Organización del Computador II
TP2

6 de mayo de 2015

Integrante	LU	Correo electrónico
Federico Beuter	827/13	federicobeuter@gmail.com
Juan Rinaudo	864/13	jangamesdev@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

El objectivo del trabajo practico es utilizar el set de instrucciones SIMD para el procesamiento de imagenes. Para esto se nos pidio implementar 2 versiones en ASM (x64) de 3 filtros diferentes (Blur, Merge, HSL), ademas se nos brindo una version en C para usar como guia.

Es importante destacar que las imagenes que vamos a procesar son multiplo de 4 pixeles y con un tamaño minimo de 16 pixeles, esto nos permite en algunos trabajar de a 4 pixeles sin preocuparnos de salir de la cantidad de memoria asignada a la imagen. Ademas el procesamiento de las imagenes se hace unicamente usando instrucciones SSE y durante el procesamiento de los mismos tratamos de mantenernos adentro de los margenes de error brindados por los test de la catedra.

Una vez implementado los 3 filtos y sus diferentes versiones se iniciara con la etapa de experimentacion, donde buscamos responder las preguntas brindadas por la catedra y de formar un mayor entendimiento de los algoritmos implementados para asi formar conclusiones y propuestas propias.

2. Filtro 1: Blur

2.1. Explicacion

El filtro de blur se aplica en cada pixel de la imagen exceptuando los del borde, para eso tomamos el pixel a modificar y los 8 adyacentes y los promediamos para obtener el valor final del pixel.

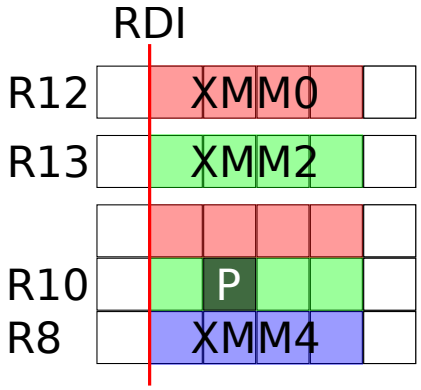
Debemos tener cuidado a la hora de procesar la imagen de no pisar los datos y operar con los pixeles de la imagen original.

2.2. Implementacion 1

La primera implementacion nos pide que trabajemos de a 1 pixel por iteracion.

Los pixeles estan compuestos por 4 bytes: A R G B, esto nos permite cargar de memoria en un registro XMM 4 pixeles, como nosotros necesitamos 9 pixeles en 3 filas de pixeles diferentes vamos a necesitar 3 registros XMM para cargar los pixeles (Para esto usamos los registros XMM0, XMM2 y XMM4). Y ademas necesito 3 registros XMM mas para luego desenpaquetar los bytes a words.

Ademas necesitaremos 6 registro de proposito general:
RDI que lo usamos para iterar sobre el eje X
R9 que lo usamos para iterar sobre y
R12 que es un puntero a la copia de la fila superior
R13 que es un puntero a la copia de la fila actual
R8 es un puntero a la inferior fila
R10 es un puntero a la fila actual de la imagen (fila que estoy modificando)



Antes de comenzar el ciclo incicializo R9 en 2 y posiciono el puntero a la imagen en la segunda fila de pixeles.

Al principio de cada ciclo de y muevo R8 a R10, aumento R8 en una fila y inicializo el iterador en x (RDI) en 0.

Muevo a los registros los 3 grupos de pixeles.
XMM0 = [R12 + RDI] = xx ; p2 ; p1 ; p0
XMM1 = [R13 + RDI] = xx ; p5 ; p4 ; p3
XMM2 = [R8 + RDI] = xx ; p8 ; p7 ; p6

Despues desenpaqueto los pixeles de byte a word para poder sumar los 9 pixeles sin saturacion, hacemos una copia de cada uno para poder desenpaquetar la parte inferior en un registro y la superior en otro (XMM1 = XMM0, XMM3 = XMM2, XMM5 = XMM4) y ademas llenamos un registro (XMM15) con ceros para expandir los componentes con ceros. Una vez desenpaquetados nos quedan los registros con los siguientes valores.
XMM0 = p1 ; p0
XMM1 = xx ; p2
XMM2 = p4 ; p3
XMM3 = xx ; p5
XMM4 = p7 ; p6
XMM5 = xx ; p8

Ahora sumamos los registros y luego hacemos la division.
Sumando XMM0, XMM2 y XMM3 en XMM15
XMM15 = p1 + p4 + p7 ; p0 + p3 + p6
Sumando XMM1, XMM3 y XMM4 en XMM14
XMM14 = x ; p2 + p5 + p8
Ahora hago una copia de XMM15 en XMM13 y la shifteo 8 bytes a la derecha
XMM13 = x ; p1 + p4 + p7
Por ultimo sumo XMM15, XMM14 y XMM13 en XMM15
XMM15 x ; p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8

Para hacer la division optamos por multiplicar por $(2^{16}/9)+1$ y luego shiftear 16 bits a la derecha cada componente. Para eso tengo el valor por el cual voy a multiplicar en memoria precalculado, al principio del programa decido guardarlo en XMM12. Hago una copia de XMM15 en XMM14 y multiplico por XMM12 guardandome la parte superior de la mutiplicacion en XMM15 y la inferior en XMM14. Luego empaqueto los dos valores juntos como doubleword y los guardo en XMM14 y shifteo a 16 a la derecha. Por ultimo empaqueto los valores obtenidos como words y luego como bytes y los guardo en la posicion de memoria del pixel actual.

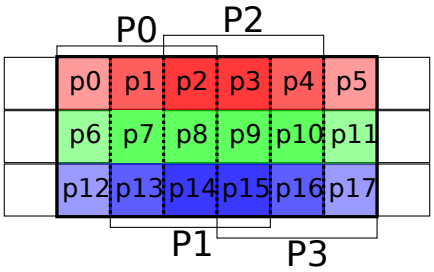
Aumento en 4 el iterador en x y comparo con el tamaño en bytes de una linea de pixeles, si es menor itero nuevamente en x.

Si es mayor o igual voy a hacer 2 copias de las lineas de pixeles siguientes (Que necesito para operar la siguiente fila).

2.3. Implementacion 2

La implementacion 2 nos pide trabajar de a 4 pixeles por iteracion.

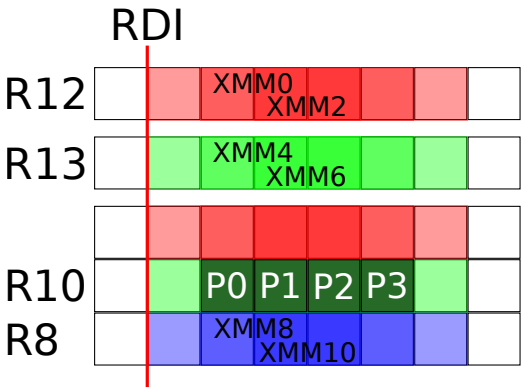
Para eso voy a calcular primero los pixeles P0 y P1, y despues el P2 y P3 tratando de minimizar la cantidad de accesos a memoria.



Para hacer los calculos vamos a tomar 6 set de 4 pixeles que nos permiten sumar en un solo registro 2 pixeles al mismo tiempo usando solo sumas, cada set ira adentro de un registro XMM. Ademas se necesitan 6 registros XMM mas para poder desenpaquetar los bytes a words y asi no tener saturacion.

Al igual que la implementacion uno necesitaremos los mismos 6 registros de proposito general.

Antes de comenzar el ciclo iniciamos el iterador en y (R9) y posicionamos el puntero a la imagen en la segunda fila de pixeles.



Al igual que en la primera implementacion nos guardamos una copia del puntero a la fila que vamos a incrementar y movemos R8 a la siguiente fila de pixeles. Ademas inicializamos el iterador de X (RDI) en 0.

Muevo a los registros los 6 pixeles.
XMM0 = p3 ; p2 ; p1 ; p0
XMM2 = p4 ; p3 ; p2 ; p1
XMM4 = p9 ; p8 ; p7 ; p6
XMM6 = p10 ; p9 ; p8 ; p7
XMM8 = p15 ; p14 ; p13 ; p12
XMM10 = p16 ; p15 ; p14 ; p13

Al igual que en la implementacion uno desenpaquetamos los registros haciendo una copia y desenpaquetando la parte superior en el registro original y la inferior en la copia. Para desenpaquetar uso un registro con ceros (XMM15)
XMM0 = p1 ; p0
XMM1 = p3 ; p2
XMM2 = p2 ; p1
XMM3 = p4 ; p3
XMM4 = p7 ; p6
XMM5 = p9 ; p8
XMM6 = p8 ; p7
XMM7 = p10 ; p9
XMM8 = p13 ; p12
XMM9 = p15 ; p14
XMM10 = p14 ; p13
XMM11 = p16 ; p15

Ahora sumo los registros XMM0, XMM1, XMM2, XMM4, XMM5, XMM6, XMM8, XMM9, XMM10 uno por uno con XMM15, al terminar XMM15 = p1 + p2 + p3 + p7 + p8 + p9 + p13 + p14 + p15 ; p0 + p1 + p2 + p6 + p7 + p8 + p12 + p13 + p14, esto como se ve en la imagen es igual a P1 — P0.

Ahora hago la division de cada pixel igual que en la primera implementacion, antes de dividir me guardo una copia de XMM15 en XMM0, divido por 9 y luego shifteo 8 bytes a la derecha y divido nuevamente.

Luego muevo los pixeles procesados P0 y P1 a memoria, posicionandolos con el offset correcto.

Ahora para procesar los pixeles P2 y P3 muevo 3 grupos mas de pixeles de la memoria a los registros XMM0, XMM4 y XMM8.
XMM0 = p5 ; p4 ; p3 ; p2
XMM4 = p11 ; p10 ; p9 ; p8
XMM8 = p17 ; p16 ; p15 ; p14

Ademas hago copias en XMM1, XMM5 y XMM9 y desenpaqueto de byte a word.
XMM0 = p3 ; p2
XMM1 = p5 ; p4
XMM4 = p9 ; p8

XMM5 = p11 ; p10
XMM8 = p15 ; p14
XMM9 = p17 ; p16

Nuevamente sumo los registros XMM0, XMM1, XMM3, XMM4, XMM5, XMM7, XMM8, XMM9, XMM11 uno por uno con XMM15, XMM15 = p3 + p4 + p5 + p9 + p10 + p11 + p15 + p16 + p17 ; p2 + p3 + p4 + p8 + p9 + p10 + p14 + p15 + p16, esto es P3 — P2.

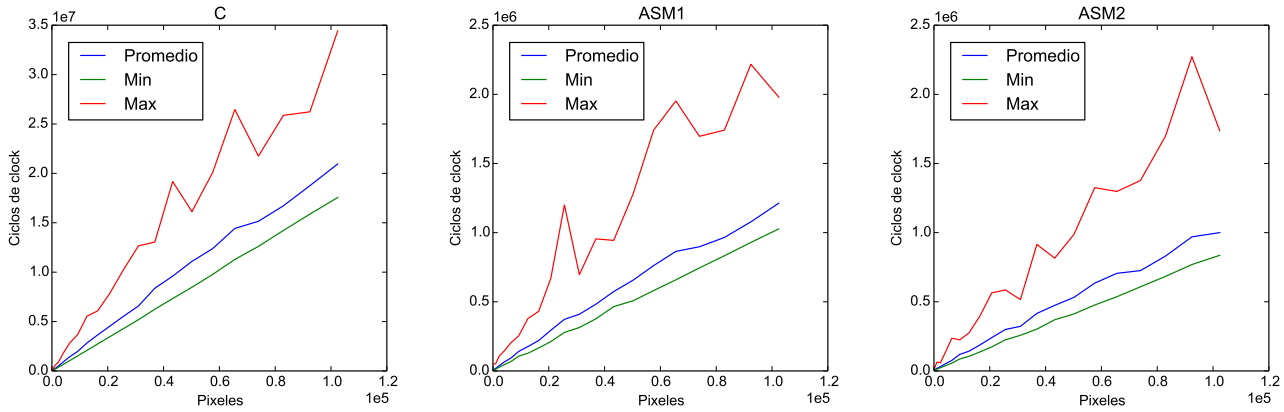
Hago la division al igual que arriba y guardo los pixeles procesados P2 y P3 en memoria.

Comparo el iterador con la cantidad de pixeles en una fila y repito hasta que llegue hasta los ultimos 16 bytes de la fila, en los ultimos 16 bytes me fijo de cargarlos en memoria en solo 3 registros XMM y procesarlos al igual que la primera parte del ciclo para asegurarme de no pasarme.

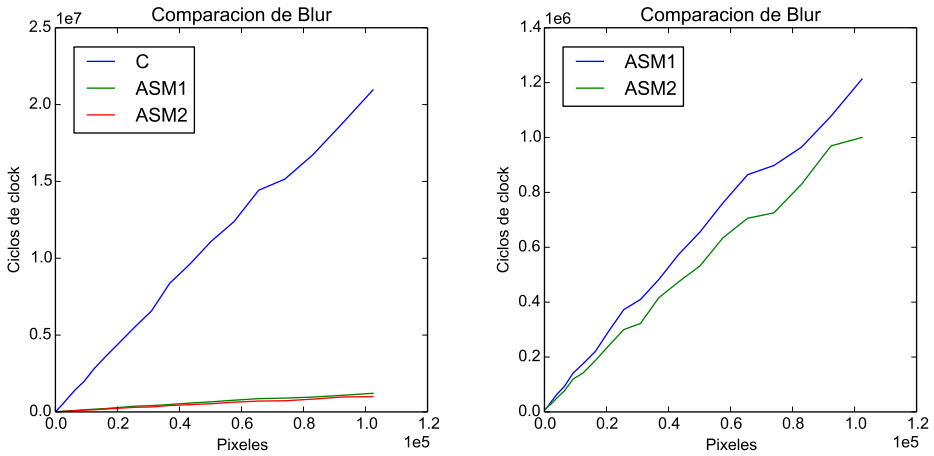
Al final del ciclo hago las copias, incremento el iterador de filas y checkeo si llegue al final.

2.4. Resultados

Para la experimentacion vamos a correr las 3 implementaciones (La version de C compilada con optimizaciones de nivel 3) con la imagen de lena brindada por la catedra (multiplos de 16x16, hasta 320x320), se corren 100 veces cada tamaño y despues se saca un promedio y se grafica el maximo, el minimo y el promedio.

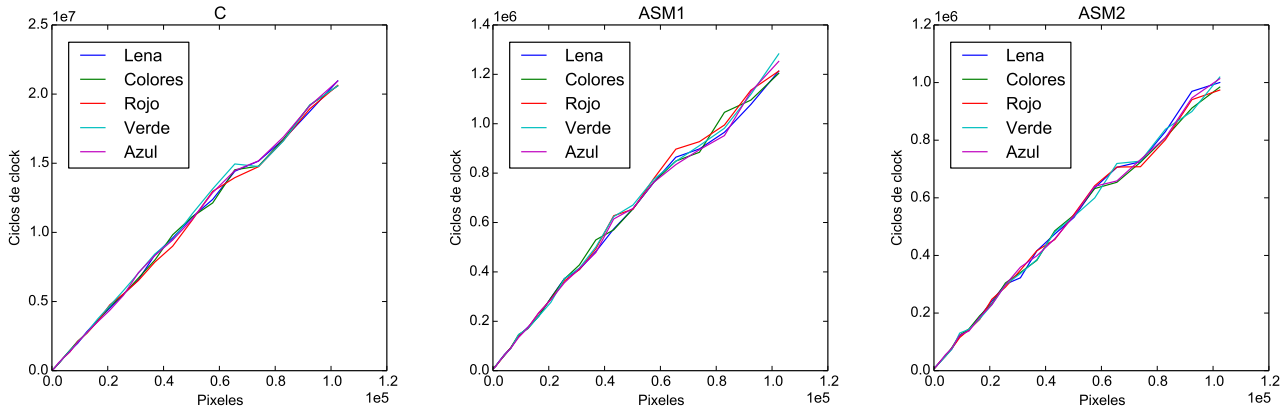


Ahora graficamos los 3 valores promedios de blur para ver cual de ellos es el mas rapido en general. Para ver mejor la diferencia entre ASM1 y ASM2 los graficamos aparte.



Este grafico nos muestra que el codigo de ASM2 pareceria ser el mas rapido. Antes de confirmar esto queremos ver que no haya ningun tipo de factor en la imagen que influya en el tiempo de ejecucion que no sea el tamaño.

Tanto en el codigo de C como en el ASM1 y ASM2 no poseen ninguna operacion que dependa de los pixeles a procesar, para probar que esto es realmente asi en la practica decidimos correr los test al igual que al principio pero esta vez usando la imagen de lena, colores y 3 imagenes mas completamente rojas, azules y verdes.



2.5. Conclusion

Luego de la experimentacion podemos concluir que ASM2 es el mas rapido en general.

La imagen a procesar no afecta en nada el tiempo de ejecucion y el tamaño lo afecta solo por que se deben procesar mas pixeles.

3. Filtro 2: Merge

3.1. Explicacion

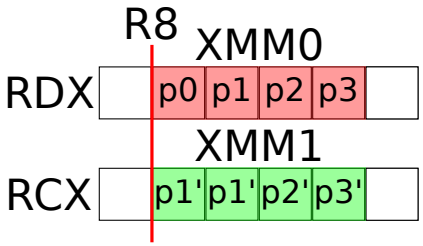
El filtro de merge consiste en tomar 2 imagenes del mismo tamaño y un float value (Con valor entre 0 y 1) y promediar los pixeles de cada imagen para crear una tercer imagen usando como parametro value.

3.2. Implementacion 1

La primera implementacion pide trabajar con floats procesando la mayor cantidad de pixeles posibles por iteracion.

Como sabemos que la cantidad de pixeles de las imagenes es un multiplo de 4 vamos a trabajar de a 4 pixeles que nos permite operar sin tener que procuparnos por salirnos de la memoria alocada a la imagen.

Para procesar los pixeles del merge vamos a usar 2 registros XMM para guardar los 4 pixeles como bytes que obtenemos de la memoria y 2 registros mas para guardar copias de los mismos. Ademas como vamos a utilizar una funcion auxiliar vamos a usar 4 registros XMM mas para guardar los resultados de la funcion. Y por ultimo tenemos los dos valores guardados en float en los registros XMM15 (value) y XMM14 (1 - value).



Ademas vamos a usar los siguientes registros de proposito general:
RDX puntero a la primera imagen
RCD puntero a la segunda imagen
R9 iterador en y R8 iterador en x (En bytes)

Es importante aclarar que antes de entrar en la iteracion creamos los registros XMM14 y XMM15 moviendo el value (XMM0) a XMM15 sufleandolo a todo el registro para que me quede v ; v ; v ; v y luego moviendo a XMM14 4 floats (1.0 ; 1.0 ; 1.0 ; 1.0) que tenemos almacenado en memoria y luego restandole a XMM14 XMM15.

Antes de comenzar a iterar en y inicializo el iterador de y en 0.

Al principio de cada ciclo de x inicializo el iterador en x en 0.

Luego muevo a los registros XMM0 y XMM1 los datos de la primera y segunda imagen. Ademas hago una copia de cada uno en XMM2 y XMM3 (XMM2 = XMM0, XMM3 = XMM1).

Ahora procedo a llamar a addPixels 4 veces y a guardar los valores en los registros XMM4, XMM5, XMM6, XMM7.

addPixels es una funcion auxiliar que hace lo siguiente:

Mueve los valores de las copias a XMM0 y XMM1 (XMM0 = XMM2, XMM1 = XMM3). Luego los desenpaqueta de byte a word y de word a doubleword usando el registro XMM10 el cual posee todos ceros. Los convierte a float, luego multiplica a XMM0 por XMM15 y a XMM1 por XMM14.

Luego de hacer esto queda en XMM0 a * v ; r * v ; g * v ; b * v y en XMM1 a * (1 - v) ; r * (1 - v) ; g * (1 - v) ; b * (1 - v).

Suma ambos y los pasa de float a doubleword, de doubleword a word y de word a byte.

Al final shiftea las copias (XMM2 y XMM3) 4 bytes a la derecha para que quede el siguiente pixel a procesar en los primeros 4 bytes de los mismos.

Una vez que tengo los valores lo unico que me falta hacer es pasarlo a memoria, incremento el iterador de x (R8 += 16) y hacer la comparacion, si no llegue al final de la fila de pixeles itero nuevamente.

Una vez que termine con la fila aumento el puntero de RDX en una fila, y hago lo mismo con el de RCX. Incremento en uno el iterador de y (R9++) y me fijo si llegue al final de la imagen, en caso de no ser asi itero nuevamente en y.

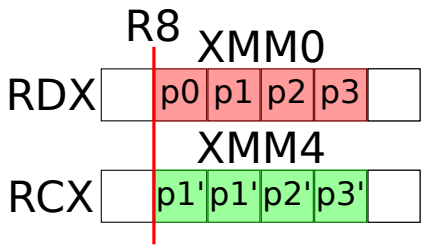
3.3. Implementacion 2

La segunda implementacion pide trabajar con enteros procesando la mayor cantidad de enteros posibles por iteracion.

Al igual que en la primera implementacion vamos a trabajar de a 4 pixeles al mismo tiempo por que nos permite iterar de manera segura.

Al igual que en la primera implementacion vamos a usar los registros XMM14 y XMM15 para guardar los valores por los cuales vamos a multiplicar los pixeles. Para calcularlos vamos a mover value (XMM0) a XMM15, luego lo shufleamos para que los 4 floats que entran en XMM15 sean v, movemos a XMM14 los floats en memoria (8192.0 ; 8192.0 ; 8192.0 ; 8192.0) y multiplicamos XMM15 por XMM14. Para finalizar le restamos a XMM14 XMM15 y convertimos ambos registros de float a doubleword. Al terminar esto nos queda:
XMM15 = v * 8192 ; v * 8192 ; v * 8192 ; v * 8192 XMM14 = 8192 - v * 8192 ; 8192 - v * 8192 ; 8192 - v * 8192 ; 8192 - v * 8192 = 8192 * (1 - v) ; 8192 * (1 - v) ; 8192 * (1 - v) ; 8192 * (1 - v)
Se elijio 8192 que es 2^{13} por que es el numero mas chico posible para usar y que el error que tenga al procesar la imagen este dentro de los parametros de los test.

Ademas vamos a usar los registros XMM0 a XMM7 para tomar los valores de memoria y desenpaquetar. Y XMM10 que es un registro con ceros tambien para desenpaquetar.



Los registros de proposito general que vamos a usar son los mismos que en la primera implementacion.

Antes de comenzar a iterar en incializamos el iterador de y (R9) en 0.

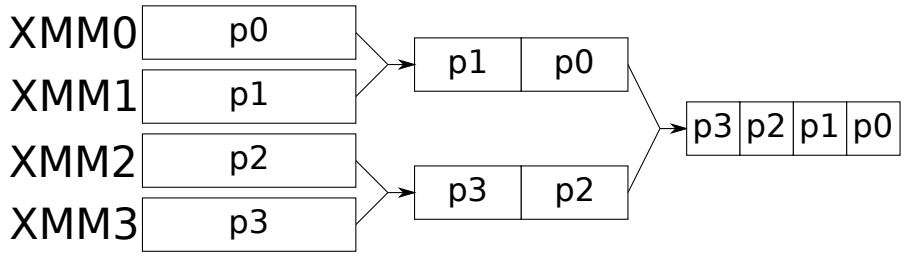
Al principio de cada iteracion de x inicializo el iterador de x (R8) en 0.

Luego muevo a XMM0 y XMM4 los grupos de pixeles de la primera y segunda imagen. Luego desenpaqueto de byte a word y de word a doubleword para que me queden los siguientes registros: XMM0 = p0 XMM1 = p1 XMM2 = p2 XMM3 = p3 XMM4 = p0' XMM5 = p1' XMM6 = p2' XMM7 = p3'

Despues multiplico XMM0, XMM1, XMM2 y XMM3 por XMM15 y lo shifteo 13 bits a la derecha, esto me da (p * 8192v) / 8192 = p * v

Hago lo mismo con XMM4, XMM5, XMM6 y XMM7 pero esta vez multiplicando por XMM14 y me quedan (p * 8192(1 - v)) / 8192 = p * (1 - v)

Luego me quedan en los registros cada pixel multiplicado por v o (1 - v) segun corresponda. Con un poco de paciencia y ingenio podemos enpaquetar las cosas para que queden los 4 pixeles como byte en un solo registro como se muestra en la siguiente imagen. Lo mismo funciona tanto para los dos grupos de pixeles.



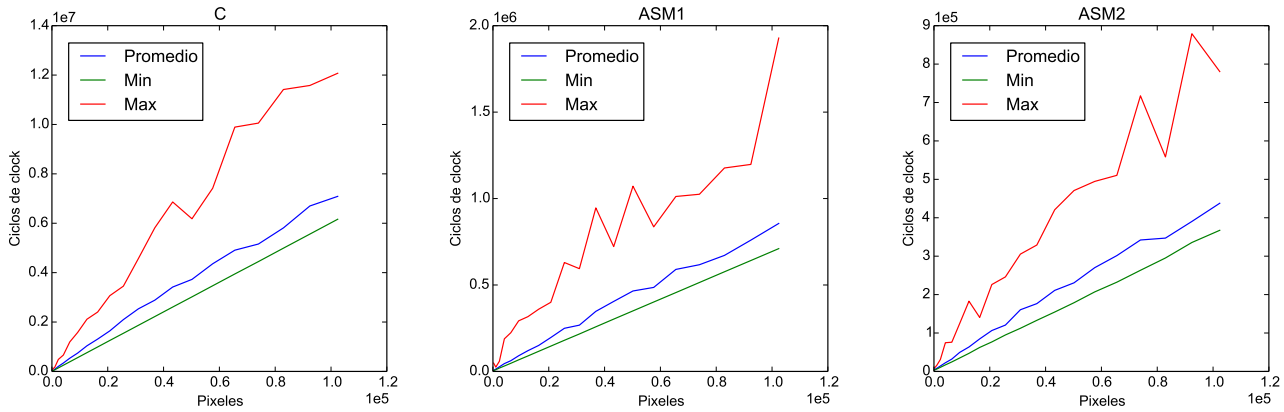
Al final nos queda en XMM0 p3 * v ; p2 * v ; p1 * v ; p0 * v y en XMM4 p3' * (1 - v) ; p2' * (1 - v) ; p1' * (1 - v) ; p0' * (1 - v)

Para terminar solo necesito sumarlos y moverlos a memoria. Esto se puede hacer con un solo mov.

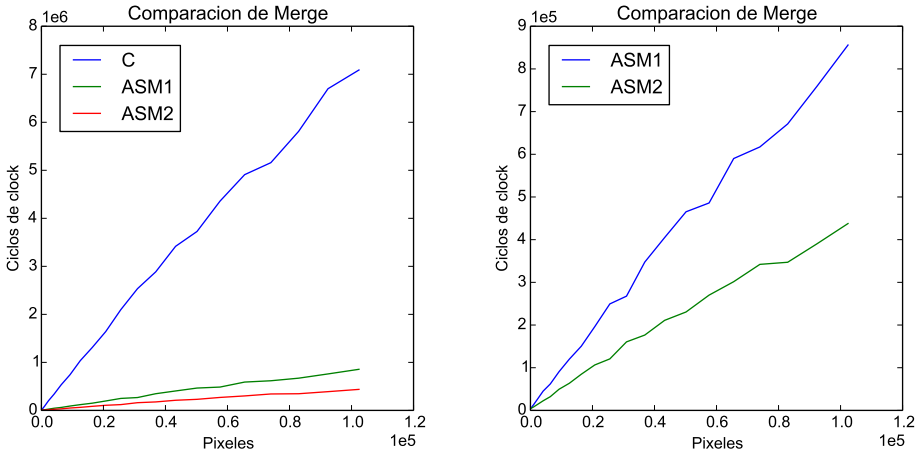
Luego al igual que en la primera implementacion checkeo si llegue al final sino itero en x nuevamenete y hago lo mismo con y.

3.4. Resultados

Para la experimentacion vamos a correr las 3 implementaciones (La version de C compilada con optimizaciones de nivel 3) con la imagen de lena brindada por la catedra (multiplos de 16x16, hasta 320x320), se corren 100 veces cada tamaño y despues se saca un promedio y se grafica el maximo, el minimo y el promedio.

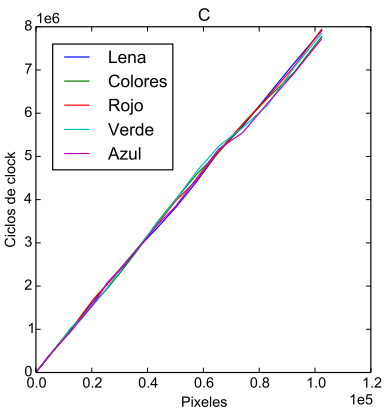


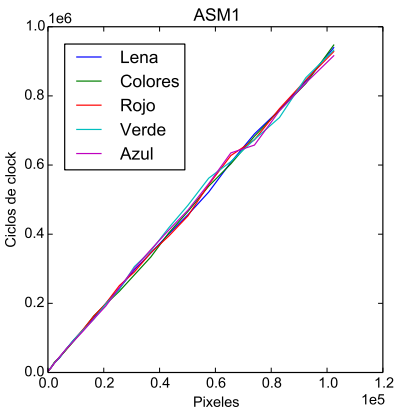
Tambien graficamos los 3 promedios en un grafico para ver cual de ellos es el mas rapido en general. Y para ver mejor la diferencia entre ASM1 y ASM2 los graficamos aparte.

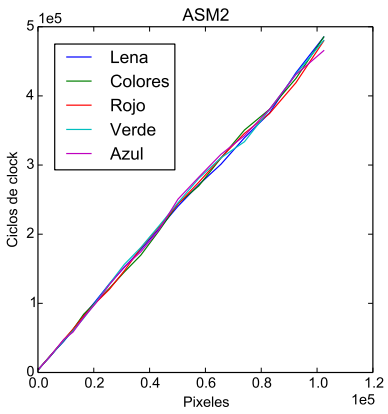


Tambien queremos ver que la imagen no modifique el tiempo de ejecucion, tanto el codigo de C com el de ASM no poseen ningun salto condicional que depende de el valor de los pixeles.

3.5. Conclusion


C
1e6
Lena
Colores
Rojo
Verde
Azul
Ciclos de clock
Pixeles
1e5


ASM1
1e6
Lena
Colores
Rojo
Verde
Azul
Ciclos de clock
Pixeles
1e5


ASM2
1e5
Lena
Colores
Rojo
Verde
Azul
Ciclos de clock
Pixeles
1e5