DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Grupo: Spyro_/_Playstation_1

Organización del Computador II

Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Iván Arcuschin	678/13	iarcuschin@gmail.com
Martín Jedwabny	885/13	martiniedva@gmail.com
Dan Zajdband	144/10	dan.zajdband@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

Índice

1. Introducción

Este Trabajo Práctico consistió de varias partes:

- En primer lugar, se realizaron las 2 implementaciones requeridas en lenguaje ensamblador para 3 filtros de imágenes distintos (un total de 6 implementaciones). Estos filtros fueron: *Blur*, *Merge* y *HSL*.

  El filtro **Blur**[1] consiste en reemplazar el valor de cada pixel de la imagen por el promedio de si mismo más sus 8 vecinos. En la primera implementación se realizó calculando pixel por pixel. En la segunda implementación se realizó calculando de a 4 pixeles a la vez.

  El filtro **Merge** consiste en, dado un valor $0 \leq v \leq 1$, combinar dos imágenes usando el valor v para decidir que porcentaje de cada una utilizar en la imagen final. Por ejemplo, dada una imagen *A* y una imagen *B*, $v = 0{,}4$, un pixel *i,j* resultante sería igual a $A_{i,j} * v + B_{i,j} * (1 - v)$. La primera implementación se realizó haciendo las cuentas necesarias para los pixeles con Floats Single Precision, mientras que en la segunda implementación se realizó haciendo las cuentas con Enteros (32 bits).

  Para el filtro **HSL**[2], las distintas implementaciones responden a las distintas etapas del filtro. La primera implementación es la etapa *Suma*: dada una imagen en *HSL*, 3 valores *HH*, *LL* y *SS*, incrementa cada canal de los pixeles en esos valores, saturando solo los canales *Saturation* y *Lightness* (el canal *Hue* va de 0 a 360). La segunda implementación es las etapas *RGBtoHSL* y *HSLtoRGB* que calculan para una imagen la respectiva transformación de un espacio de colores al otro. Adicionalmente:

  - Para el filtro Blur se realizo una implementación basada en el blur assembler version 1 en el cual se reemplazaron las instrucciones desempaquetadoras $punpck$ que hacian la conversion de enteros de 8 bit a 16 bit para calcular los resultados parciales por instrucciones shuffle $pshuffb$.
  - Para el filtro Merge se hizo una versión alternativa del merge assembler version 1 que cambia el orden de las cuentas de manera que en la operación del enunciado:
    $m[j][i][k] = m1[j][i][k] * value + m2[j][i][k] * (1 - value)$
    $m[j][i][k] = m1[j][i][k] * value + m2[j][i][k] - m2[j][i][k] * value$
    $m[j][i][k] = value * (m1[j][i][k] - m2[j][i][k]) + m2[j][i][k]$

- En segundo lugar, se realizó un análisis de los filtros implementados (en todas sus versiones: **C**, **ASM1** y **ASM2**. Siendo las realizadas en lenguaje **C** dadas por la cátedra). Éste análisis consistió en medir el tiempo de ejecución para distintas imágenes y comparar los resultados obtenidos.

- Por último, hicimos experimentaciones sobre varias imagenes en distintos tamanos para medir el tiempo que tarda cada cada algoritmo en procesarlas. Tomamos varias precauciones para realizar estas mediciones (detalladas en su seccion correspondiente) y una limpieza de outliers de manera que las comparaciones tengan sentido.

[1] Referencia `http://en.wikipedia.org/wiki/Gaussian_blur`
[2] Referencia `http://es.wikipedia.org/wiki/Modelo_de_color_HSL`

2. Metodología para medición de tiempos

A continuación vamos a detallar la metodología utilizada para recolectar los datos que luego se utilizarán en el análisis de las diferentes implementaciones.

La medicion del tiempo de ejecución se realizó utilizando la instrucción de assembler *rdtsc* (utilizando como ayuda el archivo *rdtsc.h* provisto por la cátedra), que permite obtener el valor del Time Stamp Counter (TSC) del procesador. Este registro se incrementa en uno con cada ciclo del procesador. Obteniendo la diferencia entre los contadores antes y despues de la llamada a la funcion, podemos obtener la cantidad de ciclos de esa ejecucion. Esta cantidad de ciclos no es siempre igual entre invocaciones de la funcion, ya que este registro es global del procesador y se ve afectado por una serie de factores, a ser:

1. La ejecucion puede ser interrumpida por el scheduler para realizar un cambio de contexto, esto implicará contar muchos mas ciclos (outliers) que los estrictamente necesarios para ejecutar nuestra función. En otras palabras, no tenemos garantizado, en principio, que nuestra función corra de inicio a fin sin que en el medio el procesador no se utilize para alguna otra función.

2. La frecuencia del reloj del procesador puede variar en el medio de la ejecución. Los procesadores modernos varían este valor para adecuarse a la demanda concreta en un tiempo determinado.

3. El procesador en el cual corre nuestra función puede ser utilizado por el sistema para atender alguna Interrupción (ya sea de Hardware o de Software). Esto se debe a que el sistema trata de *balancear* las interrupciones entre los distintos CPUs de una máquina para que todos funcionen de manera homogenea.

Para minimizar estas problematicas ideamos una serie de estrategias que ayuden a tener una medición en tiempo real más fiable:

1. Se utilizó la herramienta *cpuset*[3] que permite controlar la ubicacion de un proceso en la memoria y en los distintos procesadores. De esta forma, definimos un cpuset *sys* para los primeros $n-1$ cores que contuviera a todos los procesos del sistema y otro cpuset *rt* para el n-esimo core sin ningun proceso asignado.

   Al iniciar el script de medicion de tiempos, le asignamos su *PID* a los procesos del cpuset *rt*, lo que lo aisla de cualquier interferencia que pueda provenir del scheduler del sistema, ya que este no puede iniciar procesos en el cpuset *rt*.

2. Se utilizó la herramienta *cpufreq-selector*[4] que permite controlar la frecuencia de un procesador en particular. De esta forma, fijamos la frecuencia del n-esimo core a la más baja posible..

3. Para mitigar las interrupciones al n-esimo core, se reemplazaron los valores en las *smp_affinity* de todas las interrupciones listadas en la carpeta */proc/irq*, de tal forma que el core aislado no esa elegible para procesar las interrupciones.

   Si bien esto es útil, sigue habiendo interrupciones mínimas en el n-esimo core, como por ejemplo sincronizacion con los otros cores e interrupciones de timer.

Para más información, se agrega en el apéndice A el script utilizado para inicializar el entorno de medición.

[3] Referencia `http://man7.org/linux/man-pages/man7/cpuset.7.html`
[4] Referencia `http://csurs.csr.uky.edu/cgi-bin/man/man2html?cpufreq-selector+1`

3. Ejercicio 1 - Blur

3.1. Desarrollo

El filtro de Blur consiste en representar en los pixel de la imagen original como el promedio de este con sus vecinos inmediatos.

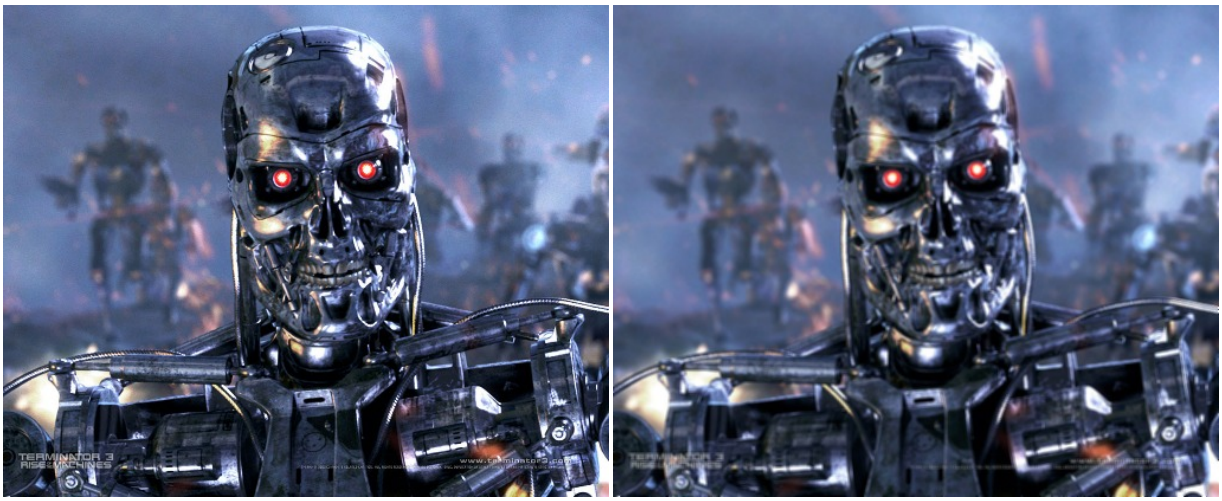


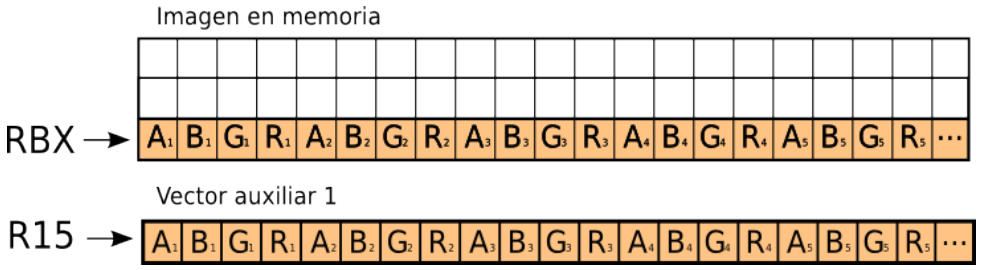
Imagen original | Resultado obtenido al aplicarle el filtro Blur

3.1.1. Blur 1

La primera implementación del filtro Blur se enfoca en entender como generar cada pixel de la imagen resultante y construir este proceso de la manera más sencilla posible. En esta implementación, cada pixel nuevo se genera en una iteración diferente del algoritmo.

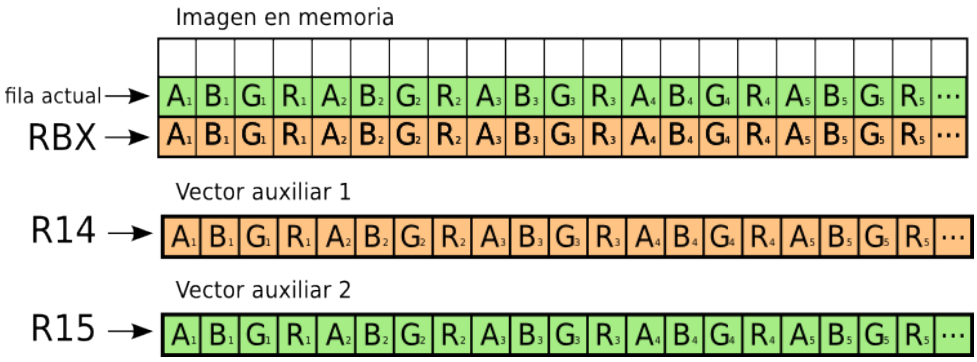
Procedimiento general Al comienzo del filtro se pide memoria para almacenar dos vectores del tamaño de una fila (ancho de la imagen), de modo que se puedan guardar los resultados parciales de filas que se tengan que reutilizar más adeltante sin polucionar la imagen original. Los punteros a estos dos vectors se guardan en `R14` y `R15`. También se guarda el puntero al inicio de la imagen en el registro `RBX`.

Inicialmente, copiamos la primera fila de la imagen en el vector que apunta `R15`.



Luego, el algoritmo recorre las filas de la imagen y dentro de cada fila itera por cada columna de la misma.

Al momento de empezar a operar con una nueva fila, el algoritmo intercambia los punteros en `R14` y `R15`, y luego carga en `R15` la fila siguiente a la actual. De esta forma, tenemos en `R14` la fila actual original y en `R15` la fila siguiente a la actual (la fila en la cual deberiamos ir escribiendo los pixel resultantes), por lo que no perdemos información durante los calculos de la fila.



Al finalizar todas las iteraciones, se libera la memoria pedida y cuyos punteros tenemos en `R14` y `R15`.

Iteración Por cada iteración se cargan los vecinos del pixel a procesar en 3 registros `XMM` utilizando la instrucción `MOVDQU` (sabemos hay un pixel de más cargado en cada registro). La fila anterior a la actual se lee tomando como base el puntero en `R14` y se guarda en el registro `XMM0`, la fila actual se lee tomando como base el puntero en `R15` y se guarda en el registro `XMM1`, y la fila siguiente a la actual se lee tomando como base la fila actual más el ancho de la imagen y se guarda en el registro `XMM2`.

Ya que estamos procesando imagenes en RGBA, los valores almacenados están en formato de Enteros de 8 bits, por lo que los convertimos a Enteros de 16 bits (1 Word) utilizando las instrucciones `PUNPCKLBW` y `PUNPCKHBW`. Esto es necesario ya que si realizaramos las operaciones aritmeticas en 8 bits correriamos el riesgo de perder precisión. Los nuevos valores se almacenan de la siguiente forma:

- La parte baja de `XMM0` en Enteros 8 bits va a `XMM0` como Enteros 16 bits.
- La parte alta de `XMM0` en Enteros 8 bits va a `XMM3` como Enteros 16 bits.
- La parte baja de `XMM1` en Enteros 8 bits va a `XMM1` como Enteros 16 bits.
- La parte alta de `XMM1` en Enteros 8 bits va a `XMM4` como Enteros 16 bits.
- La parte baja de `XMM2` en Enteros 8 bits va a `XMM2` como Enteros 16 bits.
- La parte alta de `XMM2` en Enteros 8 bits va a `XMM5` como Enteros 16 bits.

Ahora procedemos a sumar las componentes en los registros `XMM` mencionados utilizando la instrucción `PADDW`, que suma numeros empaquetados de a *Words*. El resultado de sumar las componentes de los pixel de la primera y segunda columna de la matriz de 3x3 quedan almacenados e la parte baja y alta respectivamente de `XMM0`. Y el resultado de sumar las componentes de los pixel de la tercera columna queda almacenado en la parte baja del registro `XMM3`.

Notemos que en la parta alta de `XMM3`, ahora tenemos la sumatoria de los pixel que seguian a nuestra matriz de vecinos de 3x3, por lo que limpiamos esa parte para poder realizar los siguiente pasos sin acarrear basura. Esto se realiza simplemente con la instrucción `PAND` y una máscara que tiene ceros en la parte alta y unos en la parte baja

Una vez realizado lo anterior, se procede a seguir sumando: utilizamos la instrucción `PADDW` y `PSRLDQ` para que nos quede en la parte baja de `XMM0` la sumatoria de todos los pixel de la matriz.

El siguiente paso es hacer la division de estos valores por 9, pero para eso necesitamos: primero convertirlos a Enteros de 32 bits y luego convertirlos a Floats SP. Lo primero se realiza con la instrucción `PUNPCKLWD` y lo segundo con `CVTDQ2PS`. Una vez que hicimos esto, cargamos en el registro `XMM8` una mascara que tiene nueves en formato Floats SP, y realizamos la division de floats empaquetados entre `XMM0` y `XMM8` con la instrucción `DIVPS`, lo que nos deja el promedio de cada canal (R, G, B y A) en formato Float SP en `XMM0`.

Ahora, volvemos a convertir los datos a Entero de 8 bits con las instrucciones: `CVTPS2DQ`, `PACKUSDW` y `PACKUSWB` (en ese orden), lo que nos deja en los 32 bits mas bajos de `XMM0` los valores. Utilizamos los `PACK` sin signo ya que sabemos que nuestros valores no contenian numeros negativos.

Por último, se escriben los 32 bits más bajos de `XMM0` en el pixel del centro de la matriz con la instrucción `MOVD`, utilizando para esto el registro `RBX`. Notese que para los calculos de los pixel que siguen

este valor escrito en la matriz no va a influir, ya que tenemos su valor original almacenado en el vector auxiliar al que apunta R15.

Observación: el último pixel de la imagen a procesar requiere un caso especial en el algoritmo, ya que si cargaramos los datos como para cualquier otro pixel estariamos leyendo datos que se van de la posicion de la imagen en memoria. Es por esto que lo que hacemos es restarle al contador de columnas un pixel, levantar cuatro pixel como haciamos en una iteración normal, y luego shiftear los registros XMM0, XMM1 y XMM2 4 bytes (1 pixel) con la instrucción PSRLDQ. Hecho esto, sigue la ejecución normal de la función.

3.1.2. Blur 2

La segunda implementación del filtro Blur ahora aprovecha la estructura espacial de la imagen para procesar 4 pixel en una sola iteración. En este sentido, optimizamos el algoritmo al utilizar todo el potencial que tenemos con las instrucciones de SSE.

Procedimiento general La primera parte del algoritmo es similar a la del primero, al punto que se reutiliza el código de Blur 1. La diferencia principal consiste en que las iteraciones saltean de a 4 pixel en vez de a 1, siendo un punto no menor ya que se realizan 1/4 de las iteraciones que en el caso de Blur 1.

En cuanto a los vectores auxiliares, se utilizan los mismos registros para mantener sus punteros: R14 y R15, y durante el algoritmo se manejan de la misma manera a la descripta en el Blur 1.

Iteración En este algoritmo, para poder procesar 4 pixel en la misma iteración se utilizan 6 registros XMM del siguiente modo:

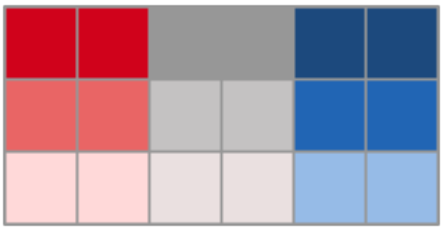


Figura 2: Registros XMM en Blur 2

En la figura 1 se muestra los pixel ocupados por cada registro R15. Los rojos corresponden a las columnas 1, 2, 3 y 4 mientras los azules corresponden a las columnas 3, 4, 5 y 6. Es por eso que los pixel de las columnas 3 y 4 se ven en gris, ya que se encuentran en ambos registros correspondientes a la misma fila.

La forma exacta en la que se guardan los pixel (todos en formato Enteros 8 bits) es:

- En XMM0 los pixel 1 a 4 de la fila anterior a la actual.
- En XMM1 los pixel 1 a 4 de la fila actual.
- En XMM2 los pixel 1 a 4 de la fila siguiente a la actual.
- En XMM3 los pixel 3 a 6 de la fila anterior a la actual.
- En XMM4 los pixel 3 a 6 de la fila actual.
- En XMM5 los pixel 3 a 6 de la fila siguiente a la actual.

Luego, convertimos los valores a Enteros de 16 bits (1 Word) utilizando las instrucciones PUNPCKLBW y PUNPCKHBW. La disposicion de los valores queda (todos en formato Enteros 16 bits):

- En XMM0 los pixel 1 y 2 de la fila anterior a la actual.
- En XMM6 los pixel 3 y 4 de la fila anterior a la actual.
- En XMM9 los pixel 5 y 6 de la fila anterior a la actual.

- En XMM1 los pixel 1 y 2 de la fila actual.
- En XMM7 los pixel 3 y 4 de la fila actual.
- En XMM10 los pixel 5 y 6 de la fila actual.
- En XMM2 los pixel 1 y 2 de la fila siguiente a la actual.
- En XMM8 los pixel 3 y 4 de la fila siguiente a la actual.
- En XMM11 los pixel 5 y 6 de la fila siguiente a la actual.

A continuación, se realizan las sumas con la instrucción PADDW, quedandonos:

- En la parte baja de XMM0 la suma de la primera columna.
- En la parte alta de XMM0 la suma de la segunda columna.
- En la parte baja de XMM6 la suma de la tercera columna.
- En la parte alta de XMM6 la suma de la cuarta columna.
- En la parte baja de XMM9 la suma de la quinta columna.
- En la parte alta de XMM9 la suma de la sexta columna.

Luego, pasamos los valores a Enteros de 32 bits con las instrucciones PUNPCKLWD y PUNPCKHWD, quedandonos:

- En XMM0 la suma de la primera columna.
- En XMM1 la suma de la segunda columna.
- En XMM6 la suma de la tercera columna.
- En XMM7 la suma de la cuarta columna.
- En XMM9 la suma de la quinta columna.
- En XMM10 la suma de la sexta columna.

Ahora, tenemos que juntar los valores de las columnas correspondientes para formar los resultados de los pixel que queremos procesar. Para el primer pixel, necesitamos sumar las columnas 1, 2 y 3, para el segundo las columnas 2, 3 y 4, y así siguiendo. Utilizando la instrucción PADDD nos queda:

- En XMM0 la suma de las columnas primera, segunda y tercera.
- En XMM1 la suma de las columnas segunda, tercera y cuarta.
- En XMM6 la suma de las columnas tercera, cuarta y quinta.
- En XMM7 la suma de las columnas cuarta, quinta y sexta.

Una vez realizado esto, solo queda dividir por 9 de la misma manera que se hizo en el Blur 1 (una máscara y la instrucción DIVPS), y convertir de vuelta a Enteros de 8 bits, utilizando las instrucciones: CVTPS2DQ, PACKUSDW y PACKUSWB (en ese orden).

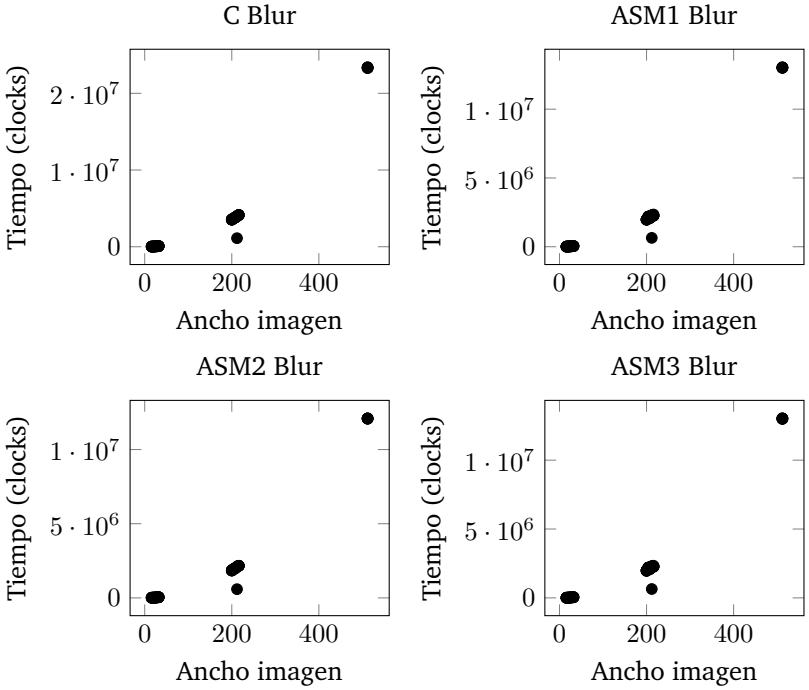
Por último, se acomoda los diferentes resultados para los pixel en el registro XMM0 (utilizando instrucciones PSLLDQ y POR) y se baja a memoria utilizando la instrucción MOVDQU, ya que estamos escribiendo 4 pixeles a la vez.

Observación: similarmente a lo que pasaba en la función Blur 1, la última iteración del algoritmo requiere un caso especial, ya que si cargaramos los datos como para cualquier otro pixel estariamos leyendo datos que se van de la posicion de la imagen en memoria. Es por esto que lo que hacemos es levantar los primeros 4 pixel de forma normal en los registros XMM0, XMM1 y XMM2, para luego copiarlos a los registros XMM3, XMM4 y XMM5 y shiftearlos 8 bytes a derecha con la instrucción PSRLDQ. De esta forma, podemos continuar la parte principal de la iteración de forma normal, ya que nos quedaría:

- En `XMM0` los pixel 1 a 4 de la fila anterior a la actual.
- En `XMM1` los pixel 1 a 4 de la fila actual.
- En `XMM2` los pixel 1 a 4 de la fila siguiente a la actual.
- En la parte alta de `XMM3` ceros, y en la parte baja los pixel 3 a 4 de la fila anterior a la actual.
- En la parte alta de `XMM4` ceros, y en la parte baja los pixel 3 a 4 de la fila actual.
- En la parte alta de `XMM5` ceros, y en la parte baja los pixel 3 a 4 de la fila siguiente a la actual.

3.2. Análisis y Resultados

Rendimiento (varias imagenes por tamano): En este caso se compara el rendimiento de un mismo algoritmo con varias imagenes en distintos tamaños



Comparacion entre versiones: Los siguientes graficos muestran para una misma imagen, los rendimientos obtenidos corriendo las diferentes versiones del algoritmo.
Observacion : las imagenes de “terminator” y “llamas” son las que aparecen al principio de la seccion 4 y la llamada “toda roja” esta compuesta solamente por ese color solido.

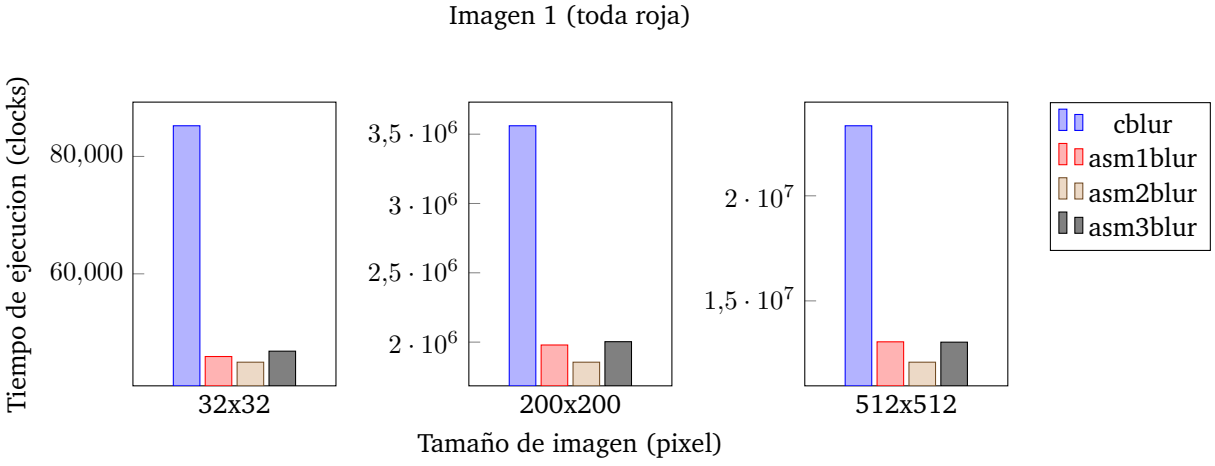


Imagen 2 (terminator)

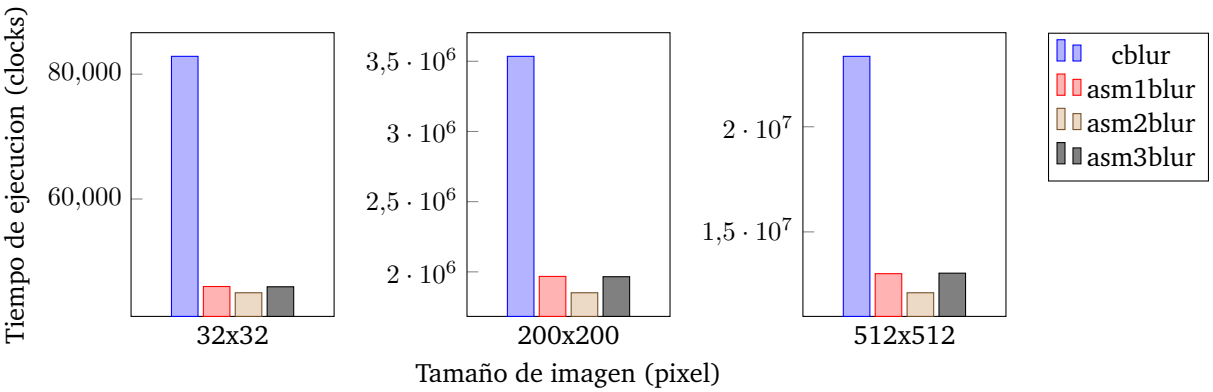
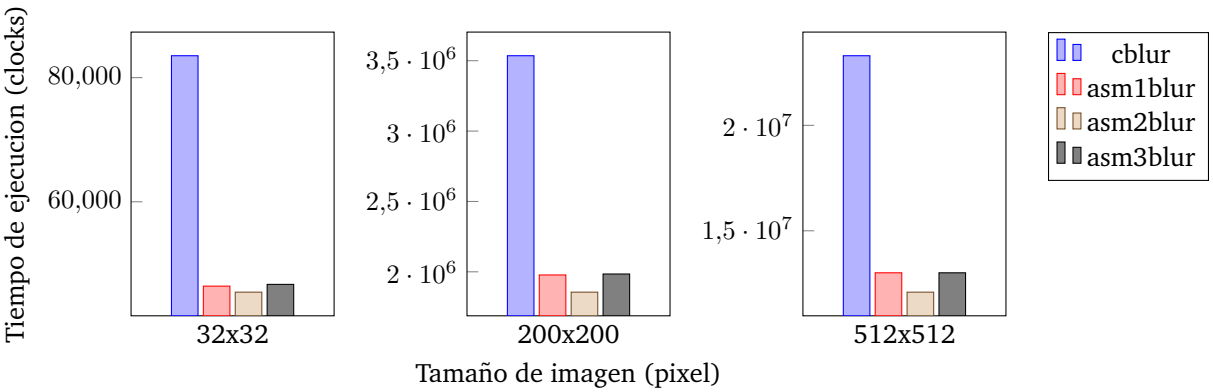


Imagen 3 (llamas)



Comparacion imagenes en tamaño 512x512				
Imagen	Tiempo cblur (clk)	Tiempo asm1blur (clk)	Tiempo asm2blur (clk)	Tiempo asm3blur
1	23333352	13051774	12082518	13035642
2	23351553	13012521	12105084	13037410
3	23298512	13008870	12091236	13008861

Se observa que en general la implementación de mejor rendimiento es ASM2, seguida por ASM1 y luego por C. La implementación en C es en promedio apróximadamente 15 veces más lenta que ASM1, que a su vez tarda en promedio %6 más que ASM2.

El resultado más predecible es la diferencia de velocidad entre C y ASM. Esta gran diferencia la atribuimos principalmente a la capacidad de realizar sumas en bloques en ASM que en C se deben realizar de a 1. Además, la posibilidad de almacenar y manipular la información en registros xmm evita una gran cantidad de accesos a memoria que en C se hacen individualmente por dato.

En las imágenes más pequeñas de 16x16 se nota una disminución en la diferencia de velocidad de procesamiento, esta cambia de 15 a 14 veces más rápida la implementación de ASM1 que la de C, mientras que en las más grandes de 256x256 la diferencia en favor de ASM1 es de aproximadamente 15.2. Pensando en una imagen de 16x16, donde el cambio de fila es muy frecuente, tiene sentido que el copiado de las filas auxiliares cobre más relevancia dentro del rendimiento del algoritmo, emparejando en cierto punto las implementaciones. Aún así, la diferencia 1/14 es suficientemente grande en favor de ASM1.

A su vez, la optimización de ASM2 sobre ASM1 a la hora de trabajar de a 4 pixel en vez de a 1 no parece ser tan importante como el pasaje de C a ASM.

No solo la implementación ASM2 no es sustencialmente más rápida que ASM1 sino que para imagenes pequeñas es más rápida la función ASM1 en ˜ %1. Si bien ASM2 aprovecha la vecindad de las sumas de

los pixel para hacer entre 2 y 3 menos sumas por pixel que ASM1 los accesos a memoria requeridos, que son a su vez más esparsos que los de ASM1, dañando el hit rate del cache, nivelan las implementaciones.

Se realizó una implementación alternativa de ASM1 llamada ASM3 enfocada en analizar los efectos de utilizar una máscara y funciones de shuffle para realizar el desempaquetado de datos en vez de usar las funciones punpck, sabiendo que las instrucciones de shuffle son lentas por naturaleza. Los resultados arrojan una leve diferencia en ASM1 respecto a ASM3 atribuída totalmente al uso de instrucciones de desempaquetado, ya que es lo que diferencia ASM1 de ASM3, sin embargo el cambio no es sustancial ya que el desempaquetamiento de datos es una parte pequeña del algoritmo.

Otra implementación interesante a realizar es la de variar la cantidad de vecinos que intervienen en la producción de cada nuevo pixel. Desde la utilización de sus 4 vecinos (izquierdo, derecho, arriba y abajo) hasta quizás 16 vecinos. A priori, mientras más vecinos se usen más suave será el blur y más lento el proceso, pensando en los accesos a memoria y la aritmética requerida.

4. Ejercicio 2 - Merge

4.1. Desarrollo

El filtro Merge genera una nueva imagen a partir de 2 imágenes origen, generando por cada pixel de las imágenes originales uno nuevo que posee un porcentaje de la primera imagen y otro de la segunda, donde dicho porcentaje es un parámetro del filtro. Si por ejemplo el parámetro es 0.1, cada componente RGB de la imagen resultante posee un 10 % de la componente de la primera imagen y un 90 % de la segunda.

El efecto resultante refleja la imagen cuyo componente es más fuerte con la imagen r̈ecesivaẗdifuminada.

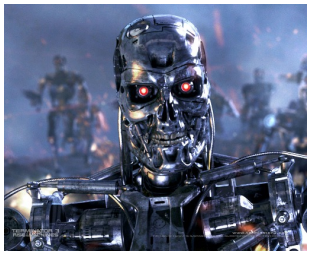
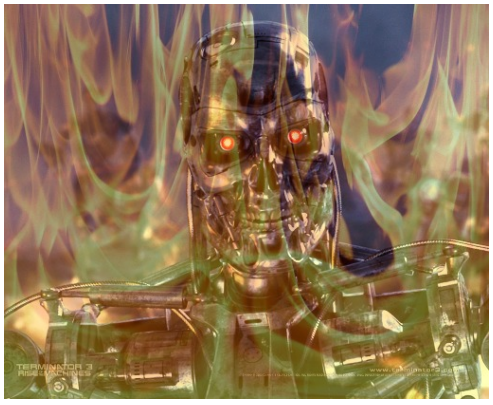


Imagen base 1

Imagen base 2



Resultado obtenido al aplicarle el filtro Merge (porcentaje de mezcla 0.42)

4.1.1. Merge 1

La primera implementación del filtro Merge se enfoca en entender como generar varios pixel de la imagen resultante y construir este proceso de la manera más sencilla posible. En ambas implementaciones, el algoritmo va a procesar de a 4 pixel a la vez.

Procedimiento general Lo primero que hace el algoritmo es empaquetar el *valor* Float SP pasado por parametro (porcentaje de mezcla de las imagenes) en el registro `XMM0` utilizando las instrucciones `MOVDQU`, `PSLLDQ` y `ADDPS` (esta última se podría utilizar de la misma forma por la instrucción `POR`).

Luego, preparamos en el registro `XMM15` 4 Floats SP empaquetados con *1-valor*. Para esto, cargamos en `XMM15` una mascara con 4 unos en formato Float SP, y le restamos el registro `XMM0` utilizando la instrucción `SUBPS`.

Además, calculamos en `RAX` la cantidad de bytes a procesar ($width * height * 4$) que nos va a servir para determinar cuando llegamos al final de la imagen.

Iteración El algoritmo del filtro Merge va a procesar de a 4 pixel a la vez en cada iteración. Para esto, cargamos 4 pixel de la imagen 1 en `XMM1` y 4 pixel de la imagen 2 en `XMM2` utilizando la instrucción `MOVDQU`.

Luego, convertimos los valores a Enteros de 16 bits (1 Word) utilizando las instrucciones `PUNPCKLBW` y `PUNPCKHBW`. La disposicion de los valores queda (todos en formato Enteros 16 bits):

- En XMM1 los pixel 1 y 2 de imagen 1.
- En XMM3 los pixel 3 y 4 de la imagen 1.
- En XMM2 los pixel 1 y 2 de imagen 2.
- En XMM4 los pixel 3 y 4 de la imagen 2.

Ahora, debemos realizar el cálculo de los 4 pixeles nuevos. Vamos a mostrar como sería para el primer pixel, ya que los otros se construyen de forma análoga:

1. Copio XMM1 a XMM5 y XMM2 a XMM6 utilizando la instrucción MOVDQU.
2. Convierto la parte baja de XMM5 y XMM6 a Enteros de 32 bit en los mismos registros (para el segundo y cuarto pixel tendriamos que convertir la parte alta de su respectivo registro).
3. Convierto XMM5 y XMM6 a formato Floats SP en los mismos registros, utilizando la instrucción CVTDQ2PS.
4. Multiplico XMM5 por XMM0 y dejo el resultado en XMM5, utilizando la instrucción MULPS. Esto significa multiplicar cada canal del primer pixel de la imagen 1 por el $valor$ Float SP pasado por parametro.
5. Multiplico XMM6 por XMM15 y dejo el resultado en XMM6, utilizando la instrucción MULPS. Esto significa multiplicar cada canal del primer pixel de la imagen 2 por $1 - valor$.
6. Luego, sumamos los canales del pixel en XMM6 a los del pixel en XMM5, con la instrucción ADDPS.

Al final, tenemos en XMM5, XMM7, XMM9 y XMM11 los 4 pixel resultante en formato Float SP.

Lo siguiente que hacemos es convertirlos a formato Enteros 8 bits, utilizando las instrucciones: CVTPS2DQ, PACKUSDW y PACKUSWB (en ese orden).

Por último, se acomoda los diferentes resultados para los pixel en el registro XMM5 (utilizando instrucciones PSLLDQ y POR) y se baja a memoria utilizando la instrucción MOVDQU, ya que estamos escribiendo 4 pixeles a la vez.

4.1.2. Merge 2

La segunda implementación del filtro Merge varía de la primera al reemplazar la aritmética que antes se realizaba con valores en formato Float SP por aritméta de enteros. Como el valor dado por parámetro para el porcentaje de mezcla es un Float, necesitamos idear un método para perder la menor precisión posible a la hora de hacer las cuentas:

Si tenemos un canal $R1$ y otro $R2$ entonces queremos calcular $R1 * v + R2 * (1 - v)$, pero podemos reescribir esa expresión como: $(R1 * v + R2 * (1 - v)) * (2^k/2^k) = (R1 * v * 2^k + R2 * (1 - v) * 2^k)/2^k$. Lo interesante es que calculando $v * 2^k$ y $(1 - v) * 2^k$ con un k suficientemente grande se obtiene un rango aceptable para la definición en enteros, reduciendo el error cometido. En nuestro caso consideramos que $k = 8$ es un valor ideal para generar buenos resultados, ya que el rango de valores que podemos expresar es de 0 a $2^8 - 1$ que es exactamente el rango que tiene cada canal en la representación RGB/RGBA.

Procedimiento general Analogamente a la implementación del Merge 1, debemos preparar dos registros especial que contengan los valores necesariso empaquetados para depués realizar las cuenta de a 4 pixel a la vez.

En primer lugar, calculamos 2^k cargando una máscara que tiene un uno en formato Entero 32 bits y lo shifteamos a izquierda k bits ($k = 8$). Luego, calculamos $v * 2^k$. Para esto, convertimos el 2^k a formato Float SP con la instrucción CVTSI2SS y lo multiplicamos por el $valor$ con la instrucción MULSS. A continuación, convertimos este valor obtenido a formato Entero de 32 bits con la instrucción CVTSS2SI y lo dejamos empaquetado en el registro XMM4 con MOVDQU, PSLLDQ y PADDD. Identicamente, dejamos empaquetado en XMM3 el 2^k en formato Entero de 32 bits con las instrucciones MOVDQU, PSLLDQ y PADDD.

Ahora, preparamos en XMM15 $(1 - v) * 2^k$ empaquetado en formato Entero de 32 bits con las instrucciones haciendo la resta empaquetada entre XMM3 y XMM4 con la instruccion PSUBD.

Antes de comenzar el ciclo, copiamos a XMM14 el empaquetado de $v*2^k$ que estaba en XMM4 y cargamos en XMM13 un 8 (el k elegido) en formato Entero 32 bits para luego usarlo para shiftear los resultados.

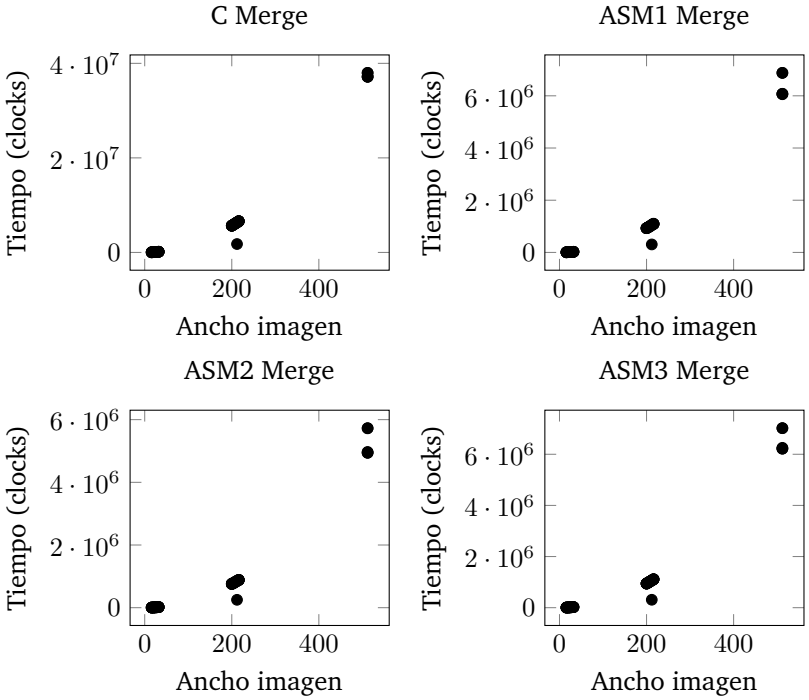
Iteración La iteración del Merge 2 es identica hasta la parte del calculo del pixel, en la cual hacemos calculamos 4 pixel de la siguiente forma (mostramos solo el primero ya que los otros se construyen de forma análoga):

1. Copio `XMM1` a `XMM5` y `XMM2` a `XMM6` utilizando la instrucción `MOVDQU`.
2. Convierto la parte baja de `XMM5` y `XMM6` a Enteros de 32 bit en los mismos registros (para el segundo y cuarto pixel tendriamos que convertir la parte alta de su respectivo registro).
3. Multiplico `XMM5` por `XMM14` y dejo el resultado en `XMM5`, utilizando la instrucción `PMULLD`. Esto significa multiplicar cada canal del primer pixel de la imagen 1 por el $v * 2^k$ y guardar la parte baja de esa operación en el *DST*. No nos molesta solo guardar la parte baja ya que sabemos que original mente teníamos canales de 8 bits y el $v * 2^k$ también es de 8 bits, así que a lo sumo nos puede llegar a quedar algo de 16 bits (a lo sumo se duplica la longitud).
4. Multiplico `XMM6` por `XMM15` y dejo el resultado en `XMM6`, utilizando la instrucción `PMULLD`. Esto significa multiplicar cada canal del primer pixel de la imagen 2 por el $(1 - v) * 2^k$ y guardar la parte baja de esa operación en el *DST*.
5. Sumamos los canales del pixel en `XMM6` a los del pixel en `XMM5`, con la instrucción `PADDD`.
6. Shifteamos a derecha cada *DWORD* en `XMM5` k bits (almacenado en `XMM13`) con la instrucción `PSRLD`. Esto es equivalente a dividir cada Entero de 32 bits en `XMM5` por 2^k.

A partir de acá realizamos exactamente lo mismo que realizabamos para el Merge 1.

4.2. Análisis y Resultados

Rendimiento (varias imagenes por tamano): En este caso se compara el rendimiento de un mismo algoritmo con varias imagenes en distintos tamaños



Comparacion entre versiones: Los siguientes graficos muestran para una misma imagen, los rendimientos obtenidos corriendo las diferentes versiones del algoritmo

Imagen 1 (toda roja)

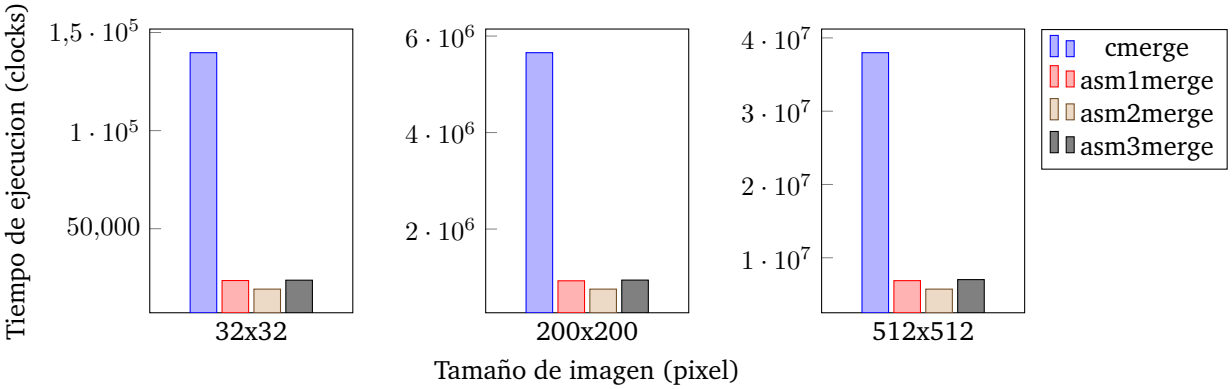


Imagen 2 (terminator)

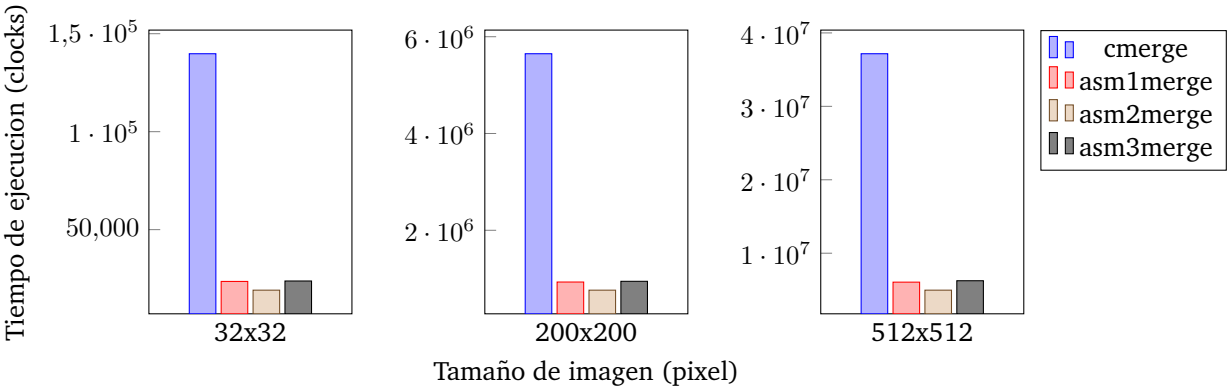
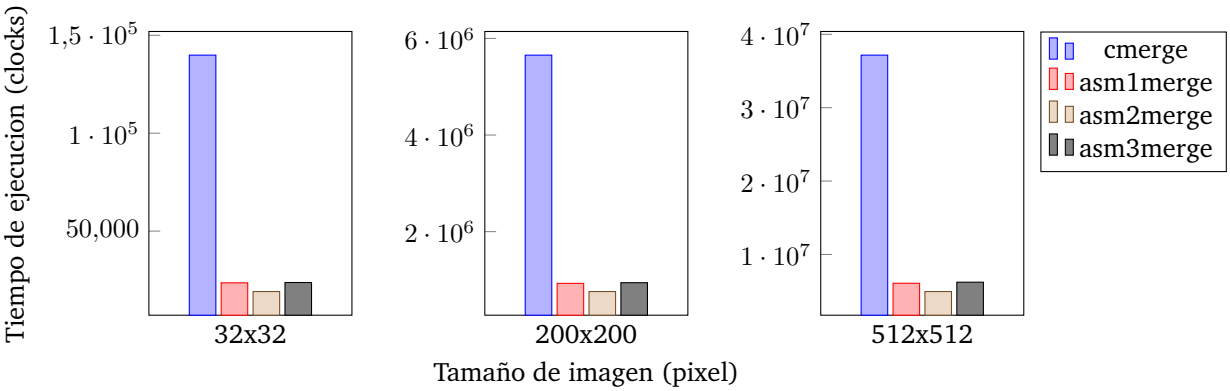


Imagen 3 (llamas)



Comparacion imagenes en tamaño 512x512				
Imagen	Tiempo cmerge (clk)	Tiempo asm1merge (clk)	Tiempo asm2merge (clk)	Tiempo asm3merge (clk)
1	37983067	6878114	5722524	7019372
2	37164583	6059792	4969306	6252390
3	37160997	6073302	4942517	6219515

La implementación más lenta es la de C, siendo esta aproximadamente 11 veces más lenta que ASM1. A su vez ASM1 es un 20 % más lenta que ASM2. Esto es consistente tanto para imagenes pequeñas como grandes.

Ambas implementaciones (tanto C como ASM1) son muy simples y si miramos solamente las operaciones aritméticas de suma y producto, deberiamos ver una diferencia de 4x en favor de ASM teniendo

en cuenta que con instrucciones de SIMD se pueden realizar sumas y productos de a 1 pixel a la vez. En cambio en C se puede realizar una suma o producto de una sola componente RGBA a la vez. Tanto C como ASM1 deben convertir los valores de uint8 a float, con lo cual la diferencia de tiempo entre el 4x esperado inicialmente y el 11x registrado lo atribuimos a la disminución en accesos a memoria. Para acceder a 4 pixel en ASM1 alcanza con almacenar en 1 registro xmm, en cambio en C se realiza un acceso por cada componente de cada pixel. Esto mejora sustancialmente los tiempos en ASM1.

En ASM2 notamos una mejora consistente de un 20 % respecto a ASM1. Esta mejora interesante está dada por el manejo de los datos en enteros en vez del uso de aritmética de punto flotante. Como se comenta en la sección de desarrollo, la idea principal es realizar los cálculos en enteros, llevando los números a órdenes que no alteren demasiado los resultados por el uso de enteros en vez de floats y luego dividir por el mismo factor para devolver los números al órden inicial (2^k).

Este 20 % de mejora demuestra el costo de convertir valores enteros a punto flotante y viceversa. En compensación, el código de ASM1 es más simple de entender y preciso, ya que no posee el error variable de usar 2^k como factor de multiplicidad.

Como implementación alternativa pensamos como modificar las cuentas para realizar una menor cantidad de operaciones de punto flotante, en eso se basa ASM3. Este filtro es 24 % más lento que ASM2, justificado simplemente por una mayor cantidad de operaciones aritméticas realizadas. ASM3 resultó más lento que ASM1 y no posee mejoras a la hora de leer el código, con lo cual sería descartado para una implementación futura del filtro Merge.

5. Ejercicio 3 - HSL

5.1. Desarrollo

La idea detrás de este filtro es llevar la representación de los pixel de RGB a HSL. Las componentes HSL (Matiz, Saturación y Luminosidad) tienen más que ver con los aspectos a retocar de una imagen a ojos de un humano que los componentes de Rojo, Azul y Verde. Una vez transportados los pixel a esta representación se manipulan los valores de HSL sumandole los valores pasados por patrametro: HH, SS y LL. Los últimos dos se manipulan con saturación, mientras que el primero no (si $H + HH > 360$ entonces se el resta 360). Luego, se retorna a la representación RGB para depositar los pixel en la imagen resultante.

Imagen original

Resultado obtenido al aplicarle el filtro HSL

5.1.1. HSL 1

En la primera implementación del HSL se utiliza las funciones auxiliares en C *rgbTOhsl* y *hslTOrgb* dadas por la catedra, y su mayor dificultad consiste en realizar los calculos para sumar los parametros HH, SS y LL a los pixeles convertidos a HSL.

Procedimiento general Antes de comenzar el ciclo de iteraciones se prepara un vector *hsl_params_dato* de 128 bits (almacenado en la sección *.data*) conteniendo en 4 números en formato Float SP la información de HSL utilizando las instrucciones `MOVDQU`, `PSLLDQ` y `POR`. El resultado obtenido queda (empezando por la parte más baja):

- 0.0
- H
- S
- L

Luego, calculamos en `R15` la cantidad de bytes a procesar ($width * height * 4$) que nos va a servir para determinar cuando llegamos al final de la imagen.

Iteración Por cada pixel se llama, al comienzo de la iteración, a la función de conversión de RGB a HSL (`CALL` *rgbTOhsl*). Los parametros que se le pasan son: en `RDI` un puntero a la dirección del pixel actual y en `RSI` un puntero a un vector *hsl_temp_dato* de 128 bits (almacenado en la sección *.data*). Esto produce un vector con números de punto flotantes de 32 bits.

Luego se suman en formato Float SP los valores del pixel convertido a HSL con los valores pasados por parámetro. Para esto, se copia la información en *hsl_params_dato* y *hsl_temp_dato* a los registros XMM1 y XMM0 respectivamente, para luego realizar la suma con la instrucción ADDPS.

A continuación, se copia el resultado obtenido en XMM0 a un vector *hsl_suma_dato* de 128 bits (almacenado en la sección *.data*). Este vector nos va a ser útil para ir guardando los resultados definitivos. Además, cargamos en XMM3 una mascara que tiene un 360 en formato Float SP en la segunda DWORD (contando desde la parte baja) y lo demás todo ceros.

Ahora pasamos a hacer los chequeos para que los resultados de la suma sean correctos. Necesitamos que los resultados de las componentes L y S saturen entre 0 y 1, mientras que la componente H puede pasarse, pero siempre entre 0 y 360 (menor estricto que 360). Para esto, tenemos dos partes de código por separado: una que verifica los limites superiores y otra que verifica los limites inferiores.

Para verificar los **limites superiores**:

- Copiamos en XMM1 una mascara en formato Float SP que tiene los valores máximos permitidos para cada canal: 360, 1 y 1.
- Con la instrucción CMPLTPS hacemos la comparación por menor estricto entre los valores en XMM0 y los valores en XMM1.
- Este resultado lo copiamos en el vector *hsl_temp_dato* con la instrucción MOVDQU.
- Ahora, copiamos en EDI el resultado de $H < 360$, en ESI el resultado de $S < 1$ y en EDX el resultado de $L < 1$ con la instrucción MOVD.
- Comparamos EDI con 0 (FALSE) (con la instrucción CMP). Si no es igual, pasamos a evaluar ESI (el salto se realiza con la instrucción JNE). Si es igual, utilizamos la instrucción SUBPS para dejar en XMM0 los valores de antes pero con el Hue restado en 360 (la resta se hizo contra el registro XMM3) y se pisa el vector *hsl_suma_dato* con el contenido en XMM0 con la instrucción MOVDQU.
- Comparamos ESI con 0 (FALSE) (con la instrucción CMP). Si no es igual, pasamos a evaluar EDX (el salto se realiza con la instrucción JNE). Si es igual, simplemente movemos el máximo para S desde la máscara de valores máximos a su posición correspondiente en el vector *hsl_suma_dato* con la instrucción MOVD.
- Realizamos lo mismo para EDX que para ESI, excepto que en el caso de que no sea igual a FALSE, pasamos a chequear los limites inferiores.

Para verificar los **limites inferiores** se realiza de forma análoga a la anterior:

- Copiamos en XMM1 una mascara en formato Float SP que tiene los valores mínimos permitidos para cada canal: 0, 0 y 0.
- Con la instrucción CMPNLTPS hacemos la comparación por menor estricto entre los valores en XMM0 y los valores en XMM1.
- Este resultado lo copiamos en el vector *hsl_temp_dato* con la instrucción MOVDQU.
- Ahora, copiamos en EDI el resultado de $H \geq 0$, en ESI el resultado de $S \geq 0$ y en EDX el resultado de $L \geq 0$ con la instrucción MOVD.
- Comparamos EDI con 0 (FALSE) (con la instrucción CMP). Si no es igual, pasamos a evaluar ESI (el salto se realiza con la instrucción JNE). Si es igual, utilizamos la instrucción ADDPS para dejar en XMM0 los valores de antes pero con el Hue sumado en 360 (la suma se hizo contra el registro XMM3) y se mueve el nuevo Hue a su posición correspondiente en el vector *hsl_suma_dato* con la instrucción MOVD.
- Comparamos ESI con 0 (FALSE) (con la instrucción CMP). Si no es igual, pasamos a evaluar EDX (el salto se realiza con la instrucción JNE). Si es igual, simplemente movemos el mínimo para S desde la máscara de valores mínimos a su posición correspondiente en el vector *hsl_suma_dato* con la instrucción MOVD.

- Realizamos lo mismo para EDX que para ESI, excepto que en el caso de que no sea igual a FALSE, pasamos a la parte final del ciclo.

Para terminar, llamamos a la función de conversión de HSL a RGB (CALL *hslTOrgb*). Los parametros que se le pasan son: en RDI un puntero al vector vector *hsl_suma_dato* y en RSI un puntero a un vector *rgb_result* de 32 bits (almacenado en la sección *.data*). Esto produce un vector con números en Entero de 8 bits, que luego se copia a su posición correspondiente en la imagen utilizando la instrucción MOVD.

5.1.2. HSL 2

En la segunda implementación del HSL se mantiene lo implementado para HSL 1 y se agregan los algoritmos *rgbTOhsl* y *hslTOrgb* en lenguaje ensamblador.

Procedimiento general e Iteración El procedimiento general e Iteración se mantiene respecto a HSL 1 ya que la diferencia está en la implementación de las funciones de conversión de RGB a HSL y viceversa.

RGB a HSL La función $rgbTOhsl$ recibe en RDI un puntero a un pixel en forma de Enteros sin signo de 8 bits (pixel de 32 bits) y en RSI un puntero a un pixel en forma de Floats SP (pixel de 128 bits). Lo primero que hacemos es cargar los valores apuntados por RDI en XMM0 con la instrucción MOVD, para luego convertir la representación de Enteros de 8 bits a Enteros de 32 bits con las instrucciones PUNPCKLBW y PUNPCKLWD en el mismo registro XMM0.

Una vez que tenemos esto, calculamos el máximo y mínimo de los componentes RGB utilizando las instrucciones PMAXUD, PMINUD y PSLLDQ lo que nos deja en los 32 bits más altos de XMM1 $cmax = max(R, G, B)$ y en los 32 bits más altos de XMM2 $cmin = min(R, G, B)$. Utilizando la instrucción PSUBD dejamos en los 32 bits más altos de XMM3 $d = max(R, G, B) - min(R, G, B)$. Lo movemos a los 32 bits más bajos de XMM3 con la instrucción PSRLDQ y lo copiamos en ECX usando MOVD.

Antes de empezar con los calculos de cada componente de HSL, dejamos en los 32 bits más bajos de XMM4 $cmax + cmin$ y en los 32 bits más bajos de XMM1 $cmax$ utilizando las instrucciones PADDD y PSRLDQ. Además, limpiamos el registro XMM14 con la instrucción PXOR para ir dejando los resultados de cada canal.

Calculo de L

- Convetimos el valor en XMM4 ($cmax + cmin$) a formato Float SP con la instrucción CVTDQ2PS.
- Cargamos en el registro XMM13 una mascara que en los 32 bits más bajos un 510 en formato Float SP y lo demás todos unos en formato Float SP.
- Dividimos XMM4 por XMM13 con la instrucción DIVPS.
- Copiamos el valor obtenido a los 32 bits más altos de XMM14 con las instrucciones MOVDQU y PSLLDQ.

Calculo de S

- Comparamos ECX (d) con 0 (instrucción CMP). Si es igual, no hacemos nada y pasamos a calcular H (salto con instrucción JE). Caso contrario, seguimos.
- Multiplicamos el valor obtenido para L por 2 y le restamos 1. Esto lo hacemos cargando en XMM5 una mascara que tiene en los 32 bits más bajos un 1 en formato Float SP y en el resto ceros, luego se lo suma contra si mismo con ADDPS para obtener el 2. Con la instrucción MULPS multiplicamos el 2 por L y lo dejamos en XMM5. Luego, volvemos a cargar la máscara con el 1 en los 32 bits más bajos y se la restamos a XMM5 usando SUBPS. Este resultado parcial queda en XMM4.
- Lo siguiente es aplicarle la función $fabs$ a el valor en XMM4. Para esto, cargamos una máscara que tiene todos unos excepto en el primer bit de cada DWORD ($0x7fffffff$), y se la aplicamos con la instrucción PAND a XMM4, con lo que efectivamente estamos seteando el bit de signo de cada Float en XMM4 a cero, convirtiendolos a positivos en caso de que no lo fueran.

- Le restamos a 1 el resultado de $fabs$ de la misma forma que antes le restabamos 1 a $2 * L$. El resultado parcial queda en XMM5.
- A continuación, dividimos d por el valor en XMM5. Primero, cargamos una máscara que tiene todos unos excepto en los 32 bits más bajos y le hacemo un POR con XMM5. Segundo, convertimos el valor d que estaba en XMM3 en Entero 32 bits a XMM4 en formato Float SP con la instrucción CVTDQ2PS. Y luego dividimos XMM4 por XMM5 con la instrucción DIVPS.
- Por último, cargamos en XMM5 una máscara que tiene todos unos excepto en los 32 bits más bajos, que tiene el valor $255{,}001f$ (todos en formato Float SP), y dividimos XMM4 por XMM5 con la instrucción DIVPS, lo que nos deja efectivamente en XMM4 el valor $d/(1 - fabs(2 * L - 1))/255{,}0001$.
- Copiamos el valor obtenido en los 32 bits siguientes a L en XMM14 con las instrucciones PSLLDQ y ADDPS.

Calculo de H Para este calculo de H nos dimos cuenta que las cuentas necesarias se pueden llevar a cabo en paralelo y simplemente elegir el resultado final una vez realizadas las cuentas, comparando el valor de $cmax$ con los distintos canales RGB.

- Comparamos ECX (d) con 0 (instrucción CMP). Si es igual, no hacemos nada y pasamos a calcular H (salto con instrucción JE). Caso contrario, seguimos.
- Con las instrucciones MOVDQU, PADDD, PSRLDQ y PSLLDQ preparamos en XMM5 los valores: [G,B,R,-]. Esto se lo restamos a los datos iniciales ([R,G,B,A]) en XMM0 con la instrucción PSUBD y nos queda en XMM4: [R-G,G-B,B-R,-].
- Luego, con las instrucciones CVTDQ2PS, MOVDQU, PADDD y PSLLDQ preparamos en XMM3 un empaquetado con formato Float SP con los valores: [d,d,d,d] y esto se lo dividimos alo que habiamos obtenido en XMM4 con la instrucción DIVPS.
- Cargamos en XMM6 una máscara con los valores [4,6,2,0] en formato Floats SP y se lo sumamos a XMM4 con la instrucción ADDPS.
- Cargamos en XMM6 una máscara con los valores [60,60,60,60] en formato Floats SP y se lo multiplicamos a XMM4 con la instrucción MULPS.
- Ahora, preparo en EDX, EAX, R8D y R9D los valores de $cmax$ y los canales B, G y R respectivamente, para hacer las comparaciones.
- Comparo EDX con R9D. Si es igual, movemos el valor $60 * ((G - B)/d + 6)$ en XMM4 a EDX con las instrucciones PSRLDQ y MOVD y saltamos al final. Sino, pasamos al siguiente caso.
- Comparo EDX con R8D. Si es igual, movemos el valor $60 * ((B - R)/d + 2)$ en XMM4 a EDX con las instrucciones PSRLDQ y MOVD y saltamos al final. Sino, pasamos al siguiente caso.
- Acá no hace falta comparar, ya que $cmax$ no era igual a R o G, entonces es igual a B. Movemos el valor $60 * ((R - G)/d + 4)$ en XMM4 a EDX con las instrucciones PSRLDQ y MOVD y saltamos al final. Sino, pasamos al siguiente caso.
- Ahora, tenemos que chequear que el valor en EDX sea menor igual a 360. Esto lo podemos hacer comparando EDX con 360 y con la instrucción JL. En caso de que sea mayor, cargamos una máscara con un 360 en los 32 bits más bajos y hacemo la resta en los registros XMM con la instrucción SUBPS
- Al final, copiamos el valor obtenido en los 32 bits siguientes a S en XMM14 con las instrucciones PSLLDQ y ADDPS.

Por último, copiamos el Alpha original (en formato Float SP) en los 32 bits más bajos de XMM14 con las instrucciones PSLLDQ, PSRLDQ, CVTDQ2PS y POR. El registro XMM14 se escribe en la dirección apuntada por RSI con la instrucción MOVDQU.

HSL a RGB El pasaje de HSL a RGB es muy similar al de RGB a HSL respecto al tipo de operaciones aritméticas y de conversión utilizadas, con la salvedad de que se realizan en otro orden.

Como primer paso, limpiamos el registro `XMM14` con la instrucción `PXOR` para ir almacenando los resultados parciales. Luego, cargamos el pixel en formato HSL que apunta `RDI` y movemos cada canal a la parte más baja de un registro distinto: L a `XMM1`, S a `XMM2` y H a `XMM3` (se usan las instrucciones `PSLLDQ`, `PSRLDQ` y `MOVDQU`).

Calculo de c El calculo de c es identico al cálculo de S en la función $rgbTOhsl$, excepto que una vez obtenido $1 - fabs(2 * L - 1)$ lo multiplicamos por S utilizando la instrucción `MULPS`, dejando el resultado en los 32 bits más bajos de `XMM14`.

Calculo de x

- Primero, cargamos en `XMM6` una máscara en formato Float SP con los valores: [60,60,60,60] y la utilizamos para dividir `XMM4` (H) con la instrucción `DIVPS`, dejando el resultado en `XMM4`.
- Ahora tenemos que calcular la función $fmod(H/60, 2)$. Para esto, primero hacemos cargamos en los 32 bits más bajos de `XMM5` un 2 en formato Float SP y se lo dividimos a `XMM4`, dejando el resultado en este último. Luego, copiamos `XMM4` a `XMM6` con la instrucción `MOVDQU` y convertimos `XMM6` a Entero 32 bits con la instrucción `CVTTPS2DQ`, que realiza una conversión truncada, quedandonos en `XMM6` la parte entera de $(H/60)/2$. Volvemos a convertir `XMM6` a Float SP, y se lo restamos a `XMM4` con las instrucciones `CVTPS2DQ` y `SUBPS`. Una vez tenemos esto, solo falta multiplicar lo obtenido por 2 con la instrucción `MULPS`. El resultado parcial queda en `XMM4`.
- A continuación se realizan las restas de unos y el calculo de $fabs$ de forma identica a como se hizo en el cálculo de S en la función $rgbTOhsl$.
- Por último, multiplicamos el resultado parcial por c con la instrucción `MULPS` y dejamos el resultado en los 32 bits más bajos de `XMM13`.

Calculo de m El cálculo de m es bastante sencillo comparado al recíen hecho. Utilizamos la instrucción `DIVPS` para dividir por 2 el valor de c y luego se lo restamos a L con la instrucción `SUBPS`. El resultado lo dejamos en los 32 bits más bajos de `XMM12`.

Elección de los canales RGB Ahora que tenemos los valores de c, x y m, solo nos queda asignarselos a cada canal R, G y B, dependiendo del valor de H. Para esto, vamos a tener una serie de condicionales. Como H es un valor en formato Float SP, tenemos que hacer la comparación usando los registros `XMM`. Es por eso que tenemos una serie de máscaras en formato Float SP que tienen los valores: [0,0,0,60], [0,0,0,120], ..., [0,0,0,300]. Dependiendo de en que caso cayamos, vamos a ir guardando en `XMM9`, `XMM10` y `XMM11` los valores de R, G y B respectivamente.

Empezando por la primera de las máscaras, se carga en el registro `XMM6` con la instrucción `MOVDQU` y se compara con el valor de H en `XMM3` con la instrucción `CMPLEPS`. Nos queda entonces en los 32 bits más bajos de `XMM6` el valor de $60 \leq H$. Luego, movemos este valor al registro `EAX` y lo comparamos con cero (FALSE).

Si `EAX` no es igual a cero (instrucción de salto `JNE`), quiere decir que $60 <= H! = FALSE \iff 60 <= H == TRUE$ y pasamos a evaluar la siguiente máscara (120). Caso contrario, copiamos `XMM9` a `XMM14` ($R = c$), `XMM10` a `XMM13` ($G = x$) y seteamos en cero `EAX` ($B = 0$) (con la instrucción `PXOR`) y saltamos a la última parte del algoritmo.

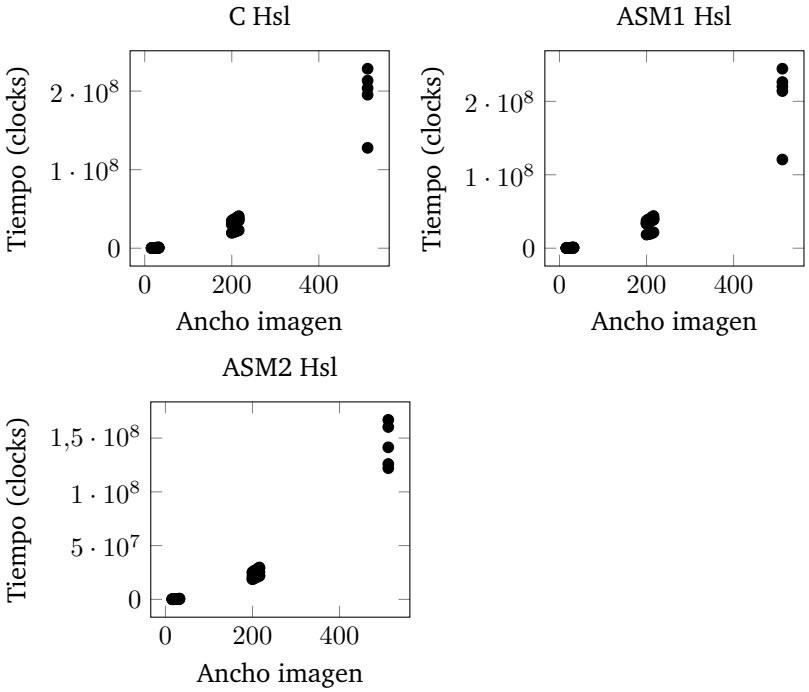
El resto de los casos condicionales se desarrolla de manera identica a la descrita para el primer caso. En el caso que $300 <= H$ no nos hace falta evaluar $360 <= H$ y asignamos directamente los valores correspondientes.

A continuación, le sumamos a `XMM9`, `XMM10` y `XMM11` el valor de m en `XMM12` con la instrucción `ADDPS`. Luego, acomodamos los valores de R, G y B en el registro `XMM9` de forma: [R,G,B,0] y lo multiplicamos por una máscara cargada en `XMM6` con [255,255,255,255] en formato Float SP.

Convertimos `XMM9` y el Alpha original a Enteros 32 bits con la instrucción `CVTPS2DQ` y los juntamos en el registro `XMM0` con las instrucciones `PSLLDQ`, `PSRLDQ` y `PADDD`. Por último, convertimos los valores en `XMM9` a enteros de 8 bits con las instrucciones `PACKUSDW` y `PACKUSWB`, y escribimos el resultado en la dirección apuntada por `RSI` usando `MOVD`.

5.2. Análisis y Resultados

Rendimiento (varias imagenes por tamaño): En este caso se compara el rendimiento de un mismo algoritmo con varias imagenes en distintos tamaños



Comparacion entre versiones: Los siguientes graficos muestran para una misma imagen, los rendimientos obtenidos corriendo las diferentes versiones del algoritmo

Imagen 1 (toda roja)

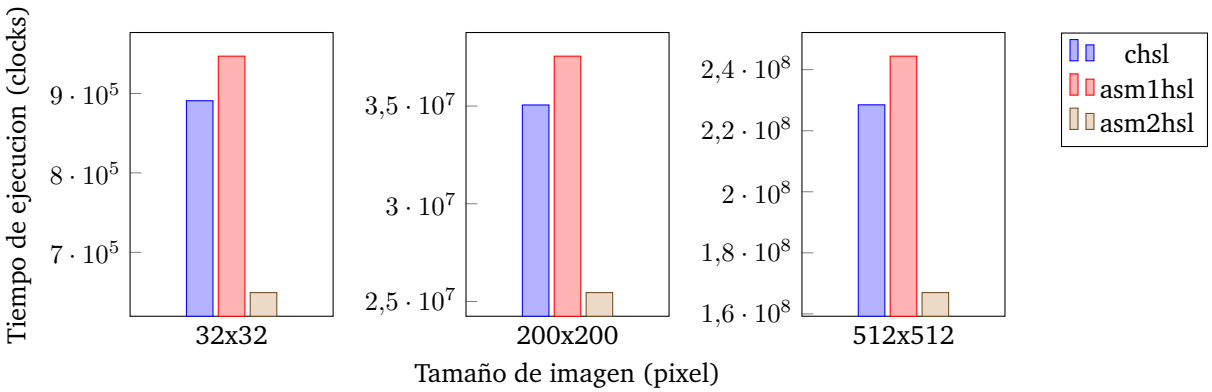


Imagen 2 (terminator)

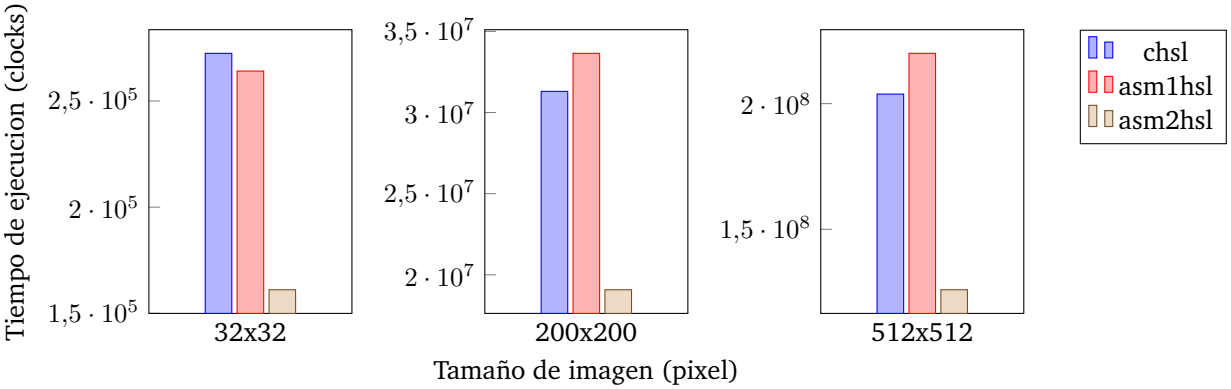
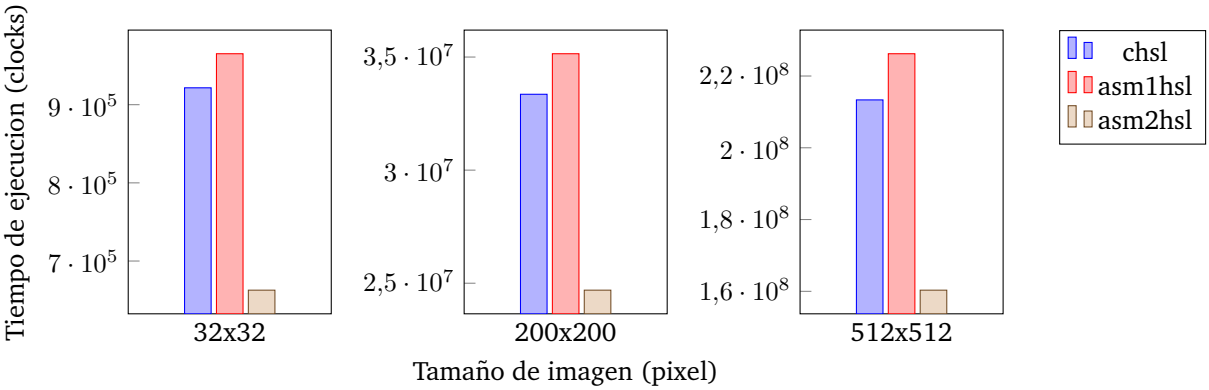


Imagen 3 (llamas)



Comparacion imagenes en tamaño 512x512					
Imagen	Tiempo chsl (clk)	Tiempo asm1hsl (clk)	Tiempo asm2hsl (clk)	Diferencia asm1 vs c	Diferencia asm2 vs c
1	228485618	244421897	166971553	7 % peor	36 % mejor
2	203807587	220020342	125928252	8 % peor	61 % mejor
3	213343065	226211244	160321381	6 % peor	33 % mejor

Como se puede apreciar en la tabla, el rendimiento de C es mejor que el de HSL1. Esta baja en la performance era predecible y la justificación está dada por el esfuerzo extra, medido en cantidad de operaciones necesarias, para operar sobre componentes particulares de una imagen y porque la función tiene que llamar a las rutinas externas de conversión entre RGB y HSL escritas en C.

Las instrucciones de SIMD son ideales para trabajar sobre operaciones iguales sobre datos contiguos pero no para hacer operaciones distintas que tiene más sentido realizar directamente sobre escalares en registros regulares de 64bits.

La mejoría de ASM2 respecto a C la vemos relacionada con el predictor de saltos y accesos a memoria. En $rgbTOhsl$ para el caso de C el algoritmo pasa por un conjunto de guardas hasta que entra en una rama de if, accede a memoria y realiza las cuentas del cálculo de H. En cambio, la versión de ASM2 realiza las 4 cuentas posibles para el cálculo de H y luego decide cual usar para el resultado, mejorando el flujo del pipeline. La ventaja en este caso de SIMD es la posibilidad no de realizar cuentas sobre múltiples datos distintos al mismo tiempo, sino distintos posibles resultados al mismo tiempo y luego elegir el más conveniente.

Habiendo realizado la experimentación resulta interesante replantear la implementación de $hslTOrgb$, realizando los cálculos y luego tomando ramas. Según nuestra hipótesis debería ser más rápida que la implementación actual.

Las llamadas a las rutinas de $rgbTOhsl$ y $hslTOrgb$ se hacen mediante call, este overhead es muy pequeño ya que no se arma todo el stackframe en cada llamado sino que simplemente se apila y luego desapila la dirección de retorno y rbp. La carga por llamar a las funciones es muy pequeña respecto a la

cantidad de operaciones realizadas dentro de las rutinas, con lo cual no es un factor determinante a la hora de medir tiempos.

6. Conclusión

Partiendo desde las implementaciones en lenguaje C de los algoritmos, realizamos distintas versiones en lenguaje Assembler x86. Si bien se pierde la expresividad que facilita la implementación, se ven mejoras en prácticamente todos los casos, debido a las ventajas de operar en bajo nivel. Particularmente, pudimos ver mejoras significativas de rendimiento mediante el uso de instrucciones SIMD. Esto se debe a dos factores clave: el acceso a memoria de a varios bloques, que minimiza la cantidad de accesos (operación cara) y la paralelización de los procesos. La ganancia de performance mas significativa entre versiones de Assembler la tuvimos cuando pudimos acceder a mayor cantidad de pixeles por ciclo de procesamiento. Ademas, la paralelización de operaciones como sumar, dividir, etc.. que en vez de realizarlas para cada dato por separado, pudimos realizar para varios datos a la vez, nos llevo a una mejora de performance significativa aun cuando no pudimos acceder a varias pixeles en memoria a la misma vez, por ejemplo, para operar sobre las diferentes componentes de un pixel.

A. Anexo - metodología para las mediciones

Script utilizado para inicializar el entorno de mediciones:

```
#!/bin/bash
# From SO: http://stackoverflow.com/questions/9072060/one-core-
    exclusively-for-my-process

mkdir /cpuset
mount -t cpuset none /cpuset/
cd /cpuset

mkdir sys # create sub-cpuset for system processes
/bin/echo 0-2 > sys/cpus # assign cpus (cores) 0-2 to this set
/bin/echo 1 > sys/cpu_exclusive
/bin/echo 0 > sys/mems

mkdir rt # create sub-cpuset for my process
/bin/echo 3 > rt/cpus # assign cpu (core) 3 to this cpuset
/bin/echo 1 > rt/cpu_exclusive
/bin/echo 0 > rt/mems
/bin/echo 0 > rt/sched_load_balance
/bin/echo 1 > rt/mem_hardwall

# move all processes from the default cpuset to the sys-cpuset
for T in `cat tasks`; do echo "Moving " $T; /bin/echo $T > sys/tasks;
    done

# disable interruptions for the isolated cpu
find /proc/irq/ -name "smp_affinity" -exec sh -c "echo 7 > {}" \;

# set fixed frequency for the isolated cpu
cpufreq-selector -c 3 -f 800000

# start task in specific isolated cpu
$1 & /bin/echo $! > /cpuset/rt/tasks    # Move the just created task PID
     to the tasks of the exclusive CPU
```