Organización del Computador II

Segundo cuatrimestre 2019

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

TP Nº 2 - Modelo de procesamiento SIMD

Grupo:

Integrante	LU	Correo electrónico
Hernán Gagliardi	810/12	hg.gagliardi@gmail.com
Edgardo Mosqueira Caballero	808/13	edgar.r.caballero@gmail.com
Francisco Ciarliero	574/14	fciarliero@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

En siguiente trabajo tiene como objetivo experimentar con el set de instrucciones SIMD de la arquitectura IA-32 de Intel. Para ello vamos a desarrollar variadas implementaciones de filtros de imágenes.

Para cada tipo de filtro, se realizaron dos tipos de soluciones. La primera es por medio de una implementación en **C**. La segunda en **ASM** usando, como se mencionó inicialmente, instrucciones **SIMD** para operar sobre la mayor cantidad de pixel's posible.

En la siguiente sección vamos a realizar un breve análisis sobre todas las implementaciónes realizadas para cada filtro, en particular veremos como fueron implementadas las funciones en ASM.

2. Desarrollo

2.1. Objetivo

El objetivo de los experimentos, es sacar una conclusión de cuál es la forma más óptima de realizar el proyecto. Para esto queremos analizar:

- Los rendimientos de cada algoritmo y sus diferencias. Porque una parte importante de la optimización que estamos buscando es la velocidad de ejecución de dichos algoritmos.
- Los tiempos de implementación de cada algoritmo y sus diferencias ya que no siempre vale la pena tener el código más rápido si su implementación toma demasiado tiempo.

2.2. Experimentos:

Realizaremos un experimento que consta en comparar las implementaciones de C/ASM en todos los filtros para ver cuál es la de mayor complejidad, o sea la que más tiempo tarde en procesar. Como hipótesis, deberíamos obtener mejores resultados en las implementaciones realizadas con ASM, ya que usando SIMD estaríamos procesando información de una mayor cantidad de pixel's que en los ciclos usados en C.

2.3. Filtro Nivel

2.3.1. Descripción

El objetivo del filtro Nivel es generar una imágen que, para cada una de sus componentes, detecte que bits del número (en su representación bunaria) están en 1.
Para eso utiliza un índice que es pasado por parámetro y nos sirve para saber que bit de la componente debemos chequear.
Luego, si la comparación es correcta y el bit indicado por el índice de la imagen origen es efectivamente 1, debemos saturar dicha componente a 255. Caso contrario, debemos setear el valor en cero.
A continuación adjuntamos el pseudocódigo que clarifica la operación mencionada:

mask = 1 « indice
»**for** $i = 0$ to $i < h$ **do**
»»**for** $j = 0$ to $j < w$ **do**
»»»$dst[i][j].b = (src[i][j]\ and\ mask)\ ?\ 255 :\ 0$
»»»$dst[i][j].r = (src[i][j]\ and\ mask)\ ?\ 255\ :\ 0$
»»»$dst[i][j].g = (src[i][j]\ and\ mask)\ ?\ 255\ :\ 0$
»»**end for**
»**end for**

2.3.2. Código en C

El código en C que nos permite resolver este filtro es alcanzable de forma casi trivial luego de haber planteado el pseudocódigo en la subsección anterior.
En principio se obtiene la máscara en un int de 8 bits *shifteando* el dígito 1 "n veces hacia la izquierda"para colocarlo en la posición del bit que queremos chequear.
Luego se recorre la imagen pixel a pixel mediante un ciclo ancho/largo para poder hacer la verificación de cada componente del pixel. Es decir, si el componente tiene un 1 en el índice que buscamos, se lo satura en 255, sino se lo setea en 0.

2.3.3. Código en ASM

La clave para resolver este ejercicio usando instrucciones en paralelo va a ser el armado de máscaras adecuadas al comienzo de nuestro código.
Guardamos el parámetro **N** (que recibimos a través de la pila) en un registro de propósito general de 8 bits, para luego realizar un shifteo a izquierda *shl* correspondiente sobre un registro previamente seteado en 1.
Una vez hecho esto, procedemos a *broadcastear* nuestro registro shifteado a un registro **xmm** (previamente formateado) para poder tenerlo en todas las posiciones de byte, es decir, 16 veces. Este procedimiento lo hacemos mediante un previo **pinsrb** y luego un **pshufb** que nos permite extender nuestro valor a cada byte del XMM.
Por último, seteamos un registro XMM en 0 y otro en 0xFF, que nos van a servir para hacer la comparación correspondiente luego.
Una vez hecho esto, se empieza a recorrer la imagen tomando de a 4 pixel's, es decir los 16 bytes que entran en un registro XMM.
Durante cada iteración repetiremos un procedimiento sencillo que consta de los siguientes pasos:
1: Hacer un primer *pand* que nos permita conocer en que componentes de esos pixel's coincide que haya un 1 en la posición señalada por el índice del parámetrro.
2: Comparar el resultado del paso 1 con la máscara que previamente seteamos en 0. De esta manera

conoceremos que componentes debemos saturar y cuales setear en 0.
3: Comparar el resultado del paso 1 con la máscara que previamente seteamos en 0. De esta manera conoceremos que componentes debemos saturar y cuales setear en 0.
4: Por último, utilizando la máscara resultante del punto anterior donde están seteados en 1 los bytes que debemos saturar y en 0 los que no, sólo resta realizar un *pand* con la máscara de unos para tener los valores resultantes para guardar nuevamente en memoria como imagen destino.
De esta manera podemos observar que mediante la utilización de las instrucciones SSE, en nuestra implementación podemos procesar 4 pixel's en simultaneo.

2.4. Bordes

2.4.1. Descripción

El siguiente filtro, como su nombre indica, nos va a servir para hacer una detección de los bordes que contiene una imagen. Se busca dentro de la imagen los cambios abruptos de los valores de los pixeles. Para esto, vamos a utilizar el operador Sobel, que calcula el gradiente de la intensidad de una imagen en cada punto. Así vamos a obtener, para cada punto, la mayor magnitud de cambio posible, la dirección de ese cambio brusco y el sentido de claro a oscuro o viceversa.

Operador Y Operador X

$$\begin{matrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{matrix} \qquad \begin{matrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{matrix}$$

Figura 1: Operador Sobel para el filtro Bordes

La idea es obtener el valor del pixel de destino usando estas 2 matrices saturando con la siguiente formula.

$$dst_{[i,j]} = \begin{cases} 255 & \text{si } |\text{OpX}| + |\text{OpY}| > 255 \\ |\text{OpX}| + |\text{OpY}| & \text{si no} \end{cases} \tag{1}$$

Donde el valor de OpX y OpY estan definidos por

$$OP_Z(x,y) = \sum_{k=0}^{2} \sum_{l=0}^{2} src(x+k-1, y+l-1) * M_Z(k,l)$$

Este filtro, a diferencia del resto, trabaja con imagenes en blanco y negro. Es decir cada pixel esta compuesto por una única componente.

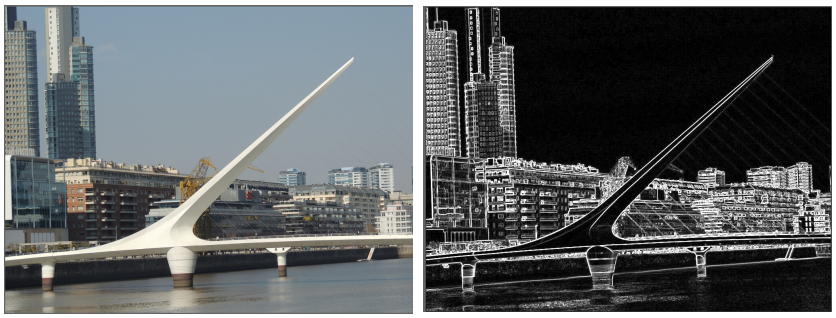


Figura 2: Aplicando el filtro Bordes

2.4.2. Código en C

Podemos observar de las ecuaciones descriptas anteriormente que en este filtro no podremos procesar todos los pixel's de la misma forma, ya que dado un pixel se necesitan los pixel's que vecinos que lo rodean, cosa que no podremos hallar para los que se encuentran en el borde de la imagen.

Luego con cada pixel simplemente realizamos la operación mencionada:

- Sumamos cada pixel por su correspondiente valor en la OpY
- Sumamos cada pixel por su correspondiente valor en la OpX
- Aplicamos modulo a los dos Op y los sumamos
- Saturamos ese valor y lo movemos al pixel de la imagen *dst*.
- Finalmente 'pintamos' el borde de la imagen destino de blanco recorriendo los pixel's de los extremos.

2.4.3. Código en ASM

Como mencionamos el ciclo que nos permite aplicar el filtro Bordes en una imagen tiene algunas particularidades. Vamos a explicarlo de manera simple a continuación.

Como los pixel's ocupan un byte cada uno (pues son imágenes monocromáticas), se quiere procesar 16 pixel's. Es por ello que en el ciclo avanzamos de a 16 bytes. Para poder procesar esta cantidad de pixel's, sabiendo que para procesar cada uno necesitamos de los 8 vecinos, usamos los registros desde $xmm1$ hasta $xmm9$ para guardar los 16 pixel's de cada posición de la matriz. De esta forma podemos ver como se forma una especie de cubo en el cual toda la capa exterior son pixel's que se usan para procesar el pixel del medio.

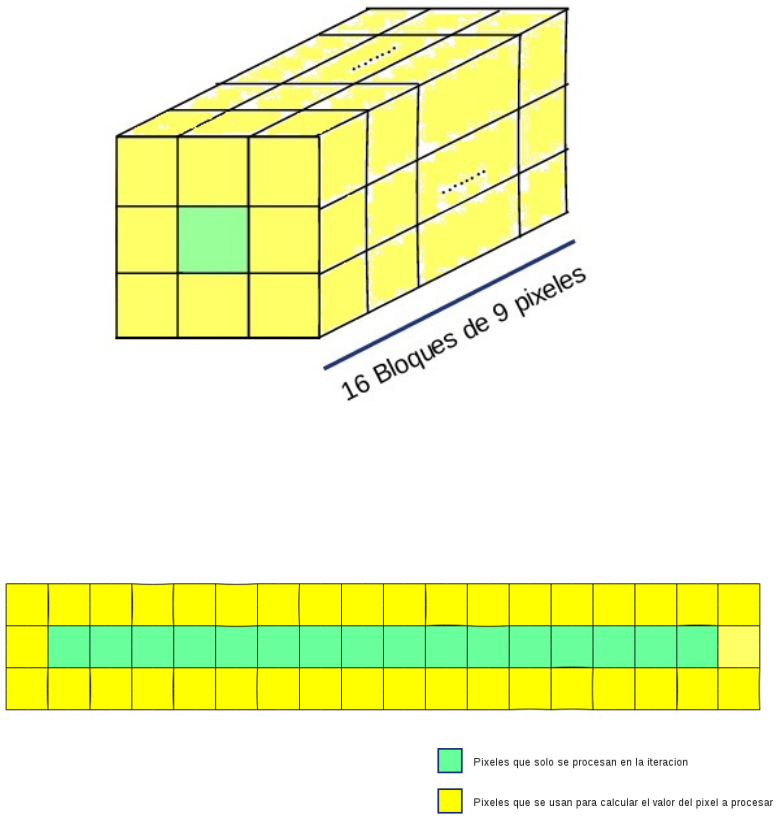


Figura 3: Ubicación de los punteros y datos necesarios

Para evitar problemas en los bordes, vamos a chequear si estamos en la primera o ultima fila, en las cuales ponemos todos los pixel's en $0xFF$.

Despues, dada la falta de registros XMM que tenemos, hicimos primero una busqueda de los pixel's que se tienen que usar para las dos matrices de la misma forma, por ejemplo en los dos Op se usa en la posición 0,0 se usa como valor multiplicativo -1 en los dos casos. Y para no tener ningún tema de cambio de signo lo único que hacemos es multiplicar los valores por 2 y para el -2 usamos *psub*. Primero lo que hacemos es sumar y restar las esquinas para liberar algunos registros luego multiplicamos por 2 los centros y despues los agregamos a las sumatorias de OpX y OpY. La sumatoria va a tener un tamaño máximo no mayor a un word, por eso no tengo que hacer mas que pasar de byte a word e ir sumándolos.

Luego nos falta multiplicar por 2 los centros e ir sumándolos o restándolos dependiendo el caso. Ya que terminamos de usar las esquinas, los registros que usábamos para esas partes ya los podemos sobreescribir. Después de sumar y restar los centros solo nos queda hacer *pabsw* a los 4 registros donde tenemos las OpX (High y Low) y las OpY (High y Low). hacemos padd de OpX High con OpY High y de OpX Low con OpY Low. Hacemos un packuswb para lograr la saturación solicitada en la ecuacion (4). Finalmente guardamos el valor en el registro de destino y ya esta completado el filtro. De esta forma conseguimos procesar 16 pixel's en una iteración y procesando, en paralelo, 8 pixel's simultáneos.

2.5. Rombos

2.5.1. Descripción

Este filtro genera una serie de marcas con forma de rombo sobre la imagen. Modificando los valores de cada píxel en función de una formula que depende de las coordenadas del píxel en la imagen. Veamos mejor en el siguiente pseudocódigo lo que el filtro realiza.

```
size = 64
     Para i de 0 a height - 1:
         Para j de 0 a width - 1:
           ii = ((size/2)-(i%size)) > 0 ? ((size/2)-(i%size)) : -((size/2)-(i%size));
           jj = ((size/2)-(j%size)) > 0 ? ((size/2)-(j%size)) : -((size/2)-(j%size));
           x = (ii+jj-(size/2)) > (size/16) ? 0 : 2*(ii+jj-(size/2));
           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)
```

2.5.2. Código en ASM

Definimos algunas mascaras que nos van a ser útil para operar sobre este filtro. Las mismas las usaremos para reordenar un registro.

```
Shuffle_1:  db 0x00,0x01,0x00,0x01
            db 0x00,0x01,0xFF,0xFF
            db 0x04,0x05,0x04,0x05
            db 0x04,0x05,0xFF,0xFF

Shuffle_2:  db 0x08,0x09,0x08,0x09
            db 0x08,0x09,0xFF,0xFF
            db 0x0C,0x0D,0x0C,0x0D
            db 0x0C,0x0D,0xFF,0xFF
```

Antes que nada tenemos que calcular el resto de dividir el nro de filas (**i** que itera sobre el alto de la imagen) por **size**. Y también el resto entre el nro de columnas **j** y **size**, **j** itera sobre el ancho de las imágenes. Además vamos a operar siempre sobre numeros enteros.

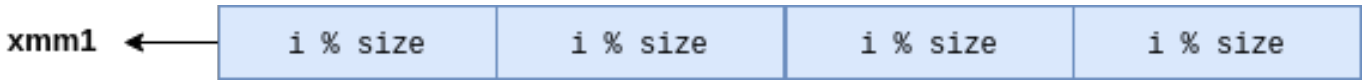
Utilizaremos la operación **div** sobre registros de propósito general de 64bits, esto nos devuelve en rdx el resto de dividir i entre size.

Para calcular **j % size**, tenemos una información adicional, sabemos que el ancho de las imágenes es siempre mayor a 16 píxeles y múltiplo de 8 píxeles. Y como tenemos size una constante que vale **64**, esto nos da libertad de saber siempre cual es el resto de dividir este numero por otro, la misma siempre estará entre 0 y 63, entonces usamos un contador en ese rango que nos devuelve el resultado que queremos en cada iteración sobre las columnas de la imagen origen.

Procedemos a recorrer la imagen ciclando por fila y por columna. Procesando 16bytes en cada iteración, es decir trabajaremos con 4 píxeles en simultaneo

Una vez dentro del ciclo principal, iteramos sobre la cantidad de filas, calculamos i % size como habíamos dicho. Este resultado lo movemos en un registro SSE elijamos **xmm1**. Como vamos a

trabajar sobre 4 píxeles en simultaneo, necesitamos replicar este valor en las demás posiciones en el registro xmm1, para esto utilizamos la instrucción **PSHUFD** y un inmediato indicándole la posición del valor que queremos copiar.



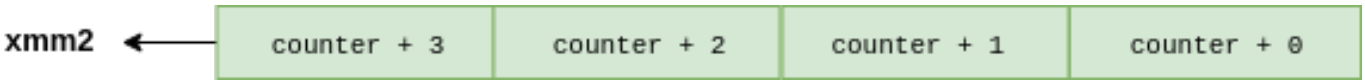
Ahora procedemos a realizar operaciones básicas sobre enteros empaquetados:
1. **PSUBD** entre empaquetados word, restamos size>>1 que copiamos a xmm2 - i %size. Sabemos que size>>1 = **32**.
2. Me quedo con el valor absoluto del resultado anterior, y lo guardo en un registro xmm

Dentro del ciclo interno, recorremos el ancho de la imágen, como ya dijimos calculamos j %size, estos valores ya los tenemos precalculados.

Counter ← [0..,63] en donde tenemos el resto de j %size

Aplicamos nuevamente operaciones sobre empaquetados de word.

1. Como procesamos 4 píxeles a la vez, le sumamos al contador, que previamente lo replicamos en cada doubleword el offset necesario para obtener el resto de la operación mencionada para cada columna.



Donde conuter + k = j % size.

2. Le restamos size>>1 a estos valores calculados y aplicamos PABSW para quedarnos con el valor absoluto de estos como hicimos con i en el ciclo principal.

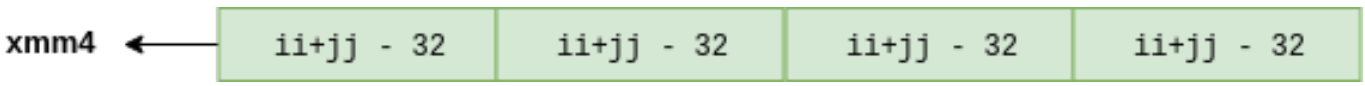
```
PSUBD xmm4, xmm3
PABSD xmm4, xmm3
```

Entonces tenemos los siguientes valores obtenidos:

```
ii = ABS[32-(i%64)], ABS[32-(i%64)], ABS[32-(i%64)], ABS[32-(i%64)]
jj = ABS[32-(j+3%64)], ABS[32-(j+2%64)], ABS[32-(j+1%64)], ABS[32-(j+0%64)]
```

Trabajamos sobre estas dos ultimas en doubleword, realizamos la suma entre estas y nuevamente restamos por el valor size>>1.

```
PADD xmm4, xmm7
PSUBD xmm4, xmm14
```

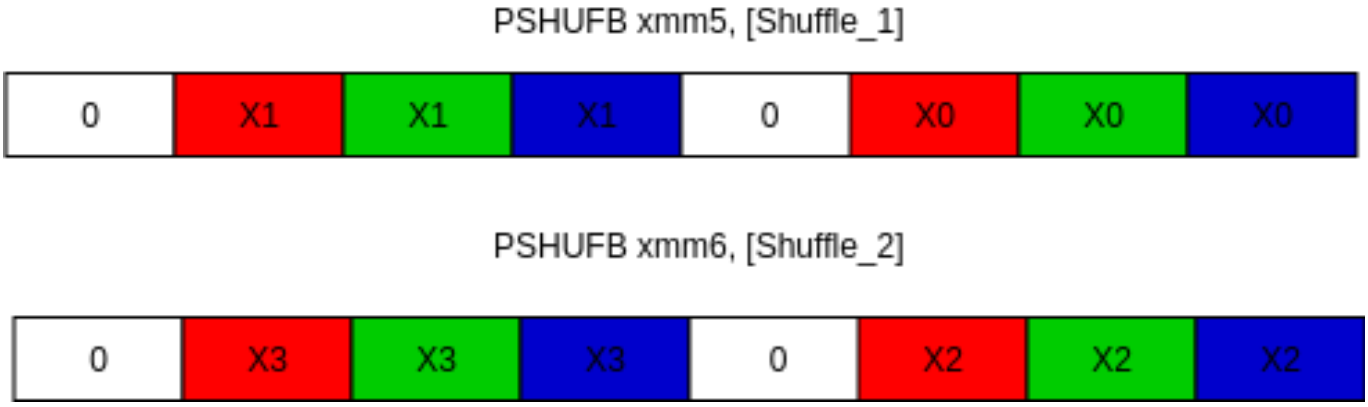


Ahora queremos quedarnos con aquellos resultados que sean menores o iguales a 4, si es así duplicamos el resultado: Para eso utilizamos las operaciones de comparación, y filtros que nos permiten filtrar los valores que buscamos. Caso contrario devolvemos cero.

XMM5 $\leftarrow [X3, X2, X1, X0]$.

Ahora procedemos a cargar 4 píxeles de rdi que es la imagen origen a un registro SSE, la desempaquetamos de byte a word utilizando las operaciones **PUNPCKHBW** para la parte alta Y **PMOVZXBW** parte baja.

Procedemos a shiftear los valores obtenidos en el paso anterior, es decir tener X3, X2, X1, X0 en las posiciones correspondientes para sumar a cada componente de cada píxel. En este caso utilizamos las mascaras que definimos al comienzo **Shuffel_1** , **Shuffle_2** y la operación de **PSHUFB**.



Para finalizar realizamos la suma entre estas ultimas, volvemos a empaquetar a byte con la operación **PACKUSWB** y copiamos a la imagen destino.

3. Experimentación

3.1. Hipótesis planteadas

En esta sección plantearemos algunas hipótesis que nos servirán de guía para la posterior experimentación y análisis de las implementaciones detalladas en la sección anterior.
En principio intentaremos ver si la utilización de las instrucciones vectoriales en la implementación con ASM mejora el rendimiento en cuanto a la implementación otorgada por la cátedra en C. En esta linea también buscaremos ver como se comporta el rendimiento de la implementación por pixel. Además, nos interesará buscar el *speedup* para saber cuanto más rápido es el algoritmo vectorial por sobre su implementación en C y buscar si está relacionado con la cantidad de pixel's que tomamos en simultaneo.
Por otro lado vamos a analizar el costo temporal por pixel de cada implementación y ver si este dato está correlacionado con el tamaño de la imágen.

3.1.1. Nivel: midiendo diferencias ASM vs C

Usaremos el filtro Nivel para hacer un estudio más detallado que cuantifique la superioridad de la implementación en ASM con respecto a la versión en C. Habiendo podido desarrollar un algoritmo con instrucciones SIMD sencillas y baja utilización de memoria, conjeturamos que se debería explotar fuertemente el caracter vectorial de la solución a medida que crece la cantidad de pixel's de la imágen.

3.1.2. Rombos: comparando distintas versiones de ASM

El objetivo de este experimento es cuantificar la diferencia de desempeño haciendo el cálculo con 2 y 4 píxeles en simultaneo. Nuestra hipótesis es que la versión de 4 pixel's debería ser mas performante, pero teniendo en cuenta el overhead de las lecturas en memoria y código esperamos que la versión de 2 pixel's sea solamente alrededor de un 30-20 % menos veloz.

3.1.3. Rombos: Comparación con versión mejorada de C

Sabemos que cuando comparamos la implementación de los algoritmos entre C y ASM no estamos realizando una comparación justa. Notamos que el código provisto en C calcula varias veces un valor que es constante, realizando operaciones innecesarias en cada iteración.

Por lo que modificamos el código, utilizando esta información para optimizarlo.

Entonces la Hipótesis que nos planteamos es que esta versión mejorada sera mas rápida que la anterior. Pero aun así la implementación en ASM sera superior.

Cuanto mejoro?....

3.1.4. Bordes: impacto de la memoria cache

Experimentaremos con el filtro Bordes para medir si el desempeño por pixel se mantiene con el crecimiento de la imágen. El costo de los algoritmos debería mantenerse igual, pero a través de este experimento queremos ver si no hay otras variables en juego para el rendimiento como por ejemplo la memoria cache. Entonces vamos a chequear si puede procesar imágenes de menor tamaño haciendo pleno uso de esta memoria y si hay alguna diferencia cuando la imágen es más grande al llenarse y tener que implementar políticas de desalojo más costosas.

3.2. Diseño de los experimentos

Para estudiar las hipótesis planteadas vamos a realizar una serie de experimentos que nos permitan acercarnos a una respuesta. En principio nos va a interesar hacer comparaciones entre la implementación en C y ASM para cada filtro. Utilizaremos para esto imágenes generadas de forma aleatoria de aspect ratio 1:1 con tamaños variables por experimento pero en el rango de entre 200px y 6000px.
De esta forma podremos evaluar desde la comparación lisa y llanamente de rendimiento por imágen, hasta el desempeño del costo por pixel.
Cabe aclarar que, salvo que se aclare expresamente, los experimentos de la implementación en C los corremos con una versión compilada con $-O0$.

Por otro lado haremos un análisis de SpeedUp, que se refiere a cuánto más rápido es un algoritmo vectorial (a.k.a. las implementaciones en ASM que hacen uso de las instrucciones SIMD) que su correspondiente implementación en C (secuencial).

Este tipo de análisis nos pareción razonable para dar una idea de la magnitud por la cual es mejor una implementación paralela que secuencial. Si bien este valor se desprende de los mismos datos mencionados anteriormente, creímos mejor formalizarlo usando el concepto de *speedup* entre las dos implementaciones. Se calcula con la fórmula:

$$\frac{T_i}{T_j} \tag{2}$$

En nuestro caso, será:

- T_i: cantidad de ticks del algoritmo en C
- T_j: cantidad de ticks del algoritmo en ASM

3.2.1. Rombos: asm 2 vs 4 pixeles

Específicamente para rombos hicimos una implementación adicional a la entregada que procesa de a 2 píxeles (lo mínimo pedido por la cátedra). esta implementación se puede encontrar en "../filtros/Rombos_asm_2_pixel.asm". se corrió este experimento en imágenes cuadradas iguales de 128, 256, 384, 512, 640, 768, 896 y 1024 píxeles un total de 500, 500, 500, 250, 250, 250, 125 y 125 veces respectivamente.

Debido a que el sistema operativo conmuta entre distintas tareas, interrumpiendo su ejecución introduciendo ruido en las mediciones, calculamos la media y la varianza muestral que graficamos como barras de errores siguiendo la siguiente formula: La media $\frac{1}{n}\sum X_i$ con n el tamaño de a muestra y X_i el resultado de cada experimento individual y la varianza como $\sqrt{\frac{1}{n-1}\sum(X_i-\mu)^2}$ siendo μ la media calculada previamente.

3.2.2. Aspectos técnicos

En principio podemos afirmar que realizamos todos los experimentos con la misma PC que cuenta con un procesador Intel Core I5-4210U.
Para generar los experimentos usaremos tanto la librería **time** de Python y además la cantidad de ticks del procesador, que medimos utilizando el código que la cátedra nos proporcionó usando la instrucción **rdtsc**. El registro Time Stamp Counter (que utiliza la instrucción) se incrementa en uno

con cada ciclo del procesador, de modo que la cantidad de ciclos total equivale a la diferencia del valor después y antes de ejecutar cada filtro.
Este registro es global y por ende cuenta ticks que todos los procesos del sistema consumen, no solo el de los filtros y tests que nosotros corremos, por lo cual no sería correcto hacer solo una medición, en vez de eso haremos una cierta cantidad de iteraciones (varía según los filtros) y tomamos el promedio, para suavizar los ruidos"que se puedan generar.

3.3. Resultados

3.3.1. Filtro Nivel: comparando implementaciones

Abordamos la experimentación de este primer filtro intentando comparar en una primera instancia la solución provista por la cátedra compilada con flag -O0 con la implementada por nosotros utilizando instrucciones SIMD. Para realizar esta comparación planteamos un primer experimento que compare los tiempos de ejecución de las implementaciones mencionadas.
De acuerdo a la descripción realizada en la sección anterior, planteamos una hipótesis que indica que la implementación realizada con SIMD debería ser más rápida que la de C por el paralelismo utilizado en los cálculos.

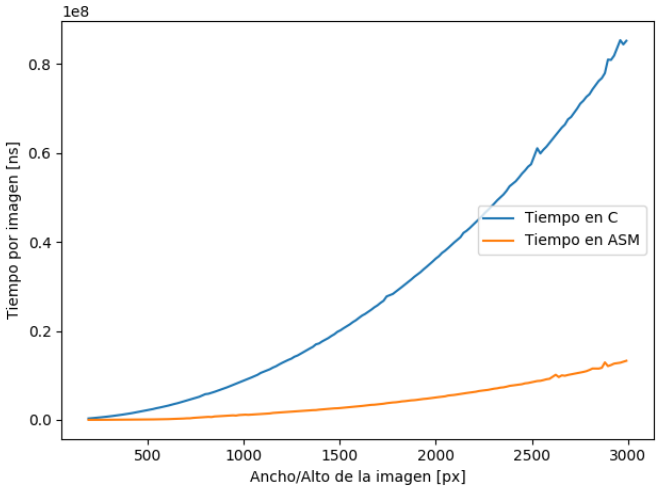


Figura 4: Tiempo por imagen de las implementaciones del filtro Nivel

El primer resultado de este experimento nos indica la superioridad temporal de la implementación en ASM. Habiendo planteado un primer experimento basado en el comportamiento sobre la cantidad de pixel's en imágenes de aspect ratio 1:1, vemos como la diferencia temporal se va agrandando a medida que la imagen crece en tamaño.

Se deduce que este crecimiento de la diferencia a medida que se agranda la imagen se debe a que logramos una implementación sencilla en ASM que además hace valer la característica vectorial de sus instrucciones.

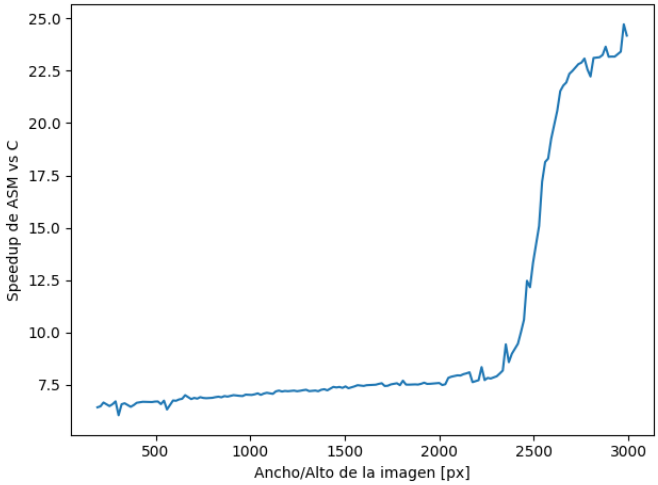


Figura 5: SpeedUp C vs ASM filtro Nivel

Acá se puede ver explícitamente en el gráfico como queda reflejada la diferencia de velocidad de las implementaciones que explicábamos anteriormente.
Vamos a probar por último si la comparación que hicimos anteriormente es 'justa' ya que compilamos nuestros filtros en C con -o0 que no aporta ninguna ventaja, por lo cual repetiremos el experimento anterior pero compilando con el flag -o3.

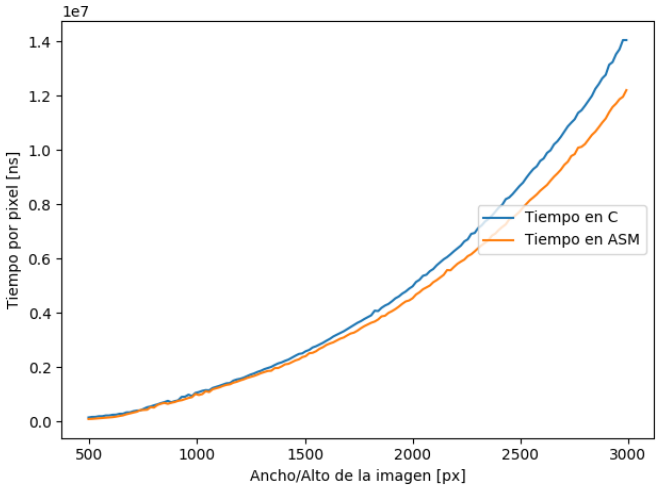


Figura 6: C (con -o3) vs ASM filtro Nivel

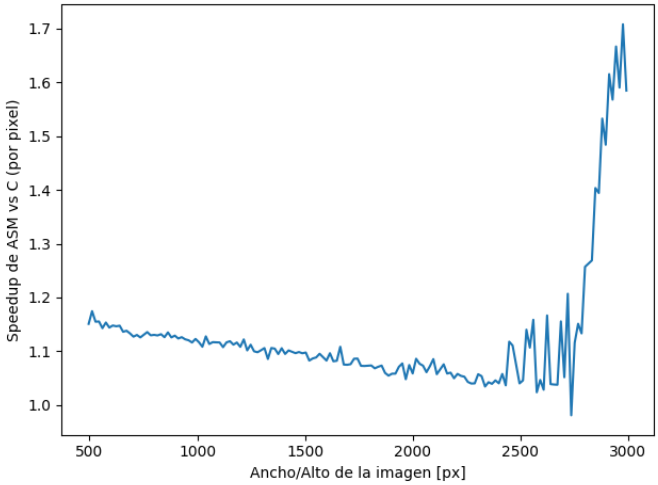
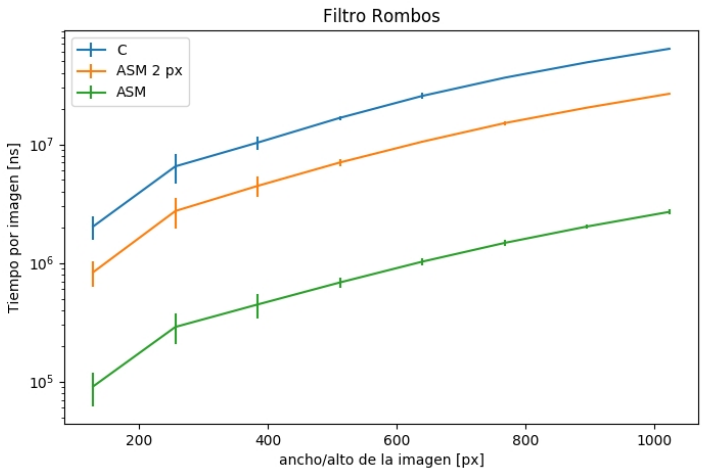


Figura 7: SpeedUp C (con -o3) vs ASM filtro Nivel

Podemos ver como la suposición era correcta y vemos que el distinto modo de optimización en la compilación cambia drásticamente el tiempo de ejecución en la implementación de C.
A medida que la imagen va creciendo se va agrandando una pequeña diferencia a favor de la implementación de ASM, casi empatada en su totalidad por la implementación en C (con -o3) que se ve gráficamente en el segundo gráfico de SpeedUp. Una de las razones posibles es que la opción de compilación con optimización -o3 activa, entre otros flags, el -ftree-vectorize que "vectoriza"la operación para ciclos con ciertas características usando (en el caso de la PC con la que hicimos los experimentos) los registros XMM. Esto, entre otras cosas, permite mejorar notoriamente el rendimiento de la versión de C, obteniendo una comparación más 'justa' que la planteada anteriormente.

3.3.2. Rombos: asm 2 vs 4 pixeles

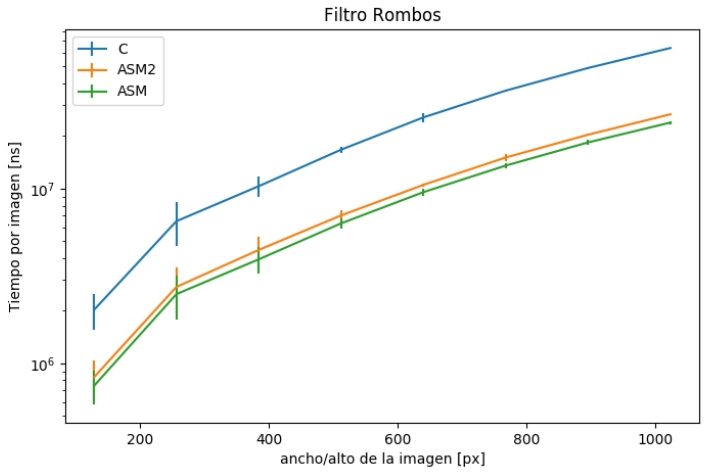
Como se muestra en el gráfico siguiente, la versión de 2 píxeles del filtro corre mucho mas lento de lo esperado estando mas cerca del rendimiento de la implementación en c que de la otra versión.



Teniendo un resultado tan diferente al esperado se analizó mas detenidamente el código de las dos implementaciones en assembler del filtro: la versión que opera con 4 pixeles tiene cerca de la mitad de instrucciones por pixel a la que opera con 2 (9,7 vs 17,5) y la mitad de lecto-escrituras a

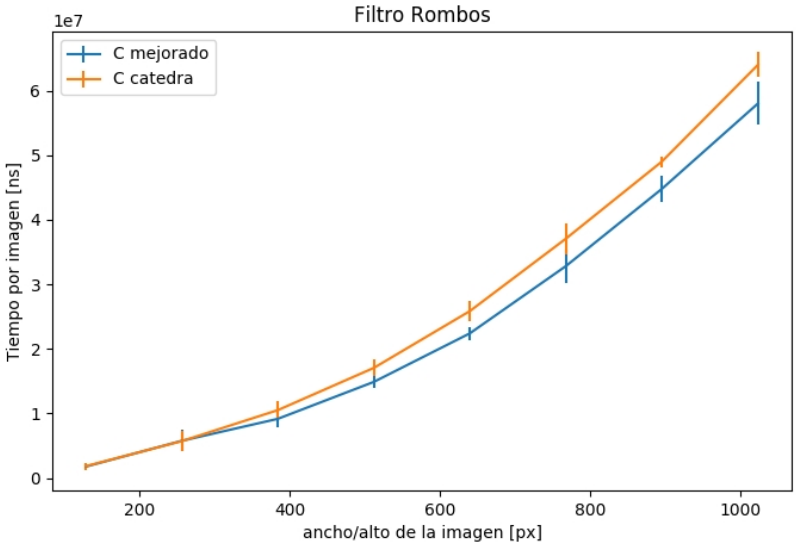
memoria. Esta diferencia en operaciones por pixel no es suficiente para explicar el gráfico anterior, fue necesario analizar que instrucciones son ejecutadas por cada pixel. De estas instrucciones la que mas podía impactar en el resultado de las mediciones era DIV que se ejecuta 1 vez por pixel procesado en la versión del filtro asm lento. para probar esta hipótesis se agrego instrucciones DIV al ciclo del filtro mas veloz para igualar la cantidad de veces que se ejecuta DIV por pixel con el otro filtro.

Como se ve en el siguiente gráfico, parecería que el uso excesivo de DIV en el segundo filtro fue la razón de la discrepancia tan grande en performance entre implementaciones.



3.3.3. Rombos: Comparación con versión mejorada de C

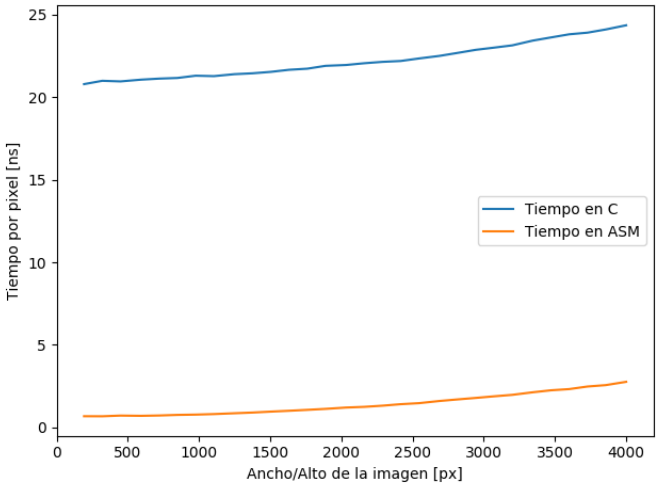
Los resultados que obtuvimos no fueron lo esperado, una de las razones posibles es que al momento de realizar las comparaciones, si el Scheduler se ejecuta justo cuando se esta corriendo el filtro y el sistema conmuta la tarea el tiempo de medición sera mayor a lo esperado. Ya que el tiempo de ejecución de un filtro es mucho menor al tiempo en que ocurre la conmutación entre tareas. También hay que tener en cuenta que cuando se compila el código en C se hacen optimizaciones, y el CPU mismo realiza Branch Prediction para optimizar cálculos y operaciones como un ÏF".



3.3.4. Bordes: impacto de la memoria cache

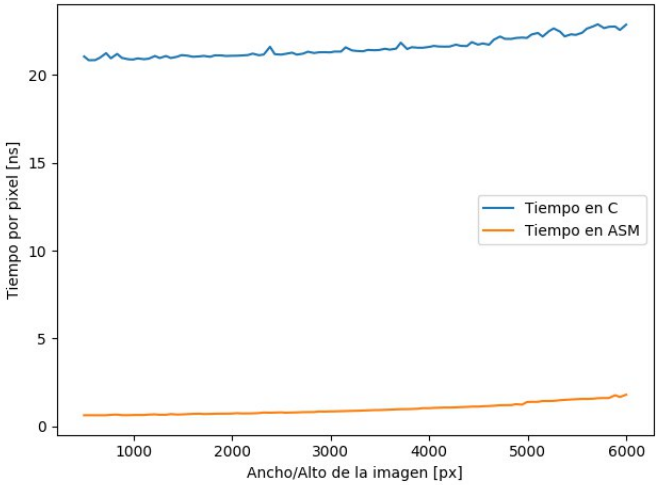
Como anticipamos en la sub sección 3.1.4 de la sección de hipótesis, los experimentos relacionados con el filtro Bordes estuvieron direccionados a medir si el tiempo de ejecución de las implementaciones tenía otras variables que pudieran modificarlo además del tamaño de la imagen y los detalles de la propia implementación

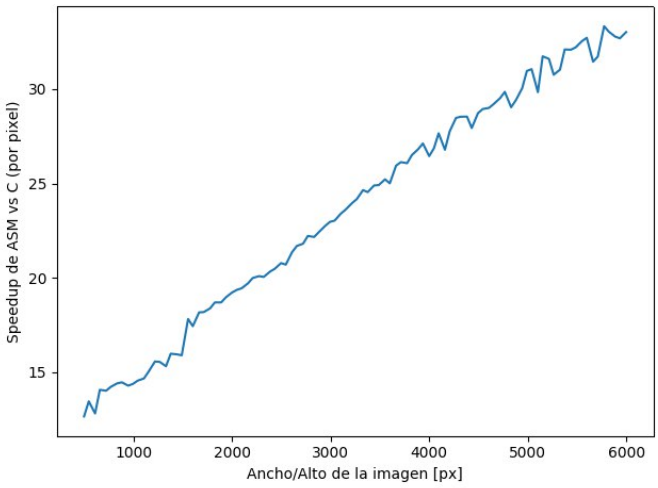
Para esto planteamos el siguiente experimento que consta de medir el tiempo de ejecución de las implementaciones para cada pixel, habiendo conjeturado que debería mantenerse igual independientemente del tamaño de la imagen que los contenga.



Como podemos ver, cuando el tamaño de la imagen empieza a crecer en cantidad de pixel's, el tiempo de implementación sube para ambas implementaciones, lo que nos hace pensar que es probable que para imágenes chicas se estaría haciendo uso pleno de la memoria cache, mientras que para imágenes que podemos considerar grandes (mayores a 3000px) esta memoria se llena y entra en un proceso más costoso de desalojo para ser rellenada.

Veremos en el siguiente gráfico como se comporta llevando las imágenes a tamaños aún más grandes y midiendo el SpeedUp de las implementaciones, que nos permitirá ver si alguna de las implementaciones hace mejor uso de la memoria cache.





De los gráficos anteriores podemos tomar dos conclusiones. Por un lado vemos que ampliar la imagen con la que experimentamos nuestras implementaciones también aumenta el costo por pixel para el filtro Bordes. Si tomamos en cuenta que estamos evaluando el tiempo que tarda el algoritmo en procesar cada pixel, y que este tiempo no dependería directamente del tamaño de la imagen, podemos notar que hay un factor externo que condiciona el tiempo de ejecución.

4. Conclusión

A partir de los resultados podemos deslizar algunas conclusiones generales que aplican a todos los filtros testeados:

- Las implementaciones hechas en *Assembler* son en su mayoría bastante más rápidas que las implementaciones hechas en *C*. Esto cumplió nuestras expectativas ya que las implementaciones en *Assembler* hacen uso de las instrucciones SSE y se procesan mas de un pixel a la vez.
- Par el filtro Nivel pudimos notar como detalles de la implementación afectan considerablemente el rendimiento, como ser el llamado a instrucciones çaras.ª nivel temporal como DIV. Además generamos dos implementaciones en ASM que operaban con distinta cantidad de pixel's en simultaneo y que implicancia tiene esto en la velocidad de ejecución.

- En el algoritmo de Bordes pudimos ver como la diferencia en la velocidad de resolución fue mayor porque al usar imágenes monocromáticas, cada pixel ocupaba menos bytes y por ende podíamos procesar mayor cantidad simultáneamente. Pudimos ver en el filtro Nivel, además, como mejora notablemente la implementación de C compilando el código con optimizaciones como o3.

- Medimos el impacto de la cache en el algoritmo de Bordes lo que nos permitió conocer como afecta esta variable externa a la implementación. Como una próxima posible experimentación queda pendiente agrandar el tamaño de la imagen y buscar el punto donde el tiempo se vuelve a estabilizar, lo que nos indicaría que se regulariza la situación de desalojo de la memoria cache.

En lineas generales, la realización de este trabajo además nos introdujo a una manera de pensar como implementar ciertas funciones de una manera distinta a la que estamos acostumbrados. Además, nos ha hecho conocer la sintaxis y algunos detalles para la implementación del modelo *SIMD*.

La programación vectorial es muy útil para optimizar funciones ya que se ve claramente la ganancia en performance que hay al programar de esta manera. Sin embargo, se debe pagar el costo de la complejidad de los algoritmos ya que el número de instrucciones aumenta significativamente y se debe pensar cuidadosamente como se cargan los datos de memoria. Muchas veces los datos deben ser transformados y reacomodados para poder usarlos con las operaciones *SIMD*. Otra desventaja que tiene SIMD es la atadura a una arquitectura específica haciendo nuestro programa dificil de exportar.

4.0.1. Apéndice Rombos: asm 2 vs 4 píxeles

Con el resultado de la segunda medición se vio que la diferencia entre implementaciones fue mucho menor a la esperada por nuestra hipótesis. Creemos esto se debe a que el overhead de las instrucciones ancilares es mas grande de lo que previnimos y eclipsa la diferencia obtenida por hacer menos cuentas vectoriales al operar con 4 píxeles.

De la anomalía observada en la primera medición también concluimos que no basta con hacer cuentas con vectores mas grandes para apreciar una mejora en desempeño, sino que es igual o mas importante tener en cuenta el overhead que genera cada instrucción. Por ejemplo es el caso de DIV.