DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 2

Segundo cuatrimestre de 2018 — Organización del computador 2

Integrante	LU	Correo electrónico
Nicolás Hertzulis	811/15	nicohertzulis@gmail.com
Cynthia Liberman	443/15	cynthia.lib@gmail.com
María Belén Ticona	143/16	ticona.belu@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, se realiza un estudio comparativo entre la eficiencia del procesamiento vectorial de filtros de imágenes en lenguaje ensamblador vs su implementación en el lenguaje de alto nivel C, el cual procesa la información individualmente. Para ello, se estudió la relación de tiempos de ejecución con respecto al tamaño de las imágenes procesadas.

A partir de los resultados obtenidos, se puede afirmar que ASM optimiza la performance gracias al paralelismo bajo el modelo SIMD. Además se analizaron comparativamente la complejidad de los filtros entre sí, teniendo en cuanto qué tipos de datos operan y las lecturas de memoria realizadas. Como resultado, los filtros que operan con datos flotantes y realizan lectoras a memoria con mayor frecuencia obtienen menor eficiencia en ambos lenguajes.

Índice

> The monkey's sneakiest trick is when there are no deadlines... Everyone is procrastinating on something
>
> *Tim Urban*

1. Introducción

1.1. Marco teórico

En computación, un procesador vectorial es una unidad central de proceso que implementa un conjunto de instrucciones que operan sobre vectores (arreglos de una dimensión de tamaño fijo), a diferencia del procesamiento escalar el cual se caracteriza por instrucciones que operan únicamente sobre datos individuales.

Algunos programas informáticos pueden ser implementados en lenguaje ensamblador utilizando exclusivamente instrucciones escalares o utilizando también instrucciones vectoriales. Para estos programas, el procesamiento vectorial constituye una optimización, dado que se percibe un incremento en la performance (i.e. tiempo neto de procesamiento).

Por otro lado, los compiladores modernos de lenguajes de alto nivel, como GCC, incluyen la posibilidad de realizar diversas optimizaciones apreciables en el código ensamblado, entre las cuales se encuentra la utilización de instrucciones de procesamiento vectorial (siempre que el compilador sea capaz de detectar la posibilidad de su empleo).

El presente estudio resulta importante porque a pesar de que la velocidad de procesamiento y la velocidad de acceso a memoria aumentan a lo largo de los años, el ritmo de crecimiento del nivel de procesamiento se ha incrementado considerablemente con respecto a la velocidad del acceso a memoria [1]. Esto significa que la brecha es cada vez mayor y se traduce en un costo porcentual creciente de los accesos a memoria sobre el costo total de procesamiento de un programa.

Asimismo, esta investigación de caracter educativo es importante para programadores que se desempeñan en la academia o en la industria porque motiva a la optimización del código, a la medición de performance de los programas y a la profundización del conocimiento de lo que sucede en el procesador en la ejecución de programas escritos en lenguajes de alto nivel.

1.2. Objetivos

En el siguiente trabajo práctico se aplica el modelo de procesamiento vectorial `SIMD` (*Single Instruction, Multiple Data*) para el procesamiento de imágenes mediante filtros. Los objetivos propuestos para el mismo son:

- Implementar filtros gráficos en lenguaje ensamblador bajo el modelo `SIMD`.
- Realizar un estudio comparativo de la performance de dichas implementaciones en relación a sus correspondientes en el lenguaje de alto nivel C.

2. Desarrollo

La implementación de los filtros se realizó en lenguaje ensamblador x86_64. Para la misma se tuvieron en cuenta las siguientes pautas generales para obtener una mayor eficiencia temporal:

- Optimizar el paralelismo en el procesamiento de pixels.
- Minimizar las lecturas de memoria realizadas (especialmente dentro de los ciclos).
- Efectuar juntas las lecturas de posiciones de memoria adyacentes, considerando el principio de localidad espacial en el uso de memoria caché.

- Emplear las instrucciones más específicas posibles de acorde a su propósito, teniéndose en cuenta que a menor poder de una instrucción le corresponden menos ciclos de reloj y por tanto menor tiempo de procesamiento (por ejemplo, evitar uso excesivo de *shuffles*).

Los filtros con los cuales se trabajó son: *Tres Colores*, *Efecto Bayer*, *Cambia Color* y *Edge Sobel*. A continuación se describen los detalles de sus respectivas implementaciones en lenguaje ensamblador (véase el enunciado para mayor detalle de cada filtro].

2.1. Consideraciones de Implementación

Exceptuando el filtro *Edge Sobel*, los demás operan recibiendo y retornando una imagen a color. Para especificar la graduación de color de cada píxel se tienen las siguientes cuatro componentes: B (blue), G (green), R (red) y A (alpha), todas en el rango de valores entre 0 y 255. Cada una en representación binaria ocupa un byte, por lo que cada píxel tendrá un tamaño de 4 bytes en total.

Las imágenes se encuentran almacenadas como porciones de píxeles contiguos en memoria y se las trata como matrices a nivel esquemático. La modificación de los pixeles originales dada una serie de pautas para cada filtro es lo que dará la imagen resultante. Cabe resaltar que los píxels se almacenan de izquierda a derecha y sus componenten se guardan en el siguiente orden: B, G, R, A.

2.2. Filtro Tres Colores

El filtro *Tres Colores* se caracteriza por modificar el color de un píxel en función de su brillo, trabajando sobre cada uno de forma independiente del resto de la imagen. Para poder determinar la coloración de cada píxel destino, se procedió a efectuar una combinación lineal dada por el brillo y un nuevo color de entre las siguientes opciones: rojo, verde y crema.

2.2.1. Pre-proceso de imagen

En primer lugar, el algoritmo implementado guarda en memoria tanto las componentes de los nuevos colores así también como tres constantes necesarias para determinar qué combinación lineal aplicar en cada píxel. De esta forma, antes de proceder a procesar la imagen se resguardan estas constantes definidas en `.RODATA` en registros `XMMn`, de modo que ya se encuentren disponibles en cada iteración -resultando innecesario buscar sus valores a memoria-.

En cuanto al procesamiento de la imagen, en cada iteración se trabaja con 4 píxels dado que el tamaño de cada uno es de 4 bytess -entrando 16 bytesss en un registro `XMMn`-. La finalización de la ejecución del ciclo se fija mediante una comparación entre la dirección a leer (resguadado en RDI) y la dirección inmediatamente posterior al final de la imagen. Esta última se determina sumando a la dirección fuente de la imagen, el tamaño de las filas multiplicado por la altura de la imagen en pixels: $src + row\_size * height$.

2.2.2. Iteración

Por cada ciclo el procesamiento consta de las siguientes partes:

1. Cálculo de Brillo
2. Máscaras según intensidad de Brillo
3. Obtención de Pixels Destino por Combinación Lineal
4. Inserción de Transparencia

En primer lugar, para iniciar con el *Cálculo de Brillo* se obtienen de los 4 pixels levantados las componentes de cada color por separado, de modo que ocupen un Double Word cada una. Para ello se emplea un shift lógico empaquedado de Double Word dado que el mismo completa con ceros las posiciones redefinidas, quedando inalterado el valor de cada componente. De esta forma se obtiene un registro XMM por cada set de bytess rojo, verde y azul de los 4 píxels en proceso. Luego, se procede a sumar de a Word

dichos registros y a convertir los enteros obtenidos a float para su división por 3. Comos los datos en las imágenes son enteros y dado que para finalizar el cálculo de brillo resta tomar parte entera inferior del resultado obtenido, no resulta necesario una precisión mejor que un *single-precision floating-point*.

Para la obtención de las *Máscaras según intensidad de Brillo* se trabaja por comparación entre las constantes reservadas (véase Pre-proceso de imagen) y los brillos obtenidos mediante las instrucciones pcmpgtd y pcmpeqd. Las mismas setean en 1 en la Double Word donde se guarda cada brillo en caso de cumplirse la condición de mayor o igual según lo que corresponda en cada caso. En particular cabe resaltar que para el caso $85 < W \leq 170$, se efecúa un OR lógico entre las máscaras de bits de las condiciones $85 \geq W$ y $W > 170$, negando la obtenida y quedando así la máscara de brillos de comprendidos entre 85 y 170 inclusive.

En cuanto a la obtención de los píxels destino, primero se filtró cada color resguardado (rojo, verde y crema) según se cumpliese o no su condición de intensidad de brillo correspondiente. Como la combinación lineal resulta de tomar 3/4 de cada color y sumarle 1/4 del correspondiente brillo, se procedió efectuando la división por 4 al final para evitar pérdida de precisión. Ahora bien como al realizar la multiplicación por 3 de los colores se puede producir *rollover*, se extendió cada componente de los colores de tamaño bytes a Word. Es por ello que resulta necesario desdoblar también el procesamiento de los 4 píxels para realizarlo en partes Low y High (cada una con 2 píxels fuente). Tanto para la multiplicación por tres de los colores como para su suma con el brillo se trabaja con la intrucción ADDW, habiendo previamente efectuado un *merge* entre las partes Low y High de los colores. Una vez realizada la división por 4 de cada Word mediante un shift lógico a la derecha, se empaquedan ambas partes volviendo a trabajar con 4 píxels en un solo registro.

Resulta importante remarcar que en todo momento la componente correspondiene a la transparencia se encuentra con valor 0. Para que su valor sea 255 se setean todos los bits en 1 mediante una comparación de un registro con sí mismo y se realiza un shift lógico hacia la izquierda para setear en 0 los 3 bytess menos significativos de cada Double Word. Finalmente se realiza un OR para terminar de obtener los píxels destino, combinando así transparencia y colores obtenidos.

2.3. Filtro Efecto Bayer

El filtro de *Efecto Bayer* toma la imágen original donde cada pixel es una combinación de graduaciones de los colores rojo, verde y azul y aplica sobre la misma un filtro con un patrón que respeta el orden mostrado en la Figura 1 Cada pixel obtiene uno solo color en su graduacíon original de acuerdo al patrón mientras que los otros dos colores se anulan, la transparencia se setea a 255.
En consonancia con el código en C, se asume que tanto el alto como el ancho de las imágenes de origen es múltiplo de 8 pixels.

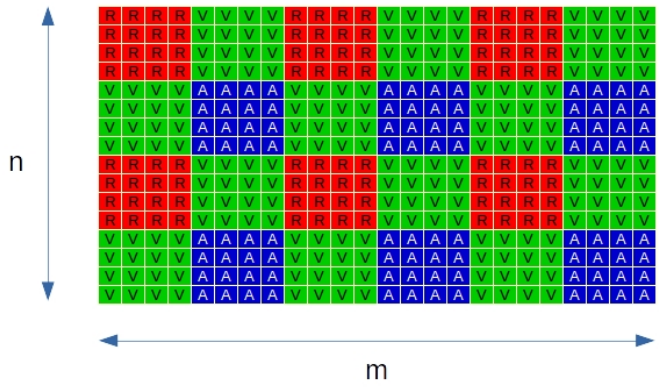


Figura 1: Ejemplo de patrón de pixeles al aplicar el filtro de Efecto Bayer con n=16 y m=24.

2.3.1. Pre-proceso de imagen

Debido a que el patrón que se debe generar es uniforme y se repite, lo ideal es utilizar ciclos para el proceso de la imágen. Los pixeles de 4 bytes cada uno se pueden levantar de a 4 de memoria a un registro `XMMn` de 16 bytes. Trabajando así en forma paralela con 4 pixeles al mismo tiempo.
Por otra parte es necesario crear una única máscara para la transaparencia que se carga a un registro `XMMn` antes de ingresar al ciclo, para poder utilizarla evitando accesos innecesarios a memoria.

2.3.2. Iteración

El proceso de iteración consta de tres ciclos:

- Gran Ciclo
  - Ciclo Verde-Azul
  - Ciclo Rojo-Verde

La imagén se recorre desde la esquina inferior izquierda, por lo tanto se comienza con el **ciclo Verde-Azul**. Las dimensiones de la imágen son n pixeles de alto (filas), y m pixeles de ancho (columnas). Se levantarán primero 4 pixeles verdes se procesarán y luego 4 azules, en total 8 pixeles en una misma fila, como este patrón se repite a lo largo de toda la fila, se repetirá este proceso $m/8$ veces, pero además esto se replica en 4 filas continuas por lo tanto la cuenta queda $(m \times 4)/8 = m/2 = columnas/2$, que es la cantidad total de veces que ciclará el ciclo Verde-Azul. Equivalentemente ocurre con el **ciclo Rojo-Verde**. Las operaciones realizadas dentro de estos ciclos son levantamiento desde memoria de 4 pixeles por vez en un registro `XMMn`, shifteos lógicos para obtener en cada *dword* del registro el único color deseado y luego un packet or lógico bit a bit para agregar la transaparencia contenida en otro reigistro `XMMn`. Este resultado se mueve a la posición de memoria de la imágen destino. Esto se efectúa primeramente para el color Verde y luego para el Azúl, (o para los colores Rojo y Verde) y se vuelve a ciclar.
El **Gran Ciclo**: como se ha mencionado, comenzando desde la última fila y recorriendo la imágen hacia arriba habrán cuatro filas con el patrón Verde-Azul, seguidas de cuatro filas con el patrón Rojo-Verde. De este modo se repite el mismo esquema cada 8 filas y por lo tanto este gran ciclo que contiene a los dos anteriores, deberá efectuar n/8 = filas/8 ciclos en total.

2.4. Filtro Cambia Color

El filtro *Cambia Color* se caracteriza por modificar un determinado color *target* por uno nuevo, dependiendo de la distancia entre el mismo y el pasado por parámetro. El color resultante se obtiene por una combinación lineal dependiendo del cociente entre un valor umbral lim ingresado y la distancia calculada (d).

Para simplificar la descripción se denominan a las componentes nuevas rojo, azul y verde del color ingresado Nr, Nb y Ng, mientras que a las correspondientes al color *target* se las denomina Or, Ob y Og.

2.4.1. Pre-proceso de imagen

En primer lugar, en pos de minimizar los accesos a memoria el algoritmo preserva en `XMM11-XMM13` las componentes del *target* insertando cada uno de modo que ocupen una Double Word (se efectúa una lectura única desde la pila para preservar en un registro de propósito general desde el cual se inserta en los registros `XMMn`). Por otro lado, como lim^2 ocupa una Double Word, se decidió calcularlo una vez adentro dentro del ciclo, trabajando con lim directamente como float (resultando innecesario mover 64 bits a un registro `XMMn` en caso de haber realizado el cálculo antes del ciclo).

En cuanto al procesamiento de la imagen, dado para el caso $d < lim$ se trabaja en el rango $0 \leq c < 1$, resulta necesario realizar las operaciones de d como float para evitar pérdida de precisión de c, y por ende, $1 - c$ (ambos factores de la combinación lineal del color destino). De allí que se vio innecesario calcular la raíz cuadrada de d^2 (cálculo que habría disminuido la precisión del color de los pixels destino) puesto que para la obtención de c se requiere hacer el cociente entre d^2/lim^2 y la inecuación $d < lim$ resulta equivalente a hacer $d^2 < lim^2$.

2.4.2. Iteración

Por cada ciclo el procesamiento consta de las siguientes partes:

1. Cálculo de d^2 en float
   a) Obtención de ΔG^2, ΔB^2 y ΔR^2
   b) Cuentas con r como entero
   c) Divisón por 256 como float
   d) Suma total
2. Combinación Lineal
   a) Obtención de c y $1 - c$ como floats
   b) Máscara según $d \geq lim$ y $d < lim$
   c) Saturación
   d) Combinación por Merge

Para calcular la distancia al cuadrado se obtuvo cada componente de los pixels en un registro `XMMn` de la misma forma que se trabajó en `TresColores`(véase Sección 2.1.2).

Luego por medio de una resta de Double Word con los parámetros resguardados en `XMM11-XMM13`, se calculó los delta de color (ΔR, ΔG y ΔB). Cabe resaltar que en el caso $d \geq lim$ se debe retornar el pixels sin alteración, razón por la cual se los preserva en el registro `XMM0` para evitar leer de memoria nuevamente.

A la hora de obtener los deltas se empleó la instrucción `PMULDQ` del registro que contiene las cuatro Double Words, trabajando con este tamaño dado que se procesan 4 pixels en paralelo. Como dicha instrucción multiplica solo la primera y la tercera Double Word escrbiendo el resultado como Quadword, en otro registro se calculó el la multiplicación del segundo y el cuarto por medio de un shifteo lógico a derecha de 4 bytes.

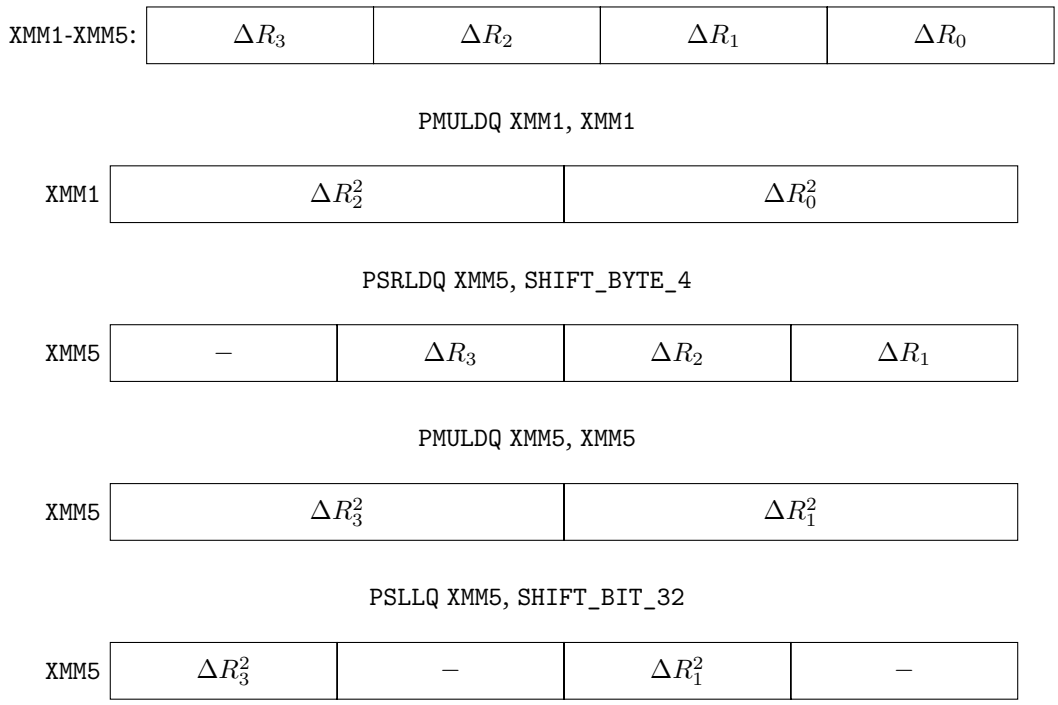


Figura 2: Diagrama de obtención de delta color cuadrado

Como estrictamente cada delta ocupa una Word su cuadrado entra en una Double Word, razón por la cual vale el shifteo a izquierda de `XMM5`. Luego, se realiza el merge entre `XMM5` y `XMM1`, obteniéndose los cuatro resultados en un solo registro.

Una vez obtenido los deltas correspondientes, se procedió a calcular el múltiplo correspondiente de cada color por shift lógico a la derecha de 1 bit o 2 bits en los casos rojo y verde respectivamente. Para la multiplicación por 3 para el caso azul, una vez obtenido el doble por shifteo, se sumó el valor inicial.

Con respecto a las cuentas vinculadas al promedio de los rojos, una vez calculada la diferencia entre ΔR^2 y ΔB^2 en Double Word y la suma de los rojos *Nr* y *Or*, se procede con su multiplicación empleando la instrucción `PMULDQ` como se detalló anteriormente. Luego, se convirtió a float para efecutar la division por 512 con precisión simple. Cabe recordar que el cálculo de d^2 se efectúa en float por lo que la conversión resulta necesaria.

Finalmente, se procede a realizar una suma de float entre $2\Delta R_3^2$, $4\Delta G_3^2$, $3\Delta B_3^2$ y los registros obteniendose d^2.

Se procedió a realizar el cálculo de c y las máscara correspondiente para los casos $d \geq lim$. Cabe resaltar que resultó necesario definir la máscara de 1 en float de precisión simple dado que la representación en enteros y float son distintas, no puediendo efectuar un `PCMPED` de un registro consigo mismo.

Una fez efectuados los cálculos de la combinación lineal para el caso $d < lim$ para la cual se empleó `MULPS`, se procedió a filtrar los resultados según se deba saturar o no cada pixels. Para determinar cuáles era las componentens que debían saturarse insertando 255 como valor, se realizó una comparación de DoubleWord habiendo previamente convertido los datos de enteros a float.

Finalmente se realizó el merge entre el resultado final y los pixels fuente resguardados según las máscaras de cada condición.

2.5. Filtro Edge Sobel

El filtro *Edge Sobel* modifica la imagen de entrada resaltando los bordes, a partir de un algoritmo diseñado para tal fin. Trabaja con imágenes en escala de grises, por lo cual la imagen de entrada es transformada a ese formato de manera previa a la aplicación del filtro.

El procesamiento principal consiste en recorrer cada uno de los píxeles (excepto los bordes) realizando cálculos con los píxeles adyacentes para resaltar los bordes. Los últimos 14 píxeles se procesan por fuera del ciclo principal para evitar la lectura de memoria no correspondida. Finalmente, se escribe el color negro en el borde de la imagen. En las subsecciones siguientes se explica en detalle la implementación.

2.5.1. Construcción del ciclo principal

Se trabaja con una *ventana* de 3x3 píxeles alrededor de cada pixel. El ciclo comienza con un puntero al borde inferior izquierdo de la imagen, que a su vez corresponde al píxel inferior izquierdo de la primera ventana de 3x3, es decir, el píxel de abajo a la izquierda del primer píxel que se debe procesar. A partir de este puntero y utilizando el esquema de $base + índice * escala + desplazamiento$ se leerá el resto de los píxeles necesarios de la ventana.

Como el borde no se procesa y el puntero del ciclo corresponde al píxel de abajo a la izquierda del píxel que se procesa en cada iteración, el ciclo debe cortar dos filas antes de terminar la imagen.

El puntero donde cortar el ciclo, por lo tanto corresponde al índice $(row\_size - 2) * height$ de la matriz de la imagen fuente. Adicionalmente, a ese valor se le restan 16 bytes (los bytes correspondientes a un registro xmm) para evitar las lecturas de memoria por fuera del rango de la matriz.

Al final de cada iteración, se suma 16 a los punteros a las imágenes fuente y destino. Si el puntero a la imagen fuente es menor que el calculado anteriormente, se continúa a una nueva iteración. Si no, el ciclo termina.

2.5.2. Ciclo principal

El procesamiento se realiza en registros vectoriales (xmm) de 16 bytes. Como los píxeles en escala de grises ocupan un byte, se realiza el procesamiento de 16 píxeles en paralelo en cada iteración. A partir de ahora se hará referencia a un solo píxel para facilitar la comprensión, pero debe entenderse como 16 píxeles en paralelo.

En primer lugar, se realiza la lectura de memoria de los 8 píxeles que rodean al píxel actual, cada una en un registro xmm. Se podrían haber utilizado menos registros para reducir lecturas de memoria (algo

positivo), pero reduciendo la cantidad de píxeles a procesar en paralelo (esto es negativo). Se optó por hacer las 8 lecturas porque como son celdas cercanas, se debería aprovechar la caché mediante el principio de localidad espacial, logrando así una mayor eficiencia en relación a la otra forma de procesamiento.

Los cálculos se realizan en enteros de 16 bits (llamados $short$ en lenguaje C) para evitar el *overflow*. Por lo tanto, cada vector de 16 píxeles se desempaqueta en dos vectores de 8 píxeles con el doble de tamaño. De esta manera, se designan dos registros xmm para el operador X y otros dos para el operador Y. Cada uno de los 8 píxeles que rodean al píxel actual también se desempaquetan en dos registros y se realizan las sumas o restas correspondientes sobre los registros acumuladores.

Luego de realizar todas las operaciones correspondientes a los píxeles adyacentes, se calcula el módulo de los operadores. Pasos para calcular el módulo:

- Máscara para distinguir los positivos de los negativos.
- Copiar el resultado y calcular su opuesto aditivo.
- Con la máscara, pisar con cero los números que eran positivos originalmente, haciendo que sobrevivan solo los que eran originalmente negativos, que ahora son positivos. Es decir, en este registro tenemos el módulo de los negativos y el resto está completado con ceros.
- Con la máscara original invertida, pisar con cero los números que eran negativos originalmente. Es decir, en este registro tenemos solo los positivos y el resto está completado con cero
- Unimos estos registros y tenemos el módulo de todos los valores.

Para finalizar el cálculo, se suman los módulos de X y de Y, se empaquetan y se escribe en la matriz de la imagen destino.

2.5.3. Procesamiento final

Al finalizar el ciclo, deben procesar los últimos 16 bytes de la imagen. Como el puntero corresponde al píxel de abajo a la izquierda del píxel a procesar, los 16 bytes corresponden a 15 píxeles. Adicionalmente el borde derecho resta otro píxel, por lo que finalmente los 16 bytes corresponden a tan solo 14 píxeles.

Se realiza un ciclo para estos 14 píxeles que es análogo al del ciclo principal pero se procesa de a un píxel por vez. No se realiza procesamiento en paralelo porque la cantidad de píxeles es constante y tan pequeña que su tiempo de procesamiento es despreciable.

Finalmente se escriben los bordes horizontales y verticales en color negro (cero). Los bordes horizontales se realizan en paralelo pues están ubicados de forma contigua en memoria (filas). Los bordes verticales no se realizan en paralelo pues no están almacenados en forma contigua. El tiempo de procesamiento de los bordes verticales es despreciable de todos modos porque el ciclo principal tiene complejidad cuadrática y los bordes tienen complejidad lineal.

3. Procedimiento Experimental

A continuación se analizará comportamiento temporal de los algorítmos programados para los filtros en lenguaje ASM y las implementaciones en C otorgadas por la cátedra. La diferencia sustancial dentre ellos en cuanto al procesamiento de imagen radica en que en ASM -al programar en SIMD-, se están paralelizando la aplicación de sendos filtros, mientras en lenguaje C se opera sobre un único pixel a la vez.

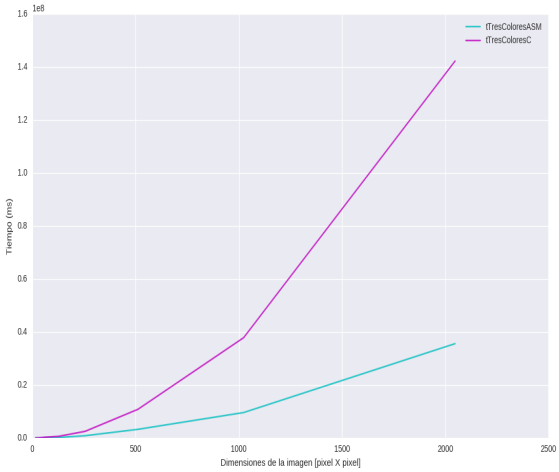
Para la experimentación se propuso analizar la relación entre el tamaño de las imágenes a procesar y la performance de su ejecución en las implementación en ASM y C. Para ello se tomó una misma imagen en distintas dimensiones[1] y se graficó el tiempo de ejecución en función del tamaño de la imágen, pudiendo así efectuar comparaciones entre ambas implementaciones. Para esta experimentación se espera que la performance en ASM sea más eficiente que la correspondiente a C debido a la mencionada paralelización del procesamiento de pixels.

[1]Se trabajó con imágenes cuadradas de los siguientes tamaños: [16x16, 32x32, 64x64, 128x128, 256x256, 512x512, 1024x1024, 2048x2048] en pixel x pixel.

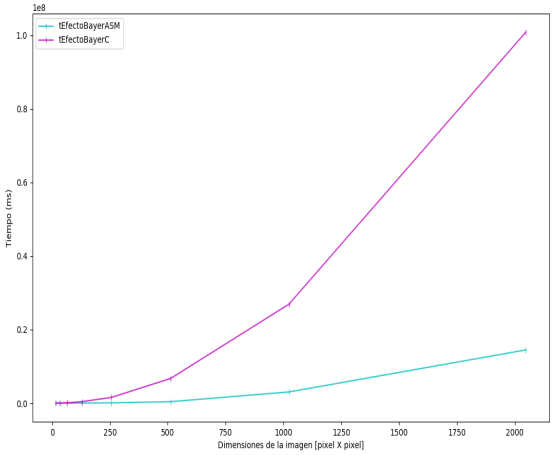
Por otro lado, también se procedió a analizar experimentalmente el costo de una lectura a memoria en el ciclo de procesamiento en comparación con una lectura única previa a la iteración, resguardándose el dato en un registro disponible para ser consultado una vez dentro del ciclo. Dicha experimentación se llevó a cabo en la implementación del filtro *Cambia Color*, en el cual se requerió utilizar la mayoría de los registros `XMMn` en la ejecución del algoritmo (lo cual dificultaba preservar las máscaras necesarias en registros antes de ciclar). Esta es la razón por la cual se decide efectuar una segunda implementación en pos de optimizar su eficiencia. Cabe resaltar que a costas de realizar una lectura única a memoria se debe efectuar múltiples shifteos e inserciones dentro del ciclo. Como resultado de esta experimentación se espera una mejora en el rendimiento temporal de ejecución para el caso de lectura por fuera del ciclo.

4. Resultados y Análisis

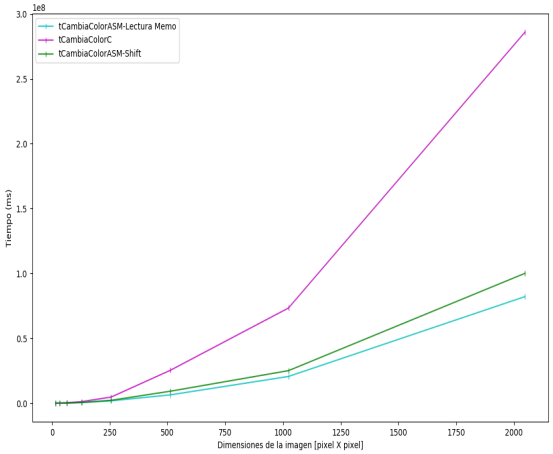
A continuación se presentan los resultados obtenidos para ambas experimentaciones, apreciándose la relación entre tiempo de ejecución y tamaño de iamgen para las implementaciones realizadas en ASM y las otorgadas por la cátedra en C.



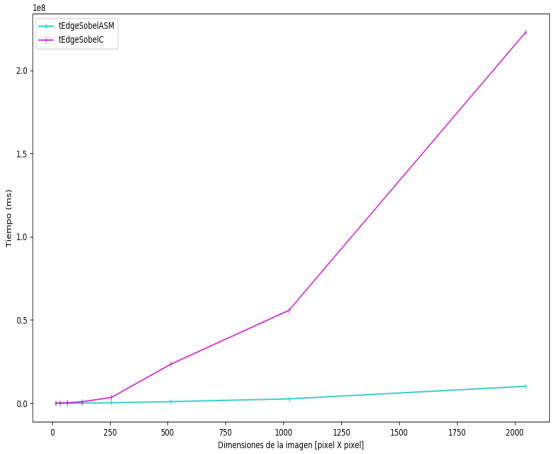
(a) Filtro Tres Colores



(b) Filtro Efecto Bayer



(c) Filtro Cambia Color



(d) Filtro Edge Sobel

Figura 3: Comparación de tiempos de ejecución ASM y C

En primer lugar, en la Figura 3 se puede observar que para todos los filtros la performance en ASM fue superior a la misma en C, lo cual se corresponde con los resultados esperados en términos generales: la

paralelización en el procesamiento de los pixeles en ASM optimiza los tiempos de ejecución con respecto a C, el cual procesa los pixeles individualmente.

Sin ir más lejos, las pendientes de las curvas correspondientes a la implementación en C aumentan a mayor tamaño de imagen procesada, lo cual responde a la magnificación de ineficiencia del procesamiento de pxels individualmente. En contraposición, la implementación en ASM no resulta tener cambios de pendiente pronunciados, siendo más eficiente el caso del Edge Sobel: su curva es la que mejor se ajusta cualitativamente a una recta lineal resultando ser la performance no dependiente del tamaño de la imagen procesada según los resultados obtenidos.

A la hora de analizar la diferencia de los filtros entre sí, se aprecia el siguiente orden creciente de performance con respecto a los tiempos de ejecución: $EfectoBayer < EdgeSobel < TresColores < CambiaColor$.

Resulta llamativo que *Cambia Color ASM* alcance tiempos de ejecución similares a *Efecto Bayer C*, lo cual se adjudica al alto costo del procesamiento de datos flotantes de precisión simple, viéndose disminuida la eficiencia en ASM y tomando tiempos de ejecución más altos en comparación con los otros filtros.

El filtro de *Efecto Bayer* es el que presenta los tiempos de ejecución más óptimos entre los cuatro filtros analizados, esto se justifica en que se realiza una escasa cantidad de operaciones de shifteo lógico para procesar la imagen y las operaciones en el pre-proceso de la misma son de menor complejidad con respecto a los demás filtros.

Por otro lado, a la hora de comparar *Edge Sobel* con *Tres Colores ASM* se obtiene que este último resulta ser menos eficiente. Este resultado se lo atribuye a las seis lecturas de constantes definidas en `.RODATA` por fuera del ciclo y además, a la una operación de float que se realiza dentro del mismo. Además, *Edge Sobel* toma ventaja por las lecturas a memoria de posiciones adyacentes haciendo uso del principio de vecindad.

Sin ir más lejos, como ya se ha mencionado, el filtro *Cambia Color* resulta tener los mayores tiempos de ejecución por la cantidad de operaciones aritméticas de float en precisión simple.

En cuanto a la experimentación sobre el costo de una lectura a memoria en el ciclo de procesamiento en comparación con una lectura única previa a la iteración (véase Figura 3.(c) Filtro Cambia Color), se obtuvo que la lectura dentro del ciclo resultó ser mas eficiente que la lectura previa a la iteración. De dicho resultado se infiere que el costo adicional de las instrucciones de shift e inserción agregadas superan el costo de realizar una lectura de memoria por cada ciclo ejecutado. Esta es la razón por la cual la implementación final de *Cambia Color* realiza estas lecturas, las cuales no pudieron ser evitadas dado que no se encontró el modo de tener registros disponibles para resguardarlos previo a la interación.

5. Conclusiones

En conclusión, en este trabajo se logró realizar la implentación en ASM de los filtros propuestos bajo el modelo `SIMD` y compararlas con las otorgadas por la cátedra en lenguaje C. A la hora de analizar la performance del procesamiento de imagen en cuanto a tiempo de ejecución, se obtuvo como resultado que ASM optimiza su eficiencia gracias a la paralelización del proceso.

A la hora de comparar los filtros entre sí, se obtuvo que el tratamiento de datos flotantes requiere de una mayor complejidad temporal considerable, así como también lo requiere la lectura de memoria.

Cabe resaltar que durante la realización del trabajo se propuso llevar a cabo una segunda implementación para el filtro *Cambia Color*, en pos de disminuir una lectura a memoria dentro del ciclo de procesamiento. De esta experimentación se concluye que para este filtro en particular dada su complejidad, una lectura resulta ser más eficiente que la re-estructuración del código para evitar la misma.

Por último, se consideran algunas propuestas de experimentación a futuro para poder comprender mejor lo analizado en este trabajo:

1. Variar la cantidad de pixeles procesados en los filtros. De esta forma se analizaría mejor la eficiencia del paralelismo.

2. Indagar qué instrucciones de tipo float requieren de mayor complejidad. Se supone que mover datos y convertirlos es menos costoso que instrucciones aritméticas (i.e. multiplicación y división).

Referencias

[1] Sthephen A. Edward, *Fundamentals of Computer Systems*, Columbia University, Spring, 2012. Disponible en: http://www.cs.columbia.edu/ sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf

[2] Tim Urban, *Inside the mind of a master procastinator*, Evento TED2016, Febrero, 2016. Disponible en: https://www.ted.com/talks/tim_urban_inside_the_mind_of_a_master_procrastinator