DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 1

Viernes 14 de Abril de 2017 — Organización del Computador

Integrante	LU	Correo electrónico
Guido Maceira	231/15	guidomaceira@gmail.com
Nicolás Martín Obesio	258/15	obesio.nicolas@gmail.com
Ivan Vercinsky	141/15	ivan9074@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - Pabellón I
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Resumen

El objetivo de este trabajo practico es la implementación de filtros de imágenes utilizando el conjunto de instrucciones de Intel SSE, que responden a la técnica SIMD (Múltiples Datos, Una Instrucción).

Palabras claves
◦ Filtros Imagenes
◦ C
◦ ASM

Índice

1. Introducción Teórica

1.1. Introducción

El objetivo de este trabajo practico es la implementación de filtros de imágenes utilizando el conjunto de instrucciones de Intel SSE, que responden a la técnica SIMD (Múltiples Datos, Una Instrucción).

Mediremos las mejoras de rendimiento que se obtienen al implementar los filtros en lenguaje ensamblador, utilizando el conjunto de instrucciones SSE, en contraposición a su implementación en lenguaje C.

1.2. Filtros de imágenes

Los filtros de imágenes son procesos que permiten, a partir de una imagen fuente, obtener otra que posea propiedades deseables como bordes realzados, menor ruido o modificación de colores.

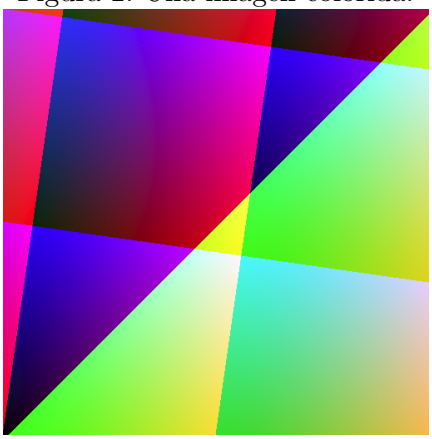
1.3. Materiales

Todos los experimentos fueron realizados utilizando la misma computadora para que los resultados fueran comparables. La computadora posee un procesador Intel(R) Core(TM) i5-2500K con 4 núcleos y una velocidad de clock de 3.30GHz. Este procesador cuenta con una cache de 3 niveles con capacidades de 32 KB, 256 KB y 6 MB respectivamente. La computadora cuenta con 8 GB de memoria RAM y utiliza Ubuntu 14.04 LTS en su versión de 64 bits.

Se utilizaron las siguientes imágenes, en distintas dimensiones, para la aplicación de los filtros:

Figura 1: Lena Söderberg

Figura 2: Una imagen colorida.



La computadora contaba con una versión de Python 3 y la librería gráfica Matplotlib. Los mismos eran requeridos para correr el script, que diseñamos, para automatizar la toma de mediciones.

1.4. Metodología de las mediciones

Se realizaron 100 ejecuciones por cada implementación de cada filtro por cada imagen. Luego de realizar las mediciones utilizando las herramientas provistas por la cátedra para la interacción con el "Time Stamp Counter".

Se realizaron 100 mediciones debido a la gran variabilidad generada por diversos factores como, por ejemplo, el Process Scheduler del sistema operativo.

Se eliminaron los outliers y se calculó la mediana de los datos con su desvío estándar. Creamos un Script, en Python 3, para automatizar las mediciones, pre-procesamiento de los datos y generación de gráficos. El mismo puede ser encontrado en el repositorio, que se encuentra adjunto a este documento.

Tambien se ponen a disposición en el repositorio, las mediciones obtenidas en cada uno de los experimentos individualmente, como así también gráficos de las mismas que nos indican los tiempos de ejecución en ciclos de clocks o tiempo en milisegundos.

2. Filtro ConvertRGBtoYUV

2.1. Descripción

El filtro a implementar convierte imágenes del espacio de color RGB al espacio de color YUV, y viceversa. Este filtro es análogo a la siguiente transformación lineal:

$$\begin{bmatrix} Y' \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0{,}299 & 0{,}587 & 0{,}114 \\ -0{,}14713 & -0{,}28886 & 0{,}436 \\ 0{,}615 & -0{,}51499 & -0{,}10001 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1{,}13983 \\ 1 & -0{,}39465 & -0{,}58060 \\ 1 & 2{,}03211 & 0 \end{bmatrix} \begin{bmatrix} Y' \\ U \\ V \end{bmatrix}$$

La cátedra nos dio una implementación de estos filtros utilizando operaciones aritméticas sobre enteros.

$$RGBtoYUV(R,G,B) = \begin{bmatrix} Y = satura((((66 \cdot R + 129 \cdot G + 25 \cdot B + 128) >> 8) + 16) \\ U = satura((((-38 \cdot R - 74 \cdot G + 112 \cdot B + 128) >> 8) + 128) \\ V = satura((((112 \cdot R - 94 \cdot G - 18 \cdot B + 128) >> 8) + 128) \end{bmatrix}$$

$$YUVtoRGB(Y,U,V) = \begin{bmatrix} R = satura((298 \cdot (Y-16) + 409 \cdot (V-128) + 128) >> 8) \\ G = satura((298 \cdot (Y-16) - 100 \cdot (U-128) - 208 \cdot (V-128) + 128) >> 8) \\ B = satura((298 \cdot (Y-16) + 516 \cdot (U-128) + 128) >> 8) \end{bmatrix}$$

2.2. Implementación en C

Idea: La idea era recorrer la matriz utilizando las diferentes transformaciones lineales para cada componente.

Código:

```
...
    void  C_convertPixelFromRGBtoYUB(RGBA * src_pixel, YUVA * dest_pixel) {
        uint8_t a_comp = src_pixel->a;
        uint8_t r_comp = src_pixel->r;
        uint8_t g_comp = src_pixel->g;
        uint8_t b_comp = src_pixel->b;

        uint8_t y_comp = aplica_tl_saturando(r_comp, g_comp, b_comp, tlY);
        uint8_t u_comp = aplica_tl_saturando(r_comp, g_comp, b_comp, tlU);
        uint8_t v_comp = aplica_tl_saturando(r_comp, g_comp, b_comp, tlV);

        dest_pixel->a = a_comp;
        dest_pixel->y = y_comp;
        dest_pixel->u = u_comp;
        dest_pixel->v = v_comp;

    }
    ...
```

2.3. Implementación en ASM

Idea: La idea era definir un ciclo generico para todas las componentes y darle el comportamiendo para transformar adecuadamente via mascaras. Para cualquiera de las dos conversiones.

Código:

```
.V:
movdqu xmm1, xmm0;  XMM1 = [ r1 * RM  | g1 * GM   | b1 * BM  | a1 * AM ]
pmulld xmm1, [vMaskRGBtoYUV]; XMM1 = [ r1 * RM | g1 * GM | b1 * BM | a1 * AM ]
movdqu xmm2, xmm1; XMM1 = [ r1 * RM  | g1 * GM   | b1 * BM  | a1 * AM ]
pslldq xmm2, 4;     XMM2 = [ g1 * BM  | b1 * GM   | a1 * AM  | 0  ]
paddd xmm2, xmm1;      XMM2 + XMM1 = [ -    | b1 * BM + g1 * GM  | -      | - ]
pslldq xmm2, 4;  XMM2 = [ b1 * BM + g1 * GM   | -   | -  | 0  ]
paddd xmm2, xmm1;   XMM2 + XMM1 = [  b1 * BM + g1 * GM + r1 * RM | -   | -  | -  ]
paddd xmm2, [addition128];     XMM2  = [  (b1 * BM + g1 * GM + r1 * RM ) + 128 | -   | -  | -  ]
pand xmm2, [only127_95];
psrad xmm2, 8;     XMM2  = [  ((b1 * BM + g1 * GM + r1 * RM ) + 128 ) >> 8 | -   | -  | -  ]
paddd xmm2, [addition128];      XMM2  = [  ((b1 * BM + g1 * GM + r1 * RM ) + 128 ) >> 8)+128
| -   | -  | -  ]
pand xmm2, [only127_95]; XMM2 = [ V | 0 | 0 | 0 ]
```

2.4. Implementación Alternativa en ASM

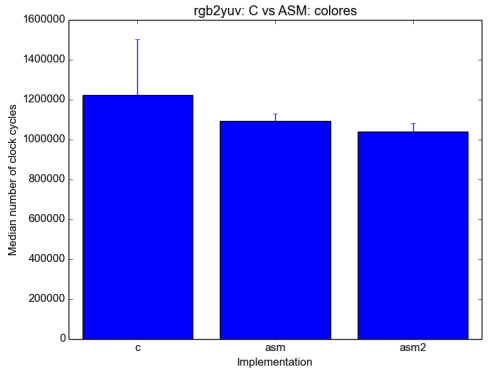
Motivacion: La idea era no cargar las mascaras en el loop. Sino tenerlas pre-cargadas en registros antes del loop.

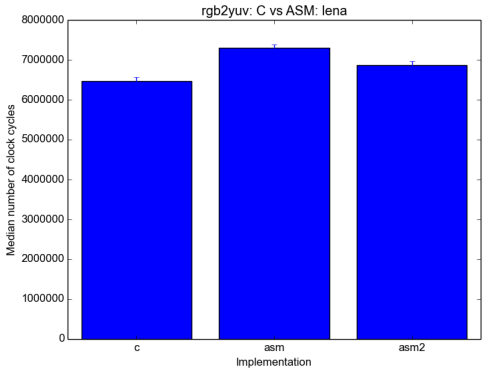
Hipotesis Nuestra hipotesis es que este cambio va a dismunuir el tiempo de ejecucion de la funcion. Ya que no va a tener que acceder a memoria tantas veces.

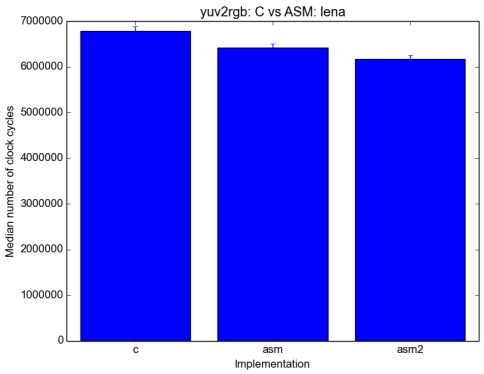
Codigo:

```
movdqu xmm6, [only31_0]
movdqu xmm8, [only127_95]
movdqu xmm9, [addition128]
movdqu xmm10, [sumBMaskYUVtoRGB]
movdqu xmm11, [bMaskYUVtoRGB]
movdqu xmm12, [sumGMaskYUVtoRGB]
movdqu xmm14, [gMaskYUVtoRGB]
movdqu xmm15, [sumRMaskYUVtoRGB]
    ...
```

2.5. Resultados

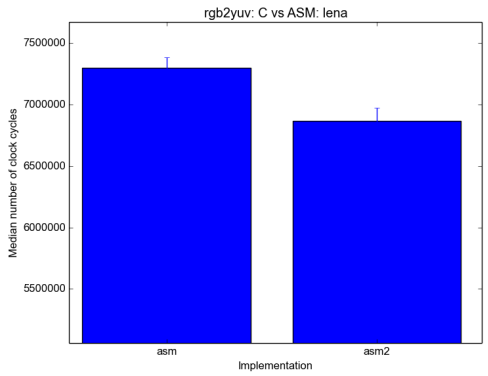


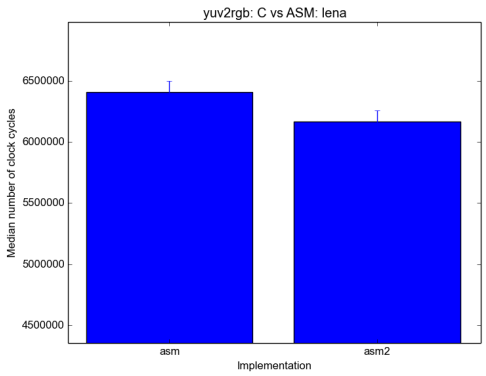




Como se puede observar en los graficos superiores los resultados son los esperados. Donde la implementacion en C es la mas lenta, la primer implementacion en ASM es aproximadamente 4 % mas rapida, y finalmente la implementacion alternativa es aproximadamente 15 % mas rapida que la de C donde no accedemos a memoria en el loop.

Excepto en el caso de lena rgb2yuv CvsASM donde se puede ver que en el grafico que C fue el mas rapido. Creemos que esto de se puede deber a optimizaciones que realizo el compilador a la hora de leer los datos de memoria. O, que el compilador haya encontrado una mejor manera de paralelizar las operaciones.





La implementacion alternativa efectivamente fue más rapida en ambos casos. Esto confirma nuestra hipotesis. Esta es que no acceder a memoria en el loop mejora la performance del algoritmo.

3. Filtro Four Combine

3.1. Descripción

El filtro a implementar reordena los pixeles de la imagen fuente y los organiza de acuerdo a la siguiente ilustración:

11	12	13	14	15	16	17	18
21	22	23	24	25	26	27	28
31	32	33	34	35	36	37	38
41	42	43	44	45	46	47	48
51	52	53	54	55	56	57	58
61	62	63	64	65	66	67	68
71	72	73	74	75	76	77	78
81	82	83	84	85	86	87	88

11	13	15	17	12	14	16	18
31	33	35	37	32	34	36	38
51	53	55	57	52	54	56	58
71	73	75	77	72	74	76	78
21	23	25	27	22	24	26	28
41	43	45	47	42	44	46	48
61	63	65	67	62	64	66	68
81	83	85	87	82	84	86	88

Como se puede observarm podemos separar los pixeles correspondientes a las filas pares e impares en 4 cuadrantes, teniendo en cada fila (pares e impares) los valores que irán a dos de ellos (primer y segundo / tercero y cuarto).

3.2. Implementación en C

Idea: La idea fue recorrer la matriz y para cada pixel calcular su posición en la matriz destino en base a su posición en la matriz origen.

Código:

```
...
void C_fourCombine(uint8_t* src, uint32_t srcw, uint32_t srch, uint8_t* dst, uint32_t dstw __attr
    RGBA (*matrix_src)[srcw] = (RGBA(*)[srcw])src;
    RGBA (*matrix_dst)[dstw] = (RGBA(*)[dstw])dst;
    uint32_t f = 0;
    uint32_t c = 0;
    for(f=0;f<srch;f++) {
        for(c=0;c<srcw;c++) {
            pos dest_pos = get_destPos(f,c,srcw);
            matrix_dst[dest_pos.f][dest_pos.c] = matrix_src[f][c];
        }
    }

}
...
```

En el código superior podemos observar la parte mas importante del ciclo, que es la que se encarga de completar los cuadrantes de la imagen destino.

3.3. Implementación en ASM

Idea: A la hora de realizar la programación de este filtro en assembly decidimos, en un mismo ciclo, escribir los 4 cuadrantes de la imagen destino, para esto leiamos 8 pixeles en cada uno. Los primeros 4 correspondientes a las filas pares, para completar los cuadrantes inferiores, y luego, los siguientes, 4 que eran corespondientes a las filas impares, para completar los cuadrantes superiores.

Para ello utilizabamos dos indices para el destino, uno para los cuadrantes inferiores y otro para los superiores. Estos índices, cada vez que se terminaba de leer una fila, se salteaban la siguiente.

Como se puede notar en la imágen de 'Descripción', de los 4 pixeles obtenidos de memoria, el 1º y el 3º corresponden a un cuadrante y el 2º y el 4º al otro. Para poder escribirlos en memoria adecuadamente, utilizamos un shuffle que nos los posicionó en orden para poder completar los cuadrantes como correspondia.

Código:

```
...
; Cuadrantes inferiores
movdqu xmm2 , [r12] ; Agarro los 4 en memoria
pshufd xmm3, xmm2 , 11011000b; Shuffle para invertirme los del medio
movq [r14],xmm3 ; Los copio a destino (Primer cuadrante)

psrldq xmm3, 8
movq [r14 + 2*rsi ],xmm3 ; Los copio a destino (Segundo cuadrante)

; Cuadrantes superiores
movdqu xmm4 , [r12+4*rsi] ; Agarro los 4 en memoria
pshufd xmm5, xmm4 , 11011000b; Shuffle para invertirme los del medio
movq [r9],xmm5 ; Los copio a destino (Tercer cuadrante)

psrldq xmm5, 8
movq [r9 + 2*rsi ],xmm5 ; Los copio a destino (Cuarto cuadrante)
    ...
```

En el código superior podemos observar la parte mas importante del ciclo, que es la que se encarga de completar los cuadrantes de la imagen destino.

3.4. Implementación Alternativa en ASM

Motivación: Como objetivo se buscaba comparar el costo de utilizar shuffles contra realizar shifts y utilizar mascaras (leyendolas de memoria en cada ciclo).

Hipótesis: Conjeturamos que, dado que tenemos que acceder a memoria a la hora de utilizar las mascaras y que, por lo comentado en clase, los shifts resultan (en términos generales) mas costosos que los shuffles, la implementación original debe ser más rápida.

Código:

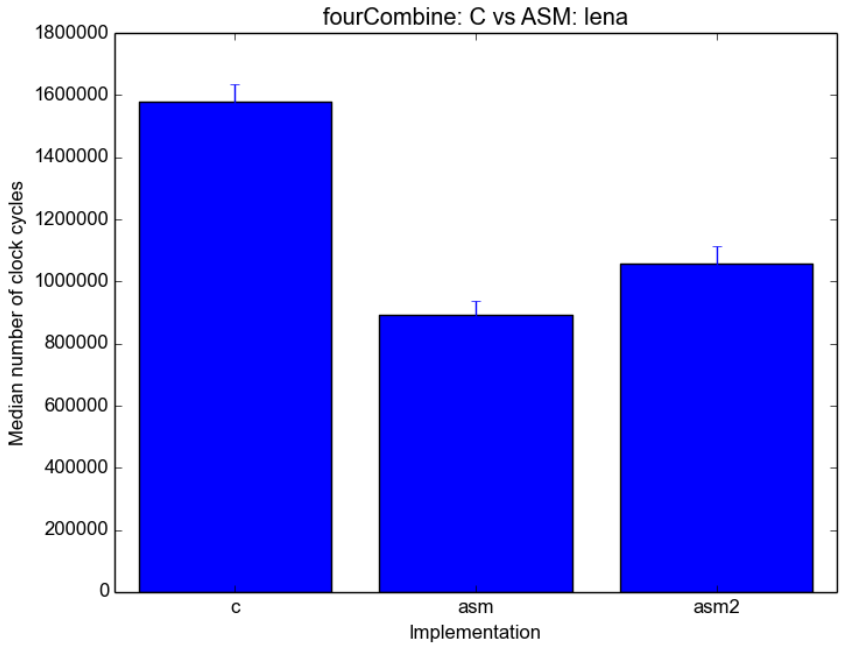
```
onlyP3P1: dd 0xFFFFFFFF, 0, 0xFFFFFFFF, 0
    ...
; Cuadrante inferior
movdqu xmm4 , [r12+4*rsi]  ; xmm4 = [p4,p3,p2,p1]
movdqu xmm5, xmm4 ; xmm5 = [p4, p3, p2, p1]
pand xmm5, [onlyP3P1] ; xmm5 = [00, p3, 00, p1]
movdqu xmm15, xmm5 ; xmm15 = [00, p3, 00, p1]
psrldq xmm15, 4 ; xmm15 = [00, 00, p3, 00]
por xmm5, xmm15 ; xmm5 = [00, 00, p3, p1]
movq [r9],xmm5  ; Los copio a destino
    ...
```

En esta porción de código, correspondiente al tercer cuadrante, se ve como fue implementado el cambio.

3.5. Resultados

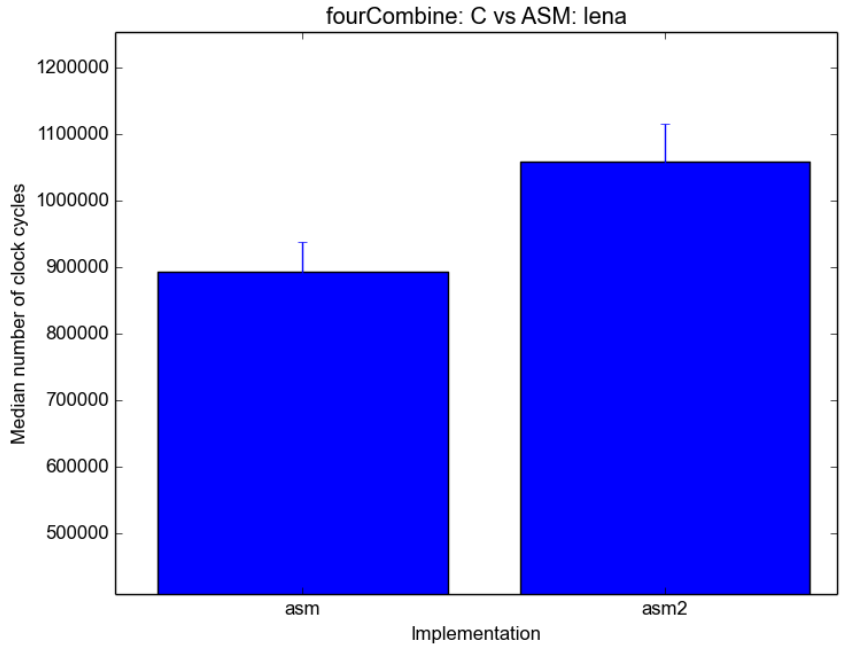
Los resultados obtenidos son correspondientes a la cantidad de ciclos de clock que tomó cada uno de los códigos utilizando la imágen 'Lena'.

Antes de comenzar a testear, supusimos que la perfomance de nuestras implementaciones en Assembler iban a ser superiores a las de C.



Cómo esperabamos, la implementación en C tomó una cantidad de ciclos de clock considerablemente superior (44 % aproximadamente) a la implementación en Assembly.

Esto se puede deber, principalmente, a la cantidad de accesos a memoria y las operaciones realizadas ya que, para en el caso de C, por cada pixel tenemos que leer de memoria de a 1 pixel y en base a ese, realizar todas las operaciones pertinentes para poder volver a escribirlo en memoria en su posición adecuada. En cambio, la lectura en ASM la realizamos de a 4 pixeles a la vez y, con operaciones mas simples, que nos permiten estimar la posición de memoria en la que grabaremos cada pixel.



Las distintas implementaciones en Assembly también resultaron como deseabamos pero con la diferencia que esperabamos una diferencia mayor ya que el acceso a memoria es 'costoso'. Luego de realizar experimentos para otros filtros (en este caso para LinearZoom), pudimos llegar a la conclusión que el acceso a las mascaras no implica el cambio mas significativo en este caso ya que, por la arquitectura del procesador, en la mayoria de los casos deben estar alojadas en caché (mas detallado en el filtro siguiente). Debido a esto podemos concluir que lo que tuvo mas peso en este caso fueron las operaciones intermedias que agregamos

para poder llegar al mismo resultado que al realizar un shuffle y, principalmente, los shifts utilizados para cada uno de los pixeles involucrados.

4. Filtro Linear Zoom

4.1. Descripción

El filtro a implementar permite duplicar las dimensiones de la imagen original. El filtro completa los nuevos espacios vacíos entre los pixeles con una interpolación lineal entre los pixeles lindantes. A continuación se puede observar una ilustración:

A	$(A+B)/2$	B
$(A+C)/2$	$(A+B+C+D)/4$	$(B+D)/2$
C	$(C+D)/2$	D

Para que el método no se indefina, se completa el borde superior y el borde izquierdo con una copia los extremos.

11	I	12	I	13	I	14	X
I	I	I	I	I	I	I	X
21	I	22	I	23	I	24	X
I	I	I	I	I	I	I	X
31	I	32	I	33	I	34	X
I	I	I	I	I	I	I	X
41	I	42	I	43	I	44	X
X	X	X	X	X	X	X	X

4.2. Implementación en C

Idea: Para implementarlo decidimos primero recorrer toda la imagen original que, a medida que leemos cada pixel, lo colocamos en la imagen destino en su correspondiente lugar. Todo esto, evitando la ultima fila que será completada al final, copiando la anteúltima fila.

Código:

```
for( uint32_t f = 0; f < srch ; f++){
for (uint32_t c = 0; c < srcw; c++) matrix_dst[(f*2)+1][c*2] = matrix_src[f][c];
}
```

Una vez recorrida la imagen original, comenzamos a recorrer las filas de destino, teniendo dos opciones: completar las filas que tienen los valores originales (impares) o las filas que no los tienen (pares).

En el caso de las primeras, recorremos los pixeles a completar y realizamos el promedio de los pixeles a sus costados.

En el otro caso, tenemos que completar todos los valores de la fila. Los pixeles que están en el medio de dos pixeles de la imagen original se completan, como en el caso anterior, sacando el promedio del pixel superior con el pixel inferior. Los otros, se obtienen mediante el promedio de los 4 pixeles de la imagen original mas cercanos (que es el promedio entre el valor de la fila superior e inferior a este).

Los extremos a completar, simplemente copian los pixeles de la columna/fila mas cercana.

4.3. Implementación en ASM

Idea: La idea implementada consiste en ir leyendo de memoria de a 4 pixeles para, primero, completar todas las filas pares. Ya con los cuatro pixeles, descartamos uno y trabajamos con los que me quedan. Con ellos realizamos las operaciones de promedio (pavg); para los primeros dos, el promedio es 'directo', luego, con el tercero que fue guardado, ocupamos la ultima posición disponible del registro con el promedio del segundo con ese, obteniendo: XMM = [P1 | (P1 + P2) / 2 | P2 | (P2 + P3) / 2]. Repitiendo este proceso hasta terminar de recorrer toda la imagen original y completando las filas pares de la imagen destino. En el caso que se deba guardar el último valor de la fila, copiamos directamente el anteultimo valor del registro al ultimo.

Luego, mediante otro ciclo, recorremos las filas impares que tendrán siempre, en la fila superior y en la inferior los valores previamente calculados. Tomamos de a 4 pixeles de ambas filas, calculamos el promedio

entre los valores de ambos entre pixel y pixel y los guardamos en la imagen destino.

Para la ultima fila realizamos un ciclo que copia directamente la anteúltima fila.

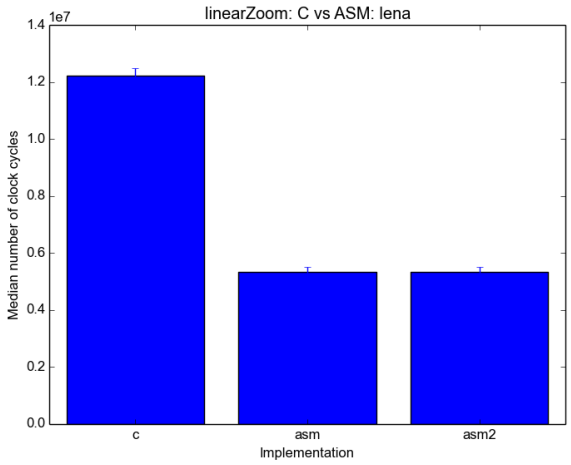
4.4. Implementación Alternativa en ASM

Motivación: El objetivo en esta implementación fue comprobar como afectaba el tiempo de ejecución, leer los datos de memoria de a uno a la vez y guardandolos en el archivo destino de a uno también.

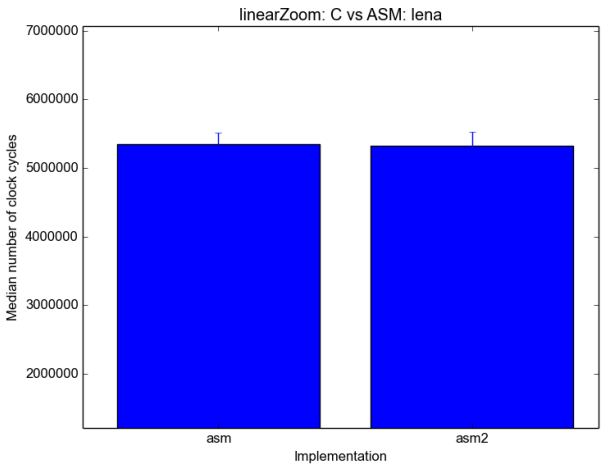
Hipótesis: Al aumentar la cantidad de lecturas y escrituras en memoria, el tiempo de ejecución debe verse afectado.

4.5. Resultados

En los siguientes gráficos veremos los resultados obtenidos:



Como esperabamos, la implementación en Assembly demandó menor cantidad de ciclos de clocks. Además de que haya sido por las ventajas que nos trae el lenguaje en si (por ejemplo, a la hora de realizar el promedio ya contabamos con una instrucción que lo realizaba), creemos que poder trabajar de a 4 pixeles al mismo tiempo en vez de a uno como en el caso de la implementación en C, tuvo gran importancia en lo que ha ciclos de clocks se refiere.



Por otro lado, la implementación alternativa en ASM no representó cambio alguno en el tiempo de ejecución. Asumimos que la manera en la que el procesador utiliza la memoria caché es de vital importancia, principalmente a la hora de leer los datos de memoria, como por ejemplo, la utilización de un principio de vecindad que a la hora de volver a pedir un dato en la memoria, produzca un hit en la memoria caché y nos evite ir a buscarlo a la memoria principal, ahorrandonos así ciclos de clock.

5. Filtro Max Closer

5.1. Descripción

El filtro a implementar calcula los máximos componentes de color para los pixeles contenidos en un kernel de dimensión 7x7 centrado sobre el pixel a procesar (Notar que no se puede aplicar a los pixeles que se encuentran a 3 o menos pixeles del borde). Luego de obtener los máximo valores para los canales "red", "green" y "blue" se calculan las nuevas componentes del pixel que se esta procesando de la siguiente forma:

$$\mathsf{dst}[i][j][R] = \mathsf{src}[i][j][R] \cdot (1 - val) + maxR \cdot (val)$$
$$\mathsf{dst}[i][j][G] = \mathsf{src}[i][j][G] \cdot (1 - val) + maxG \cdot (val)$$
$$\mathsf{dst}[i][j][B] = \mathsf{src}[i][j][B] \cdot (1 - val) + maxB \cdot (val)$$

Este procedimiento se repite para todo pixel no contenido en el borde.

5.2. Implementación en C

Idea: La idea detrás de la implementación en C fue recorrer toda la imagen. Si el pixel se encontraba a 3 o menos pixeles de cualquier margen este era remplazado por un pixel blanco en la imagen de destino. Caso contrario, utilizando dos for anidados se recorrían todos los pixeles del kernel de dimensión 7x7 centrado en el pixel a procesar y se calculaba el máximo valor para cada componente (rojo, verde, azul). Una vez calculado estos valores se aplicaba la transformación descripta en la sección anterior al pixel en cuestión.

Código:

```
...
    RGBA (*matrix_src)[srcw] = (RGBA(*)[srcw])src;
    RGBA (*matrix_dst)[dstw] = (RGBA(*)[dstw])dst;

    for (uint32_t x=0; x<srcw; x++) {
      for (uint32_t y=0; y<srch; y++) {
        RGBA pixelNuevo;
        if (2<x && x<srcw-3 && 2<y && y<srch-3){
        uint8_t max_r = 0, max_g = 0, max_b = 0;
        for (int i=-3; i<=3; i++) {
          for (int j=-3; j<=3; j++) {
            if (matrix_src[x+i][y+j].r > max_r) max_r = matrix_src[x+i][y+j].r;
            ...
          }
        }

        pixelNuevo.a = matrix_src[x][y].a;
        pixelNuevo.r = min((matrix_src[x][y].r * (1 - val)) + (max_r * val), 255);
        ...
        matrix_dst[x][y] = pixelNuevo;
      } else {
        pixelNuevo.a = 255;
        pixelNuevo.r = 255;
        ...
        matrix_dst[x][y] = pixelNuevo;
      }
      }
    }
...
```

5.3. Implementación en ASM

Idea: La idea detrás de la implementación en ASM fue cargar todos los pixeles del kernel de dimesión 7x7 en los registros del procesador para posteriormente poder calcular el máximo de forma simultanea.

Código:

```
...

    ;               ************************
    ;         xmm1  *         *    *         *  xmm8
    ;               ************************
    ;         xmm2  *         *    *         *  xmm9
    ;               ************************
    ;         xmm3  *         *    *         *  xmm10
    ;               ************************
    ;         xmm4  *         *HERE*         *  xmm11
    ;               ************************
    ;         xmm5  *         *    *         *  xmm12
    ;               ************************
    ;         xmm6  *         *    *         *  xmm13
    ;               ************************
    ;         xmm7  *         *    *         *  xmm14
    ;               ************************

  ; Cargamos los pixeles en memoria
    ...

    pmaxub xmm2, xmm1
    ...
    pmaxub xmm7, xmm6
    ; XMM7 <- Max R pixels (column wise)

    pmaxub xmm9, xmm8
    ...
    ; XMM14 <- Max L pixels (column wise)
...
```

5.4. Implementación Alternativa en ASM

Motivación: La idea detrás de la implementación alternativa en ASM fue disminuir el tiempo de ejecución al crear un "buffer de escritura" utilizando un registro de 128 bits donde poder alojar los pixeles procesados. Una vez que este registro se llenaba, lo escribíamos en el archivo destino. De esta manera reducíamos 4 veces la cantidad de accesos al disco.

Nos basamos en la premisa de que los accesos a memoria RAM son mas lentos que los accesos a los registros.

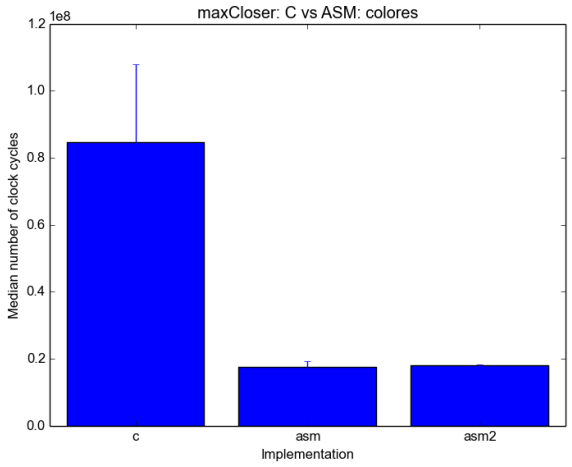
Código:

```
...
cmp r9, 4 ; r9 contiene la cantidad de pixeles en el buffer.
je .dump_cache
pslldq xmm15, 4
jmp .end_dump_cache

.dump_cache:
mov r12, r15
```

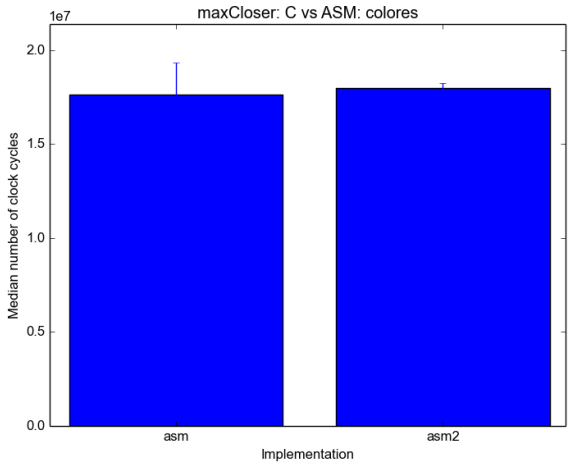
```
sub r12, 3
     imul r12, 4    ; r12 <- current w in pixels * pixel size in bytes

    mov rbx, r14
     imul rbx, rsi
     imul rbx, 4    ; rbx <- current h in rows * pixel per row * pixel size in bytes

    add rbx, r12   ; rbx <- current offset to pixel
     mov rax, rbx   ; rax <- current offset to pixel

pshufd xmm15, xmm15 , 00011011b ; Reacomodamos los pixeles del buffer (Se insertaron LIFO)

movdqu [rcx+rax], xmm15 ; Escribimos al disco
mov r9, 0
pxor xmm15, xmm15
.end_dump_cache:
...
```

5.5. Resultados

A continuación se puede observar los resultados de nuestras mediciones:



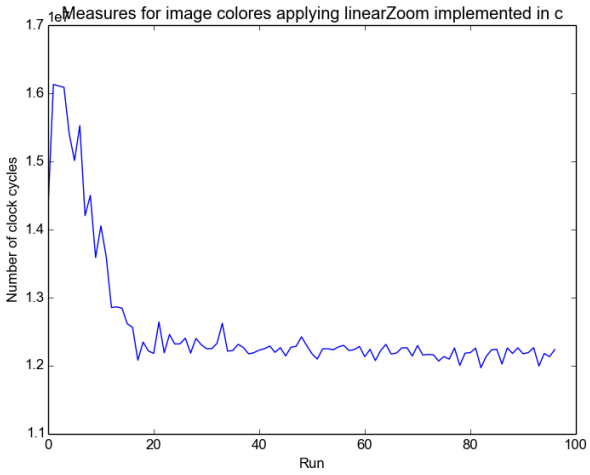
Como se puede observar en el gráfico superior nuestra hipótesis no se cumplió. Nuestra implementación alternativa "asm2" tuve un rendimiento inferior a su contra parte "asm". Ambas implementaciones en ASM tuvieron un tiempo de ejecución significativamente (aproximadamente un 325 % menos) inferior a la implementación en C. Creemos que esto se debe a que el costo para la utilización del "buffer de escritura" supera los beneficios que este nos ofrece.

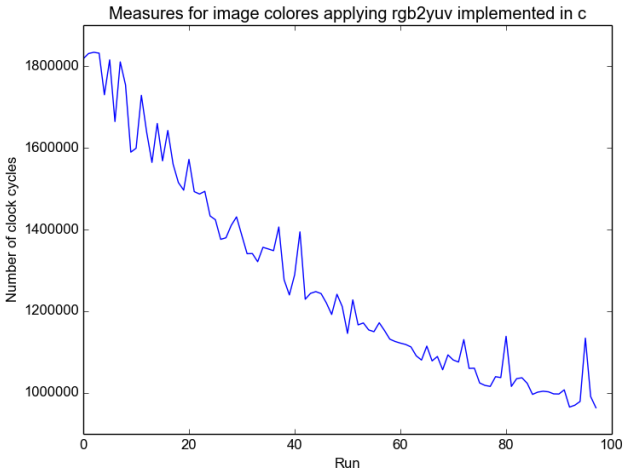


Observando detenidamente los gráficos notamos que las mediciones de la implementación alternativa en ASM posee un desvío estándar menor. Esto se debe a que tenemos menos accesos a la memoria principal, y por ende, menos variables que puedan afectar los tiempos de ejecución. Una variable que puede afectar, por ejemplo, es la cantidad de espera producidas por que el bus de datos a memoria principal este ocupado.

6. Observación general de experimentación

A la hora de ejecutar nuestros algoritmos para realizar las pertinentes comparaciones y experimentaciones, mediante nuestro script en Python, visualizamos un 'patrón' que explicaremos a continuacion y que esta a disposición en nuestro repositorio.





En las imágenes superiores tenemos las mediciones obtenidas en dos de nuestras implementaciones de filtros en C.

Como se puede observar, en ambos casos, siendo la mas notoria la implementación de C, va disminuyendo considerablemente la cantidad de ciclos de clocks que demoró el compilador en ejecutar el código, cosa que nos sorprendió ya que no lo esperabamos, o no con una diferencia tan marcada.

Luego de investigar, pudimos llegar a conjeturar que esto puede deberse a la optimización temporal del compilador, en este caso gcc.

La misma consiste en reducir el tiempo de ejecución de un programa, llegando internamente a un código optimizado que nos garantiza obtener la misma salida pero optimizando el tiempo. Esta mejora se puede dar gracias a la optimización del flujo de datos, de los bucles, reordenamiento de bloques, entre otros.

7. Conclusiones

Los resultados de las mediciones demostraron que la implementación en Assembly nos otorga una perfomance mejor (en la mayoria de los casos) y como vimos en este caso, en el procesamiento de imagenes. Para los casos en los que la perfomance no fue mejor, en gran parte se debe por la optimización detras del compilador y, de todos modos, la diferencia no es considerablemente grande.

Si bien puede no resultar cómodo o fácil para algunos de nosotros que estamos acostumbrados a programar en alto nivel, programar en Assembly y utilizando SIMD de manera adecuada como técnica empleada para conseguir paralelismo a nivel de datos, puede resultar muy efectivo.

Hay que destacar también que resulta de vital importancia cuales son nuestros objetivos finales a la hora de programar y cuando es realmente importante programar en Assembly. Tal vez la mejora que podemos obtener es muy poca o hasta nula, y probablemente hayamos perdido mayor tiempo que programando a alto nivel.

Por mas que uno pueda acostumbrarse al lenguaje, la utilización de comentarios en el código es importante ya que nos permitirá comprenderlo mas rapidamente debido a que es menos legible, como así tambien es mas sencillo cometer errores y el proceso de debugging es mas costoso.

Optimizar adecuadamente el código a veces puede implicar un gran costo, modificando gran parte de lo escrito, como a veces puede ser simplemente cambiar instrucciones.