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

TP2

Organización del Computador II

Primer Cuatrimestre de 2014

Grupo: **Colombia/Arepa**

Apellido y Nombre	LU	E-mail
Cisneros Rodrigo	920/10	rodricis@hotmail.com
Rodríguez, Agustín	120/10	agustinrodriguez90@hotmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Contents

1 Introducción

El lenguaje C es uno de los más eficientes en cuestión de performance, pero esto no quiere decir que sea óptimo para todos los casos o que no haya campos en los que pueda utilizarse una opción mejor. Para comprobar esto, experimentamos con el set de instrucciones SIMD de la arquitectura Intel. Vamos a procesar imágenes mediante la aplicación de ciertos filtros y estudiaremos la posible ventaja que puede tener un código en Assembler[1] con respecto a uno en C. Implementaremos los filtros en ambos lenguajes para luego poder comparar la performance de cada uno y evaluar las ventajas y/o desventajas de cada uno.

[1] Durante el desarrollo del informe puede verse referido a Assembler como ASM haciendo un abuso de notación siendo que este es solo el nombre de la extensión.

2 Desarrollo y Resultados

2.1 Filtro Tiles

2.1.1 Objetivo

El objetivo del filtro "tiles" es seleccionar una porción de la imagen fuente e ir copiándola en otra imagen destino, la cual tiene el mismo tamaño que la fuente.
Logrando así, que una parte de la imagen fuente se vea repetidamente. Formalizando lo anterior, el filtro debe respetar la siguiente fórmula:

$$dst(i,j) = src[(i.mod\ tamy) + offsety][(jmodtamx) + offsetx]$$

2.1.2 Desarrollo Implementacion C

Sintesis de Desarrollo

Recorremos la imagen "destino" mediante dos ciclos anidados, un ciclo para filas, y otro para columnas. Dentro de estas iteraciones, vamos copiando pixel por pixel desde fuente a destino. Para llevar la cuenta de lo que vamos recorriendo de la imagen fuente, tenemos 2 contadores, uno para fila y otro para columna. Cuando se recorrió la "porción" de imagen que se solicitó (por parámetros), se vuelven a reiniciar esos contadores.

Explicación detallada de Implementación

En esta implementación, lo que se realizo primero fue crear dos punteros de matriz uno correspondiente a src y otro a dst.
Luego, realizamos dos ciclos, uno añidado dentro del otro recorriendo de la forma $'(int\ i\_d = 0;\ i\_d < filas;\ i\_d++)'$ y $(int\ j\_d = 0;\ j\_d < cols;\ j\_d++)$.
Dentro del segundo ciclo, creamos dos punteros rgb_t donde uno apuntará a
rgb_t $*p\_d = (rgb\_t*)\ \&dst\_matrix[i\_d][j\_d * 3]$, y el otro a $rgb\_t * p\_s = (rgb\_t*)\ \&src\_matrix[f][c * 3]$
donde f es igual a offsety y c es igual a offsetx que nos los dan como parámaetro.
Copiamos el puntero p_s en p_d. Y aumentamos en 1 c.
Posteriormente, chequeamos si $c >= (tamx + offsetx)$, en caso verdadero, c pasa a valer offsetx.
Fuera de este segundo ciclo, y dentro del primero, a c le damos el valor offsetx.
Luego, incrementamos f en 1, y chequeamos si $f >= (tamy + offsety)$ en caso verdadero, f pasa a valer offsety.
Estos dos ciclos realizaran $i\_d = filas - 1$ de itereaciones y el añidado $j\_d = cols - 1$ iteraciones.
Con lo mencionado, obtenemos el filtro tiles en c correctamente, con los valores que nos pasan como parámetros.

2.1.3 Desarrollo Implementacion en ASM

Nuestro algoritmo para el filtro TILES en su versión ASM es simple, aunque requiere varios chequeos por casos borde.
Recibimos 10 parámetros: Matriz fuente, Matriz destino, número de filas, número de columnas, tamaño de filas, tamaño de columnas, tamaño de fila del cuadro a replicar, tamaño de columna del cuadro a replicar y sus respectivos offset x e y.

Lo primero a tener en cuenta es que el tamaño de la fila del cuadro a replicar en realidad hay que multiplicarla por tres, ya que los píxeles de la imagen son de tres bits cada uno.
Por éste motivo también es que, como cada píxel son tres bits, podremos procesar de a 5 por ciclo (15 bytes).

Ahora lo próximo a hacer, antes de empezar el ciclo principal, es ubicar el puntero a la matriz fuente en el principio del cuadro a replicar.
Para esto al registro RDI que contiene la matriz le sumamos tres veces el offsetx, para quedar ubicado en la columna indicada, y luego le multiplicamos el offsety para finalmente obtener la posición de la matriz que queríamos.

Ya estamos en condiciones de realizar el ciclo principal del algoritmo. Tendremos dos ciclos anidados. Uno va a ser para ciclar por fila de la matriz destino (ciclo_por_linea_dst), ya que ésta va a ser toda lo ancho de la matriz original, y luego dentro de éste otro ciclo para la fila del cuadro a replicar (ciclo_linea_cuadro).

Lo que hacemos es simple, tener un contador con el largo de el tamaño de filas de la matriz (EAX) que irá decreciendo cuando cicle ciclo_por_linea_dst. Luego entramos a ciclo_linea_cuadro donde tendremos otro contador
para ver cuando termina la fila del cuadro (r13d). Tenemos que preguntarnos dos cosas antes de levantar los píxeles: Primero si el contador de linea DST (EAX) es menor a 15, ya que si es menor lo tratamos como un caso especial, levantando 15 bytes (5 píxeles) pero adelantando el puntero sólo los lugares que nos faltan para terminar la linea, para luego pasar el puntero de DST a la próxima fila y además colocar el puntero de SRC a la próxima linea del cuadro.
Por último, saltamos a ciclo_por_lina_dst, ordenando y restando los valores correspondientes de los contadores.
Algo parecido nos vamos a preguntar pero con respecto a lo que queda por recorrer de la linea del cuadro a replicar,
si es menor a 15 lo tratamos como en el caso anterior, aunque en este caso tenemos que volver a colocar el
puntero al principio de la linea del cuadro, pues lo queremos replicar hasta que termine la linea de la matriz.
Después ciclar a ciclo_linea_cuadro. Cuando copiamos todo el cuadro volvemos a reiniciar el puntero al principio del mismo Si pasa estas preguntas, significa que no estamos en el final de alguna fila, levantamos 15 píxeles de [RDI] con un registro xmm y los copiamos a [RSI], adelantamos los punteros sumándoles 15 y le restamos a los contadores respectivos antes de volver a ciclar a ciclo_linea_cuadro.

Repetimos este proceso hasta que llegamos a la última fila de DST, en este caso tenemos otro caso especial.
El problema es que cuando queremos completar los últimos lugares de la última fila (cuando el contador EAX es menor a 15) no podemos realizar el movdqu xmm0, [RDI] ya que éste nos toma 16 lugares y esto nos estaría trayendo posiciones de memoria que no tenemos asignadas. Para solucionar este problema lo que hicimos es levantar de a 1 byte hasta terminar la fila.

2.1.4 Experimento 1 - análisis el código generado

Utilizar la herramienta `objdump` para verificar como el compilador de C deja ensamblado el código C. Como es el código generado, ¿cómo se manipulan las variables locales?¿le parece que ese código generado podría optimizarse?

- Usando la herramienta *objdump* desensamblamos el archivo de extensión .o de código del filtro de Color. Al observar este código, lo primero que notamos es que el compilador no usó instrucciónes de SIMD a pesar de que el procesador tenga esa característica. Esto se debe a que al escribir en lenguaje C no se puede hacer uso de estas instrucciones a menos que usemos una librería aparte que haga uso de estas como puede ser *libSIMDx86*.
  En cuanto a como se manipulan las variables locales, se respetan las convenciones de pushear registros como r15 - r12 y rbx. Pero en casi todo el código se hace uso de las variables por parámetros y se usa mucho la pila moviendo el registro rbp para recorrerla.

- Existen algunas optimizaciones que se pueden realizar a la hora de compilar el código en C. Estas son -O1, -O2 o -O3, las cuales, siendo agregadas como flags a la hora de compilar, mi código

ensamblado queda mucho más óptimo.

2.1.5 Experimento 2 - optimizaciones del compilador

Compile el código de C con optimizaciones del compilador, por ejemplo, pasando el flag `-O1`[2]. ¿Qué optimizaciones realizó el compilador? ¿Qué otros flags de optimización brinda el compilador? ¿Para qué sirven?

En particular el flag -O1 hace que el tamaño del código ensamblado sea mucho menor que el código ensablado sin optimización. En particular al mirar el objdump de código con -O1 se pudo apreciar que el código era mucho menor en cantidad de lineas y que la cantidad de registros pusheados a la pila fue mayor.
El flag -O2 hace todas las optimizaciones que pueda en el código que no estén involucradas con optimizaciones de espacio-tiempo. Y por último el flag -O3 abre todas las opmitizaciones de -O2 más algunas extras con el fin de hacer aun más optimo el código.[3]
Haciendo uso de las optimizaciones mencionadas anteriormente, se realizaron experimentos de performance usando el código en ASM, en C y en C compilado con optimizacion -O1, -O2 y -O3.

2.1.6 Experimento 3 - secuencial vs. vectorial

Realice una medición de las diferencias de performance entre las versiones de C y ASM (el primero con -O1, -O2 y -O3).
¿Como realizó la medición?¿Cómo sabe que su medición es una buena medida?¿Cómo afecta a la medición la existencia de outliers?¿De qué manera puede minimizar su impacto?¿Qué resultados obtiene si mientras corre los tests ejecuta otras aplicaciones que utilicen al máximo la CPU? Realizar un análisis **riguroso** de los resultados y acompañar con un gráfico que presente estas diferencias.

- Haciendo uso de las optimizaciones mencionadas anteriormente, se realizaron experimentos de performance usando el código en ASM, en C y en C compilado con optimizacion -O1, -O2 y -O3. El siguiente gráfico muestra como el código en C optimizado y el ASM son casi idénticos.

  Esta medición fue realizada cambiando la cantidad de iteraciones y midiendo cuantos ticks de procesador consume procesar el video completo. El problema de realizar las mediciones de esta manera es que el procesador switchea entre distintos procesos todo el tiempo haciendo que mi contador aumente al ejecutar procesos que no pertenecen a mi función y se cuenta en intervalos muy grandes provocando que la probabilidad de contar ticks de procesos exteriores sea mayor. Es por esta razón que nuestra medición no es lo suficientemente precisa. Una forma de hacerla mas precisa sería evaluando un promedio de la cantidad de Ticks que consume por cada frame de cada iteración. Pero al trabajar en ordenes tan grandes deberíamos procesar demasiada información que no viene al caso de lo que se queire mostrar.

2.1.7 Experimento 4 - cpu vs. bus de memoria

Se desea conocer cual es el mayor limitante a la performance de este filtro en su versión ASM.

¿Cuál es el factor que limita la performance en este caso? En caso de que el limitante fuera la intensidad de cómputo, entonces podrían agregarse instrucciones que realicen accesos a memoria y la performance casi no debería sufrir. La inversa puede aplicarse si el limitante es la cantidad de accesos a memoria.

[2] agregando este flag a `CCFLAGS64` en el makefile
[3] Para mas información revisar el http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Realizar un experimento, agregando múltiples instrucciones de un mismo tipo y realizar un análisis del resultado. Acompañar con un gráfico.

Nuestro algoritmo de Tiles, en su versión ASM, lo que hace es simple:

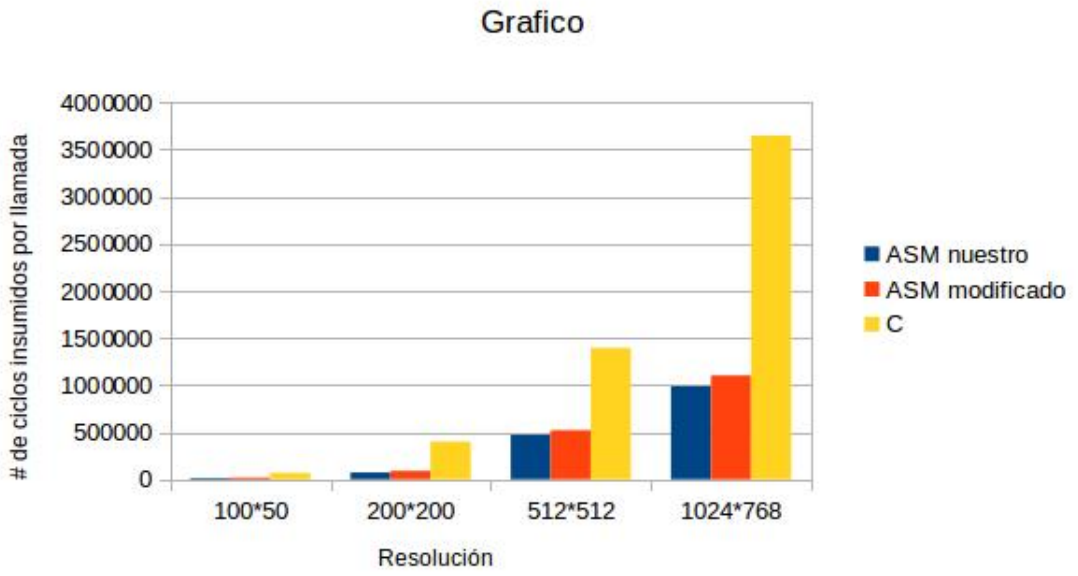
Hace unos pequeños cálculos para localizar la parte de la matriz fuente a copiar en la matriz destino. Luego lo que hace sencillamente es copiar a secas esa submatriz de la matriz destino en la matriz a devolver, esto lo hace repetidas veces. Entonces nuestro algoritmo hace muy pocas instrucciones de procesamiento (desempaquetar, dividir, aplicar máscaras, etcétera) y accede muchas veces a memoria, lo que nos hace pensar que el limitante de nuestro algoritmo es acceder tanto a memoria.

Para tratar de probar o negar esto, modificamos el código agregando varias instrucciones de procesamiento dentro del ciclo principal del algoritmo y comparamos su eficiencia en cuanto a tiempo con respecto al original.

Lo que vamos a hacer es agregarle al código, en su ciclo principal, las siguientes operaciones de procesamiento:

```
MOVDQU XMM1, XMM0
PXOR XMM7, XMM7
PUNPCKLBW XMM0, XMM7
PUNPCKHBW XMM1, XMM7
PACKUSWB XMM0, XMM1
```

Estas instrucciones las ponemos justo después de levantar los píxeles en xmm0, no modifican en nada el resultado del algoritmo.



En el gráfico tenemos el tiles de ASM modificado y lo comparamos con el tiles nuestro y con el tiles de C.

Podemos ver que la cantidad de ciclos insumidos por llamada del algoritmo modificado, aunque aumentan, este aumento no es demasiado (nunca aumenta más de un 15%) con respecto a nuestro ASM, y mucho menos se nota si lo comparamos con los valores del algoritmo hecho en C.

Esto nos deja en claro que el limitante de nuestro algoritmo son los accesos a memoria, ya que no realiza casi ninguna instrucción de prosesamiento en el ciclo principal del algoritmo

2.2 Filtro Popart

2.2.1 Objetivo

El objetivo de este filtro es transformar el color de una forma determinada llamada "popart".
Se trata de sumar los 3 colores de cada pixel, y según el resultado de esa suma, ir poniendo colores predeterminados. Formalizando, según el resultado de la suma:

$$dst_{(x,y)} < r,g,b >= \begin{cases} < 0,0,255 > & \text{si } suma(i,j) < 153 \\ < 127,0,127 > & \text{si } 153 \leq suma(i,j) < 306 \\ < 255,0,255 > & \text{si } 306 \leq suma(i,j) < 459 \\ < 255,0,0 > & \text{si } 459 \leq suma(i,j) < 612 \\ < 255,255,0 > & \text{si no} \end{cases}$$

2.2.2 Desarrollo Implementacion C

Sintesis de Desarrollo

La implementación en C fue simple. Mediante 2 ciclos anidados recorremos la imagen fuente, uno para filas y otro para columnas.
Dentro de la iteración sumamos los colores de cada pixel de la imagen fuente, y según el resultado, pegamos valores predeterminados en la imagen destino.

Explicación detallada de Implementación

En esta implementación, inicialmente,se crearon dos punteros de matriz uno correspondiente a src y otro a dst.
Ademas, un entero llamado *suma* inicializado en 0.
Luego, realizamos dos ciclos, uno añidado dentro del otro recorriendo de la forma $'(int\ i\_d = 0;\ i\_d < filas;\ i\_d++)'$ y $(int\ j\_d = 0;\ j\_d < cols;\ j\_d++)$.
Dentro del segundo ciclo, creamos dos punteros rgb_t donde uno apuntará a
rgb_t $*p\_d = (rgb\_t*)\ \&dst\_matrix[i\_d][j\_d * 3]$, y el otro a $rgb\_t * p\_s = (rgb\_t*)\ \&src\_matrix[i\_d][j\_d]$.
Al entero *suma* le guardamos la suma de los correspondientes r, g y b del puntero p_s $suma = p\_s \rightarrow r + p\_s \rightarrow b + p\_s \rightarrow g$
Posteriormente, chequeamos el valor correspondiente de suma:
Primero si $suma < 153$, en caso verdadero, guardamos en el puntero p_d la posicion 0 de la matriz colores $*p\_d = colores[0]$.
Segundo, si $(153 <= suma\ \&\&\ suma < 306)$, en caso verdadero, guardamos en el puntero p_d la posicion 1 de la matriz colores $*p\_d = colores[1]$.
Tercero, si $(306 <= suma\ \&\&\ suma < 459)$, en caso verdadero, guardamos en el puntero p_d la posicion 2 de la matriz colores $*p\_d = colores[2]$.
Cuarto, si $(459 <= suma\ \&\&\ suma < 612)$, en caso verdadero, guardamos en el puntero p_d la posicion 3 de la matriz colores $*p\_d = colores[3]$.
Por ultimo, si $(612 <= suma)$, en caso verdadero, guardamos en el puntero p_d la posicion 4 de la matriz colores $*p\_d = colores[4]$.
Esta matriz colores mencionada se encuentra definida fuera de la funcion como:
rgb_t colores[] = { {255, 0, 0},
{127, 0, 127}
{255, 0, 255}
{0, 0, 255}
{0, 255, 255}}

Estos dos ciclos realizaran $j\_d = cols - 1$ por $i\_d = filas - 1$ de itereaciones.
Con lo mencionado, obtenemos el filtro popart en c correctamente, con los valores que nos pasan como parámetros.

2.2.3 Desarrollo Implementacion en ASM

En esta implementacion, inicialmente pusheamos RBP, R12 y R13, nos guardamos en R12 y R13 los respectivos valores de columnas y filas.
Luego, guardamos en EAX un 3, guardamos en R10D, R13D y procedemos a hacer la operacion MUL R10, luego realizamos lo mismo con R12 y así, obtenemos en R10 el valor completo de la cantidad de pixeles a recorrer.
Guardamos en R11 el valor de RDI donde venia *src y en R13 el valor de RSI donde venia *dst.
Luego, realizamos la seccion donde chequeamos si estamos en el final de la linea o mejor dicho en el padding, comparamos con R10D si es cero
en caso de ser 0 significa que ya recorrimos toda la imagen completa por ende saltamos a la etiqueta fin.
En caso de no ser cero, comparamos R15D con 15, en r15 teníamos guardado la cantidad de pixel que vamos a recorrer en una iteracion. Como vemos de a 15 pixeles si el valor es menor significa que estamos por tocar el padding y eso lo chequeamos a parte.
Si no es menor, procedemos a recorrer y trabajar con 15 pixeles de la siguiente forma:
Primero, nos guardamos en XMM0 los primeros 15 pixeles haciendo $movdqu\ XMM0, [RDI]$, luego, el valor de XMM0, lo guardamos en XMM1 y XMM2, limpiamos XMM7 y con 3 mascaras distintas que lo que hacen es ponernos los 5 pixeles rojos adelante y llenar con 0, los 5 pixeles azules y llenar con cero y los 5 pixeles verdes y llenar con ceros.
Realizamos la operacion Pshufb con los tres xmm y las mascaras (un xmm con cada mascara) y luego desempaquetamos de byte a word, usando el xmm7 que habiamos llenado de 0.
El desempaquetado hace que nos queden los valores en word en los registros XMM10, XMM12, XMM14 respectivamente, y luego los sumamos con PADDW guardando el valor en XMM10. De esta forma obtenemos en word $|b0+g0+r0|b1+g1+r1|b2+g2+r2|b3+g3+r3|b4+g4+r4|$.
Limpiamos todos los registros que usamos, salvo XMM10 y volvemos a guardar en XMM0 el valor de XMM10 procedemos a las comparaciones. Primero, vamos a comparar si nuestras sumas son mayores a 611, guardamos en XMM14 el valor 611 con una mascara en DW la cual tiene en todos los pack el 611. Guardamos en XMM15 el valor XMM0 y comparamos con XMM14. A partir del resultado en XMM15 realizamos lo siguiente:

$movdqu\ XMM13, [MASK\_5]$
$paddusdw\ XMM10,\ XMM15$
$movdqu\ XMM11,\ XMM15$
$pcmpeqw\ XMM14,\ XMM14$
$pxor\ XMM11,\ XMM14$
$pand\ XMM0,\ XMM11$
$pshufb\ XMM15, [MASK\_DE1WORDA3BYTES]$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM5,\ XMM13$

En MASK_5 tenemos en DW los pack de los resultados que da el caso 5 osea $0|FF|FF$. Luego, en XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos.
Lo salvamos y lo invertimos osea de 00 a FF y viceversa y luego hacemos un AND con los pack de XMM0 para quedarnos con los valores que no dieron TRUE.
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM5.

Despues de chequear este caso pasamos al caso 4:

Primero, vamos a comparar si nuestras sumas son mayores a 458, guardamos en XMM14 el valor 458 con una mascara en DW la cual tiene en todos los pack el 458. Guardamos en XMM15 el valor XMM0 y comparamos con XMM14. A partir del resultado en XMM15 realizamos lo siguiente:

$movdqu\ XMM13, [MASK\_4]$
$PADDUSW\ XMM10,\ XMM15$
$movdqu\ XMM11,\ XMM15$
$pcmpeqw\ XMM14,\ XMM14$
$pxor\ XMM11,\ XMM14$
$pand\ XMM0,\ XMM11$

$pshufb\ XMM15, [MASK\_DE1WORDA3BYTES]$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM4,\ XMM13$

En MASK_4 tenemos en DW los pack de los resultados que da el caso 4 osea $0|0|FF$. Luego, en XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos.
Lo salvamos y lo invertimos osea de 00 a FF y viceversa y luego hacemos un AND con los pack de XMM0 para quedarnos con los valores que no dieron TRUE.
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM4.

Despues de chequear este caso pasamos al caso 3:

Primero, vamos a comparar si nuestras sumas son mayores a 305, guardamos en XMM14 el valor 305 con una mascara en DW la cual tiene en todos los pack el 305. Guardamos en XMM15 el valor XMM0 y comparamos con XMM14. A partir del resultado en XMM15 realizamos lo siguiente:

$movdqu\ XMM13, [MASK\_3]$
$paddusw\ XMM10,\ XMM15$
$movdqu\ XMM11,\ XMM15$
$pcmpeqw\ XMM14,\ XMM14$
$pxor\ XMM11,\ XMM14$
$pand\ XMM0,\ XMM11$
$pshufb\ XMM15, [MASK\_DE1WORDA3BYTES]$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM3,\ XMM13$

En MASK_3 tenemos en DW los pack de los resultados que da el caso 3 osea $FF|0|FF$. Luego, en XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos.
Lo salvamos y lo invertimos osea de 00 a FF y viceversa y luego hacemos un AND con los pack de XMM0 para quedarnos con los valores que no dieron TRUE.
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM3.

Despues de chequear este caso pasamos al caso 2:

Primero, vamos a comparar si nuestras sumas son mayores a 152, guardamos en XMM14 el valor 152 con una mascara en DW la cual tiene en todos los pack el 152. Guardamos en XMM15 el valor XMM0 y comparamos con XMM14. A partir del resultado en XMM15 realizamos lo siguiente:

$movdqu\ XMM13, [MASK\_2]$
$paddusw\ XMM10,\ XMM15$
$movdqu\ XMM11,\ XMM15$
$pcmpeqw\ XMM14,\ XMM14$
$pxor\ XMM11,\ XMM14$
$pand\ XMM0,\ XMM11$
$pshufb\ XMM15, [MASK\_DE1WORDA3BYTES]$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM2,\ XMM13$

En MASK_2 tenemos en DW los pack de los resultados que da el caso 2 osea $7F|0|7F$. Luego, en XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos.
Lo salvamos y lo invertimos osea de 00 a FF y viceversa y luego hacemos un AND con los pack de XMM0 para quedarnos con los valores que no dieron TRUE.
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM2.

Y por último, el caso 1:

Este caso, es particular, ya que invertimos, y chequeamos si los packs que quedan son menores a 153:

$movdqa\ XMM14, [MASK\_153]$
$movdqu\ XMM6,\ XMM10$
$pcmpeqw\ XMM7,\ XMM7$
$pxor\ XMM6,\ XMM7$
$pand\ XMM14,\ XMM6$
$movdqu\ XMM15,\ XMM0$
$pcmpgtw\ XMM14,\ XMM15$

A partir del resultado en XMM14 realizamos lo siguiente:

$movdqu\ XMM13, [MASK\_1]$
$paddusw\ XMM10,\ XMM14$
$movdqu\ XMM11,\ XMM14$
$pcmpeqw\ XMM15,\ XMM15$
$pxor\ XMM11,\ XMM15$
$pand\ XMM0,\ XMM11$
$pshufb\ XMM14, [MASK\_DE1WORDA3BYTES]$
$pand\ XMM13,\ XMM14$
$movdqu\ XMM1,\ XMM13$

En MASK_1 tenemos en DW los pack de los resultados que da el caso 1 osea $FF|0|0$. Luego, en XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos.
Lo salvamos y lo invertimos osea de 00 a FF y viceversa y luego hacemos un AND con los pack de XMM0 para quedarnos con los valores que no dieron TRUE.
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM1.

Por finalizado, hacemos or entre los XMM1, 2, 3, 4 y 5, y movemos el valor final de XMM1 a XMM0. Comparamos si estamos chequeando el borde haciendo $cmp\ R15D, 15 jl.terminoBorde$ en caso de no ser menor movemos los packs del XMM0 al puntero en RDI $movdqu\ [RSI], XMM0$, le restamos a R15D y a r10d, la cantidad de pixeles por ciclo y le sumamos a RDI y RSI la cantidad de pixeles por ciclo.

En caso de que nuestra comparacion diera que es menor a 15, procedemos a:

$movdqu\ [RSI - pixels\_por\_ciclo + r12],\ XMM0$
$add\ R11, R8$
$add\ R13, R8$
$mov\ RDI, R11$
$mov\ R11, RDI$
$mov\ RSI, R13$
$mov\ R13, RSI$
$sub\ R10D, R15D$
$mov\ R15D, R14D$

Como habiamos enuciado inicialmente, en R11 y R13 nos habiamos guardado RDI y RSI respectivamente y en R8 tenemos por parametro src_row_size. De esta forma hacemos que en dst se guarden los ultimos pixeles de la linea y baje a la siguiente sin tocar padding.

Esto, como fue explicado, se recorre, hasta que R10D sea igual a 0.

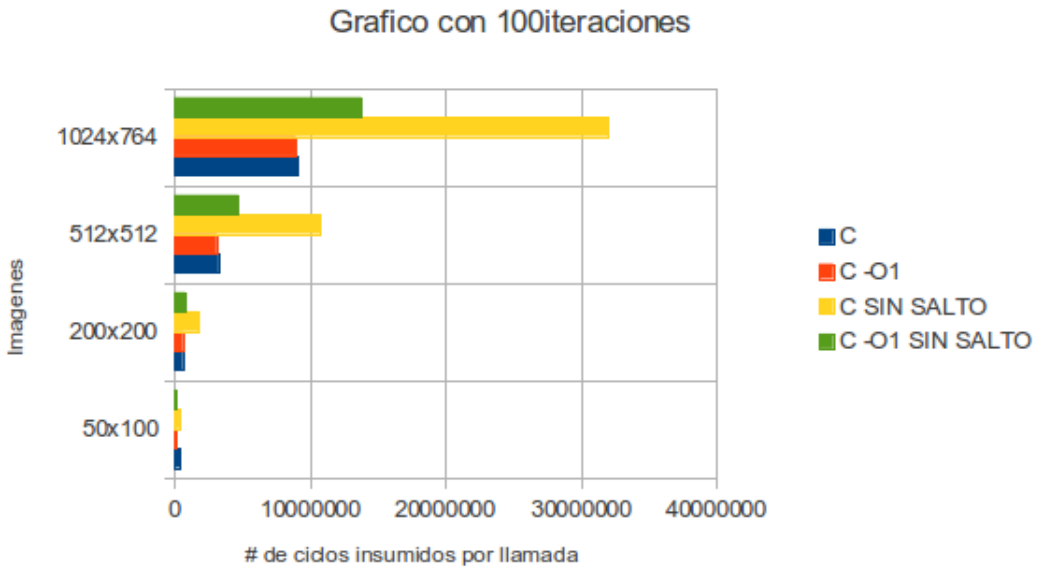
2.2.4 Experimento 1 - saltos condicionales

Se desea conocer que tanto impactan los saltos condicionales en el código del ejercicio anterior con `-O1`. Para poder medir esto, una posibilidad es quitar las comparaciones al procesar cada pixel. Por más que la imagen resultante no sea correcta, será posible tomar una medida del impacto de los saltos condicionales. Analizar como varía la performance.

Si se le ocurren, mencionar otras posibles formas de medir el impacto de los saltos condicionales.

Para poder conocer como impactaban los saltos condicionales en la implementacion realizamos distintas mediciones; quitando los saltos con y sin optimizacion obteniendo los siguientes resultados: Quitando los saltos sin optimizar obtuvimos que, para imagenes de tamaño pequeño (50x100) con 100 iteraciones, el mencionado codigo insume una cantidad de 389596 ciclos por cada llamada. A su vez, con imagenes de tamaño mas grande (200x200) utilizando la misma cantidad de iteraciones, el codigo nos insume una cantidad de 1803084 ciclos por llamada, mientras que con imagenes aun mas grande como por ejemplo 512x512 y 1024x768, demanda 10745530 y 32020980 por llamada respectivamente. Mientras que la implementación sin saltos con optimizacion -O1 cosechamos la siguiente información: Para imagenes de tamaño pequeño (50x100) con la misma cantidad de iteraciones insume 167132 ciclos por llamada. Con imagenes de mayor tamaño (200x200), demanda 805378 de ciclos. Por ultimo con imagenes de tamaño aun mayor (512x512) y (1024x768), emplea 4719143 y 13811050 de ciclos por llamada correspondientemente.

Para una mejor representación se realizo un grafico mostrando a simple vista las diferencias entre cada tipo de veRSIon de la implementacion implementacion.



Concluimos que en imagenes grandes es muy notoria la diferencia en performace entre las veRSIones con y sin saltos.

2.2.5 Experimento 2 - cpu vs. bus de memoria

¿Cuál es el factor que limita la performance en este caso?

Realizar un experimento, agregando múltiples instrucciones de un mismo tipo y realizar un análisis del resultado. Acompañar con un gráfico.

En este caso lo que limita a la performance son las comparaciones y saltos condicionales. Para probar esto agregamos instrucciones de varios tipos en la veRSIón de ASM: accesos a memoria, movimiento de

registros, sumas, comparaciones y finalmente comparaciones y saltos.

Todas estas agregadas en cada instacia de procesamiento, es decir, tomamos el dato, lo procesamos, agregamos las instrucciones (50) y así sucesivamente con cada dato.

Como resultado observamos que lo que más ciclos insume son las comparaciones y saltos condicionales. Este tipo de instrucciones incrementan con el tamaño de la imagen, ya que mientras mas grande es la imagen, más comparaciones para fijarse si llegó al final tendrá que hacer.

Los accesos a memoria también son costosos, pero quedaron en segundo lugar.

A continuación enunciaremos los respectivos ciclos insumidos por llamada dependiendo cada caso y cada resolución de imagen.

120x56

+50 accesos : 124262.398

+50 movimientos : 85166.438

+50 sumas : 100156.117

+50 comparaciones : 106732.922

+50 comp y saltos : 233104.562

200x200

+50 accesos : 748478.750

+50 movimientos : 511008.031

+50 sumas : 584339.938

+50 comparaciones : 578177.250

+50 comp y saltos : 1411856.125

512x512

+50 accesos : 5279703.500

+50 movimientos : 3266983.000

+50 sumas : 3802452.750

+50 comparaciones : 4524299.000

+50 comp y saltos : 9041962.000

1023x767

+50 accesos : 18832506.000

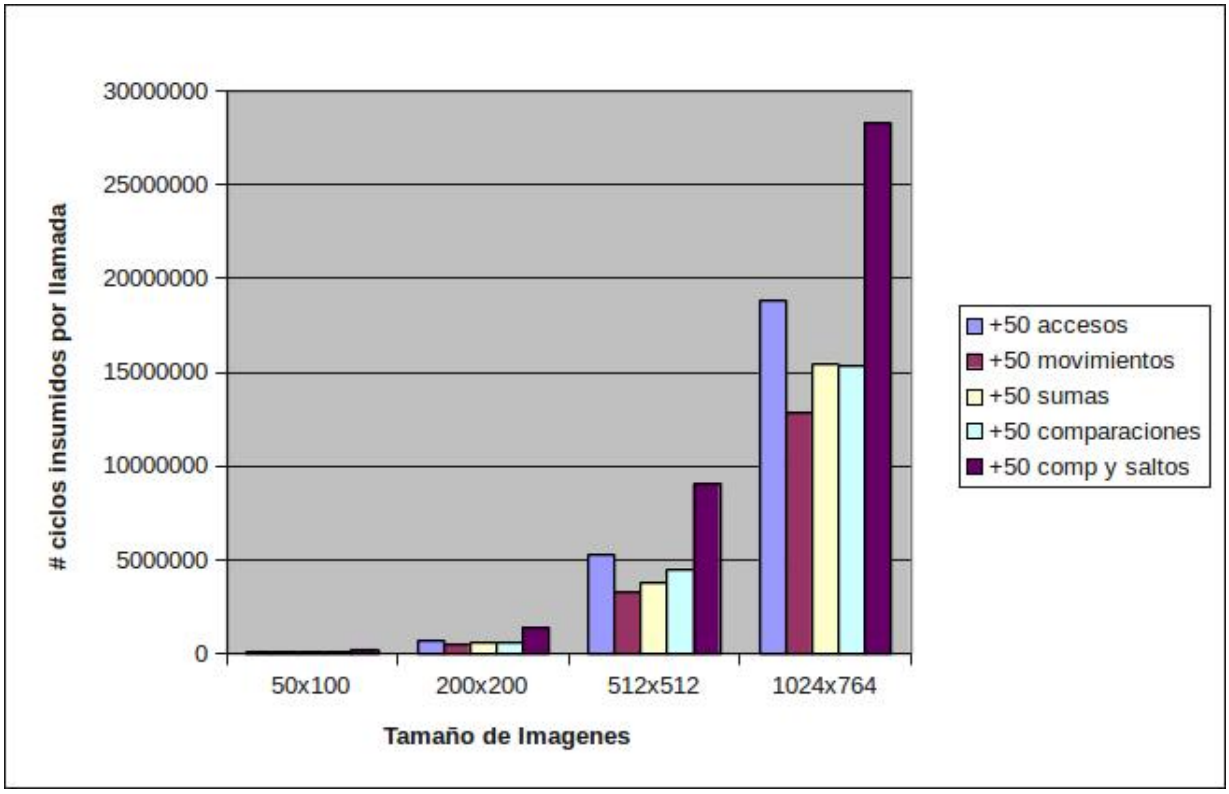
+50 movimientos : 12843421.000

+50 sumas : 15499867.000

+50 comparaciones : 15349734.000

+50 comp y saltos : 28297848.000

En el siguiente gráfico podemos distinguir los resultados.



2.2.6 Experimento 3 - prefetch

La técnica de *prefetch* es otra forma de optimización que puede realizarse. Su sustento teórico es el siguiente:

Suponga un algoritmo que en cada iteración tarda n ciclos en obtener un dato y una cantidad similar en procesarlo. Si el algoritmo lee el dato i y luego lo procesa, despeRDIciará siempre n ciclos esperando entre que el dato llega y que se comienza a procesar efectivamente. Un algoritmo más inteligente podría pedir el dato $i + 1$ al comienzo del ciclo de proceso del dato i (siempre suponiendo que el dato i pidió en la iteración $i - 1$. De esta manera, a la vez que el procesador computa todas las instrucciones de la iteración i, se estáran trayendo los datos de la siguiente iteración, y cuando esta última comience, los datos ya habrán llegado.

Estudiar esta técnica y proponer una aplicación al código del filtro en la veRSIón ASM. Programarla y analizar el resultado. ¿Vale la pena hacer prefetching?

Cuando empezamos a estudiar esta técnica, creímos que no era realizable en assembler, ya que pensamos que cada instrucción era bloqueante, es decir, que hasta que no se terminaba de ejecutar una instrucción no seguía con la siguiente.

Luego descubrimos que no era así. El procesador puede ejecutar las instrucciones fuera de orden. Si no es necesario, o no depende de la instrucción anterior, el procesador no espera a que se termine de ejecutar la instrucción actual para ejecutar la siguiente.

Para probar esto, adaptamos el prefecth a assembler. Modificamos levemente el recorrido de la imagen para no hacer accesos a memoria, al menos no directamente. Lo hicimos a través de un registro. Por ejemplo, levantamos de memoria en XMM8, luego lo pasamos a XMM0 y seguimos levantando de memoria la siguiente posición en XMM8. Así mientras se procesa el dato de XMM0 se puede ir trayendo el siguiente dato a procesar en XMM8.

A continuación, enunciaremos los valores obtenidos de nuestras mediciones en las respectivas resolu-

ciones utilizando prefecth y sin utilizarlo.

120x56

sin prefetch: 67836.359
con prefetch: 65300.320

200x200

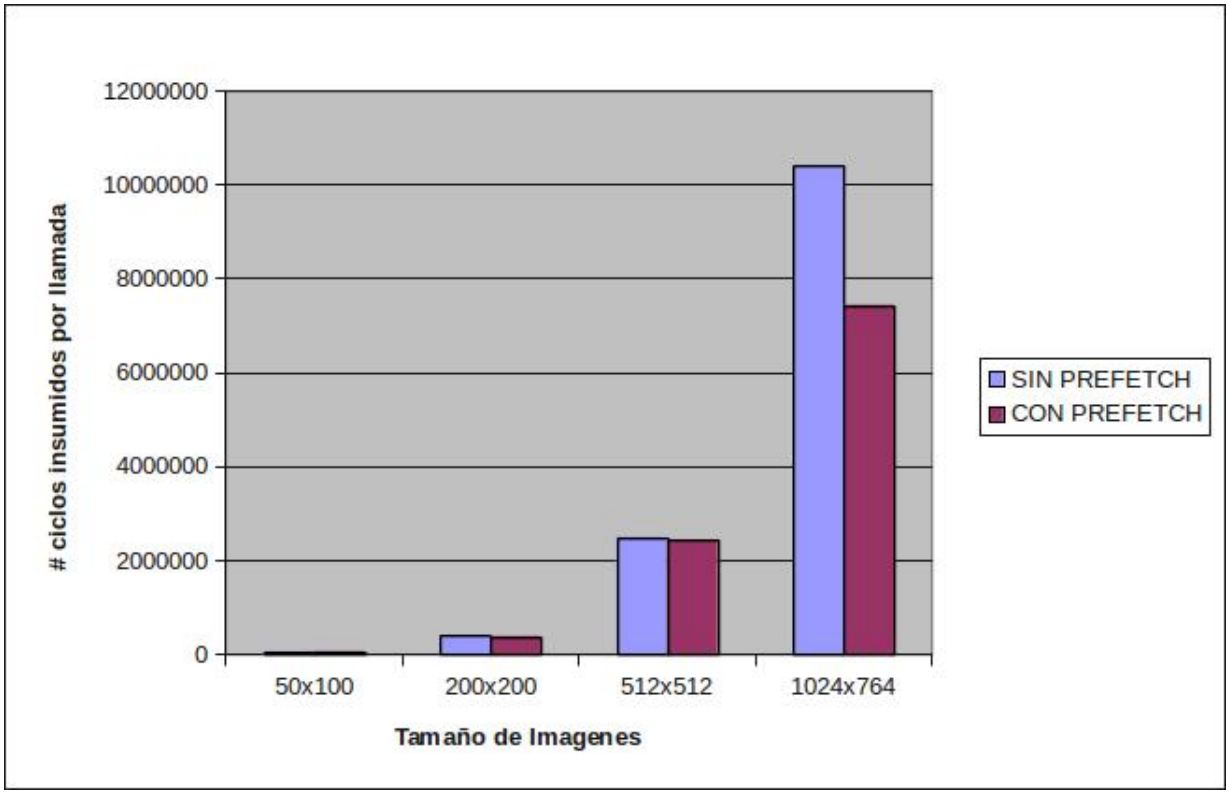
sin prefetch: 398975.594
con prefetch: 382423.562

512x512

sin prefetch: 2498557.750
con prefetch: 2438276.250

1023x767

sin prefetch: 10389265.000
con prefetch: 7434984.500



Se puede observar que hay una sutil mejora en la performance en tamaños pequeños y medianos. En cambio en la imagen grande, se puede observar una brecha más amplia.

Lógicamente, mientras más grande es la imagen, la mejora será más notoria al haber más accesos a memoria.

Como conclusión decimos que no vale la pena hacer prefetching para este filtro, ya que la diferencia se puede apreciar sólo en imágenes grandes, y además te restringe el uso de un registro XMM.

2.2.7 Experimento 4 - secuencial vs. vectorial

Analizar cuales son las diferencias de performace entre las veRSIones de C y ASM. Realizar gráficos que representen estas diferencias.

Realizando nuestras mediciones en la implementacion de C y en ASM obtenemos los siguientes resultados:

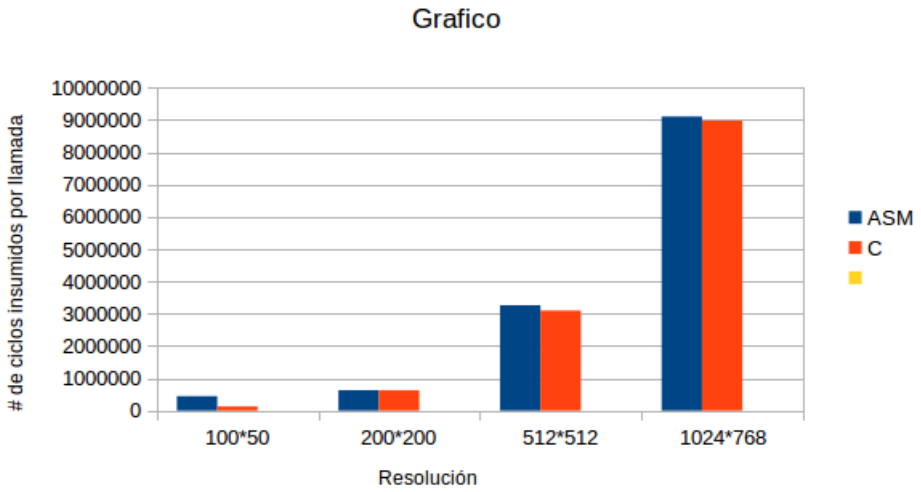
120x56
En ASM: 444959
En C: 128763

200x200
En ASM: 631827
En C: 628630

512x512
En ASM: 3261260
En C: 3098234

1023x767
En ASM: 9107332
En C: 8978195

Para una mejor representación se realizaron distintos graficos mostrando a simple vista las diferencias entre cada tipo de implementacion en cada tipo de imagen.



La diferencia principal entre el código escrito en lenguaje C y el código ASM, es la cantidad de accesos a memoria realizados en cada iteración. Otra diferencia puede verse en la cantidad de bytes que pueden ser procesados simultaneamente durante el mismo ciclo. Mientras que en C leemos y procesamos un byte por vez, el código ASM nos permite acceder y procesar 16 bytes en simultaneos. De todas maneras, en este caso trabajamos con 12 bytes que es un numero mas útil ya que es múltiplo de 3 (cantidad de bytes por píxel) y de 4 (cantidad de bytes por cada caracter). Pudiendo levantar de a 16 bytes en memoria, podría esperarse que el código asm demorase una diesiseiava parte de la cantidad de ciclos que demora el código en C. Pero debemos tener en cuenta que en nuestro caso estamos aprovechando solo 15 de los 16 bytes que leemos en cada iteración por lo tanto es más acertado evaluar como si solo leyeramos 15 bytes. Además, hay que tener en cuenta que las instrucciones SSE pueden necesitar más ciclos que las operaciones comunes que usamos en el C, eso disminuye un poco la performance en la comparación. Por mas que como hemos visto los saltos condicionales empeoran la performace en ASM, dicha implementacion sigue siendo mas optima que la de C.

2.3 Filtro Temperature

2.3.1 Objetivo

El objetivo del filtro "temperature" es, al igual que el popart, transformar el color, con la diferencia de que el resultado simula un efecto de calor. Para esto, se suman los 3 colores de cada pixel y se lo divide por 3. Y según el resultado de eso, al que llamaremos "t", se ponen valores predeterminados:

$$dst_{(x,y)} < r,g,b >= \begin{cases} < 0,0,128+t*4 > & \text{si } t < 32 \\ < 0(t-32)*4,255 > & \text{si } 32 \leq t < 96 \\ < (t-96)*4,255,255-(t-96)*4 > & \text{si } 96 \leq t < 160 \\ < 255,255-(t-160)*4,0 > & \text{si } 160 \leq t < 224 \\ < 255-(t-224)*4,0,0 > & \text{si no} \end{cases}$$

2.3.2 Desarrollo Implementacion C

Sintesis de Desarrollo

Similiar al de popart. Recorremos mediante dos ciclos la imagen fuente para sumar y dividimos por 3 cada pixel, y según el resultado de eso, pegar un valor predetermiado.

Explicación detallada de Implementación

En esta implementación, inicialmente,se crearon dos punteros de matriz uno correspondiente a src y otro a dst.
Ademas, un entero llamado *suma* y otro *divido* inicializados en 0.
Luego, realizamos dos ciclos, uno añidado dentro del otro recorriendo de la forma $'(int\ i\_d = 0;\ i\_d < filas;\ i\_d++)'$ y $(int\ j\_d = 0;\ j\_d < cols;\ j\_d++)$.
Dentro del segundo ciclo, creamos dos punteros rgb_t donde uno apuntará a
rgb_t $*p\_d = (rgb\_t*)\ \&dst\_matrix[i\_d][j\_d*3]$, y el otro a $rgb\_t*p\_s = (rgb\_t*)\ \&src\_matrix[i\_d][j\_d]$.
Al entero *suma* le guardamos la suma de los correspondientes r, g y b del puntero p_s $suma = p\_s \rightarrow r + p\_s \rightarrow b + p\_s \rightarrow g$ Al entero *divido* le guardamos el valor de suma dividido por 3 $divido = suma/3$
Posteriormente, chequeamos entre que valores se encuentra divido utilizando la funcion between provista por la catedra:

Primero si $(between(divido, 0, 31))$, en caso verdadero, guardamos en el valor r del puntero p_d el valor 0,en el valor g guardamos el valor 0 y en el valor b guardamos $(128 + (4 * divido)$.
Segundo, si $(between(divido, 32, 95))$, en caso verdadero, guardamos en el valor r del puntero p_d el valor 0,en el valor g guardamos el valor $(divido - 32) * 4$ y en el valor b guardamos 255.
Tercero, si $(between(divido, 96, 159))$, en caso verdadero, guardamos en el valor r del puntero p_d el valor $(divido - 96) * 4$,en el valor g guardamos el valor 255 y en el valor b guardamos $255 - ((divido - 96) * 4)$.
Cuarto, si $(between(divido, 160, 223))$, en caso verdadero, guardamos en el valor r del puntero p_d el valor 255,en el valor g guardamos el valor $255 - ((divido - 160) * 4)$ y en el valor b guardamos 0.
Por ultimo, si $(between(divido, 224, 255))$, en caso verdadero, guardamos en el valor r del puntero p_d el valor $255 - (divido - 224) * 4$,en el valor g guardamos el valor 0 y en el valor b guardamos 0.

Estos dos ciclos realizaran $j\_d = cols - 1$ por $i\_d = filas - 1$ de itereaciones.
Con lo mencionado, obtenemos el filtro temperature en c correctamente, con los valores que nos pasan como parámetros.

2.3.3 Desarrollo Implementacion en ASM

En esta implementacion, inicialmente pusheamos RBP, R12 y R13, nos guardamos en R12 y R13 los respectivos valores de columnas y filas.

Luego, guardamos en EAX un 3, guardamos en R10D, R13D y procedemos a hacer la operacion MUL R10, luego realizamos lo mismo con R12 y así, obtenemos en R10 el valor completo de la cantidad de pixeles a recorrer.
Guardamos en R11 el valor de RDI donde venia *src y en R13 el valor de RSI donde venia *dst.
Luego, realizamos la seccion donde chequeamos si estamos en el final de la linea o mejor dicho en el padding, comparamos con R10D si es cero
en caso de ser 0 significa que ya recorrimos toda la imagen completa por ende saltamos a la etiqueta fin.
En caso de no ser cero, comparamos R15D con 15, en r15 teníamos guardado la cantidad de pixel que vamos a recorrer en una iteracion. Como vemos de a 15 pixeles si el valor es menor significa que estamos por tocar el padding y eso lo chequeamos a parte.
Si no es menor, procedemos a recorrer y trabajar con 15 pixeles de la siguiente forma:
Primero, nos guardamos en XMM0 los primeros 15 pixeles haciendo $movdqu\ XMM0, [RDI]$, luego, el valor de XMM0, lo guardamos en XMM1 y XMM2, limpiamos XMM7 y con 3 mascaras distintas que lo que hacen es ponernos los 5 pixeles rojos adelante y llenar con 0, los 5 pixeles azules y llenar con cero y los 5 pixeles verdes y llenar con ceros.
Realizamos la operacion Pshufb con los tres xmm y las mascaras (un xmm con cada mascara) y luego desempaquetamos de byte a word, usando el XMM7 que habiamos llenado de 0.
El desempaquetado hace que nos queden los valores en word en los registros XMM10, XMM12, XMM14 respectivamente, y luego los sumamos con PADDW guardando el valor en XMM10. De esta forma obtenemos en word $|b0 + g0 + r0|b1 + g1 + r1|b2 + g2 + r2|b3 + g3 + r3|b4 + g4 + r4|$.
Limpiamos todos los registros que usamos, salvo XMM10 y volvemos a guardar en XMM0 el valor de XMM10. Luego, procedemos a realizar la division correspondiente de los packs de la siguiente manera:

$movdqu\ XMM10, [DIVIDO]$
$movdqu\ XMM1,\ XMM0$
$movdqu\ XMM2,\ XMM0$
$xorpd\ XMM15,\ XMM15$
$punpcklwd\ XMM1,\ XMM15$
$punpckwd\ XMM2,\ XMM15$
$cvtdq2ps\ XMM1,\ XMM1$
$cvtdq2ps\ XMM2,\ XMM2$
$cvtdq2ps\ XMM10, XMM10$
$divps\ XMM1,\ XMM10$
$divps\ XMM2,\ XMM10$
$cvttps2dq\ XMM1, XMM1$
$cvttps2dq\ XMM2,\ XMM2$
$packusdw\ XMM1,\ XMM2$
$movdqu\ XMM0,\ XMM1$

En la mascara DIVIDO tenemos el valor 3 en DD. Desempaquetamos de Word a Dword para luego podes convertir a float. Realizamos la respectiva division y luego volvemos a convertir de float a dword y de dword a word y volvemos a guardar el resultado en XMM0.

Limpiamos todos los registros que usamos, salvo XMM0. Posteriormente, seguimos con las comparaciones. Primero, vamos a comparar si nuestras sumas son mayores a 223, guardamos en XMM14 el valor 223 con una mascara en DW la cual tiene en todos los pack el 223. Guardamos en XMM15 el valor XMM0 y comparamos con XMM14. A partir del resultado en XMM15 realizamos lo siguiente:

$movdqu\ XMM13,\ XMM0$
$movdqu\ XMM11, [M224]$
$psubb\ XMM13,\ XMM11$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM11, [M4]$
$movdqu\ XMM7,\ XMM13$
$movdqu\ XMM8,\ XMM13$
$xorpd\ XMM9,\ XMM9$
$punpcklwd\ XMM7,\ XMM9$

$punpckwd\ XMM8,\ XMM9$
$cvtdq2ps\ XMM7,\ XMM7$
$cvtdq2ps\ XMM8,\ XMM8$
$cvtdq2ps\ XMM11, XMM11$
$mulps\ XMM7,\ XMM11$
$mulps\ XMM8,\ XMM11$
$cvtps2dq\ XMM7, XMM7$
$cvtps2dq\ XMM8,\ XMM8$
$packusdw\ XMM7,\ XMM8$
$movdqu\ XMM13,\ XMM7$
$movdqu\ XMM11, [M255]$
$pand\ XMM11,\ XMM15$
$psubb\ XMM11,\ XMM13$
$movdqu\ XMM13,\ XMM11$
$pshufb\ XMM13, [MASK_5]$
$paddusw\ XMM10,\ XMM15$
$movdqu\ XMM11,\ XMM15$
$pcmpeqw\ XMM14, XMM14$
$pxor\ XMM11,\ XMM14$
$pand\ XMM0, XMM11$
$pshufb\ XMM15, [MASK_DE1WORDA3BYTES]$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM5,\ XMM13$

Copiamos en XMM13, el valor de XMM0, guardamos en XMM11 una mascara en DW con 224, realizamos la resta de los packs y luego un and con XMM15 para quedarnos con los packs que cumplieron esta comparacion. Despues, guardamos en XMM11 una mascara en DW con 4. Procedemos a desempaquetar de WORD a DWORD el XMM13 copiandolo previamente en otros dos XMM. Luego convertimos todo de DWORD a FLOAT y realizamos la multiplicacion con XMM11. Volvemos a convertir a DWORD y a empaquetar a WORD y guardamos el resultado en XMM13.
Guardamos en XMM11 una mascara DW con 255 realizamos un and entre este XMM y el XMM15 para quedarnos con los packs que dieron TRUE la comparacion y luego restamos XMM11 con XMM13 y guardamos el valor de XMM11 en XMM13. En MASK_5 tenemos en DW los pack de los resultados que da el caso 5 osea $X|0|0$. Luego, en XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos.
Lo salvamos y lo invertimos osea de 00 a FF y viceversa y luego hacemos un AND con los pack de XMM0 para quedarnos con los valores que no dieron TRUE.
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM5.

Despues de chequear este caso pasamos al caso 4:

Primero, vamos a comparar si nuestras sumas son mayores a 159, guardamos en XMM14 el valor 159 con una mascara en DW la cual tiene en todos los pack el 159. Guardamos en XMM15 el valor XMM0 y comparamos con XMM14. A partir del resultado en XMM15 realizamos lo siguiente:

$movdqu\ XMM13,\ XMM0$
$movdqu\ XMM11, [M160]$
$psubb\ XMM13,\ XMM11$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM11, [M4]$
$movdqu\ XMM7,\ XMM13$
$movdqu\ XMM8,\ XMM13$
$xorpd\ XMM9,\ XMM9$
$punpcklwd\ XMM7,\ XMM9$
$punpckwd\ XMM8,\ XMM9$
$cvtdq2ps\ XMM7,\ XMM7$
$cvtdq2ps\ XMM8,\ XMM8$

$cvtdq2ps\ XMM11, XMM11$
$mulps\ XMM7,\ XMM11$
$mulps\ XMM8,\ XMM11$
$cvtps2dq\ XMM7, XMM7$
$cvtps2dq\ XMM8,\ XMM8$
$packusdw\ XMM7,\ XMM8$
$movdqu\ XMM13,\ XMM7$
$movdqu\ XMM11, [MASK\_4]$
$pshufb\ XMM13, [MASK\_4\_2]$
$psubb\ XMM11,\ XMM13$
$movdqu\ XMM13,\ XMM11$
$movdqu\ XMM11,\ XMM15$
$pcmpeqw\ XMM14, XMM14$
$pxor\ XMM11,\ XMM14$
$pand\ XMM0, XMM11$
$paddusw\ XMM10,\ XMM15$
$pshufb\ XMM15, [MASK\_DE1WORDA3BYTES]$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM4,\ XMM13$

Copiamos en XMM13, el valor de XMM0, guardamos en XMM11 una mascara en DW con 160, realizamos la resta de los packs y luego un and con XMM15 para quedarnos con los packs que cumplieron esta comparacion. Despues, guardamos en XMM11 una mascara en DW con 4. Procedemos a desempaquetar de WORD a DWORD el XMM13 copiandolo previamente en otros dos XMM. Luego convertimos todo de DWORD a FLOAT y realizamos la multiplicacion con XMM11. Volvemos a convertir a DWORD y a empaquetar a WORD y guardamos el resultado en XMM13.
Guardamos en XMM11 una mascara DW con los resultados que da el caso 4 y hacemos un movimiento de Packs en XMM13 de la forma $X|0x80|0x80$ realizamos la resta entre ambos y guardamos el resultado de XMM11 en XMM13. Guardamos en XMM11 el valor de XMM15 invertimos el XMM11 y luego hacemos un and con XMM0.
Luego, en XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos.
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM4.

Despues de chequear este caso pasamos al caso 3:

Primero, vamos a comparar si nuestras sumas son mayores a 95, guardamos en XMM14 el valor 95 con una mascara en DW la cual tiene en todos los pack el 95. Guardamos en XMM15 el valor XMM0 y comparamos con XMM14. A partir del resultado en XMM15 realizamos lo siguiente:

$movdqu\ XMM13,\ XMM0$
$movdqu\ XMM11, [M96]$
$psubb\ XMM13,\ XMM11$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM11, [M4]$
$movdqu\ XMM7,\ XMM13$
$movdqu\ XMM8,\ XMM13$
$xorpd\ XMM9,\ XMM9$
$punpcklwd\ XMM7,\ XMM9$
$punpckwd\ XMM8,\ XMM9$
$cvtdq2ps\ XMM7,\ XMM7$
$cvtdq2ps\ XMM8,\ XMM8$
$cvtdq2ps\ XMM11, XMM11$
$mulps\ XMM7,\ XMM11$
$mulps\ XMM8,\ XMM11$
$cvtps2dq\ XMM7, XMM7$
$cvtps2dq\ XMM8,\ XMM8$

$packusdw\ XMM7,\ XMM8$
$movdqu\ XMM13,\ XMM7$
$pshufb\ XMM13, [MASK\_3]$
$movdqu\ XMM11, [MASK\_CON255]$
$psubb\ XMM11,\ XMM13$
$pshufb\ XMM13, [MASK\_3\_2]$
$pshufb\ XMM11, [MASK\_3\_3]$
$paddb\ XMM13,\ XMM11$
$paddusw\ XMM10,\ XMM15$
$movdqu\ XMM11,\ XMM15$
$pcmpeqw\ XMM14, XMM14$
$pxor\ XMM11,\ XMM14$
$pand\ XMM0, XMM11$
$pshufb\ XMM15, [MASK\_DE1WORDA3BYTES]$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM3,\ XMM13$

Copiamos en XMM13, el valor de XMM0, guardamos en XMM11 una mascara en DW con 96, realizamos la resta de los packs y luego un and con XMM15 para quedarnos con los packs que cumplieron esta comparacion. Despues, guardamos en XMM11 una mascara en DW con 4. Procedemos a desempaquetar de WORD a DWORD el XMM13 copiandolo previamente en otros dos XMM. Luego convertimos todo de DWORD a FLOAT y realizamos la multiplicacion con XMM11. Volvemos a convertir a DWORD y a empaquetar a WORD y guardamos el resultado en XMM13.
Movemos en XMM13 con PSHUFB una mascara DW con los resultados que da el caso 3 y guardamos en XMM11 una mascara en DW con 255, realizamos la resta entre ambos. Movemos los packs de XMM11 y XMM13 con las mascaras $[MASK\_3\_2]$ $[MASK\_3\_3]$ y luego los sumamos.
Luego, en XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos. Y luego invertirmos XMM11 donde guardamos XMM15 para hacer un AND con XMM0 y dejar solo habilitados los packs que no cumplieron
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM3.

Despues de chequear este caso pasamos al caso 2:

Primero, vamos a comparar si nuestras sumas son mayores a 31, guardamos en XMM14 el valor 31 con una mascara en DW la cual tiene en todos los pack el 31. Guardamos en XMM15 el valor XMM0 y comparamos con XMM14. A partir del resultado en XMM15 realizamos lo siguiente:

$movdqu\ XMM13,\ XMM0$
$movdqu\ XMM11, [M32]$
$psubb\ XMM13,\ XMM11$
$movdqu\ XMM11, [M4]$
$movdqu\ XMM7,\ XMM13$
$movdqu\ XMM8,\ XMM13$
$xorpd\ XMM9,\ XMM9$
$punpcklwd\ XMM7,\ XMM9$
$punpckwd\ XMM8,\ XMM9$
$cvtdq2ps\ XMM7,\ XMM7$
$cvtdq2ps\ XMM8,\ XMM8$
$cvtdq2ps\ XMM11, XMM11$
$mulps\ XMM7,\ XMM11$
$mulps\ XMM8,\ XMM11$
$cvtps2dq\ XMM7, XMM7$
$cvtps2dq\ XMM8,\ XMM8$
$packusdw\ XMM7,\ XMM8$
$movdqu\ XMM13,\ XMM7$
$movdqu\ XMM11, [MASK\_2]$
$pshufb\ XMM13, [MASK\_2\_2]$

$paddb\ XMM13,\ XMM11$
$paddusw\ XMM10,\ XMM15$
$movdqu\ XMM11,\ XMM15$
$pcmpeqw\ XMM14, XMM14$
$pxor\ XMM11,\ XMM14$
$pand\ XMM0, XMM11$
$pshufb\ XMM15, [MASK\_DE1WORDA3BYTES]$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM2,\ XMM13$

Copiamos en XMM13, el valor de XMM0, guardamos en XMM11 una mascara en DW con 96, realizamos la resta de los packs y luego un and con XMM15 para quedarnos con los packs que cumplieron esta comparacion. Despues, guardamos en XMM11 una mascara en DW con 4. Procedemos a desempaquetar de WORD a DWORD el XMM13 copiandolo previamente en otros dos XMM. Luego convertimos todo de DWORD a FLOAT y realizamos la multiplicacion con XMM11. Volvemos a convertir a DWORD y a empaquetar a WORD y guardamos el resultado en XMM13.
Movemos los packs de XMM11 y XMM13 con las mascaras $[MASK\_3\_2]$ $[MASK\_3\_3]$ y luego los sumamos.
Luego, en XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos. Y luego invertirmos XMM11 donde guardamos XMM15 para hacer un AND con XMM0 y dejar solo habilitados los packs que no cumplieron
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM2.

Y por último, el caso 1:

Este caso, es particular, ya que invertimos, y chequeamos si los packs que quedan son menores a 31:

$XMM14, [MASK\_31]$
$movdqu\ XMM6,\ XMM10$
$PCMPEQW\ XMM7,\ XMM7$
$pxor\ XMM6,\ XMM7$
$pand\ XMM14,\ XMM6$
$movdqu\ XMM15,\ XMM0$
$pcmpgtw\ XMM14,\ XMM15$

A partir del resultado en XMM14 realizamos lo siguiente:

$movdqu\ XMM12,\ XMM15$
$movdqu\ XMM11, [M4]$
$movdqu\ XMM7,\ XMM12$
$movdqu\ XMM8,\ XMM12$
$xorpd\ XMM9,\ XMM9$
$punpcklwd\ XMM7,\ XMM9$
$punpckwd\ XMM8,\ XMM9$
$cvtdq2ps\ XMM7,\ XMM7$
$cvtdq2ps\ XMM8,\ XMM8$
$cvtdq2ps\ XMM11, XMM11$
$mulps\ XMM7,\ XMM11$
$mulps\ XMM8,\ XMM11$
$cvtps2dq\ XMM7, XMM7$
$cvtps2dq\ XMM8,\ XMM8$
$packusdw\ XMM7,\ XMM8$
$movdqu\ XMM12,\ XMM7$
$movdqu\ XMM13, [M128]$
$pand\ XMM13,\ xmm6$
$paddb\ XMM13,\ XMM12$
$pshufb\ XMM13, [MASK\_1]$

$paddusw\ XMM10,\ XMM14$
$pshufb\ XMM14, [MASK\_DE1WORDA3BYTES]$
$movdqu\ XMM1,\ XMM13$

Copiamos en XMM12, el valor de XMM15, guardamos en XMM11 una mascara en DW con 4. Procedemos a desempaquetar de WORD a DWORD el XMM12 copiandolo previamente en otros dos XMM. Luego convertimos todo de DWORD a FLOAT y realizamos la multiplicacion con XMM11. Volvemos a convertir a DWORD y a empaquetar a WORD y guardamos el resultado en XMM12.
Guardamos en XMM13 una mascara con 128 en cada word y luego hacemos AND con XMM6. Sumamos XMM13, con XMM6.
Shufteamos en XMM13 con una MASK_1 que contiene el orden de los valores para el caso 1.
Luego, XMM10 lo usamos para saturar en 1 los casos que ya son chequeados para así no pisarlos con los otros casos. Y luego invertirmos XMM11 donde guardamos XMM15 para hacer un AND con XMM0 y dejar solo habilitados los packs que no cumplieron
Usamos una mascara para que nos acomode todo de 1word a 3 bytes, hacemos otro AND con los valores TRUE y lo guardamos en XMM1.

Por finalizado, hacemos or entre los XMM1, 2, 3, 4 y 5, y movemos el valor final de XMM1 a XMM0. Comparamos si estamos chequeando el borde haciendo $cmp\ R15D, 15 jl.terminoBorde$ en caso de no ser menor movemos los packs del XMM0 al puntero en RDI $movdqu\ [RSI], XMM0$, le restamos a R15D y a R10d, la cantidad de pixeles por ciclo y le sumamos a RDI y RSI la cantidad de pixeles por ciclo.

En caso de que nuestra comparacion diera que es menor a 15, procedemos a:

$movdqu[RSI - pixels\_por\_ciclo + R12],\ XMM0$
$addR11, R8$
$addR13, R8$
$movRDI, R11$
$movR11, RDI$
$movRSI, R13$
$movR13, RSI$
$subR10D, R15D$
$movR15D, R14D$

Como habiamos enuciado inicialmente, en R11 y R13 nos habiamos guardado RDI y RSI respectivamente y en R8 tenemos por parametro src_row_size. De esta forma hacemos que en dst se guarden los ultimos pixeles de la linea y baje a la siguiente sin tocar padding.

Esto, como fue explicado, se recorre, hasta que R10D sea igual a 0.

2.3.4 Experimento 1

Analizar cuales son las diferencias de performace entre las veRSIones de C y ASM. Realizar gráficos que representen estas diferencias.

Realizando nuestras mediciones en la implementacion de C y en ASM obtenemos los siguientes resultados sobre la cantidad de ciclos insumidos por llamada:

120x56
En ASM: 234468
En C: 783041

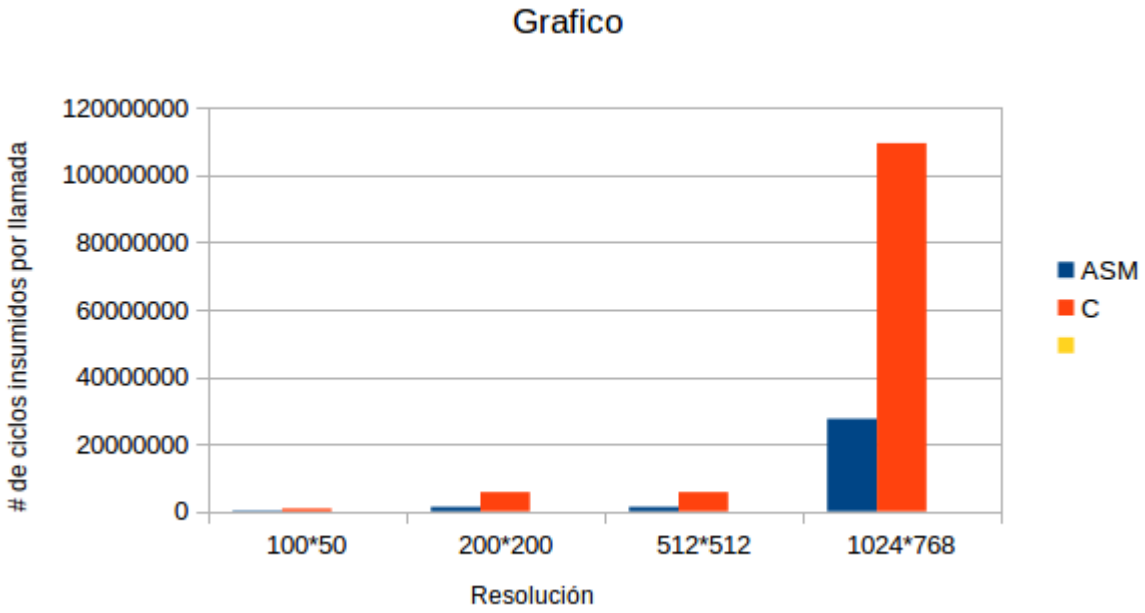
200x200
En ASM: 1289310
En C: 5708085

512x512
En ASM: 1289310

En C: 5708085

1023x767
En ASM: 27479906
En C: 109464584

Para una mejor representación se realizaron distintos graficos mostrando a simple vista las diferencias entre cada tipo de implementacion en cada tipo de imagen.



La diferencia principal entre el código escrito en lenguaje C y el código ASM, es la cantidad de accesos a memoria realizados en cada iteración. Otra diferencia puede verse en la cantidad de bytes que pueden ser procesados simultaneamente durante el mismo ciclo. Mientras que en C leemos y procesamos un byte por vez, el código ASM nos permite acceder y procesar 16 bytes en simultaneos. De todas maneras, en este caso trabajamos con 12 bytes que es un numero mas útil ya que es múltiplo de 3 (cantidad de bytes por píxel) y de 4 (cantidad de bytes por cada caracter). Pudiendo levantar de a 16 bytes en memoria, podría esperarse que el código asm demorase una diesiseiava parte de la cantidad de ciclos que demora el código en C. Pero debemos tener en cuenta que en nuestro caso estamos aprovechando solo 15 de los 16 bytes que leemos en cada iteración por lo tanto es más acertado evaluar como si solo leyeramos 15 bytes. Además, hay que tener en cuenta que las instrucciones SSE pueden necesitar más ciclos que las operaciones comunes que usamos en el C, eso disminuye un poco la performance en la comparación.

2.4 Filtro LDR

2.4.1 Objetivo

El objevtivo de este filtro, es tomar una imagen y transformar el color, pero dándole un efecto de iluminación.
Para lograrlo, se toma el valor de un pixel y le agrega un porcentaje alfa de sus pixels vecinos.
Si ese porcentaje es positivo, los pixels claros, se vuelven aún más claros y los oscuros se mantienen igual.

$$dst_{(i,j)} = \begin{cases} src_{(i,j)} & \text{si } i < 2 \vee j < 2 \vee i+2 \leq tamy \vee j+2 \leq tamx \\ min(max(src_{(i,j)} + var_{(i,j)}, 0, 255)) & \text{si no} \end{cases}$$

Donde:
$var(i,j) = \frac{src(i,j)*\alpha*sumargb(i,j)}{max}$

Donde:

$sumargb_{(i,j)} = suma_r(i,j) + suma_{g(i,j)} + suma_{b(i,j)}$

$max = 5*5*255*3*255$

Y suma_r suma_g y suma_b es la suma de los pixeles vecinos desde $-2 \leq i \leq 2$ y $-2 \leq j \leq 2$

2.4.2 Desarrollo Implementacion C

Sintesis de Desarrollo

Recorremos la totalidad de la imagen fuente mediante 2 ciclos, uno para filas, y otro para columnas. Dentro de esta iteración, hacemos dos cosas: en los bordes (de dos pixels) copiamos la imagen fuente en destino tal cual. Por otra parte, procesamos la parte central que no es borde. Para procesar este centro, tomamos cada pixel y calculamos la sumargb con 2 ciclos también. Al obtener esa suma, seguimos con lo indicado en el enunciado, multiplicamos por alfa y dividimos por max.
Finalmente, para pegar el valor final en el pixel de la imagen destino, hacemos un máximo y un mínimo, para que el valor quede entre 0 y 255.

Explicación detallada de Implementación

En esta implementación, inicialmente,se crearon dos punteros de matriz uno correspondiente a src y otro a dst.
Ademas, 4 enteros llamados *sumargb*, *varr*,*varg* y *varb* inicializados en 0 y otro entero *max* inicializado en 4876875.
Luego, realizamos dos ciclos, uno añidado dentro del otro recorriendo de la forma $'(int\ i = 0;\ i < filas;\ i++)'$ y $(int\ j = 0;\ j < cols;\ j\ ++)$.
Dentro del segundo ciclo, creamos dos punteros rgb_t donde uno apuntará a
rgb_t $*p\_d = (rgb\_t*)\ \&dst\_matrix[i][j*3]$, y el otro a $rgb\_t * p\_s = (rgb\_t*)\ \&src\_matrix[i][j*3]$.
Posteriormente, chequeamos si nos encontramos recorriendo los bordes de la imagen
$(i < 2||j < 2||(i+2) >= filas||(j+2) >= cols)$, en caso verdadero, guardamos el puntero p_s en p_d $*p_d = *p_s$.
En caso negativo, realizamos lo siguiente:
Primero guardamos en *sumargb* el valor 0, luego, crearemos dos ciclos, uno añidado del otro recorriendo de la siguiente forma:
$(int\ i\_p = i-2;\ i\_p <= i+2;\ ++i\_p)$ $(int\ j\_p = j-2;\ j\_p <= j+2;\ ++j\_p)$, dentro del ciclo interno, creamos un rgb_t $rgb\_t * s\_s = (rgb\_t*)\ \&src\_matrix[i\_p][j\_p*3]$ y posteriormente al entero *sumargb* le almacenamos la suma entre los valores r, g y b del rgb_t s_s.
Fuera de estos dos ciclos, y aun dentro de la parte falsa del if, realizamos las siguientes operaciones:
Primero, al entero *varb* le almacenamos el valor b del rgb_t p_s multiplicado por $alfa$ (dado como parametro) y por *sumargb* $varb = (p\_s \rightarrow b * alfa * sumargb)$;.

Segundo, al entero $varg$ le almacenamos el valor g del rgb_t p_s multiplicado por $alfa$ (dado como parametro) y por $sumargb$ $varg = (p\_s \rightarrow g * alfa * sumargb)$;.
Tercero, al entero $varr$ le almacenamos el valor r del rgb_t p_s multiplicado por $alfa$ (dado como parametro) y por $sumargb$ $varr = (p\_s \rightarrow r * alfa * sumargb)$;.
Luego, a cada uno de estos tres enteros los dividimos por max:
$varb = varb/max; varg = varg/max; varr = varr/max$;
Y, por ultimo, al valor b del rgb_t p_d le almacenamos $MIN(MAX(p\_s \rightarrow b + varb, 0), 255)$ donde MIN y MAX son funciones dadas por la catedra,
Al valor g del rgb_t p_d le almacenamos $MIN(MAX(p\_s \rightarrow g + varg, 0), 255)$,
Al valor r del rgb_t p_d le almacenamos $MIN(MAX(p\_s \rightarrow r + varr, 0), 255)$.
Damos por finalizada la parte falsa del if.

Estos dos ciclos realizaran $j\_d = cols - 1$ por $i\_d = filas - 1$ de itereaciones.
Con lo mencionado, obtenemos el filtro temperature en c correctamente, con los valores que nos pasan como parámetros.

2.4.3 Desarrollo Implementacion en ASM

En esta implementacion, a diferencia de en Popart y Temperature, no hubieron tantos casos por chequear, sino que trabajamos los bordes por un lado, y la parte interna por el otro.
Recorrimos, por columna de la siguiente forma:

$mov\ R12D, 0$
$mov\ R13D, 0$
$mov\ R14D, 0$
$mov\ EBX, 0$
$.recorrido\_fila:$
$cmp\ R13D, EDX$
$je\ .fin$

$.recorrido\_columna:$
$cmp\ R12D, ECX$
$je\ .siguiente\_columna$

; me fijo si es borde o interior
$cmp\ R12D, 2$
$jl\ .es\_borde$
$cmp\ R13D, 2$
$jl\ .es\_borde$
$mov\ EAX, R12D$
$add\ EAX, 2$
$cmp\ EAX, ECX$
$jge\ .es\_borde$
$mov\ EAX, R13D$
$add\ EAX, 2$
$cmp\ EAX, EDX$
$jge\ .es\_borde$

Inicializamos varios registros en 0 y comparamos si es Menor a 2 la fila y/o la columna, y en caso de ser asi, chequeamos los bordes así:
$.es\_borde:; copiolamismaimagen$
$xor\ RAX, RAX$
$mov\ EAX, EBX$
$add\ EAX, R14D$
$mov\ R15, [RDI + RAX]$
$mov\ [RSI + RAX], R15D$

$.seguir$:
$add\ R12D, 1$
$add\ R14D, R8D$; R14D = R14D + src_row_size
$jmp\ .recorrido_columna$

Estas iteraciones se realizaran hasta que R12D y R13D sean mayores a 2.
Una vez que esto ocurre pasamos a sumar cada color haciendo:

$movdqu\ XMM7, XMM0$
$movdqu\ XMM8, XMM0$
$xorpd\ XMM9, XMM9$

$pshufb\ XMM0, [MASK\_1\_COLOR]$
$movdqu\ XMM10, XMM0$
$punpcklbw\ XMM10, XMM9$
$pshufb\ XMM7, [MASK\_1\_COLORGREEN]$
$movdqu\ XMM12, XMM7$
$punpcklbw\ XMM12, XMM9$
$pshufb\ XMM8, [MASK\_1\_COLORBLUE]$
$movdqu\ XMM14, XMM8$
$punpcklbw\ XMM14, XMM9$
$paddw\ XMM10, XMM12$
$paddw\ XMM10, XMM14$
$movdqu\ XMM0, XMM10$

Primero, nos guardamos en XMM0 los primeros 15 pixeles haciendo $movdqu\ XMM0, [RDI]$, luego, el valor de XMM0, lo guardamos en XMM7 y XMM8, limpiamos XMM9 y con 3 mascaras distintas que lo que hacen es ponernos los 5 pixeles rojos adelante y llenar con 0, los 5 pixeles azules y llenar con cero y los 5 pixeles verdes y llenar con ceros.
Realizamos la operacion Pshufb con los tres xmm y las mascaras (un xmm con cada mascara) y luego desempaquetamos de byte a word, usando el XMM9 que habiamos llenado de 0.
El desempaquetado hace que nos queden los valores en word en los registros XMM10, XMM12, XMM14 respectivamente, y luego los sumamos con PADDW guardando el valor en XMM10. De esta forma obtenemos en word $|b0 + g0 + r0|b1 + g1 + r1|b2 + g2 + r2|b3 + g3 + r3|b4 + g4 + r4|$.
Esto lo realizamos 5 veces, quedandonos con los valores en XMM1, 2 ,3,4 y 5. Luego, sumamos estos XMM y nos queda el valor en XMM3. Proximo a esto, procedemos a sumar los vecinos:

$movdqu\ XMM8,\ XMM3$
$movdqu\ XMM9,\ XMM3$
$psrldqXMM9, 8D$
$paddw\ XMM3,\ XMM9$

$movdqu\ XMM9,\ XMM8$
$psrldqXMM9, 6D$
$paddw\ XMM3,\ XMM9$
$movdqu\ XMM9,\ XMM8$
$psrldqXMM9, 4D$
$paddw\ XMM3,\ XMM9$

$movdqu\ XMM9,\ XMM8$
$psrldq\ XMM9, 2D$
$paddw\ XMM3,\ XMM9$
$movdqu\ XMM9,\ XMM3$
$psrldq\ XMM9, 2D$
$pslldq\ XMM9, 2D$
$psubw\ XMM3,\ XMM9$

En esta seccion vamos moviendo bytes a la izquierda, 8 6 4 y 2 asi logramos sumar todos los vecinos

en los packs.
Guardamos en R15D el valor alfa, que nos viene por pila en $[RSP + 56]$. Y lo guardamos en XMM15
Procedemos a la multiplicación por alfa:

$movdqu\ XMM7,\ XMM3$
$xorpd\ XMM9,\ XMM9$
$punpcklwd\ XMM7,\ XMM9$
$cvtdq2pd\ XMM7,\ XMM7$
$cvtdq2pd\ XMM15,\ XMM15$
$mulpdXMM7,\ XMM15$
$cvtpd2Dq\ XMM7, XMM7$
$movdqu\ XMM3,\ XMM7$

En esta seccion desempaquetamos de WORD a DWORD y convertimos a Double para multiplicar con alfa, luego lo convertimos a DWORD. Guardamos en XMMO el valor de XMM6 y nos quedamos con el del medio, por consiguiente pasamos a multiplicar lo que tenemos con SUMARGB y dividimos por MAX:

$movdquXMM7, XMM0$
$movdquXMM8, XMM0$

$cvtdq2pd\ XMM3,\ XMM3$

$xorpd\ XMM9, XMM9$
;multiplico blue
$pshufb\ XMM0, [MASK\_1\_COLOR]$
$movdqu\ XMM10, XMM0$
$punpcklbw\ XMM10, XMM9$
$movdqu\ XMM13, XMM10$
$xorpd\ XMM9,\ XMM9$
$punpcklwd\ XMM13,\ XMM9$
$cvtdq2pd\ XMM13,\ XMM13$
$mulpd\ XMM13,\ XMM3$
$movdqu\ XMM1, XMM13$

;multiplico verde
$pshufb\ XMM7, [MASK\_1\_COLORGREEN]$
$movdqu\ XMM11, XMM7$
$punpcklbw\ XMM11, XMM9$

$movdqu\ XMM13, XMM11$
$xorpd\ XMM9,\ XMM9$
$punpcklwd\ XMM13,\ XMM9$
$cvtdq2pd\ XMM13,\ XMM13$

$mulpd\ XMM13,\ XMM3$

$movdquXMM5, XMM13$

;multiplico red
$pshufb\ XMM8, [MASK\_1\_COLORBLUE]$
$movdqu\ XMM12, XMM8$
$punpcklbw\ XMM12, XMM9$
$movdqu\ XMM13, XMM12$

$xorpd\ XMM9,\ XMM9$
$punpcklwd\ XMM13,\ XMM9$
$cvtdq2pd\ XMM13,\ XMM13$
$mulpd\ XMM13,\ XMM3$

$movdqu\ XMM7, XMM13$

En esta parte procedemos a realizar la multiplicacion de cada color por separado con lo que ya teniamos. Desempaquetando, convirtiendo a double y por finalizado multiplicando quedandonos en el XMM1, 5 Y 7 los valores obtenidos.
Luego, realizamos la division por max:

$movdqu\ XMM11, [MASK\_MAX]$
$cvtdq2pd\ XMM11, XMM11$
$divpd\ XMM1, XMM11$
$cvttpd2Dq\ XMM1, XMM1$
$packssdw\ XMM1, XMM1$
$;GREEN$
$divpd\ XMM5, XMM11$
$cvttpd2Dq\ XMM5, XMM5$
$packssdw\ XMM5,\ XMM5$
$;rojo$
$divpd\ XMM7, XMM11$
$cvttpd2Dq\ XMM7, XMM7$
$packssdw\ XMM7,\ XMM7$

$movdqu\ XMM9,\ XMM1$
$psrldq\ XMM9, 2D$
$pslldq\ XMM9, 2D$
$psubw\ XMM1,\ XMM9; restoparaqnomequedebasura$
$movdqu\ XMM9,\ XMM5$
$psrldq\ XMM9, 2D$
$pslldq\ XMM9, 2D$
$psubw\ XMM5,\ XMM9; restoparaqnomequedebasura$
$movdqu\ XMM9,\ XMM7$
$psrldq\ XMM9, 2D$
$pslldq\ XMM9, 2D$
$psubw\ XMM7,\ XMM9; restoparaqnomequedebasura$
$;en\ XMM1 tengoblue$
$;en\ XMM5 tengoGREEN$
$;en\ XMM7 tengorojo$
$;en\ XMM6 mequedeconelactual$
$movdqu\ XMM15,\ XMM6$
$pshufb\ XMM15, [MEQUEDOCONELDELMEDIO]$
$movdqu\ XMM8,\ XMM15$
$movdqu\ XMM9,\ XMM15$
$movdqu\ XMM10,\ XMM15$
$pshufb\ XMM8, [MASKCOLORROJO]$
$paddw\ XMM8,\ XMM1$
$pshufb\ XMM9, [MASKCOLORVERDE]$
$paddw\ XMM9,\ XMM5$
$pshufb\ XMM10, [MASKCOLORAZUL]$
$paddw\ XMM10,\ XMM7$

En esta seccion realizamos la division por max, a diferencia de las otras conversiones, en esta convertimos truncado a double para no perder presicion en la division. Luego de dividir realizamos distintos shifteos, y proximo a esto, shufteos con mascaras para ir quedandonos definitivamente con cada color final y sumandolos entre si para proceder a chequear maximos y minimos.
El resultado final lo tenemos en XMM8,9 y 10.

Pasamos a chequear dichos valores:
Primero, chequeando si es mayor a cero:

$movdqu\ XMM14, [M0]$
$movdqu\ XMM15, XMM0$
$pcmpgtw\ XMM15, XMM14$

Donde, M0 es una mascara con todos 0.

Y procedemos a:

$movdqu\ XMM13,\ XMM0$
$movdqu\ XMM11,\ XMM15;$
$pcmpeqw\ XMM14, XMM14$
$pxor\ XMM11,\ XMM14$
$movdqu\ XMM2, [M0]$
$pand\ XMM2, XMM11$
$pand\ XMM13,\ XMM15$
$movdqu\ XMM1,\ XMM13$

Esto, similar a nuestros chequeos en las implementaciones anteriores de Popart y Temperature, guardamos el valor de la comparación, realizamos and con los valores chequeados, invertimos el resultado de la comparacion y lo guardamos para que esos casos no vuelvan a ser chequeados. Luego, nuestro resultado nos queda en XMM1.

Segundo, si el numero es menor a 255:

$orpd\ XMM1, XMM2$
$movdqu\ XMM0,\ XMM1$
$movdqu\ XMM14, [M255]$
$movdqu\ XMM15, XMM0$
$pcmpgtwXMM14, XMM15$

Donde, M255 es una mascara con todos 255.

Procedemos a:

$movdquXMM13,\ XMM0$
$movdqu\ XMM11,\ XMM14$
$pcmpeqw\ XMM15, XMM15$
$pxor\ XMM11,\ XMM15$
$movdqu\ XMM4, [M255]$
$pand\ XMM4, XMM11$
$pand\ XMM13,\ XMM14$
$movdqu\ XMM3,\ XMM13$

Esta seccion, como mencionamos antes, similar a las implementaciones anteriores, guardamos el valor de la comparación, realizamos and con los valores chequeados, invertimos el resultado de la comparacion y lo guardamos para que esos casos no vuelvan a ser chequeados. Luego, nuestro resultado nos queda en XMM3.

Por ultimo, en nuestra implementacion, juntamos los resultados, chequeando con un contador si pertenece a verde rojo o todo:

$add\ r15, 1$
$orps\ XMM3, XMM4$
$movdqu\ XMM0, XMM3$
$cmp\ r15, 1$
$je\ .verdes$
$cmp\ r15, 2$
$je\ .rojos$

```
        cmp r15, 3
        je .juntoTodo

.verdes:
        movdqu XMM8, XMM0
        movdqu XMM0, XMM9
        jmp .chequeo
        .rojos:
        movdqu XMM9, XMM0
        movdqu XMM0, XMM10
        jmp .chequeo
        .juntoTodo:
        movdqu XMM10, XMM0
        pslldq XMM9, 1d
        pslldq XMM10, 2D
        paddb XMM8, XMM9
        paddb XMM8, XMM10
        movdqu XMM0, XMM8
```

En esta seccion, como mencionamos, realizamos la suma de todos los xmm de los distintos colores. Una ves terminado esto, guardamos en la imagen resultante los valores:

```
xor RAX, RAX
mov EAX, EBX
add EAX, R14D
xor R15, R15
movd R15D, XMM0
mov [RSI + RAX], R15D
```

Esto, finalmente, se realizará hasta que no queden columnas y filas por verse.

2.4.4 Experimento 1

Analizar cuales son las diferencias de performace entre las versiones de C y ASM. Realizar gráficos que representen estas diferencias.

En este filtro a diferencia de los anteriores existe un alfa el cual da valores positivos y negativos, por ende realizamos mediciones con alfa positivo y negativo.

Realizando nuestras mediciones en la implementacion de C y de ASM de nuestro codigos con alfa positivo obtenemos para disversas resoluciones:

120x56
En ASM: 1460376
En C:3431854

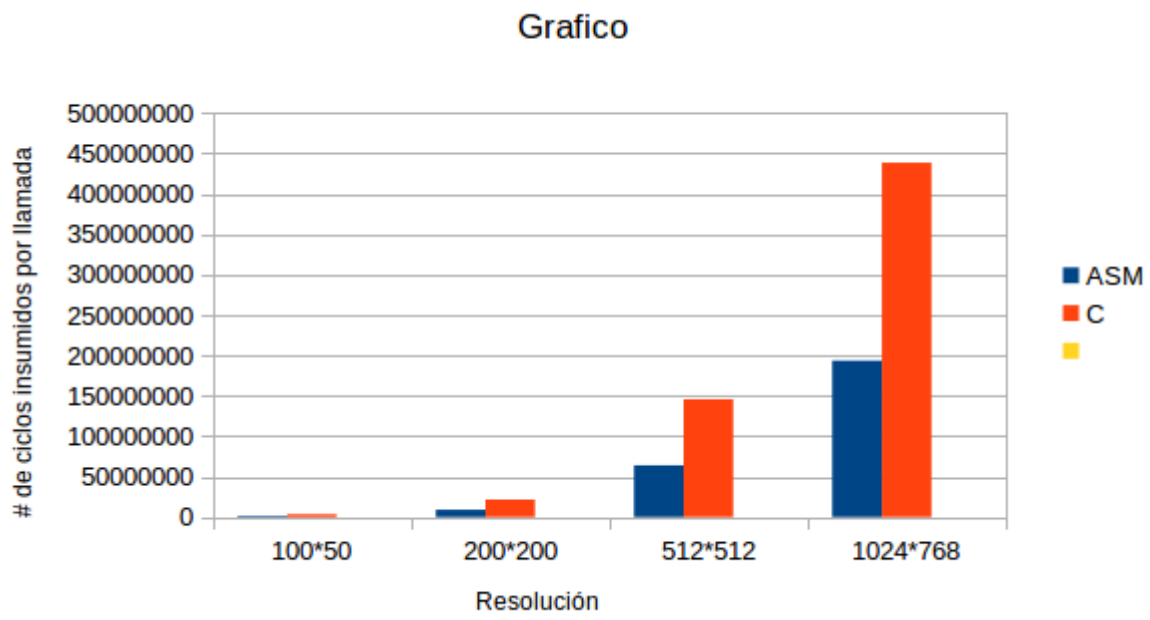
200x200
En ASM: 9112044
En C:21743294

512x512
En ASM: 64050524
En C:145683856

1023x767
En ASM :193531520
En C:438512256

Para una mejor representación se realizo un grafico mostrando a simple vista las diferencias entre

cada tipo de implementacion con dicho alfa.



Por consiguiente, con un valor alfa negativo realizamos las pertinentes mediciones en las distintas implementaciones obteniendo que:

120x56

En ASM: 1438545

En C:3450531

200x200

En ASM: 9026611

En C:21783128

512x512

En ASM: 63927320

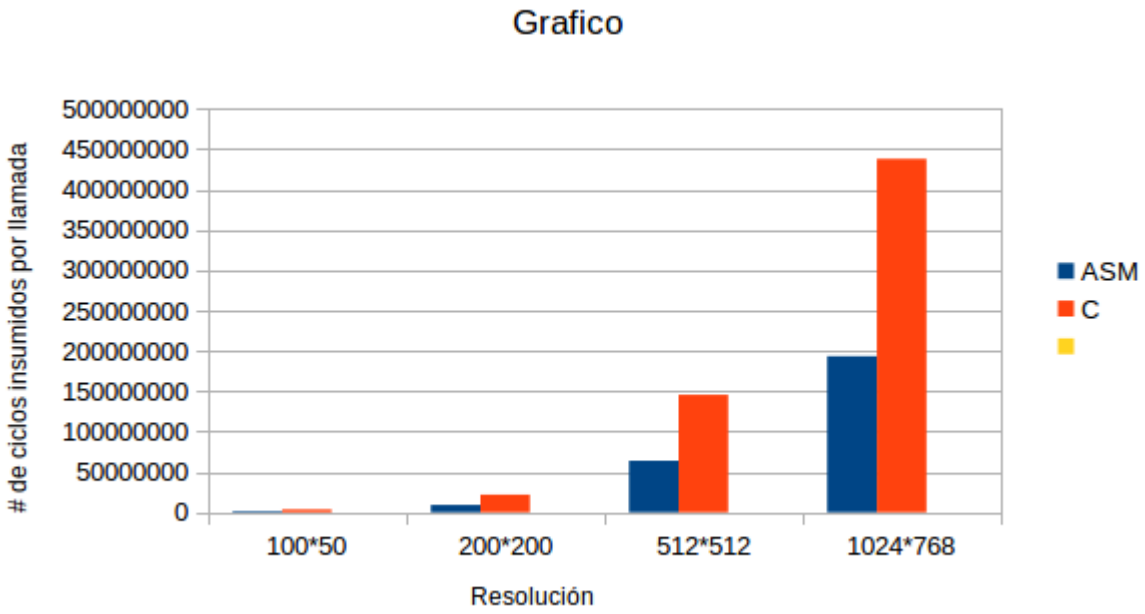
En C:145574048

1023x767

En ASM: 193388816

En C:437996160

Para una mejor representación se realizo un grafico mostrando a simple vista las diferencias entre cada tipo de implementacion con dicho alfa.



La diferencia principal entre el código escrito en lenguaje C y el código ASM, es la cantidad de accesos a memoria realizados en cada iteración.Otra diferencia puede verse en la cantidad de bytes que pueden ser procesados simultaneamente durante el mismo ciclo. En nuestra implamentacion de ASM en una sola iteracion ya obtenemos la sumas correspondientes de los pixeles vecinos, mientras que en la implamentacion del c por cada iteracion se realizan dentro de esa misma varias iteraciones mas sumando los pixeles vecinos y por este motivo principal la diferencia notoria en la performance.

Pudiendo levantar de a 16 bytes en memoria, podría esperarse que el código asm demorase una diesiseiava parte de la cantidad de ciclos que demora el código en C. Pero debemos tener en cuenta que en nuestro caso estamos aprovechando solo 15 de los 16 bytes que leemos en cada iteración por lo tanto es más acertado evaluar como si solo leyeramos 15 bytes. Además, hay que tener en cuenta que las instrucciones SSE pueden necesitar más ciclos que las operaciones comunes que usamos en el C, eso disminuye un poco la performance en la comparación.

3 Conclusión

Las instrucciones SIMD (Single Instruction Multiple Data) proveen al programador de una herramienta más efectiva para realizar el mismo conjunto de operaciones a una gran cantidad de datos.

La aplicación de filtros a imágenes era un ejemplo perfecto para probar su eficiencia.

Analizando los resultados de las implementaciones de los 4 filtros, podemos notar:

- Las operaciones básicas (padd, psub, pmul, pdiv, shifts, etc.) SIMD tienen un costo similar a sus correspondientes operaciones unitarias, pero generalmente requieren algún tipo de pre-proceso para poder trabajar con los 16 bytes (pack, unpack, shifts) en una sola iteración, por lo tanto, aunque más eficientes, no lo son en una relación directamente proporcional.
- En el caso que sí hay una relación directamente proporcional es en el acceso a memoria.
- Además, el acceso a memoria es, por lejos, la operación más costosa de las que implementamos en cada filtro.
- Por consecuencia directa del ítem anterior, las llamadas a otras funciones (que a su vez, probablemente contengan variable locales) dentro de una iteración provocan estragos en la efectividad de las implementaciones en C.
- Para poder aprovechar las instrucciones SIMD es un prerequisito que los datos estén contiguos en memoria.

Concluimos que, definitivamente, las instrucciones SIMD, cuando pueden aprovecharse, demuestran una gran eficiencia. Sin embargo, hay que tener algunas consideraciones:

Aunque las imágenes, video y sonido son los primeros candidatos a ser optimizados por paralelización, no todos los procesos pueden ser efectivos y se requiere un análisis profundo de los datos para ver si vale el esfuerzo.

Además, aunque se pueda lograr una gran optimización, no siempre es lo más importante. La optimización seguramente es indispensable en transmisiones de video en vivo, pero baja en importancia si tuviese que ser aplicado una sola vez en una aplicación tipo MS Paint.

Las desventajas que podrían opacar a la optimización son:

El código no es portable, únicamente funciona en procesadores que implementan el set de instrucciones AMD64, requiriendo reescrituras para otras plataformas. Sin embargo el código C debería funcionar perfectamente en IA-32, ARM y cualquier otro procesador que tenga un compilador de lenguaje C.

El código es mucho más largo y difícil de entender (por lo tanto mayor posibilidad de tener bugs) que en un lenguaje de más alto nivel como C. Y en pos de la optimización, se llegan a eliminar funciones (poniéndolas inline), lo que genera código repetido, largo y confuso.