DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Organización del Computador II
Primer Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Rodrigo Laconte	193/18	rola1475@gmail.com
Julia Rabinowicz	48/18	julirabinowicz@gmail.com
Amalia Sorondo	281/18	sorondo.amalia@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

Resumen

En el presente trabajo implementamos tres filtros para imágenes. En base a estas implementaciones, realizamos experimentos con el fin de explorar algunos de los factores que influyen en la cantidad de ciclos de clock insumidos por una función en ASM. Analizamos de esta forma cómo afectan en el costo temporal procedimientos como llamados a funciones, saltos condicionales o distintas formas de acceder a memoria. Junto con el análisis de los resultados intentamos sacar conclusiones contrastándolas con nuestras hipótesis iniciales.

Índice

1. Introducción

En la actualidad entre todas las funcionalidades de las computadoras, una a destacar es la edición y procesamiento de archivos multimedia, tales como audio, video o imágenes. En este contexto, los algoritmos suelen ser repetitivos ya que generalmente se quiere aplicar el mismo procedimiento a los datos de cada archivo. En la década del '60, surgió la idea de poder procesar, en una sola instrucción, varios de estos datos[1]. Con este propósito, y con la ventaja que brindó la aparición de los registros de mayor capacidad de almacenamiento en los procesadores, comenzaron a aparecer las instrucciones SIMD (Single Instruction Multiple Data).

A lo largo de este informe vamos a enfocarnos en el procesamiento de imágenes. Su utilidad e importancia consisten en el reconocimiento de patrones en imágenes por ejemplo, tomar de una foto una patente, o una cara, pero también tiene un lado más enfocado al aspecto comercial, como lo es la edición de imágenes.

En este trabajo vamos a analizar las implementaciones de tres filtros en particular: nivel, bordes y rombos. Los mismos se encuentran explicados en la sección de desarrollo. Con este objetivo, en primer lugar implementamos los filtros en lenguaje ASM, utilizando instrucciones SIMD. Esto nos sirvió como base de la experimentación que realizamos de los mismos.

Todos los experimentos estudiados en este informe comparan distintas implementaciones de los filtros mencionados, midiendo la eficiencia de cada una en función de la cantidad de ciclos de reloj. El propósito de la experimentación es determinar qué factores influyen en la performance del código de ASM, por lo que nos centramos en aspectos tales como accesos a memoria, cantidad de saltos, contraste con código de C, entre otras cuestiones.

Para cada experimento planetado, proponemos hipótesis previas sobre lo que esperamos que ocurra, es decir cómo creemos que se modificará la eficiencia de cada implementación, basada en nuestras ideas y conocimientos previos. Luego, efectuamos cada uno de ellos y analizamos los resultados, comparándolos con los esperados e intentando explicar, en los casos en los que nuestras predicciones difieren de lo obtenido, las razones por las cuales esto sucedió.

[1] https://en.wikipedia.org/wiki/SIMD

2. Desarrrollo

2.1. Filtro de Nivel

Nuestra implementación del filtro de nivel es la más simple de las tres en cuanto al código. Consiste en un ciclo en el cual en cada iteración son procesados cuatro píxeles de la imagen, levantándolos en un registro XMM. Esta cantidad de píxeles ocupa el registro de 128 bits entero ya que cada uno está compuesto por cuatro componentes de un byte: azul (b), verde (g), rojo (r) y transparencia (a).

```
XMM0 = [  PIXEL 3  |  PIXEL 2  |  PIXEL 1  |  PIXEL 0  ]
XMM0 = [A3|R3|G3|B3|A2|R2|G2|B2|A1|R1|G1|B1|A0|R0|G0|B0]
```

Previo al ciclo, nos guardamos en otros registros XMM dos máscaras que luego utilizaremos. De esta manera evitamos acceder a memoria en cada iteración. Una de las máscaras se encuentra relacionada directamente con el parámetro índice que recibimos, entre cero y siete. En la sección *.rodata* guardamos las máscaras en orden, escribiendo primero la necesaria para el índice igual a cero, luego la del uno, y así hasta el siete. De esta manera, accederemos a las máscaras direccionando con *[mask0 + índiceParámetro]*.

```
section .rodata
align 16
mask0: times 16 db 0x01
    .
    .
    .
mask7: times 16 db 0x80
```

La segunda máscara la usamos para setear la transparencia de los píxeles a 0xFF al final del proceso. También nos armamos un contador para recorrer la imagen fuente utilizando los parámetros que la describen (ancho y alto).

Luego, el cuerpo del ciclo primero se encarga de levantar 4 píxeles en un registro de 128 bits para correrle un PAND contra la máscara. Como resultado, obtendríamos un registro R1 con el bit que nos interesa seteado en 1 (en cada paquete) solo si esta en 1 en los componentes de los píxeles que levantamos. Paso siguiente, comparamos por igualdad los píxeles con la máscara de R1, así tendremos cada paquete (uno por cada componente de cada píxel) en 0xFF si está prendido el bit en cuestión. Previo a mover los píxeles a la imagen de destino, y por último, debemos setear los componentes de transparencia en 0xFF por si se hayan modificado accidentalmente. Para esto, realizamos un POR entre la máscara de la transparencia y los píxeles modificados.

2.2. Filtro Bordes

El filtro de bordes consiste en aplicar operaciones matriciales sobre los píxeles, es decir que modifica cada píxel en función de sus píxeles vecinos y las matrices operadores Gx y Gy. El filtro opera sobre imágenes en escala de grises, cuyos píxeles cuentan con una única componente y por lo tanto ocupan un solo byte. Luego, en un registro XMM se pueden almacenar 16 píxeles. Sin embargo, el filtro requiere realizar operaciones cuyo resultado podría irse de la representación en byte. Por este motivo decidimos procesar de a 8 píxeles en paralelo para poder desempaquetarlos y así evitar pérdida de información al realizar las cuentas correspondientes. Levantamos entonces datos ubicándolos en la parte baja de los registros XMM gracias a la instrucción movq.

Para poder operar con los píxeles vecinos de forma paralela, en cada iteración del ciclo levantamos datos de la siguiente forma:

Sea el cuadro 1 un fragmento de la imagen a la cual le estamos aplicando el filtro y supongamos que nos encontramos en una iteración arbitraria del ciclo y queremos procesar los píxeles ”B1,C1,D1,E1,F1,G1,H1,I1”. En cada iteración vamos incrementando rdi, el puntero a la fuente, en 8 bytes. Desreferenciamos posiciones relativas a rdi para acceder a los vecinos de los píxeles a procesar. Utilizamos inicialmente 8 registros XMM:

0xA0	0xB0	0xC0	0xD0	0xE0	0xF0	0xG0	0xH0	0xI0	0xJ0
0xA1	0xB1	0xC1	0xD1	0xE1	0xF1	0xG1	0xH1	0xI1	0xJ1
0xA2	0xB2	0xC2	0xD2	0xE2	0xF2	0xG2	0xH2	0xI2	0xJ2

Cuadro 1: Fragmento de imagen

```
XMM1 = [0xA0,0xB0,0xC0,0xD0,0xE0,0xF0,0xG0,0xH0]
XMM2 = [0xB0,0xC0,0xD0,0xE0,0xF0,0xG0,0xH0,0xI0]
XMM3 = [0xC0,0xD0,0xE0,0xF0,0xG0,0xH0,0xI0,0xJ0]
XMM4 = [0xA1,0xB1,0xC1,0xD1,0xE1,0xF1,0xG1,0xH1]
XMM6 = [0xC1,0xD1,0xE1,0xF1,0xG1,0xH1,0xI1,0xJ1]
XMM7 = [0xA2,0xB2,0xC2,0xD2,0xE2,0xF2,0xG2,0xH2]
XMM8 = [0xB2,0xC2,0xD2,0xE2,0xF2,0xG2,0xH2,0xI2]
XMM9 = [0xC2,0xD2,0xE2,0xF2,0xG2,0xH2,0xI2,0xJ2]
```

Es decir, que los registros contienen la siguiente información según las cuentas a realizar:

↖ ↑ ↗ = xmm1 xmm2 xmm3
← → = xmm4 xmm6
↙ ↓ ↘ = xmm7 xmm8 xmm9

Luego desempaquetamos: extendemos la representación de los píxeles de un byte a una word. Como las cuentas con las matrices Gx y Gy involucran las dos a los vecinos ”de las esquinas” salvamos la información duplicándola en los registros XMM10, XMM11, XMM12, XMM13. Luego realizamos las operaciones correspondientes a Gx y Gy, que consisten en mantener dos acumuladores totalGx (XMM14) y totalGy (XMM15) a los que les sumamos el valor del píxel vecino por el valor de la posición asociada al vecino en la matriz Gy o Gx. Estos acumuladores operan de forma paralela: cada word representa la acumulación de uno de los píxeles procesados. Tomamos el valor absoluto de los acumuladores y realizamos la suma saturada empaquetada. Finalmente empaquetamos: el resultado final de cada píxel pasa a ocupar un byte y será el nuevo valor del píxel que es copiado en la posición apuntada por rsi, que es el puntero al destino.

Decidimos considerar la última iteración del ciclo por separado por cuestiones prácticas, evitando de esta forma accesos a memoria no correspondiente a la imagen. Finalmente, los píxeles que forman los bordes de la imagen son puestos en 255 recorriendo primero la primer fila de la imagen, luego la última columna, seguida por la última fila y por último la primer columna.

2.3. Filtro Rombos

Al aplicar el filtro ”Rombos” a una imagen, cada uno de sus píxeles se ve modificado en función de sus coordenadas en la imagen. Cada píxel está compuesto en este caso por 4 componentes que ocupan un byte cada una, luego un píxel ocupa 4 bytes. Como los resultados de las operaciones efectuadas con las componentes pertenecen al rango [-128,127], no es necesario desempaquetar los datos ya que pueden representarse en un byte sin correr riesgo de overflow. Podemos afirmar esto en base a las cuentas: primero le restamos a size/2, es decir 32, un número entre 0 y 63. Obtenemos entonces un número entre -31 y 32, y nos quedamos con el valor absoluto: un número entre 0 y 32. Finalmente, sumamos dos valores pertenecientes al rango [0,32], obteniendo un número entre 0 y 64 al que le restamos 32.

Dependiendo de este último resultado asignamos a x el valor 0 o un número entre 0 y 64. Es decir luego de la primera cuenta realizada nuestros datos ocupan un byte como máximo. Levantamos entonces de a 4 píxeles, siendo ésta la mayor cantidad de píxeles que pueden ser procesados en simultáneo, ocupando un registro XMM completo.

Como es necesario tener en cuenta las coordenadas de cada píxel, recorremos la imagen gracias a dos ciclos anidados, llevando un contador que indica en cuál fila nos encontramos y otro que indica en cuál columna. Nos posicionamos entonces en una fila y vamos recorriendo sus columnas. En cada iteración realizamos lo detallado a continuación. Levantamos cuatro píxeles contiguos, los cuales comparten la misma fila ya que el ancho de las filas son múltiplo de 8 y por ende también múltiplo de 4. De esta forma, tenemos en un registro XMM las cuatro componentes de cada uno de los cuatro píxeles, a través del cual vamos a aplicarle las operaciones a cada componente, es decir a cada byte del registro.

Primero calculamos el x que debemos sumarle a cada componente de cada píxel, para luego guardar estos x en un registro XMM [x3, x2, x1, x0][2] donde cada uno ocupa el byte menos significativo de la double word que le corresponde. Como size es igual a 64, para representar size/2 y size/16 levantamos de memoria las máscaras conformadas por [32, 32, 32, 32] y [4, 4, 4, 4]. Por otro lado, siendo i la fila actual y j las 4 columnas actuales, tenemos en un registro [j+3, j+2, j+1, j] y en otro [i, i, i, i]. Para obtener los restos de ambos registros módulo 64, basta con quedarnos con los 6 bits más significativos y poner en 0 el resto ya que 2^6 es igual a 64. Realizamos entonces un shift lógico a izquierda y luego a derecha de los paquetes de doublewords. Le restamos al registro [32, 32, 32, 32] los restos y nos quedamos con los valores absolutos de cada doubleword. Con cuentas empaquetadas similares calculamos [i+(j+3)-32, i+(j+2)-32, i+(j+1)-32, i+j-32]. Gracias a la instrucción pcmpgtd nos creamos una máscara que tiene 1s en las doublewords donde i+(j+k)-32 es mayor a size/16. Utilizando esa misma máscara negada guardamos los i+(j+k)-32 menores a size/16. Luego comparando con una máscara de ceros, separamos positivos de negativos para restar el valor absoluto de éstos últimos y sumar el valor absoluto de los primeros a cada componente de los píxeles. Antes de realizar esto último utilizamos la instrucción pshufb para pasar de [0,0,0,x3,0,0,0,x2,0,0,0,x1,0,0,0,x0] a [x3,x3,x3,x3,x2,x2,x2,x2,x1,x1,x1,x1,x0,x0,x0,x0], ya que le queremos sumar x a cada componente de cada píxel. Movemos el resultado al destino y avanzamos los punteros a destino y a fuente y sumamos a la columna actual 4. Para actualizar el registro conteniendo [j+3, j+2, j+1, j] le sumamos la máscara [4, 4, 4, 4]. Cuando llegamos al final de la columna, salimos del ciclo interno de columna: incrementamos la fila y reseteamos las columnas a [3, 2, 1, 0].

[2]Representamos los registros de forma que a la derecha se encuentran los bits menos significativos

3. Resultados

Para todos los experimentos, analizamos el costo temporal de un filtro dado un parámetro, haciendo el promedio de la cantidad de ciclos de clock de 10000 corridas. Esto lo realizamos 5 veces, devolviendo el promedio obtenido. Esto lo corremos para la implementación modificada para el fin de cada experimento, y para la implementación original, comparando ambos resultados. Todos los experimentos fueron corridos en las computadora con las siguientes características: versión del sistema operativo Ubuntu 18.04.3 LTS, con 12GB RAM y procesador Intel core i5-8265U CPU @1.60GHz x8.

Por otro lado, para todos los experimentos utilizamos una imagen de 2048x1200. Elegimos este tamaño ya que, como analizaremos en el experimento de la sección 3.2.1, creemos que es lo suficientemente grande como para amortizar los costos fijos que en imágenes más pequeñas representan un costo mayor, obteniendo de esta manera resultados más representativos.

3.1. C vs ASM

El primer encuentro con ASM no resultó tan reconfortante, por lo menos para nosotres. Códigos poco legibles, instrucción-operando-operando, parecía que estábamos martillando la computadora. Nada parecido a los lenguajes que veníamos viendo. Pero si une se fija, al estar codeando en un nivel tan bajo, se cuenta con una interacción más cercana con los componentes del procesador y su arquitectura, ya que las instrucciones del lenguaje son provistas por los productores de estos. Por lo tanto, nos preguntamos cuál de los dos lenguajes resulta en una implementación más rápida.

Hipótesis

Como tenemos entendido, los códigos en lenguaje C deben pasar por un proceso de compilación, para transformarse en lenguaje ensamblador. Aunque no contamos con completos conocimientos sobre cómo funciona cada compilador, esta compilación es probable que genere instrucciones innecesarias, o algunas que podríamos implementar de manera más eficiente en ASM directamente nosotres. Por esta razón, suponemos que la implementación de los filtros de este trabajo en ASM se efectúe en menos ciclos de clock que la de C. Además, como C no funciona con procesamiento vectorizado como SIMD, esperamos que ese sea otro factor por el cual el filtro implementado en ASM finalice su ejecución en menos ciclos de reloj. Las implementaciones de los filtros en C pueden ser ejecutados con diferentes flags de optimización[3]: O0, 01, 02 y O3. El flag -O0 es el flag por default y optimiza el tiempo de ejecución sobre el de compilación, al revés de lo que hace el flag -O1. Los flags -O2 y -O3 optimizan aún más que el -O1 el tiempo de ejecución pero con la desventaja de que puede tardar más la compilación.

Experimentos

Comprobaremos si nuestras suposiciones son correctas tomando una imagen y realizando varias corridas con ambas implementaciones para cada uno de los filtros. Comparamos la implementación de cada filtro en assembler con su implementación en C con el flag de optimización O3.

[3] `https://www.rapidtables.com/code/linux/gcc/gcc-o.html`

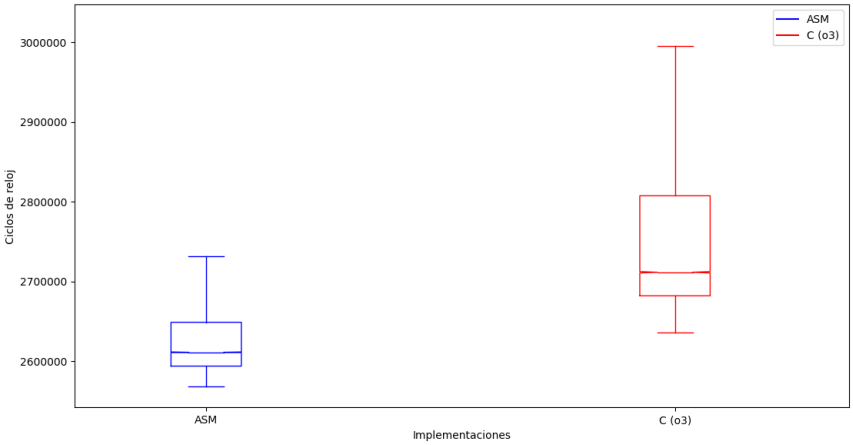

Figura 1: Comparación entre la implementación de ASM y la de C del filtro Nivel.

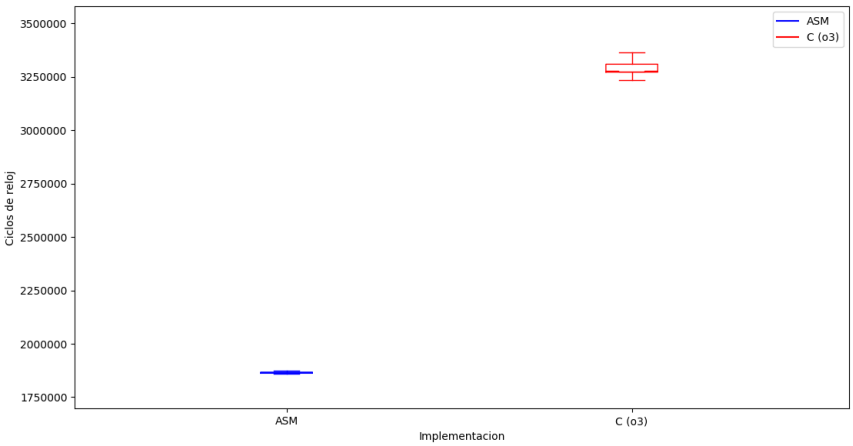

Figura 2: Comparación entre la implementación de ASM y la de C del filtro Bordes.

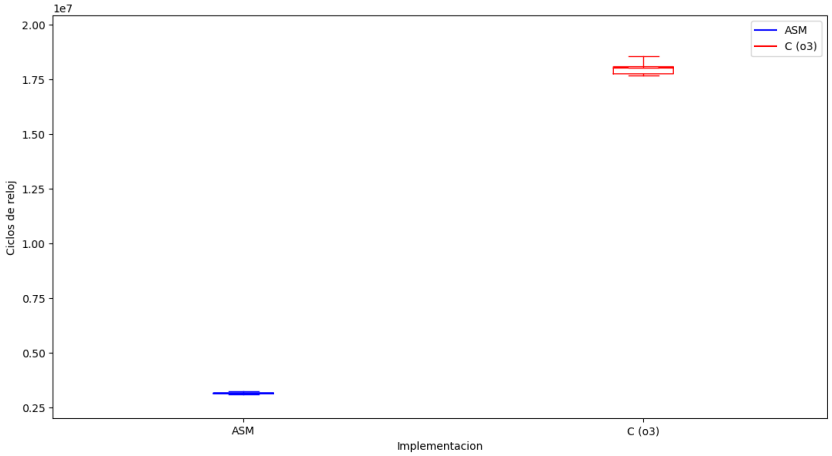

Figura 3: Comparación entre la implementación de ASM y la de C del filtro Rombos.

Análisis de resultados y conclusiones

En el caso del filtro de Nivel la cantidad de ciclos de clock insumidos en promedio por cada implementación es muy similar. La implementación del filtro en C toma 3,8 % más ciclos que la implementación en ASM. Luego, los resultados obtenidos no se contradicen con nuestra hipótesis inicial donde supusimos que el desempeño temporal de la implementación en ASM de los filtros sería superior al de la implementación en C. Si bien la implementación en C toma más ciclos, esta diferencia no resulta tan significativa y hasta podría ser asociada a un error de medición. Esto puede explicarse por la simplicidad de las operaciones realizadas en cada iteración en el filtro de Nivel. Además, los costos temporales fijos de este filtro, como los accesos a memoria para levantar las máscaras antes de comenzar a iterar, representan una proporción importante en el costo final del filtro. Por este motivo, ahorrarnos iteraciones en ASM procesando de a varios píxeles a la vez no implica necesariamente grandes cambios en el desempeño temporal total del algoritmo. Entonces, si bien la implementación en ASM resulta más eficiente, la diferencia entre ésta y la implementación en C del filtro de Nivel no es muy grande.

Como podemos observar en el gráfico de la figura 2, hay una diferencia abismal entre la implementación del filtro de Bordes en ASM y la de C; la implementación de C demoró aproximadamente 80 % más ciclos de clock que la de ASM. Efectivamente, en este caso utilizar SIMD presenta una gran ventaja al momento de procesar datos de forma repetitiva. La implementación del filtro en ASM procesa de a 8 píxeles a la vez mientras que la implementación en C procesa de a un píxel. Luego, pensaríamos que la implementación en C al realizar 8 veces más accesos a memoria y efectuar las operaciones correspondientes a cada píxel de manera individual, demoraría aproximadamente 8 veces la cantidad de ciclos de clock insumidos por la implementación de ASM. Podemos ver que esta primera intuición se contradice con los resultados obtenidos ya que éstos muestran una diferencia considerable a favor de la implementación del filtro en ASM pero no tan marcada como la esperada. Esto podría ser resultado de la optimización realizada por el flag O3; probablemente si comparásemos la implementación de ASM con la implementación de C utilizando el flag de optimización O0 la relación entre los ciclos de clock insumidos se parecería más a nuestra hipótesis.

Finalmente, la implementación en C del filtro de Rombos tomó en promedio un 414 % más de ciclos de reloj que la implementación en ASM. Este es entonces el filtro que presenta la diferencia más significativa entre las dos implementaciones. Podemos explicar este resultado a partir de la gran cantidad de operaciones que son realizadas sobre cada píxel para aplicar el filtro. Entonces al procesar de a cuatro píxeles con ASM se logra optimizar la cantidad de veces que estas cuentas son realizadas. Comparando

la implentación en ASM con la del filtro de Bordes, la cantidad de cuentas dentro del ciclo del filtro de Rombos es mayor a la del filtro de Bordes. Podemos justificar entonces porqué la diferencia entre las implementación en C y en ASM del filtro Rombos es mayor que la del filtro de Bordes.

En conclusión, implementar los filtros en ASM parece en cada caso optimizar el desempeño temporal de los filtros, reflejado en la cantidad de ciclos de clock. La diferencia entre la cantidad de ciclos de clock insumidos por la implementación de ASM y la de C varía según el filtro. Esto se explica por la cantidad de operaciones realizadas sobre cada píxel de la imagen y la cantidad de píxeles que son procesados en paralelo en la implementación en ASM.

3.2. Bordes

3.2.1. Parámetros

El filtro ”Bordes” modifica cada píxel de la imagen en función de sus píxeles cercanos, es decir que el resultado de cada píxel depende de los píxeles vecinos correspondientes a la imagen pasada por parámetro. Podemos preguntarnos entonces: ¿el desempeño de la función depende de la imagen a la cual se le aplica el filtro? El parámetro tiene dos características relevantes que podemos hacer variar para intentar responder a esta pregunta: su tamaño y las componentes de sus píxeles.

Hipótesis

El filtro realiza la misma cantidad de operaciones sin importar el valor de los píxeles. Por este motivo suponemos que aunque el valor por el cual el píxel es modificado depende de la imagen pasada por parámetro, el desempeño de la función no varía. Esperamos también que el tiempo que tarda el ciclo del algoritmo al procesar una imagen sea proporcional al tamaño de la imagen. Sin embargo, como al aumentar el tamaño de la imagen los costos fijos representan una proporción cada vez menor, suponemos que la cantidad de ciclos de reloj consumidos representarán una proporción cada vez menor respecto de la cantidad de píxeles. Es decir, que como ya mencionamos al principio de esta sección, creemos que las imágenes mayores tendrán la ventaja de una amortización en los costos fijos.

Experimentos

Para contrastar nuestras hipótesis con resultados experimentales realizamos dos experimentos.

El primero consiste en comparar la cantidad de ciclos de clock de la función con una foto blanca como parámetro y con una foto con filtro ”ruido”. Estos dos parámetros buscan representar dos opuestos; la foto blanca no presenta bordes y la otra solo tiene bordes. Comparamos estos dos extremos con la imagen ”Wargames” que nos brindó la cátedra. Esta última imagen sería un ejemplo de imagen promedio, es decir, no perteneciente a ninguno de los extremos. Observamos los resultados:

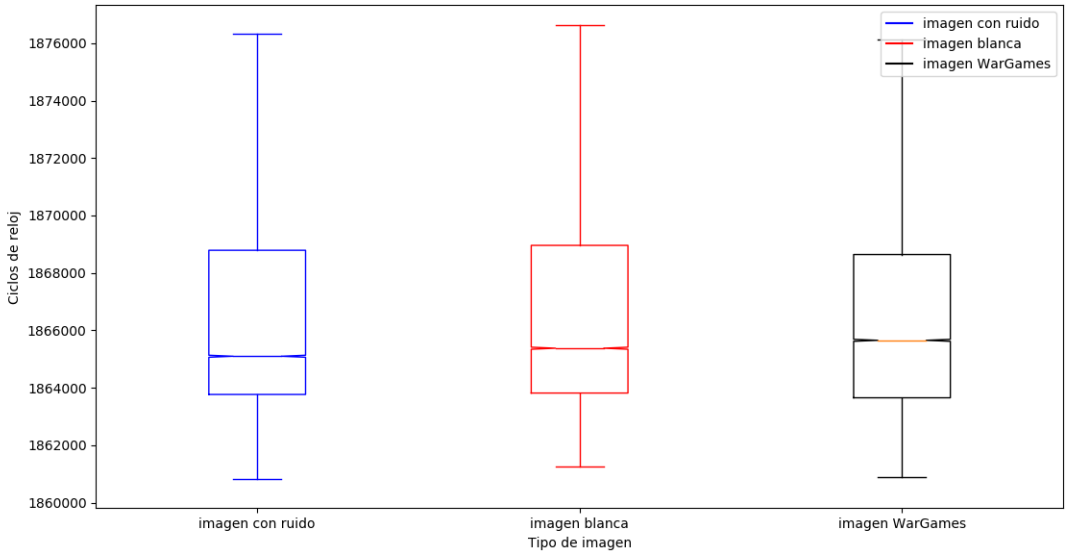

Figura 4: Experimento contenido de la imagen

El segundo experimento consiste en correr el filtro con cinco tamaños diferentes: 128x75 (9600 píxeles), 256x150 (38400 píxeles), 512x300 (153600 píxeles), 1024x600 (614400 píxeles) y 2048x1200 (2,457 megapíxeles). Para cada uno de los tamaños corrimos cinco fotos distintas generadas de forma aleatoria para hacer el experimento más representativo. Promediamos entonces la información obtenida con las distintas imágenes para cada tamaño y obtuvimos los siguientes resultados:

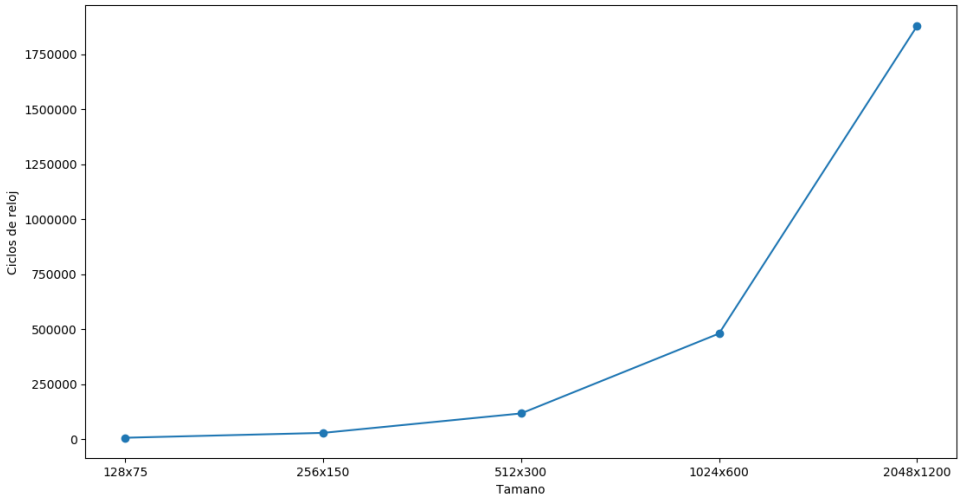

Figura 5: Experimento tamaño de la imagen

Análisis de resultados y conclusiones

Observamos gracias a la figura 4 que el contenido de la imagen no repercute en el tiempo de ejecución al aplicarle el filtro. Esto coincide a lo supuesto y se debe a que el código del filtro no está ramificado en función del contenido de los píxeles; no importa el parámetro pasado, las instrucciones ejecutadas son las mismas.

Respecto a la segunda parte del experimento, cada foto posee el cuádruple de píxeles que la anterior. Analizando los resultados exhibidos en el gráfico de la figura 5, vemos que el promedio del aumento de la cantidad de ciclos de clock insumidos por la corrida con cada imagen es aproximadamente un 290 %. Si observamos el crecimiento en función del primer tamaño vemos que fue de la siguiente manera:

	Imagen 2	Imagen 3	Imagen 4	Imagen 5
Proporción en tamaño	4	16	64	256
Proporción en cantidad de ciclos	3,9	15,5	63	247,4

Cuadro 2: Proporciones en función de la primer imagen

Como podemos ver en el cuadro 2, el tercer tamaño consume 15,5 veces más ciclos de reloj que el primero, el cuarto 63, y el último 247,4. En este sentido, parece comprobarse nuestra hipótesis ya que podemos ver que las proporciones de las cantidades de ciclos de reloj parecen ir disminuyendo en función de las proporciones en tamaño. Es decir que podemos inferir que los costos fijos efectivamente se van amortizando con el aumento de la cantidad de píxeles.

Teniendo en cuenta estas cuestiones podemos afirmar que la decisión sobre qué tamaño de imagen utilizar al experimentar resulta relevante, no así el contenido de las imágenes sobre las cuales se experimenta. A partir de los resultados experimentales, comprobamos que el costo fijo del experimento perjudicará más la medición que el riesgo mayor de ruido que corren las imágenes más grandes por lo que decidimos a partir de este experimento correr el resto de los experimentos con imágenes grandes. Además, tomamos imágenes de tamaño 2048x1200, este representa el tamaño habitual de una imagen hoy en día, luego resulta lógico querer evaluar la performance de nuestros filtros sobre éstas.

3.3. Nivel

Buscamos determinar mediante la experimentación con diferentes implementaciones en ASM del filtro de nivel los costos en términos temporales de distintos procedimientos o recursos puestos en juego al implementar funciones en este lenguaje.

3.3.1. Saltos condicionales

Como primer paso nos cuestionamos si los saltos condicionales implican un costo considerable en lo que es el tiempo de ejecución de la función, medido en ciclos de clock.

Hipótesis

Esperamos que el tiempo de ejecución de la implementación del filtro con el ciclo desarrollado sea menor a la implementación original. Esto sería producto de ahorrarnos el costo particular de realizar un salto. El procesador, para optimizar la cantidad de ciclos de clock utilizados para realizar el fetch, decode, execute de una instrucción, realiza lo que se denomina como pipeline. El pipelining consiste en superponer en el tiempo la ejecución de varias instrucciones a la vez. De esta forma los distintos bloques del procesador trabajan en paralelo y de forma simultánea con distintas instrucciones. Un salto condicional representa una discontinuidad en el flujo de ejecución, por lo que es considerado un obstáculo

en esta optimización que se caracteriza por buscar instrucciones en secuencia. El salto puede hacer que todas, o muchas de las instrucciones que se encontraban preprocesadas deban descartarse.

Experimentos

Para analizar el costo que generan los saltos en el código desarrollamos el ciclo una vez, reduciendo por la mitad la cantidad de iteraciones del ciclo, por lo que se producen la mitad de saltos condicionales. No pudimos seguir desarrollándolo ya que el ancho de la imagen es múltiplo de 8, entonces si levantábamos más de 8 píxeles por iteración podíamos tener accesos inválidos a memoria. Para realizar una comparación ”justa”, buscamos tener la misma cantidad de instrucciones dentro ciclo en ambas implementaciones, luego si hay diferencias en los costos temporales van a deberse únicamente al costo de la instrucción de salto. Para lograr esto incluimos una instrucción innecesaria en la implementación con el ciclo desarrollado.

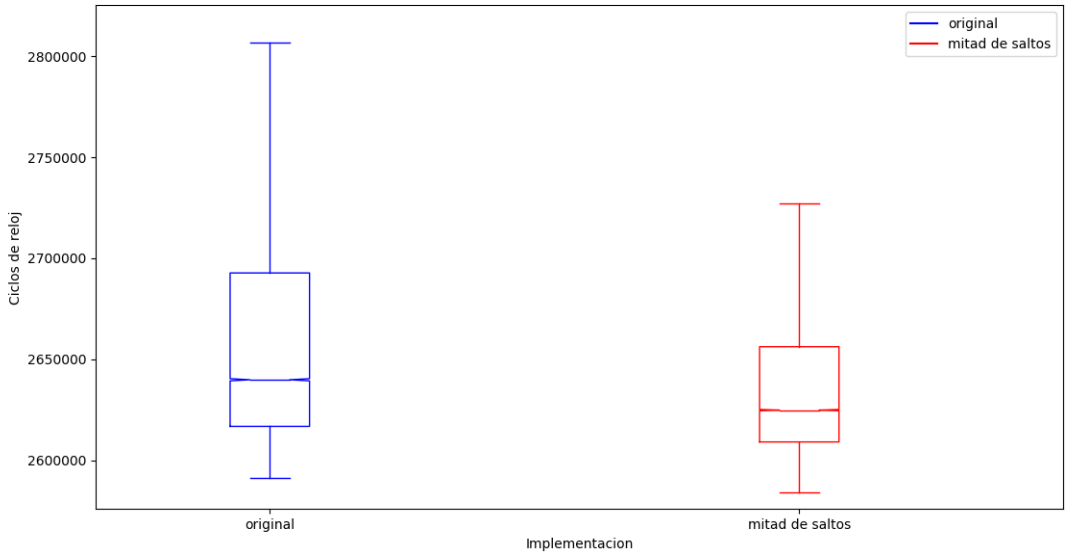

Figura 6: Experimento saltos condicionales

Análisis resultados y conclusiones

Podemos observar en el gráfico que la implementación original lleva en promedio más ciclos de clock que la nueva implementación que cuenta con la mitad de saltos condicionales. Sin embargo, esta diferencia de las medias es bastante pequeña; más específicamente, al utilizar la mitad de los saltos se consume un 0,6 % menos de ciclos de reloj que la implementación original. Sin embargo, si tenemos en cuenta la varianza y analizamos los valores máximos que se obtuvieron con cada implementación, podemos ver que la diferencia es un poco mayor (2,7 %). Ambas implementaciones tienen la misma cantidad de instrucciones en sus ciclos, luego la diferencia en la cantidad de ciclos de clock promedio que llevan se produce únicamente por el número de saltos condicionales totales que realizan. Nuestra hipótesis sobre el costo considerable de las instrucciones de saltos condicionales parece verificarse, aunque en menor medida de lo que creíamos. Este costo puede justificarse, como ya explicamos, por la obstaculización que implica un salto condicional en el proceso de pipelining.

3.3.2. Llamado a funciones

Vimos que los saltos implican un costo en cuanto al tiempo de ejecución de una función, ahora bien, ¿qué sucede con los llamados a funciones?

Hipótesis

Los llamados a funciones implican pushear la dirección de retorno a la pila, luego realizar un salto a la dirección a la cual hace referencia la instrucción ”call”. Al finalizar la función llamada, se popea la dirección de retorno y se efectúa un nuevo jump a ésta. La desventaja de hacer llamados a funciones entonces, además de pagar el costo asociado a realizar un salto que estudiamos anteriormente, es que al trabajar con la pila se está trabajando con la memoria. Por estas razones esperamos que sea más ineficiente en cuanto al tiempo.

Experimentos

Para estudiar esto hicimos tres nuevas implementaciones del filtro de nivel:

Modificación 1: el ciclo hace un llamado a una función que efectúa las operaciones que hace el ciclo en la implementación original.

Modificación 2: el ciclo llama a tres funciones distintas que efectúan cada una una operación simple realizada por ciclo en la implementación original.

Modificación 3: el ciclo llama a tres funciones distintas, donde éstas realizan a su vez dos llamados adicionales cada una, y la última función que es llamada es la que se encarga de realizar una operación del ciclo.

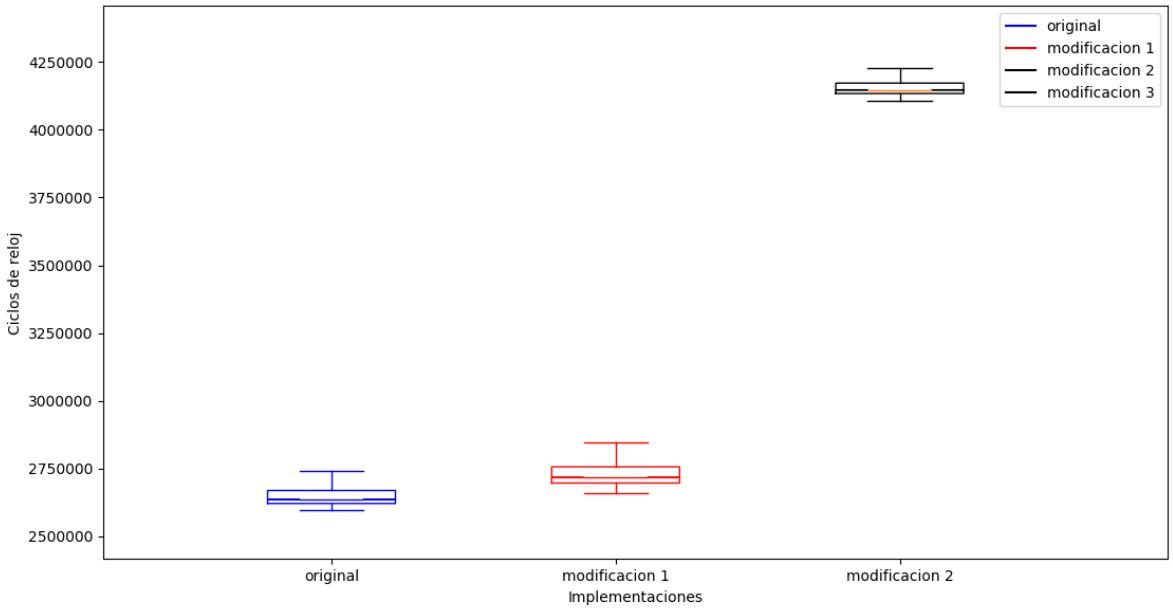

Figura 7: Experimento llamado a funciones

En la figura 7 no incluimos la representación de la modificación 3 ya que la cantidad de ciclos de clocks insumidos por ésta era tanto mayor a la de las otras modificaciones que se iba de la escala.

En la siguiente tabla comparamos el promedio de ejecutar 3 veces 10000 corridas de cada modificación y la implementación original del filtro:

	Original	Modificación 1	Modificación 2	Modificación 3
Cantidad de ciclos de clock promedio	2,6.10^6	2,7.10^6	3,6.10^6	10.10^6

Cuadro 3: Promedio de la cantidad de ciclos de clock efectuados por cada implementación

Análisis resultados y conclusiones

Podemos ver en el cuadro cómo crece notablemente la cantidad de ciclos de clock al realizar más llamados a funciones dentro del ciclo. Estudiando la tabla podemos observar que la modificación 1 consume un 3,85 % de ciclos de clock más que la original, la 2 un 38,46 % y la 3 un 284,62 %. Podemos concluir entonces que llamar a funciones representa un gran costo, lo que coincide con lo que supusimos inicialmente. Sin embargo, esto era predecible ya que anteriormente verificamos que efectuar saltos es costoso. Luego, como llamar a una función implica un jump, se deduce que llamar a funciones perjudica el desempeño temporal de una función. Decidimos entonces realizar un nuevo experimento para poner en evidencia de forma más explícita el gasto temporal en la ejecución de una función producido por el llamado a otras funciones.

Buscamos contrastar el costo de ejecutar un jump con el de realizar un llamado a función. Como los llamados modifican la pila, esperamos que estos insuman una mayor cantidad de ciclos de clock. Para experimentar esta cuestión, alteramos la modificación 1 para que en lugar de realizar un llamado a una función que realiza las operaciones del ciclo efectúe un salto a la etiqueta donde se encuentran las operaciones y luego otro salto para volver al ciclo. Alcanzamos los siguientes resultados:

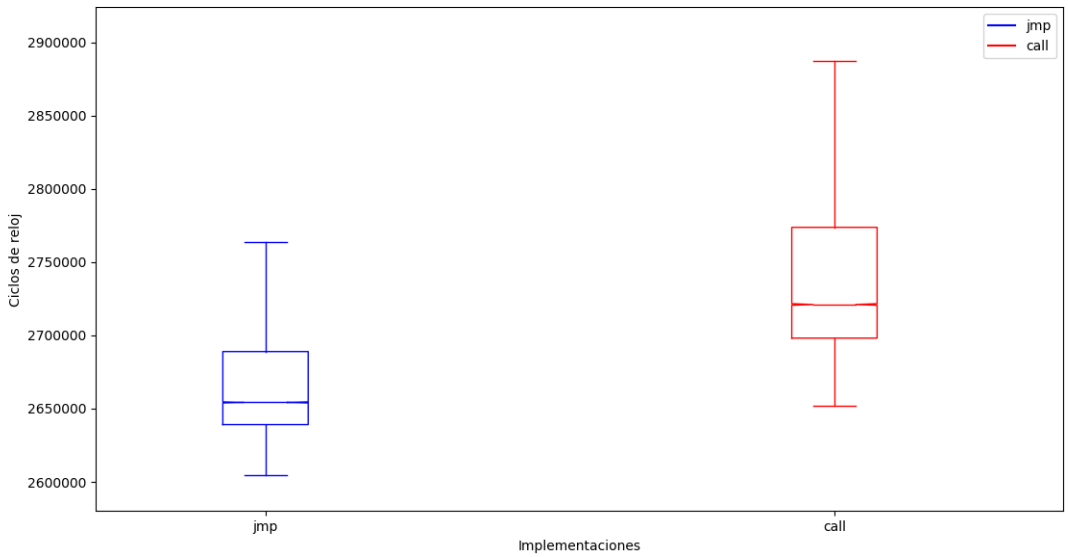

Figura 8: Experimento llamado a función vs saltos

Constatamos que en concordancia con nuestra conjetura, los llamados a función implican un mayor costo temporal que el implicado por los saltos. Esto se explica por la interacción con la pila, y por ende con la memoria, que conlleva el call a una función al pushear primero la dirección de retorno y luego popearla para regresar. Además los llamados a función implican un salto para ir a la función llamada y otro para volver. No obstante, la diferencia no es muy significativa; la implementación del filtro que utiliza llamados a funciones consumió un 2,6 % más ciclos de clock que la implementación que utiliza jumps. Podemos inferir entonces que el costo de la modificación de la pila es despreciable en comparación con los demás costos involucrados en la ejecución.

3.4. Rombos

Para el filtro ”Rombos” decidimos llevar a cabo experimentos centrados en la cuestión del acceso a memoria. Buscamos responder a las siguientes preguntas; ¿cuán costoso resulta el acceso a memoria dentro del ciclo principal de una función? ¿Es significativa la diferencia en cuanto a la performance temporal del acceso a memoria de datos alineados en comparación con la del acceso a datos desalineados?

3.4.1. Máscaras (no) alineadas

Cuando nos encontramos con la implementación del filtro de rombos, vemos que usa una gran cantidad de máscaras, al menos en comparación con los otros dos filtros. Éstas están predefinidas y escritas en memoria en la sección *.rodata* con sus respectivas etiquetas, para luego ser levantadas de memoria durante el procesamiento de los píxeles. Para este evento INTEL nos provee de dos instrucciones (entre varias), una que tiene como precondición que los datos a levantar de memoria estén alineados a dieciséis bytes mientras que la otra no, MOVDQA y MOVDQU respectivamente. Por esta razón se nos ocurrió experimentar con estas cuestiones: ¿Qué instrucción es más rápida? ¿Siempre se puede alinear la memoria? ¿Qué tan difícil es alinearla?

Hipótesis

Al ver que MOVDQA tiene una precondición, suponemos que tiene un trabajo de arquitectura que hace que el pasaje de la memoria al procesador sea más rápido, mientras que si uno levanta 16 bytes de memoria sin estar alineada, creemos que puede llegar a significar más de un ”uso” del bus. Esto se debe a que el bus siempre levanta memoria de forma alineada, por lo que para levantar datos que no se encuentran alineados va a tener que buscarlos más de una vez. En cuanto a cuándo podemos alinear la memoria y cómo, suponemos que simplemente tendremos que dejar unos bytes con ”basura” de tal manera que podamos guardar las máscaras de manera alineada.

Experimentos

Para comparar estas dos instrucciones vamos a tener dos implementaciones diferentes: una con las máscaras escritas de forma alineada que se aproveche de esto y utilice MOVDQA, y otra sin aclaraciones ni suposiciones sobre la alineación. A continuación observaremos los resultados obtenidos.

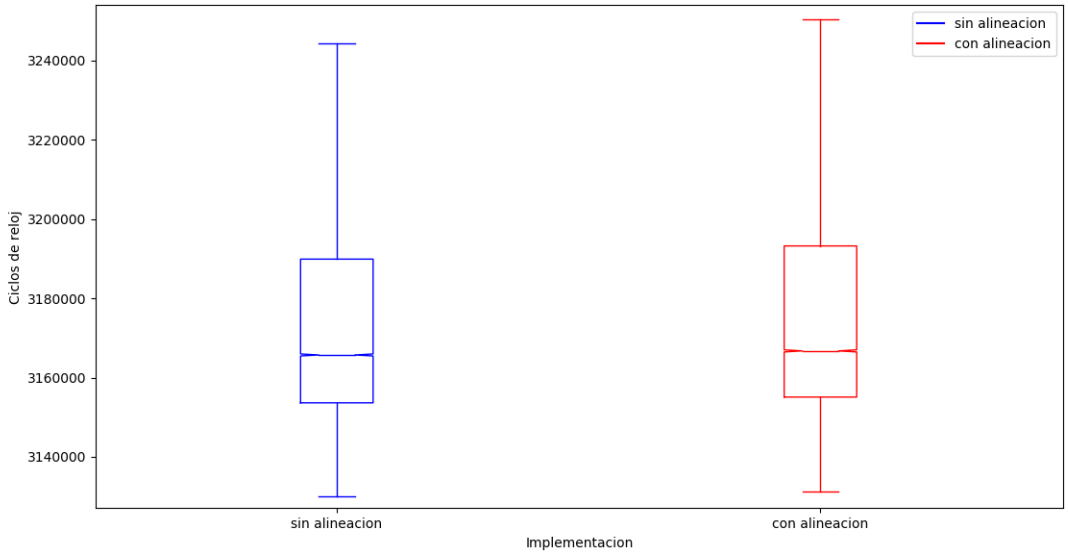

Figura 9: Experimento sobre la alineación de las máscaras

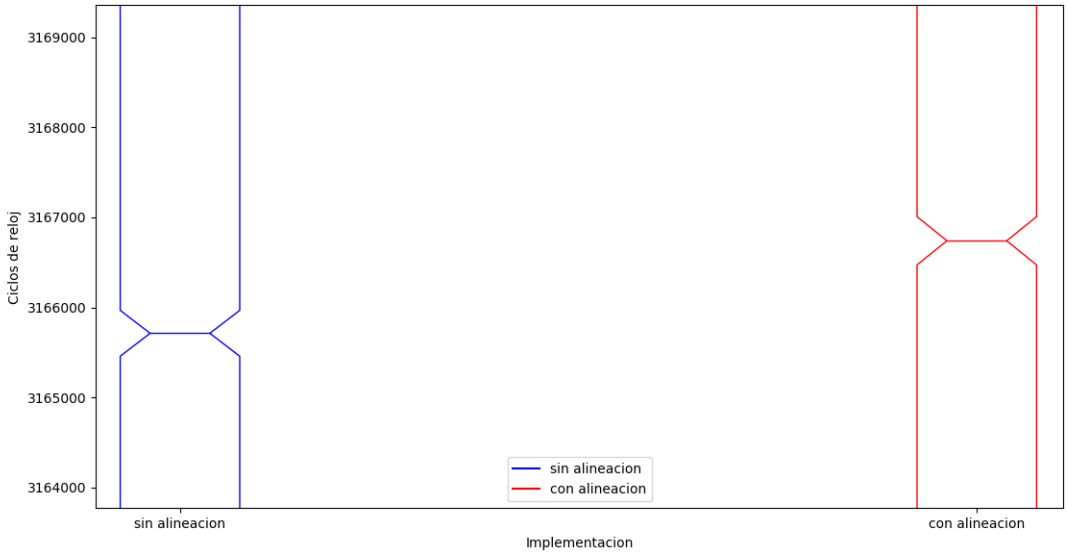

Figura 10: Medias de los resultados del experimento sobre la alineación de las máscaras

Análisis de resultados y conclusiones

Podemos observar en el gráfico que los resultados no fueron los esperados: de la figura 9 podemos concluir las dos implementaciones tardaron en general cantidades similares de ciclos de reloj, aunque si miramos con más atención, en la figura 10 vemos que la media de la implementación en la que la memoria se deja sin alinear es ligeramente menor que la que sí alinea la memoria. Sin embargo, esta diferencia no resulta significativa a fines de este análisis del tiempo de ejecución ya que siempre debemos tener en cuenta que el procesador podría estar realizando otras operaciones mientras se ejecuta nuestro programa.

Una razón por la cual pudo haber ocurrido que la alineación de la memoria no haya sido de gran influencia en la cantidad de ciclos de clock es que podría haber estado alineada, en cuyo caso no cambiaría la instrucción *"align 16"* al principio del código. Otra razón es que únicamente movemos datos de memoria a un registro al principio de la función, antes de iniciar el ciclo, y como traer cosas de memoria ya representa un costo grande en sí, la alineación no puede no ser un cambio relevante.

3.4.2. Accesos a memoria dentro y fuera del ciclo

Queremos analizar ahora el costo del acceso a memoria; ¿levantar datos en cada iteración representa un obstáculo significativo para la optimización temporal de una función?

Hipótesis

Suponemos que la cantidad de ciclos de clock de una función que accede a memoria dentro de su ciclo principal es mayor a una que realiza los accesos fuera del ciclo. Esto se debe no únicamente a que el número de instrucciones dentro del ciclo se ve incrementado por las operaciones de mover de memoria a algún registro, si no también a que el acceso a memoria representa un gran costo temporal, más aún si los datos buscados no se encuentran en la memoria cache.

Además, los accesos a memoria perjudican el buen funcionamiento del pipeline ya que en el caso de que se necesite un operando de memoria en la decodificación de una instrucción, el acceso a memoria para traer este operando interferirá con el fetch de la siguiente instrucción. De esta forma, en cada iteración la optimización buscada con pipelining se ve obstaculizada. Es decir que los accesos a memoria son incompatibles con el funcionamiento paralelo de los distintos bloques del procesador.

Experimentos

En la implementación original del filtro "Rombos" las máscaras son levantadas de memoria y almacenadas en registros XMM antes de iniciar el ciclo. Con el objetivo de analizar las ventajas o desventajas que presentan las distintas formas de levantar las máscaras creamos una nueva implementación del filtro, en la cual se levantan las máscaras en cada iteración del ciclo. Queremos comparar las dos implementaciones de forma tal que los cambios en eficacia temporal sean producto únicamente de la diferencia de acceso a las máscaras. Por este motivo comparamos la implementación que levanta las máscaras en el ciclo con la implementación del filtro original con 6 instrucciones adicionales en el ciclo. Estas instrucciones no modifican el funcionamiento del ciclo.

Comparamos entonces la performance de las dos implementaciones obteniendo los siguientes resultados:

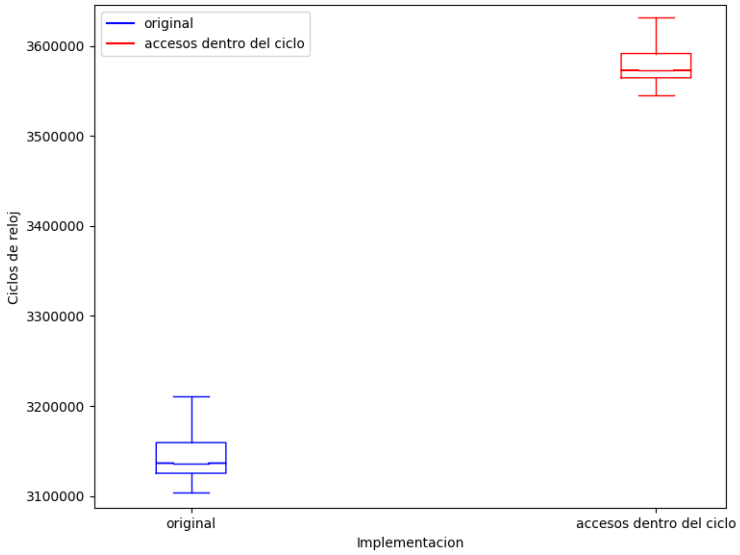

Figura 11: Resultado de levantar memoria en cada iteración del ciclo

Análisis de resultados y conclusiones

Como muestra la Figura 11, la implementación en la que se traen datos de memoria en cada una de las iteraciones del ciclo siempre está por arriba de la que trae de memoria una única vez antes de comenzar el ciclo, en lo que refiere a cantidad de ciclos de reloj. Es decir, no solo se confirmó nuestra hipótesis inicial de que acceder una mayor cantidad de veces a memoria implica una mayor cantidad de ciclos de reloj, si no que además podemos observar que este incremento en el tiempo que tarda en ejecutarse es relativamente alto: la implementación con accesos a memoria dentro del ciclo consume aproximadamente un 14 % más de ciclos de reloj que la que levanta de memoria una única vez. Esto se genera debido a que los accesos a memoria representan uno de lo que más tiempo de manda en cuanto a características de un programa (por eso la memoria cache es un tema de gran relevancia, ya que permite reducir este costo). En conclusión, por más que los datos puedan llegar a encontrarse en la memoria cache, igualmente acceder a los datos que se encuentran en memoria (ya sea en la principal o en la cache) significa un gran costo para el tiempo de ejecución.

4. Conclusión

A lo largo del trabajo estudiamos las implementaciones de distintos filtros para imágenes que trabajan con SIMD, es decir que procesan datos de forma simultánea. Analizamos cómo y en qué medida diferentes factores influyen en el tiempo de ejecución de los códigos.

Luego de realizar las experimentaciones presentadas en este informe, podemos concluir que hay ciertas características de los programas que afectan su desempeño temporal en mayor medida, como lo son el hecho de que haya sido implementado en C o en ASM, la cantidad de accesos a memoria o la cantidad de saltos.

En un primer lugar estudiamos la diferencia en la cantidad de ciclos de clock insumidos por la ejecución de los filtros al ser implementados en C o en assembler. Vimos que la implementación en C tardó en general mucho más que la de ASM. Esto en parte se debe a la forma en la que se compilan los códigos pero, en mayor medida, se debe a que las instrucciones de SIMD nos permiten procesar varios datos simultáneamente, mientras que en C modificamos de a un dato por iteración. Sin embargo, a pesar de esta gran ventaja a favor de assembler, podemos preguntarnos si no presenta a su vez ciertos inconvenientes en comparación con C. Siendo assembler más cercano a la arquitectura de la computadora, la forma de codear se vuelve menos humana. Con esto nos referimos a que se diferencia de C u otros lenguajes similares al tener un conjunto de instrucciones menos intuitivas de utilizar. En este sentido entonces, podríamos decir que C tiene la ventaja de poder ser comprendido con unas pocas leídas por lo que es más simple el mantenimiento del código o modificaciones del mismo. A esto podría argumentarse que si un código se encuentra lo suficientemente comentado, su comprensión debería resultar relativamente fácil sin importar el lenguaje en el que fue implementado. Esto podría ser verdad, pero no quita el hecho de que modificar un programa hecho en assembler es más complejo que modificar uno en C, al tener que agregar muchas más instrucciones (es decir, lineas) para lograr lo mismo que podría hacerse en unas pocas lineas en C.

En conclusión, podemos decir que assembler presenta una ventaja significativa sobre C respecto al tiempo de ejecución consumido por un programa. Sin embargo, si tenemos un programa relativamente simple como es el caso del filtro de Nivel, no se justifican los costos de implementación y mantenimiento implicados por assembler. Si queremos optimizar el desempeño temporal de un filtro como Rombos en cambio, implementarlo en assembler supone una gran ventaja debido a la cantidad de operaciones que se le aplican a cada píxel.

Por otro lado, analizando el costo temporal de los saltos condicionales y, comparándolo con el costo de los llamados a funciones, pudimos darnos cuenta de que efectivamente cambiando la implementación a una en la que se realizaban la mitad de los saltos se reducía considerablemente la cantidad de ciclos de clock. Esto no nos sorprendió ya que suponíamos que era un factor que generaría cambios en la cantidad de ciclos de reloj que lleva la ejecución al influir en gran parte en el proceso de pipelining.

También pudimos constatar que en algunos filtros implementados el desempeño temporal no depende de la foto en sí y que existe una fuerte relación entre el tamaño de la imagen y el costo temporal de aplicar el filtro, sin embargo esta relación no es lineal debido al costo fijo implicado por la ejecución del filtro.

Comparando los experimentos realizados pudimos observar que no solo cuantos más accesos a memoria, mayor es el tiempo de ejecución si no que también es, dentro de las características analizadas, una de las que más influyen en la performance del programa. Esto también fue coherente con lo que creíamos ya que la optimización de los accesos a memoria es un tema de interés: es uno de los incentivos que llevaron a incluir la memoria cache en las computadoras.

Podemos entonces concluir que utilizar herramientas como SIMD en un lenguaje como assembler puede resultar muy útil en numerosas situaciones. En particular al procesar una gran cantidad de datos que son procesados de igual manera, como es el caso de los archivos multimedia. Con el objetivo de optimizar el desempeño temporal de los algoritmos resulta fundamental tener en cuenta los distintos procedimientos e instrucciones que estuvimos analizando a lo largo del trabajo. Sin embargo, algunos de estos conceptos, como por ejemplo el pipelining, pueden ser considerados no únicamente al programar en lenguajes de tan bajo nivel como assembler. De esta forma, saber qué tenemos en la caché o evitar en lo posible discontinuidades en el flujo de ejecución puede optimizar nuestros programas más allá del lenguaje utilizado.