás parámetros estadísticos graficados fue 25.

La experimentación fue realizada en una computadora con las siguientes características:

```
Arquitectura:                         x86_64
modo(s) de operación de las CPUs:     32-bit, 64-bit
Orden de los bytes:                   Little Endian
CPU(s):                               8
Hilo(s) de procesamiento por núcleo: 2
Núcleo(s) por «socket»:               4
Nombre del modelo:                    Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
Caché L1d:                            32K
Caché L1i:                            32K
Caché L2:                             256K
Caché L3:                             6144K
Memoria Total:                        16240540 kB
Swap Total:                           999420 kB
Sistema Operativo:                    Ubuntu 18.04.3 LTS
Kernel:                               Linux 4.15.0-64-generic
```

2.3. Procedimiento del Objetivo I

La metodología empleada se basó en correr el código en ASM, que utiliza instrucciones SSE, y el código C de manera separada con distintos flags de optimización (-O0, -O1, -O2 y -O3). Adicionalmente, se analizó el código assembler generado por el compilador con las distintas flags para determinar si en alguna de ellas logra utilizar instrucciones SSE. Finalmente, se comparó el desempeño de la implementación escrita en ASM contra la implementación escrita en C compilada con el flag -O3 para distintos tamaños de imagen con el objetivo de evaluar si las diferencias en performance se mantienen.

2.4. Procedimiento del Objetivo II

Para este objetivo nos centramos en la ejecución del filtro Bordes. Como el calculo del valor de cada píxel depende de los adyacentes al mismo, consideramos que es el que más puede ser afectado por no tener los datos precargados en caché.

Llevamos a cabo la experimentación comparando la implementación mencionada previamente con una implementación alternativa, en la cual principalmente alteramos la forma en la que se recorre la imagen. Utilizando un doble ciclo, donde por cada columna iteramos sobre todas las filas, y en cada fila procesamos de a 16 píxeles de la misma forma que en el filtro original.

Para esto utilizamos un registro de propósito general para guardar el puntero a la primer fila, así volvemos a comenzar desde ahí en la siguiente iteración. Este ultimo se utiliza para empezar en la primer fila luego de haber procesado todas las filas para una columna. Además, se incluyó dentro del procesamiento de una columna a las filas superior e inferior. En estas ultimas dos filas, al comenzar y al finalizar la iteración sobre una columna se pintan de a 16 píxeles en blanco, utilizando un registro xmm donde guardamos los valores de la máscara empleada. Luego de haber procesado toda la imagen de la manera descripta anteriormente. Con un ciclo aparte pintamos de blanco los píxeles laterales de la imagen, de la misma manera que en el filtro Bordes original.

De esta forma, la única diferencia entre esta implementación y la original es si se recorre la imagen por columnas o por filas. Dado que cuando se lee un dato de memoria este se carga en la caché junto con los cercanos al mismo, por el principio de vecindad espacial, al recorrer la imagen por columnas aumenta la probabilidad de que los datos no estén precargados en la caché reduciendo el rendimiento de la implementación.

Teniendo en cuenta el tamaño limitado de la caché, se repitió el procedimiento variando el tamaño de las imágenes a procesar para evaluar si el desempeño cambia en función del tamaño de la imagen. Esto ultimo se debe a que, para imágenes chicas, la cache puede ser lo suficientemente grande para almacenar toda la imagen, evitando caché misses.

2.5. Procedimiento del Objetivo III

Se realizó una comparación de performance entre distintas versiones de código para cada uno de los filtros. Dado que el filtro Nivel es sencillo, se aprovechó para comparar el rendimiento de recorrer la imagen utilizando un ciclo simple con el impacto de recorrerla utilizando un ciclo doble. Aprovechando que el código del filtro Rombos es más complejo y por lo tanto da más margen para realizar distintas implementaciones, se compararon variaciones en la forma en que se calcula el modulo, la cantidad de instrucciones para lograr determinados cálculos y la cantidad de valores procesados en simultáneo. Por otro lado, se buscó ordenar las instrucciones de forma tal que el procesador pueda escribir los resultados lo más rápido posible, minimizando las demoras durante el proceso de ejecución fuera de orden y mejorando así la performance del código. A continuación vamos a explicar cada variación realizada.

Filtro Nivel En la implementación original de este filtro realizamos un doble ciclo, es decir por cada fila iteramos sobre todas las columnas, de esta forma se interpreta la imagen como una matriz. La posibilidad de intercambiar esa forma de recorrer por un ciclo simple se debe a que podemos interpretar la imagen como un arreglo contiguo de píxeles aprovechando como se guardan en memoria los datos. A nivel implementación sólo se agrego la utilización de varios registros de propósito general para el cálculo de la posición del ultimo píxel a procesar.

Esta implementación alternativa fue realizada para demostrar como afecta la manera de recorrer los datos a la hora de procesarlos. En la ejecución de instrucciones un branch es una discontinuidad en el flujo de ejecución lo que genera una perdida crítica de performance en la ejecución de instrucciones que realiza el procesador. Dada esta situación el procesador utiliza un branch predictor para poder predecir el flujo de instrucciones a ejecutar. Considerando esto, por cada forma de recorrer que utilizamos, los saltos que realizamos son distintos. Teniendo esto en cuenta podemos asumir que, si el branch predictor realiza las predicciones de manera correcta, no debería haber una diferencia significativa en cuanto a performance para ambas implementaciones.

Filtro Rombos Este filtro realiza la siguiente transformación sobre cada píxel de la imagen:

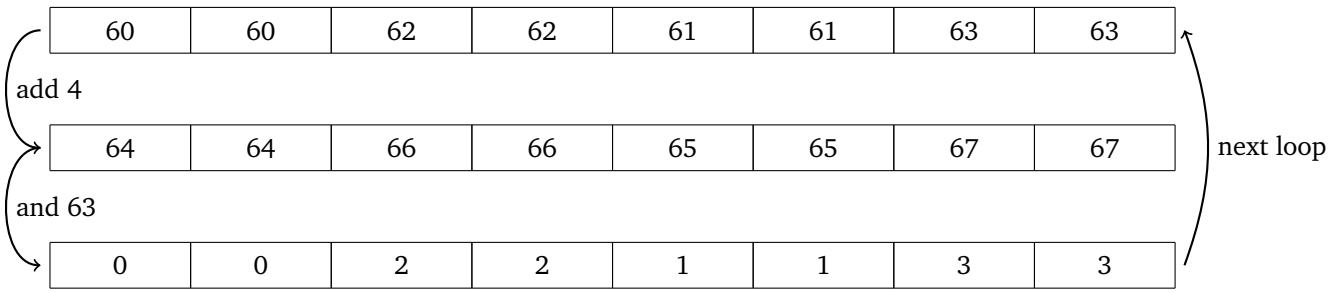
```
size = 64
Para i de 0 a height - 1:
    ii = abs(32-(i mod 64))
    Para j de 0 a width - 1:
        jj = abs(32-(j mod 64))
        x = (ii+jj-32) > 4 ? 0 : 2*(ii+jj-32)
        dst[i][j].b = SAT(src[i][j].b + x)
        dst[i][j].g = SAT(src[i][j].g + x)
        dst[i][j].r = SAT(src[i][j].r + x)
```

Los distintos experimentos que se realizaron atacan distintas partes de este pseudocódigo.

Por un lado, comparamos 3 versiones de código donde cada una procesa 4 pixeles por iteración pero difiere con las demás en como resuelve el modulo 64 que se presenta en el pseudocodigo anterior, puntualmente en "ii = abs(32-(i mod 64))" y "jj = abs(32-(j mod 64))".

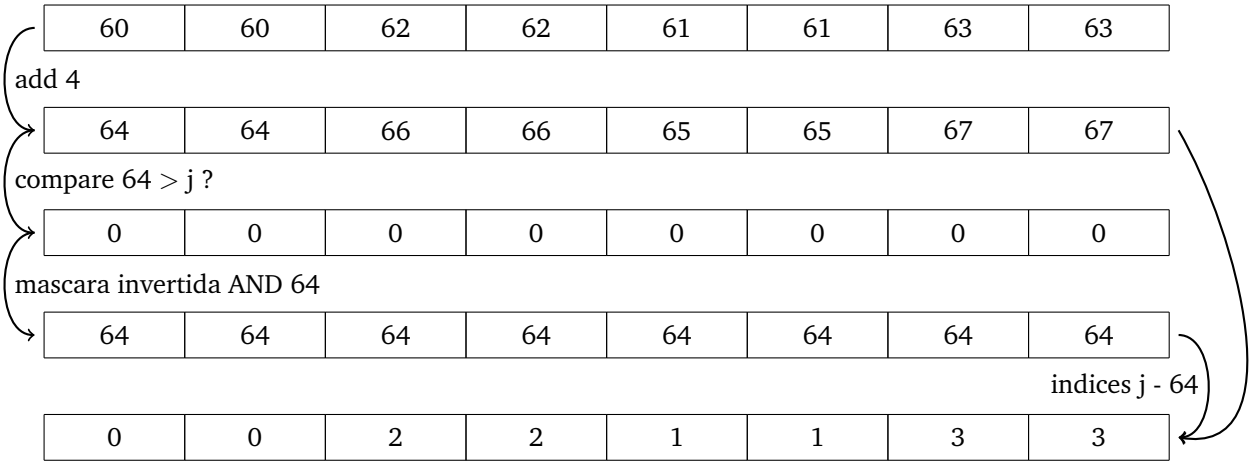
La idea para cada version es la siguiente:

1. AND 64:



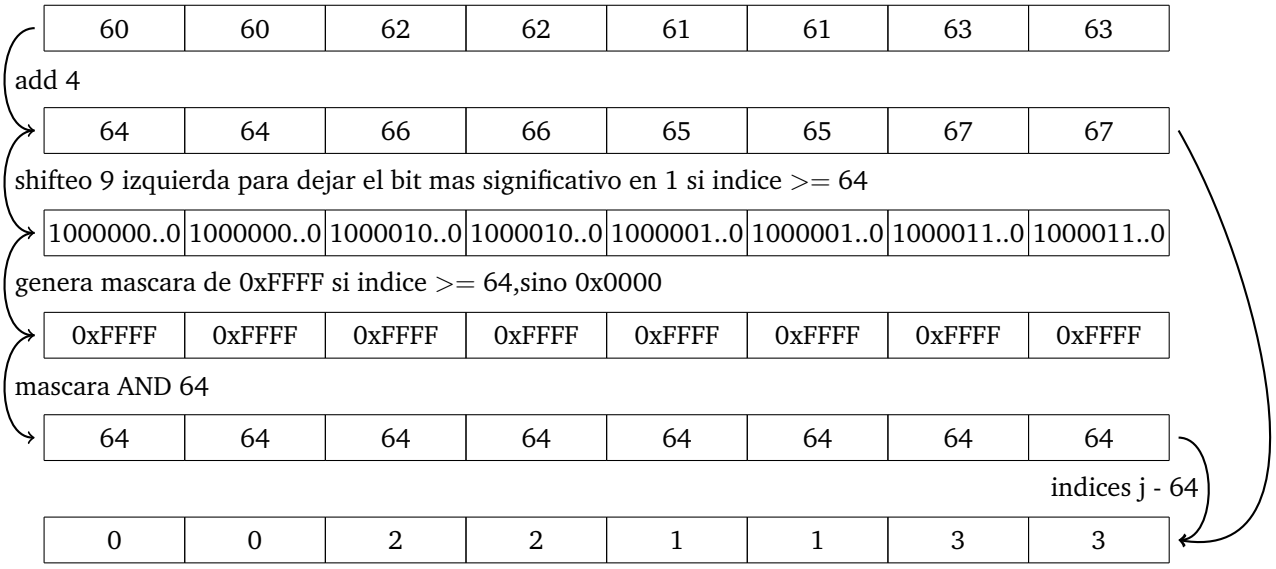
Realiza el modulo 64 en 1 paso, tomando el registro xmm que contiene a los siguientes 4 índices j (en el caso de i hay un xmm con 8 valores iguales porque se procesa sobre la misma fila) y haciéndole una AND lógica con 63, esto es equivalente a hacer mod 64.

2. Compare 64:



Esta versión, en cambio, para llevar a cabo el módulo 64 requiere primero mover la constante 64 a un xmm. Luego compara lo compara con el índice j para saber si este es mayor o igual a 64, en cual caso dejara una mascara de ceros. Finalmente, si era mayor o igual, se invierte esta mascara dejando todos unos, y se le hace una and con 64, para luego restar este resultado a los indices j.Esto es análogo para los índices i.

3. Shifting:



En esta versión, se puede observar que, como se suma 4 a los índices j en cada iteración, en la primera iteración que estos índices sean mayores a 63 serán menores a 128. Esto quiere decir que el bit 6 estará en 1 (64) en esa primera iteración y el bit 7 (128) en 0.

Utilizando este bit, se puede construir una mascara con shift lógico 9 a izquierda y luego shift aritmético 15 a derecha. Esta máscara tendrá todos 1s si el índice j es mayor igual a 64, por lo dicho anteriormente.

Finalmente, se puede utilizar esta máscara para generar un registro que tenga el valor 64 en las posiciones donde los índices j son mayor igual a 64, y luego se les puede restar a estos últimos este registro, obteniendo el modulo 64.

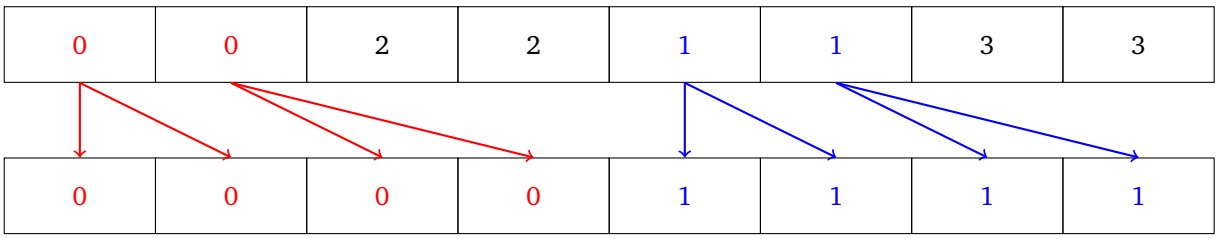
El caso es análogo para los índices i, ya que estos aumentan de a 1, y la justificación es la misma.

Por otro lado, teniendo en cuenta que se quiere ver como el OOO (Out of Order Execution) ayuda o no a la performance de un código, se incluirá en la experimentación el código de la versión 1 mencionada anteriormente, pero reordenando las instrucciones de manera que se pueda remover lo mas posible la dependencia entre instrucciones cercanas. El propósito de esto es el de permitir al procesador que pueda escribir el resultado de una instrucción lo antes posible, y en consecuencia acelerar el procesamiento.

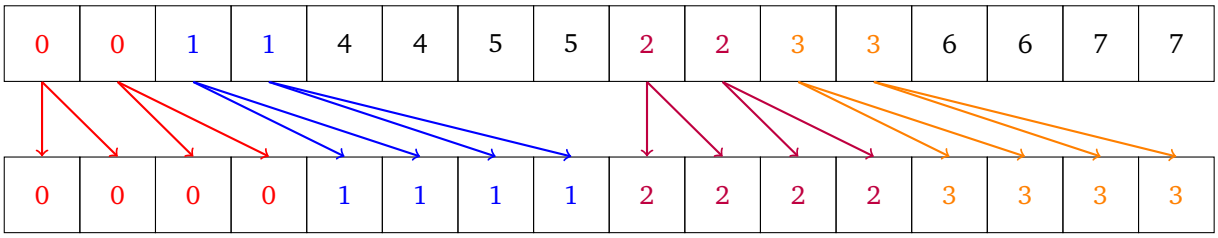
Otro aspecto que se desea ver como afecta el desempeño del código es el de aumentar la cantidad de cálculos en paralelo que se esta realizando en cada iteración. En el caso particular de este filtro se quieren calcular los valores, que se le aplicaran a los próximos X píxeles, en paralelo, como muestra el pseudocódigo del filtro Rombos. Con esto en mente, se comparara el código inicial que procesa 4 píxeles por iteración con 2 versiones distintas, donde la primera procesa 8 píxeles por iteración, y la otra procesa 16 píxeles por iteración. Estos 3 filtros a comparar, tanto el inicial como estas 2 versiones, trabajan a nivel byte, es decir, no pasan en ningún momento los datos a word para trabajar con los cálculos.

La manera en que trabajan con los índices estas 2 versiones es la siguiente:

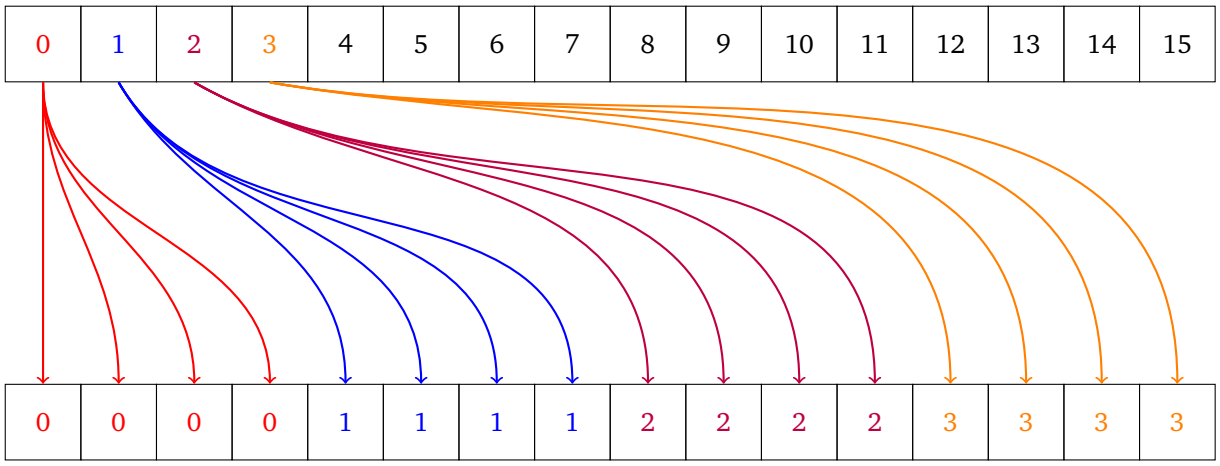
1. En esta primera versión, se tiene un registro xmm con los siguientes 4 índices almacenados como words, cada uno duplicado. Para poder trabajar sobre los primeros 2 píxeles a nivel word, dentro de cada iteración, se toman los primeros 2 índices y se los distribuye en un registro xmm utilizando shuffles. Análogamente, se hace lo mismo para los otros 2 píxeles, tomando los otros 2 índices, es decir, los 2 y 3 en el caso del ejemplo que se ve a continuación.



2. En esta versión, que procesa 8 píxeles por iteración, se tiene un registro xmm con los siguientes 8 índices. Dentro de cada iteración, se toman los primeros 4 índices y se los distribuye en un registro xmm utilizando shuffles, para poder trabajar sobre los primeros 4 píxeles. Análogamente, se hace lo mismo para los otros 4 píxeles, tomando los otros 4 índices, es decir, los 4, 5, 6 y 7.

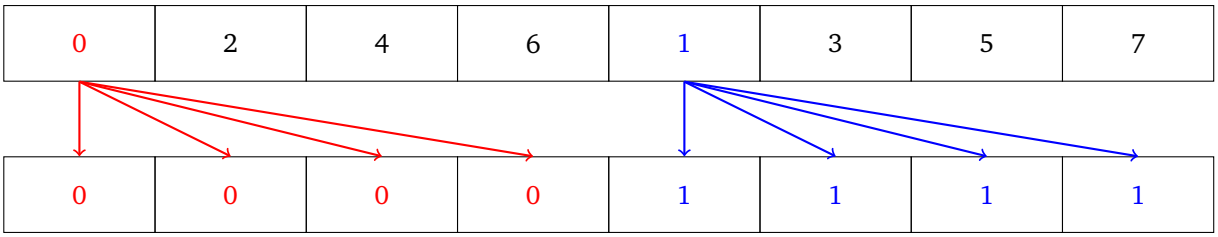


3. La ultima versión es una idea similar a la anterior, a diferencia de que se utilizan shuffles para mover los primeros 4 índices a un registro xmm y procesar los primeros 4 píxeles, y luego se repite este proceso 3 veces mas, ya que se tienen los siguientes 16 índices como se explico anteriormente.



Finalmente, se realizaron otras 2 versiones, a fines de comparar como utilizar distintas instrucciones para un mismo objetivo o realizar conversiones de datos afectan al rendimiento.

La primera de ellas tiene el objetivo de analizar el costo de las conversiones de datos. Esta procesa 8 píxeles por iteración, como la versión que fue explicada previamente, pero con la diferencia de que desempaqueta los datos de los píxeles, hace cálculos con words, y luego vuelve a empaquetar a byte. Para esto, los siguientes 8 índices estarán en forma de word en lugar de byte. Como se trabaja a nivel word, entran 8 words en un xmm. Esto equivale a 2 píxeles, entonces se tienen que utilizar 2 índices, copiando el primero y el segundo 4 veces, como muestra la figura. Este proceso se repite para los 6 píxeles restantes.



La otra versión busca comparar el impacto en utilizar distintas instrucciones para lograr el mismo objetivo. En este caso, se modificó la versión que procesa 16 píxeles por iteración. En esa versión se toman los resultados de los cálculos del pseudocódigo, los cuales ocupan un byte, y se los separa en 2 registros xmm distintos. Por un lado, se toman los bytes que contienen valores positivos y se los copia en un registro xmm respetando la posicion en la que se encuentran en el registro original. Esto se repite analogamente para los bytes que contienen valores negativos. Esta lógica de separar las cosas se hace a través de compares por si es mayor que 0 o menor que 0 en cada caso, y luego un and lógico para preservar la parte que interesa.

La idea de esta nueva versión, es cambiar la forma en que se separan estos valores, tratando de minimizar la cantidad de instrucciones utilizadas. Para esto se usó la instrucción PBLENDVB, la cual permite seleccionar entre 2 registros, que dato se quiere copiar en cada byte, dependiendo del bit mas significativo de un tercer registro. Aprovechando que se quiere separar en positivos y negativos representados en complemento a 2, se puede utilizar el bit mas significativo, que indica si es negativo, para seleccionar cual se quiere copiar.

Esto ultimo se puede observar en la siguiente figura de ejemplo:

Se espera que al hacer una comparación de todas estas versiones de filtros, se observen las siguientes cosas:

- Entre los 3 filtros que se centran en comparar métodos de realizar el módulo, el que utiliza una AND lógica sea el más eficiente, seguido por el que usa shifts lógicos y por último el que lo hace con compares.

- Comparando los 3 filtros que procesan 4, 8 y 16 píxeles por iteración respectivamente, esperamos que el mas eficiente sera el que procesa 16 píxeles y el menos eficiente el que procesa 4.
- La versión del filtro que procesa 4 píxeles con la secuencia de instrucciones ordenada de tal manera que intente beneficiar el OOO (Out of Order Execution) del procesador sea levemente mas eficiente que la que no hace esta nueva secuencia de instrucciones.
- Entre la versión que procesa 8 píxeles utilizando byte y la que realiza conversiones de byte a word, esperamos que la primera sea más rápida que la segunda.
- Esta última modificación de la versión que procesa 16 píxeles, empleando la instrucción PBLENDVB, le ganará en performance a la versión inicial que utiliza compares y AND lógicos para realizar la misma tarea.

3. Resultados

3.1. Objetivo I

En las siguientes figuras se observa el desempeño para cada filtro de la implementación en ASM comparada con la implementación en C compilada con distinto grado de optimización.

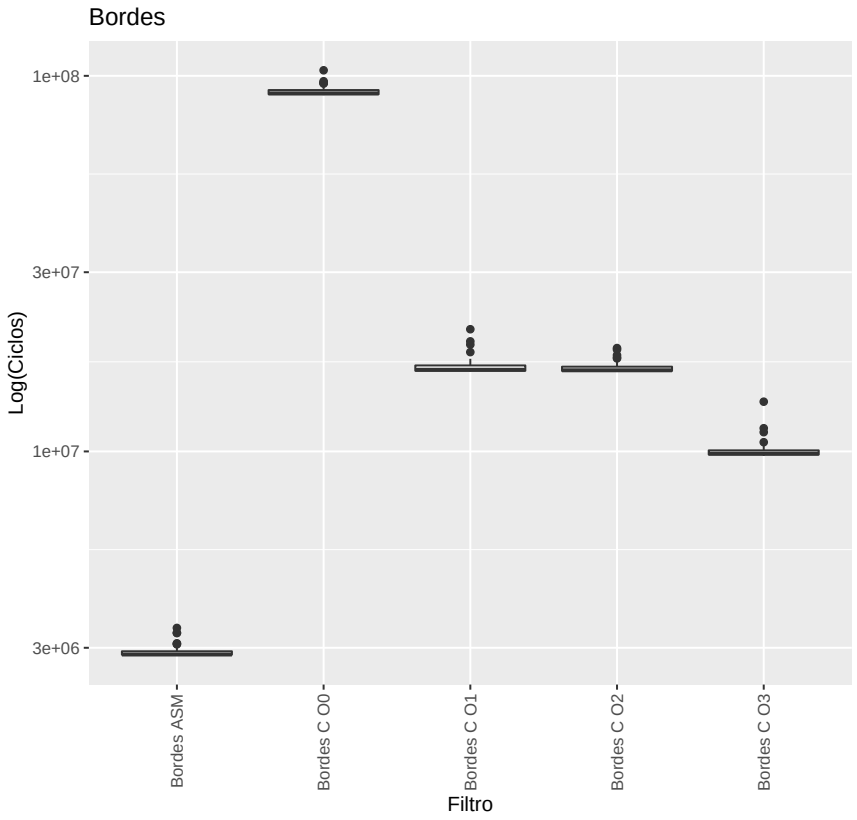


Figura 1: Boxplot del logaritmo en base 10 de la cantidad de ciclos que lleva ejecutar cada implementación del filtro Bordes.

Para el filtro Bordes (Figura 1), se observa que el mejor desempeño corresponde a la implementación en ASM lo cual corrobora nuestra hipótesis. En líneas generales, el desempeño del código C mejora a medida que se aumenta el grado de optimización, aunque no pareciera haber diferencias en el número de ciclos empleados entre el código compilado con O1 y el compilado con O2. Por otro lado, para este filtro, incluso el mayor grado de optimización tiene un peor desempeño que el código ASM. Al desensamblar los archivos objeto compilados con distinto grado de optimización, se puede observar que tanto el código O1 como el código O2 no utilizan instrucciones SIMD, sin embargo el código O3 si lo hace. Esto podría

explicar porque el desempeño de O1 es similar al de O2 pero ambos son peores que O3. De todas formas el código O3 tiene un peor desempeño que el código escrito en ASM.

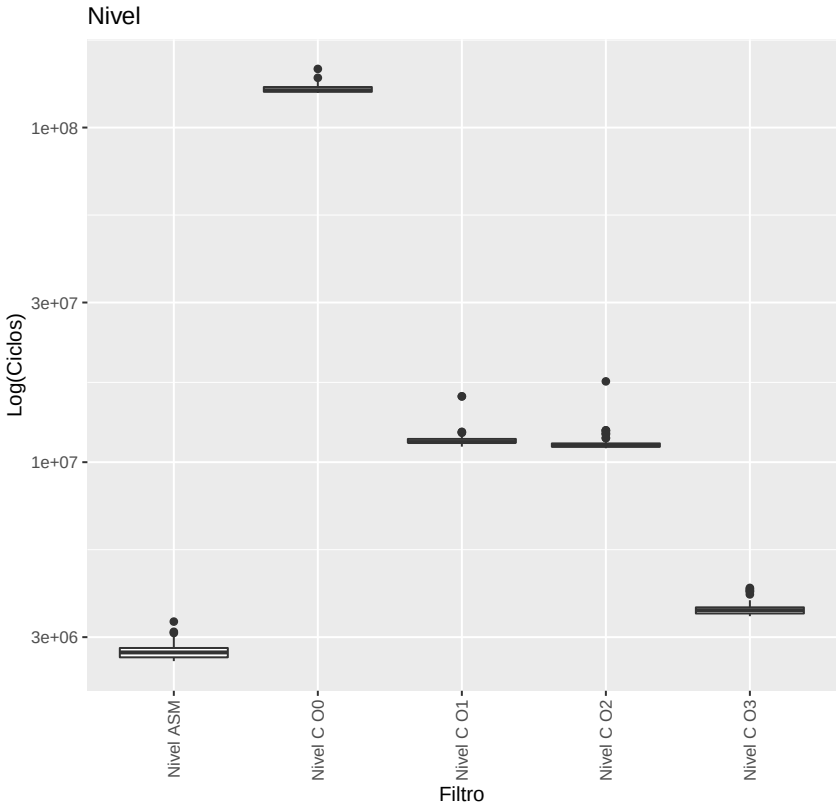


Figura 2: Boxplot del logaritmo en base 10 de la cantidad de ciclos que lleva ejecutar cada implementación del filtro Nivel.

Como se observa en la Figura 2, para el filtro Nivel, también se cumple que la implementación en ASM tiene el mejor desempeño y que el mismo mejora al incrementar el grado de optimización empleado al compilar el código c. Nuevamente observamos que no hay diferencias entre el código C compilado con los flags O1 y O2. Sin embargo, para este filtro, compilar código C con el flag O3 logra un desempeño similar al código ASM. En este filtro también sucede que las implementaciones O1 y O2 no utilizan instrucciones SIMD mientras que el código O3 si lo hace, lo cual también podría explicar las diferencias de desempeño entre los distintos grados de optimización.

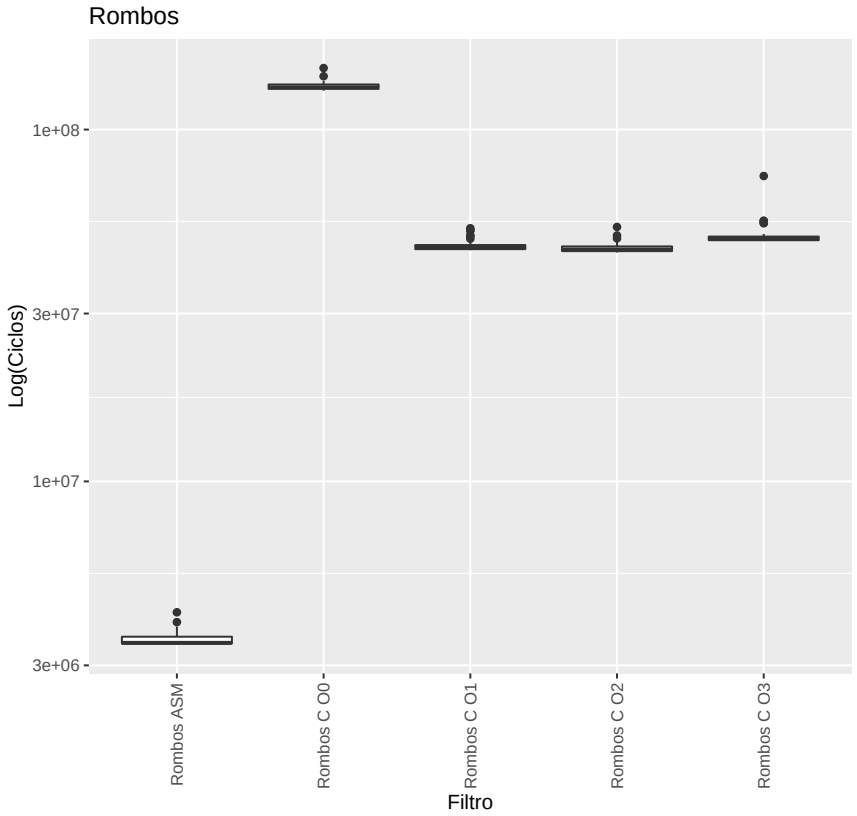


Figura 3: Boxplot del logaritmo en base 10 de la cantidad de ciclos que lleva ejecutar cada implementación del filtro Rombos.

Llamativamente, para el filtro Rombos, se obtiene un desempeño similar al compilar el código C con cualquier grado de optimización distinto a O0 (Figura 3). En este caso, la diferencia entre el desempeño del código ASM y el código C es la mayor de los tres filtros analizados. Al desensamblar los archivos objeto para los tres niveles de optimización del código C, se observa que ninguno emplea instrucciones SIMD. Esto explicaría la falta de diferencias en rendimiento entre las compilaciones con distintos flags en este filtro y la gran diferencia con el código ASM.

Estos resultados sugieren que a medida que el algoritmo del filtro se complejiza, el compilador tiene mayores dificultades para optimizar el código, incluso empleando el flag 03, y esto impacta en la performance (Figura 4). En el caso del filtro Nivel, el algoritmo es el más sencillo y el compilador logra utilizar instrucciones SIMD y conseguir una performance comparable a la del código escrito en ASM. La complejidad aumenta en el filtro de Bordes, sin embargo el compilador logra introducir el uso de instrucciones SIMD consiguiendo mejoras de desempeño respecto del código compilado con el flag O2. A pesar de ello, el número de ciclos es mayor al del código ASM sugiriendo que para lograr un rendimiento óptimo es necesario conocimiento del algoritmo que el compilador no posee y no puede deducir del código C. Finalmente, en el código de rombos el compilador no logra introducir instrucciones SIMD. Creemos que esto se debe a la complejidad asociada al calculo de los índices. Para lograr el rendimiento de la solución ASM es necesario utilizar las instrucciones SIMD para calcular varios índices en simultáneo. Claramente el compilador no logra reconocer este patrón en el código.

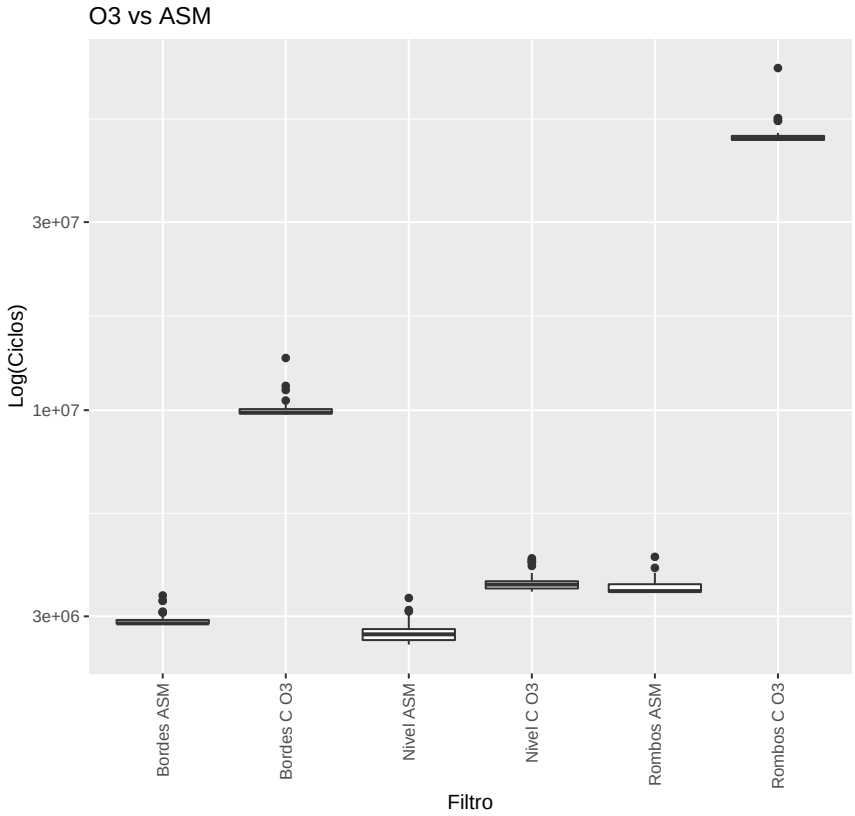


Figura 4: Boxplot del logaritmo en base 10 de la cantidad de ciclos que lleva ejecutar filtro compilado con el flag O3 o en versión ASM.

En la Figura 5 se observa, para cada filtro, el desempeño de la implementación en ASM y la implementación en C compilada con el flag O3 en función del tamaño de imagen procesada. Tanto el filtro Bordes como el filtro Rombos se comportan de forma similar. En ambos casos la implementación en ASM tiene un mejor desempeño que la implementación en C y la diferencia entre ambos se mantiene constante a medida que el tamaño de imagen aumenta. Sin embargo, la diferencia en desempeño entre ambas implementaciones para el filtro de bordes es menor a la diferencia observada para el filtro de Rombos. Esto se debe a un peor desempeño del código C del filtro Rombos y coincide con lo observado en la Figura 1. Por otro lado, para el filtro Nivel, se observa que la performance del código ASM en función del tamaño de imagen empeora con una pendiente mayor al filtro implementado en C. Esto hace que el desempeño del código ASM sea mejor que la del código C a tamaños de imagen chicos pero sea comparable a tamaños de imagen grande. Seria de interés continuar el análisis para tamaños de imagen más grandes para ver si la performance del código C logra superar al código ASM. Si esto fuera así, indicaría que el compilador está logrando una solución que escala mejor que nuestro código ASM. Claramente el compilador no es capaz de lograr esto cuando aumenta la complejidad del algoritmo, como se discutió anteriormente. Por este motivo, la diferencia entre el desempeño del código C y el código ASM permanece constante para distintos tamaños de imagen en el filtro de Bordes y el filtro de Rombos.

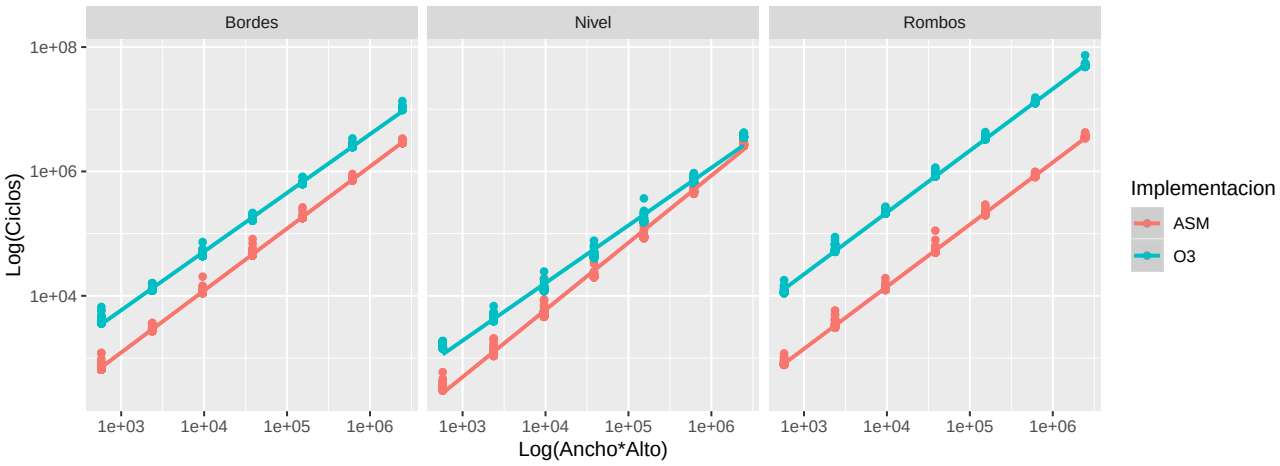


Figura 5: Logaritmo en base 10 de la cantidad de ciclos que lleva ejecutar cada filtro implementado en ASM o en C compilando con el flag -O3 en función del logaritmo de la cantidad de píxeles en la imagen.

3.2. Objetivo II

En la Figura 6 se muestran los resultados obtenidos, donde la implementación alternativa realizada para la experimentación es llamada ”Bordes ASM por columnas”. Para imágenes de menor tamaño se observa que la diferencia en el numero de ciclos de reloj empleados en ambas implementaciones no es significativa. Esto podría explicarse porque la distancia entre píxeles de diferentes columnas no es lo suficientemente amplia para que los datos dejen de estar cargados en caché para la próxima iteración. En cambio, al incrementar el tamaño de las imágenes, se empieza a incrementar la diferencia en cuanto a desempeño de ambas implementaciones. Siendo que la versión alternativa comienza a consumir más cantidad de ciclos de reloj que la versión original.

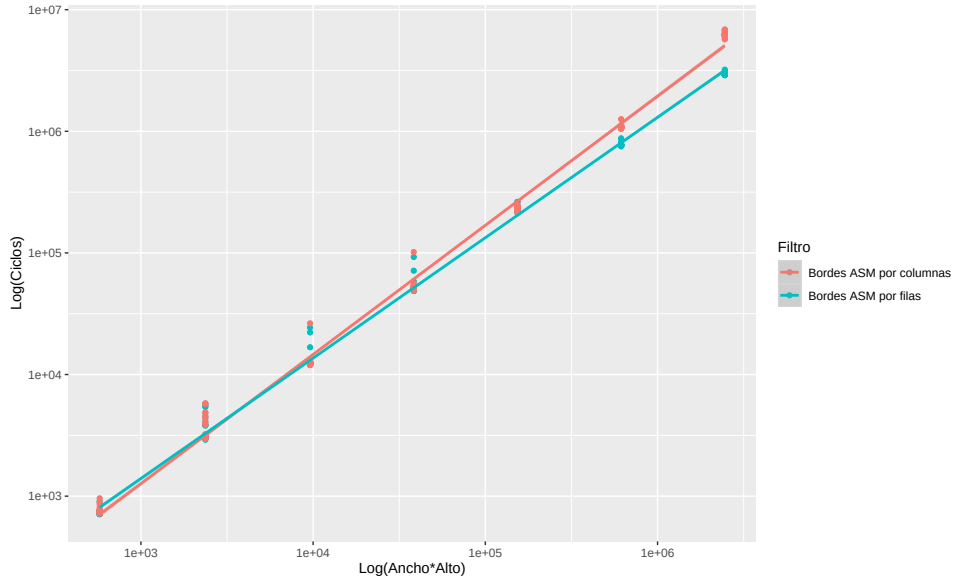


Figura 6: Logaritmo en base 10 de la cantidad de ciclos que lleva ejecutar el filtro Bordes implementado en ASM, recorriendo la imagen por filas o por columnas, en función del logaritmo de la cantidad de píxeles en la imagen.

Esto muestra el impacto de la utilización de la memoria caché en el desempeño de algoritmos de procesamiento multimedia. A pesar de que para imágenes lo suficientemente chicas, este impacto no se puede apreciar significativamente debido que el tamaño de caché es lo suficientemente grande como para almacenar los datos que vamos a procesar en el siguiente paso, si se puede ver un impacto a medida que

el tamaño de la imagen aumenta. En consecuencia una variación no radical en un algoritmo, como lo es la manera de recorrer los datos a procesar junto con la disposición de los mismos en memoria, puede tener un gran impacto en la performance gracias al aprovechamiento de la caché. Por otro lado, estos resultados argumentan a favor de la importancia de la memoria caché como recurso en el procesamiento de datos mostrando que él impacto de la caché en el rendimiento es significativo.

3.3. Objetivo III

La experimentación sobre las dos versiones del filtro nivel mostró los resultados esperados. Como se puede observar en la Figura 7 izquierda, tanto la versión con un bucle simple como la del bucle doble no presentan una diferencia significativa en performance. Esto creemos que se debe a lo que habíamos supuesto anteriormente sobre branch predictor, que la mayoría del tiempo adivina correctamente los saltos condicionales, haciendo irrelevante la presencia de un bucle doble.

Pasando a las implementaciones del filtro Rombos que procesan 4 píxeles en el ciclo principal (Figura 7 derecha) se puede observar una amplia diferencia entre la versión original (Rombos ASM 4px word), que utiliza el AND lógico con las otras dos, que utilizan compares en un caso y shifts en el otro, las cuales presentan un rendimiento inferior.

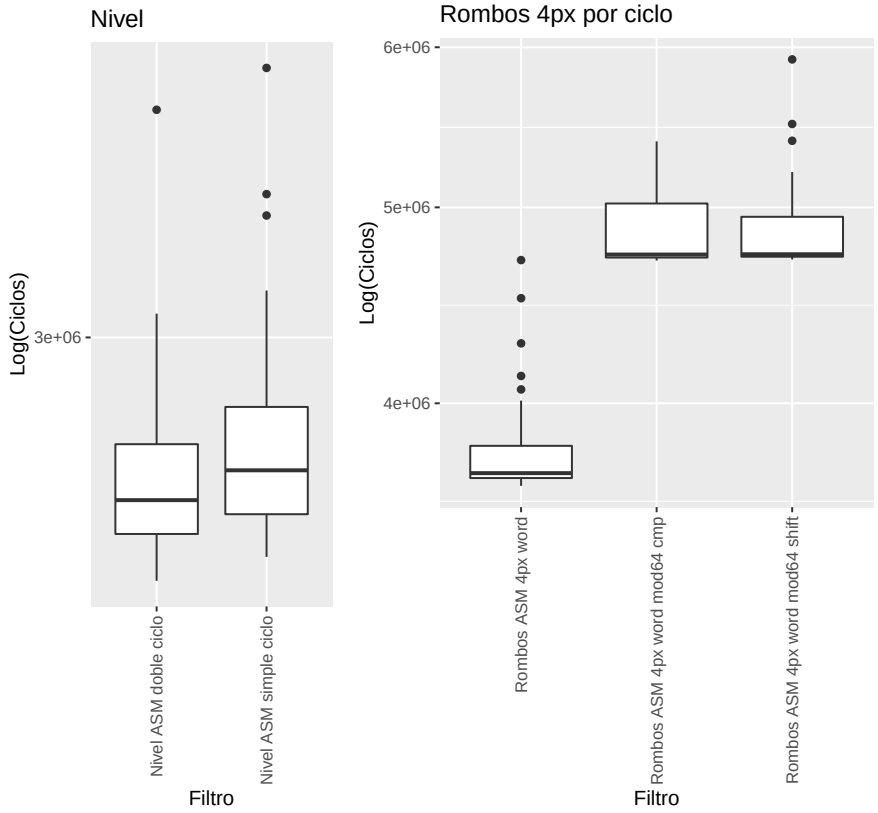


Figura 7: Logaritmo en base 10 de la cantidad de ciclos que lleva ejecutar cada filtro implementado en ASM de distintas formas.

Fijando la atención en la Figura 8, se puede ver que la diferencia de ciclos entre las implementaciones que procesaban 4, 8 y 16 píxeles se hace más notoria a medida que aumentan los píxeles procesados por iteración. Si comparamos los que procesan 4 y 8 píxeles respectivamente, vemos que la diferencia no es significativa. Sin embargo, al comparar la que hay entre 8 y 16 píxeles, se puede concluir que procesar mas píxeles por iteración, aprovechando la paralelización en los cálculos que esto implica, logra marcar una amplia diferencia.

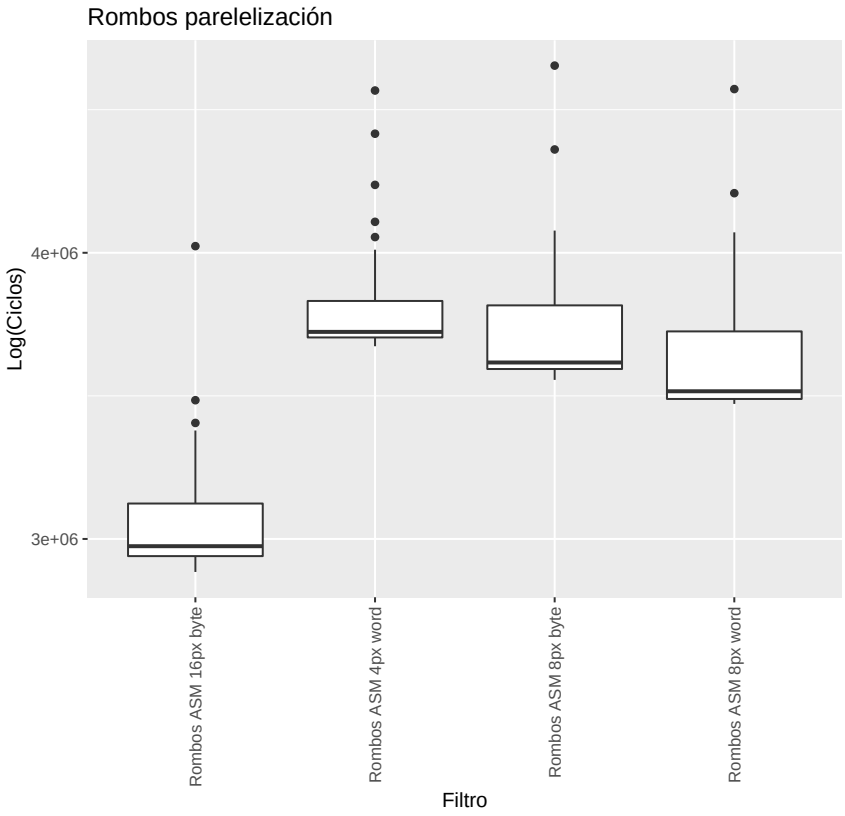


Figura 8: Logaritmo en base 10 de la cantidad de ciclos que lleva ejecutar cada filtro implementado en ASM de distintas formas.

Por otro lado, si comparamos las 2 implementaciones que procesan 8 píxeles por iteración, se puede ver que la que versión que desempaqueta a word en lugar de seguir trabajando con bytes es un poco mas eficiente. Esta observación contradice lo que se supuso que pasaría en un principio, es decir que convertir de byte a word y luego a byte nuevamente sería más costoso que trabajar con bytes directo. Esto puede deberse a que, las instrucciones utilizadas para trabajar con bytes directamente resultan mas costosas que convertir los datos. También puede haber sido un error en las mediciones, ya que la diferencia entre ambas implementaciones no es muy amplia.

En la Figura 9 izquierda podemos observar que el código que intentaba aprovechar el Out Of Order Execution que realiza el procesador para la versión ”Rombos ASM 4px word OOO”tiene un peor desempeño que la versión base que procesaba 4 píxeles. Dado que la ejecución fuera de orden es un proceso complejo y que no conocemos su funcionamiento en detalle, es probable que las modificaciones que realizamos creyendo que iban a ayudar a este proceso no lo hayan hecho.

Por último, en la Figura 9 derecha se observa que la modificación de la versión que procesa 16 píxeles por iteración utilizando pblendvb presentó una leve mejoría en el rendimiento. Esto quiere decir que, al parecer, reducir la cantidad de instrucciones para lograr un calculo puede llegar a lograr una mejoría en la performance, aunque este cambio implique en algunos casos utilizar instrucciones que requieran mas ciclos de procesador.

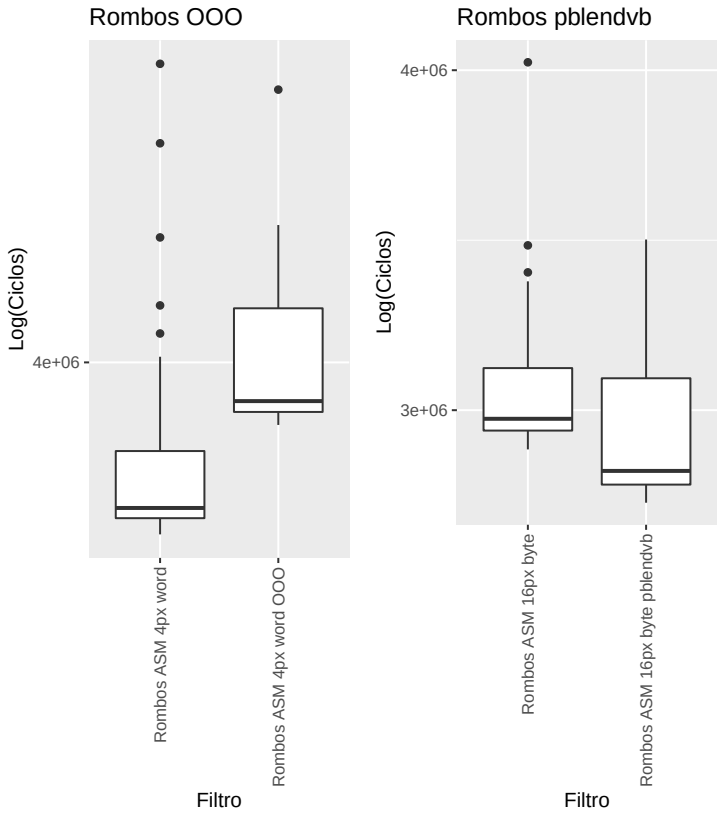


Figura 9: Logaritmo en base 10 de la cantidad de ciclos que lleva ejecutar cada filtro implementado en ASM de distintas formas.

4. Conclusión

Para todos los filtros se logró un mejor desempeño en las implementaciones escritas en ASM respecto de las implementaciones en C, incluso cuando el nivel de optimización empleado al compilar fue el máximo. Podemos concluir que el nivel de optimización que logra el compilador depende de la complejidad del algoritmo a optimizar y de que el mismo pueda reconocer patrones en el código C que puede mejorar en algunos casos implementando el uso de instrucciones SIMD. Cuando el algoritmo es sencillo, como es el caso de Nivel, logra un desempeño similar al código ASM. Sin embargo cuando el algoritmo se complejiza mucho, como es el caso de Rombos, el compilador no logra usar instrucciones SIMD. El uso de lenguajes de alto nivel permite abstraerse de estos conceptos y lograr código más compacto y fácil de entender, sin embargo se pierde performance. La existencia de flags de optimización para el compilador permite compensar hasta cierto punto esta falencia, pero cuando los algoritmos se complejizan puede que el uso de estos flags no logre las mejoras esperadas en el rendimiento.

Escribir código en ASM permite obtener un gran desempeño, sin embargo requiere un profundo conocimiento del problema y de la arquitectura a utilizar. Se puede ver por ejemplo en el caso del filtro Rombos, que al utilizar instrucciones SIMD para ciertas partes mejora ampliamente el rendimiento, pero al mismo tiempo, darse cuenta de que instrucciones utilizar y en que partes se puede hacer uso de las mismas no es sencillo. Se puede observar en esto último como, a pesar de trabajar con ASM y emplear instrucciones SIMD, es un factor de gran importancia saber estructurar el código, y ver la manera más eficiente de realizar la tarea tomando en consideración todas las opciones de implementación de la misma. Esto finalmente, termina impactando en la performance del código de bajo nivel de una manera notoria, ya que uno puede no estar haciendo buen uso del set de instrucciones SIMD en casos particulares para un algoritmo dado que se quiere implementar.

Si deseamos encontrar el mejor rendimiento posible, debemos contemplar el funcionamiento de la caché. En el caso de los algoritmos de procesamiento multimedia, el tiempo de acceso a datos no es algo despreciable en el costo total del mismo. En estos casos el contemplar dentro de la implementación

la manera mas beneficiosa de utilizar la caché, puede aportar un incremento en la performance. Dado que la imagen se almacena en memoria como una secuencia de filas y que la cache almacena posiciones consecutivas, resulta mas eficiente que los algoritmos recorran las imágenes por filas. Entonces en el caso en el que el formato de almacenamiento sea distinto, podría convenir recorrer por columnas. Por otro lado en un futuro, el avance de la tecnología utilizada en memorias y el incremento en su capacidad de almacenamiento, nos permitan evitar este tipo de contemplaciones acerca de la memoria caché. Sin embargo podemos concluir que no siempre el conocimiento profundo del problema nos puede llevar a la implementación con mejor desempeño sino que, además, conocer el funcionamiento del sistema donde va a ser ejecutado el código puede aportarnos soluciones mucho mas óptimas.