



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA



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

<b>1</b>	<b>Introducción</b>	<b>3</b>
<b>2</b>	<b>Desarrollo y Resultados</b>	<b>4</b>
2.1	Filtro Tiles . . . . .	4
2.1.1	Desarrollo Implementacion C . . . . .	4
2.1.2	Desarrollo Implementacion en ASM . . . . .	4
2.1.3	Experimento 1 - análisis el código generado . . . . .	4
2.1.4	Experimento 2 - optimizaciones del compilador . . . . .	4
2.1.5	Experimento 3 - secuencial vs. vectorial . . . . .	5
2.1.6	Experimento 4 - cpu vs. bus de memoria . . . . .	5
2.2	Filtro Popart . . . . .	7
2.2.1	Desarrollo Implementacion C . . . . .	7
2.2.2	Desarrollo Implementacion en ASM . . . . .	7
2.2.3	Experimento 1 - saltos condicionales . . . . .	10
2.2.4	Experimento 2 - cpu vs. bus de memoria . . . . .	11
2.2.5	Experimento 3 - prefetch . . . . .	12
2.2.6	Experimento 4 - secuencial vs. vectorial . . . . .	14
2.3	Filtro Temperature . . . . .	16
2.3.1	Desarrollo Implementacion C . . . . .	16
2.3.2	Desarrollo Implementacion en ASM . . . . .	16
2.3.3	Experimento 1 . . . . .	22
2.4	Filtro LDR . . . . .	24
2.4.1	Desarrollo Implementacion C . . . . .	24
2.4.2	Desarrollo Implementacion en ASM . . . . .	24
2.4.3	Experimento 1 . . . . .	29
<b>3</b>	<b>Conclusión</b>	<b>32</b>

# 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<sup>1</sup> 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.

---

<sup>1</sup>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 Desarrollo Implementacion C

En esta implementación, lo que se realizó primero fue crear dos punteros de matriz uno correspondiente a src y otro a dst.

Luego, realizamos dos ciclos, uno anidado 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 parámetro.

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 realizarán `i_d = filas - 1` de iteraciones y el anidado `j_d = cols - 1` iteraciones.

Con lo mencionado, obtenemos el filtro tiles en `c` correctamente, con los valores que nos pasan como parámetros.

#### 2.1.2 Desarrollo Implementacion en ASM

#### 2.1.3 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 instrucciones 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.

#### 2.1.4 Experimento 2 - optimizaciones del compilador

Compile el código de C con optimizaciones del compilador, por ejemplo, pasando el flag `-O1`<sup>2</sup>. ¿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 ensamblado sin optimización. En particular al mirar el `objdump` de código con `-O1` se pudo apreciar que

<sup>2</sup>agregando este flag a `CCFLAGS64` en el `makefile`

el código era mucho menor en cantidad de líneas 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 optimizaciones de -O2 más algunas extras con el fin de hacer aun más optimo el código.<sup>3</sup>

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.

### 2.1.5 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 quiere mostrar.

### 2.1.6 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.

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

---

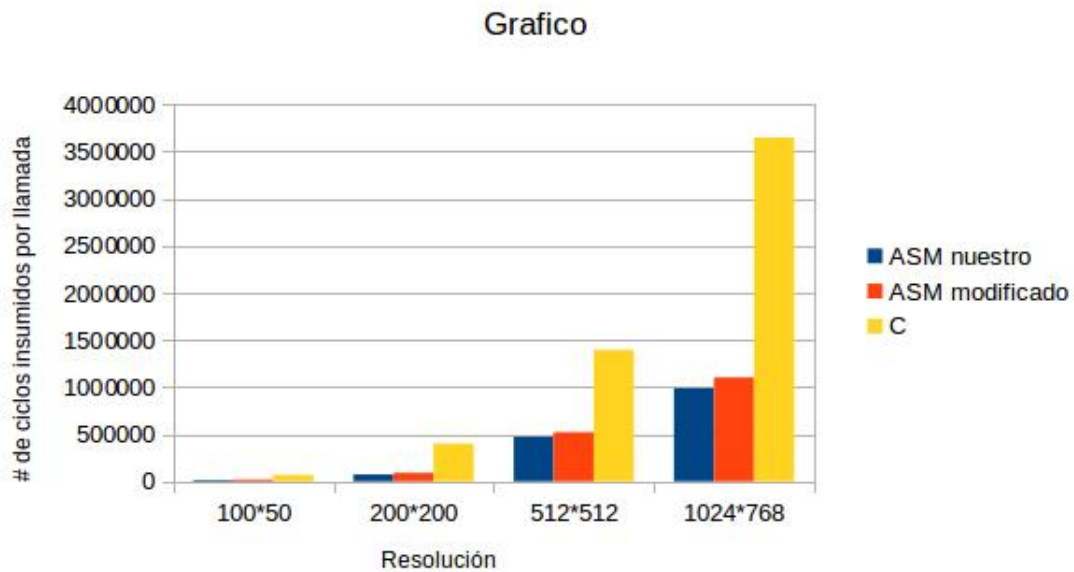
<sup>3</sup>Para mas información revisar el <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>

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 procesamiento en el ciclo principal del algoritmo

## 2.2 Filtro Popart

### 2.2.1 Desarrollo Implementacion C

En esta implementación, inicialmente, se crearon dos punteros de matriz uno correspondiente a src y otro a dst.

Además, un entero llamado *suma* inicializado en 0.

Luego, realizamos dos ciclos, uno anidado 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 → r + p_s → b + p_s → g`

Posteriormente, chequeamos el valor correspondiente de suma:

Primero si `suma < 153`, en caso verdadero, guardamos en el puntero *p\_d* la posición 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 posición 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 posición 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 posición 3 de la matriz colores `*p_d = colores[3]`.

Por último, si `(612 <= suma)`, en caso verdadero, guardamos en el puntero *p\_d* la posición 4 de la matriz colores `*p_d = colores[4]`.

Esta matriz colores mencionada se encuentra definida fuera de la función como:

```
rgb_t colores[] = { {255, 0, 0},
                   {127, 0, 127}
                   {255, 0, 255}
                   {0, 0, 255}
                   {0, 255, 255} }
```

Estos dos ciclos realizarán `j_d = cols - 1` por `i_d = filas - 1` de iteraciones.

Con lo mencionado, obtenemos el filtro popart en C correctamente, con los valores que nos pasan como parámetros.

### 2.2.2 Desarrollo Implementacion en ASM

En esta implementación, 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 operación MUL R10, luego realizamos lo mismo con R12 y así, obtenemos en R10 el valor completo de la cantidad de píxeles a recorrer.

Guardamos en R11 el valor de RDI donde venía *\*src* y en R13 el valor de RSI donde venía *\*dst*.

Luego, realizamos la sección donde chequeamos si estamos en el final de la línea 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 píxel que vamos a recorrer en una iteración. Como vemos de a 15 píxeles 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 píxeles de la siguiente forma:

Primero, nos guardamos en XMM0 los primeros 15 píxeles haciendo `movdqu XMM0, [RDI]`, luego, el valor de XMM0, lo guardamos en XMM1 y XMM2, limpiamos XMM7 y con 3 máscaras distintas que lo que hacen es ponernos los 5 píxeles rojos adelante y llenar con 0, los 5 píxeles azules y llenar con cero y los 5 píxeles verdes y llenar con ceros.

Realizamos la operación Pshufb con los tres xmm y las máscaras (un xmm con cada máscara) y luego desempaquetamos de byte a word, usando el xmm7 que habíamos 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]
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
```

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
```





```

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, 15jl.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.

### 2.2.3 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.

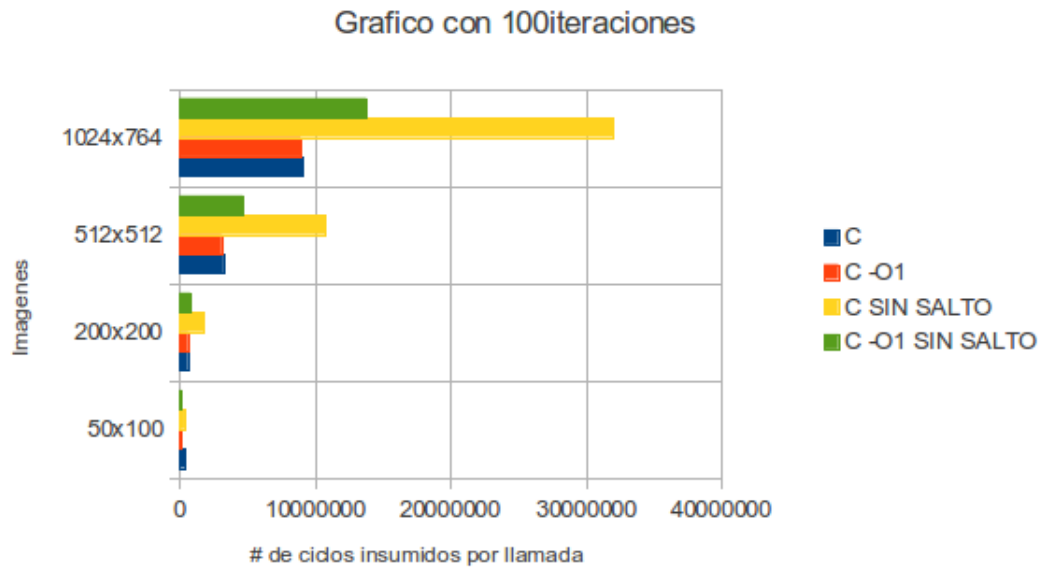
Si se le ocurren, mencionar otras posibles formas de medir el impacto de los saltos condicionales.

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 im-

plementación 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.

### 2.2.4 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 instancia 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

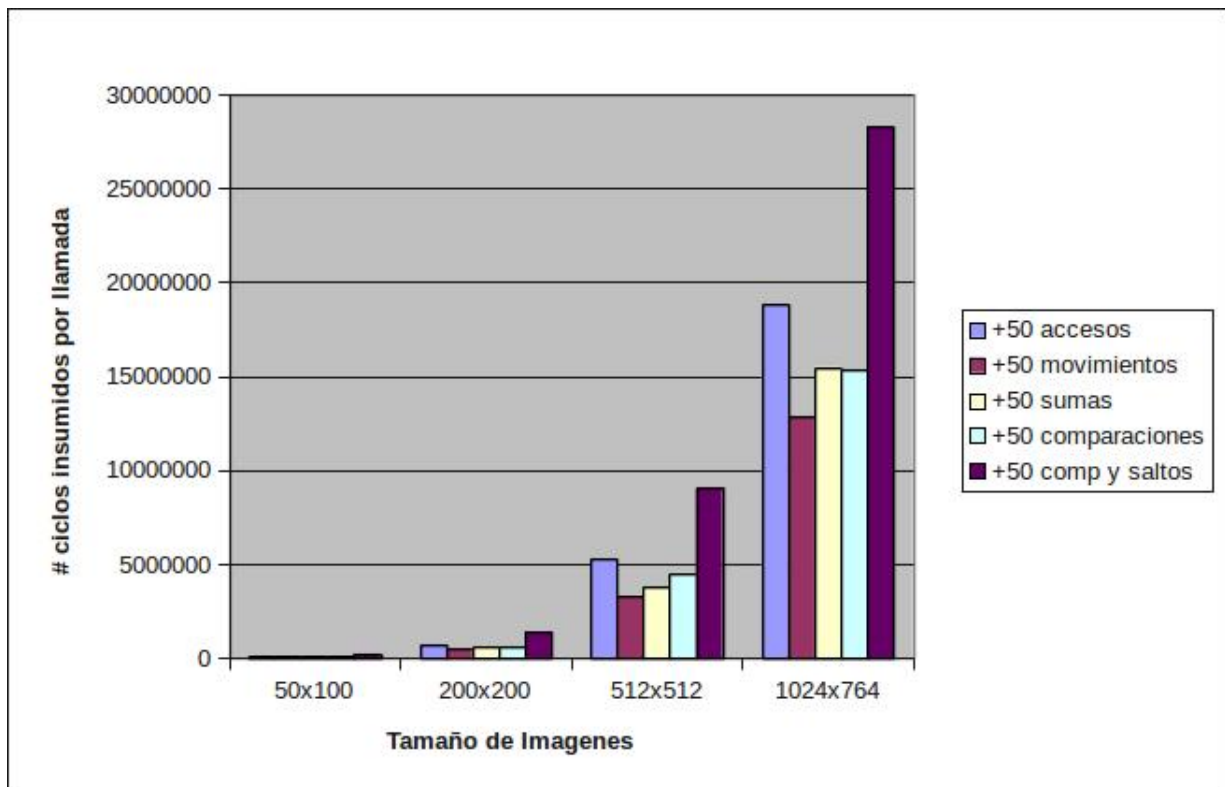
+50 movimientos : 12843421.000

+50 sumas : 15499867.000

+50 comparaciones : 15349734.000

+50 comp y saltos : 28297848.000

En el siguiente gráfico podemos distinguir los resultados.



### 2.2.5 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 estarán 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 prefetch 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 resoluciones utilizando prefetch y sin utilizarlo.

120x56

sin prefetch: 67836.359

con prefetch: 65300.320

200x200

sin prefetch: 398975.594

con prefetch: 382423.562

512x512

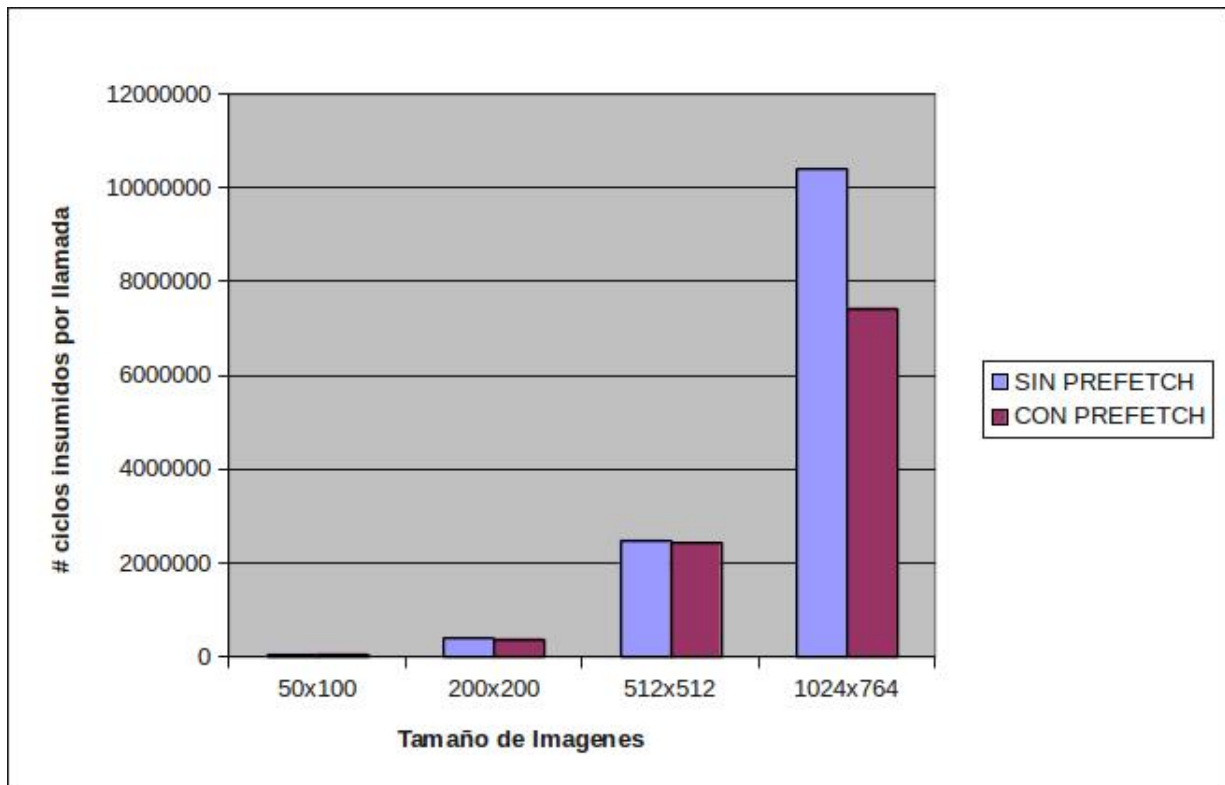
sin prefetch: 2498557.750

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.

#### 2.2.6 Experimento 4 - secuencial vs. vectorial

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

Realizando nuestras mediciones en la implementación 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

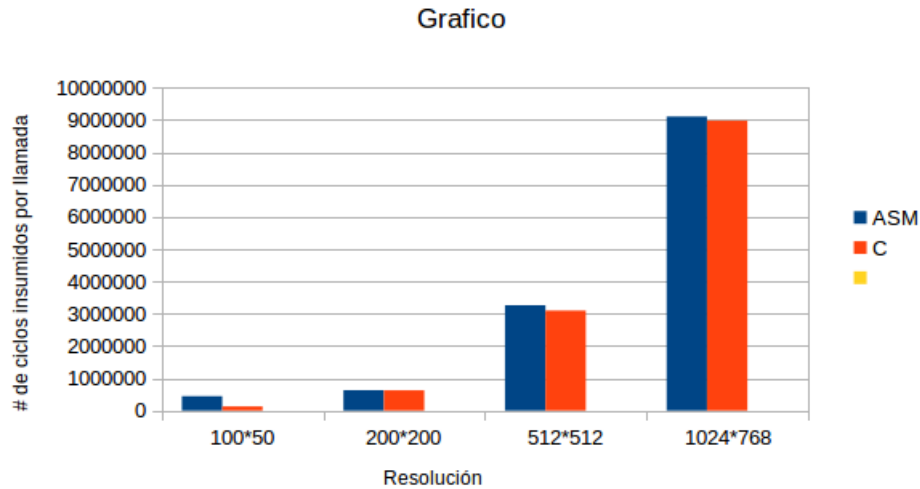
1023x767

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 simultáneamente 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 simultáneos. De todas maneras, en este caso trabajamos con 12 bytes que es un número más útil ya que es múltiplo de 3 (cantidad de bytes por píxel) y de 4 (cantidad de bytes por cada carácter). Pudiendo levantar de a 16 bytes en memoria, podría esperarse que el código asm demorase una dieciseisava 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 más que como hemos visto los saltos condicionales empeoran la performance en ASM, dicha implementación sigue siendo más óptima que la de C.

## 2.3 Filtro Temperature

### 2.3.1 Desarrollo Implementacion C

En esta implementación, inicialmente, se crearon dos punteros de matriz uno correspondiente a src y otro a dst.

Además, un entero llamado *suma* y otro *divido* inicializados en 0.

Luego, realizamos dos ciclos, uno anidado 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 función *between* provista por la catedral:

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 último, 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 realizarán  $j\_d = cols - 1$  por  $i\_d = filas - 1$  de iteraciones.

Con lo mencionado, obtenemos el filtro temperature en c correctamente, con los valores que nos pasan como parámetros.

### 2.3.2 Desarrollo Implementacion en ASM

En esta implementación, 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 operación 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 venía *\*src* y en R13 el valor de RSI donde venía *\*dst*.

Luego, realizamos la sección donde chequeamos si estamos en el final de la línea 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 iteración. 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 máscaras 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 operación *Pshufb* con los tres xmm y las máscaras (un xmm con cada máscara) y luego desempaquetamos de byte a word, usando el xmm7 que habíamos llenado de 0.

El desempaqueado 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:

```

movdquxmm10, [DIVIDO]
    movdquxmm1, xmm0
    movdquxmm2, xmm0
    XORPDxmm15, xmm15
    punpcklwdxmm1, xmm15
    punpckhwdxmm2, xmm15
    cvtdq2psxmm1, xmm1
    cvtdq2psxmm2, xmm2
    cvtdq2psxmm10, xmm10
    divpsXMM1, xmm10
    divpsXMM2, xmm10
    CVTTPS2DQxmm1, xmm1
    CVTTPS2DQxmm2, xmm2
    PACKUSDWxmm1, xmm2
    movdquxmm0, 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
punpckhwd xmm8, xmm9
cvtdq2ps xmm7, xmm7
cvtdq2ps xmm8, xmm8
cvtdq2ps xmm11, xmm11
mulps XMM7, xmm11
mulps XMM8, xmm11
CVTTPS2DQ xmm7, xmm7
CVTTPS2DQ xmm8, xmm8
PACKUSDW xmm7, xmm8
movdqu xmm13, xmm7
movdqu xmm11, [M255]
pand xmm11, xmm15
psubb xmm11, xmm13
movdqu xmm13, xmm11
pshufb xmm13, [MASK5]
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
punpckhwd xmm8, xmm9
cvt dq2ps xmm7, xmm7
cvt dq2ps xmm8, xmm8
cvt dq2ps xmm11, xmm11
mulps XMM7, xmm11
mulps XMM8, xmm11
CVT PS2DQ xmm7, xmm7
CVT PS2DQ 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, real-

izamos 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
punpckhwd 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 invertimos 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
punpckhwd xmm8, xmm9
cvt dq2ps xmm7, xmm7
cvt dq2ps xmm8, xmm8
cvt dq2ps 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
PADDDUSW 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 invertimos 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:

```
movdqa XMM14, [MASK_31]
movdqu XMM6, XMM10; me quedo con XMM10 y trabajamos el de XMM6
pcmpeqw XMM7, XMM7; XMM7 = |1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|
pxor XMM6, XMM7; deja en 1 los que son 0 y los que son 1 chequeados
pand XMM14, XMM6; en XMM14 quedan los 153 en los lugares que tengo que comparar
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
punpckhwd xmm8, xmm9
cvtqd2ps xmm7, xmm7
cvtqd2ps xmm8, xmm8
cvtqd2ps xmm11, xmm11
mulps XMM7, xmm11
mulps XMM8, xmm11
CVTQ2DQ xmm7, xmm7
CVTQ2DQ 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 invertimos 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, 15jl.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.

### 2.3.3 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

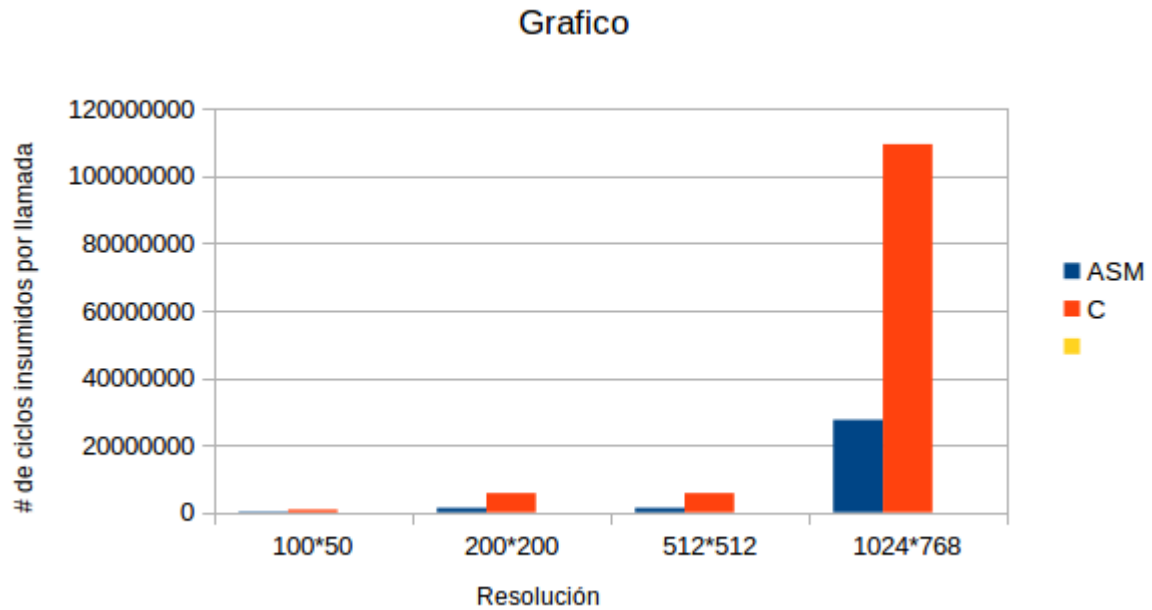
En C: 5708085

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 simultáneamente 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 simultáneos. De todas maneras, en este caso trabajamos con 12 bytes que es un número más útil ya que es múltiplo de 3 (cantidad de bytes por píxel) y de 4 (cantidad de bytes por cada carácter). Pudiendo levantar de a 16 bytes en memoria, podría esperarse que el código asm demorase una dieciseisava 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 Desarrollo Implementacion C

En esta implementación, inicialmente, se crearon dos punteros de matriz uno correspondiente a src y otro a dst.

Además, 4 enteros llamados *sumargb*, *varg*, *varg* y *varb* inicializados en 0 y otro entero *max* inicializado en 4876875.

Luego, realizamos dos ciclos, uno anidado 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 ( $i < 2 || j < 2 || (i + 2) \geq \text{filas} || (j + 2) \geq \text{cols}$ ), 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 anidado 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 → 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 → g \* alfa \* sumargb);*.

Tercero, al entero *varg* le almacenamos el valor r del *rgb\_t* *p\_s* multiplicado por *alfa* (dado como parametro) y por *sumargb* *varg = (p\_s → 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 → 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 → g + varg, 0), 255)*,

Al valor r del *rgb\_t* *p\_d* le almacenamos *MIN(MAX(p\_s → r + varr, 0), 255)*.

Damos por finalizada la parte falsa del if.

Estos dos ciclos realizarán *j\_d = cols - 1* por *i\_d = filas - 1* de iteraciones.

Con lo mencionado, obtenemos el filtro temperature en c correctamente, con los valores que nos pasan como parámetros.

### 2.4.2 Desarrollo Implementacion en ASM

En esta implementación, 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:

```
movR12D, 0; R12D = 0 contador de filas
movR13D, 0; R13D = 0 contador de columnas
movR14D, 0; R14 = offset de acceso a la imagen
movEBX, 0; EBX = contador de bytes recorridos en fila
.recorrido_fila :
cmpR13D, EDX; EDX = cols si recorri todas las columnas
je.fin

.recorrido_columna :
cmpR12D, ECX; si recorri todas las filas (en una columna)
je.siguiete_columna
```



```

; me fijosiesbordeointerior
cmpR12D, 2
jl.es_borde
cmpR13D, 2
jl.es_borde
movEAX, R12D; EAX = contadordefilas
addEAX, 2
cmpEAX, ECX; (i + 2) >= filas
jge.es_borde
movEAX, R13D; EAX = contadordecols
addEAX, 2
cmpEAX, 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 así, chequeamos los bordes así:

```

.es_borde ;; copiamos la misma imagen
xorRAX, RAX
movEAX, EBX
addEAX, R14D
movR15, [RDI + RAX]
mov[RSI + RAX], R15D

.seguir :
addR12D, 1
addR14D, R8D; R14D = R14D + src_row_size
jmp.recorrido_columna

```

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

```

movdquXMM7, XMM0; XMM1 = XMM0
movdquXMM8, XMM0; XMM2 = XMM0
XORPDXMM9, XMM9; XMM7 = 0

pshufbXMM0, [MASK_1_COLOR]
movdquXMM10, XMM0
punpcklbwXMM10, XMM9
pshufbXMM7, [MASK_1_COLORGREEN]
movdquXMM12, XMM7
punpcklbwXMM12, XMM9
pshufbXMM8, [MASK_1_COLORBLUE]
movdquXMM14, XMM8
punpcklbwXMM14, XMM9
paddwXMM10, XMM12
paddwXMM10, XMM14
movdquXMM0, XMM10

```

Primero, nos guardamos en XMM0 los primeros 15 píxeles haciendo `movdqu XMM0, [RDI]`, luego, el valor de XMM0, lo guardamos en XMM7 y XMM8, limpiamos XMM9 y con 3 máscaras distintas que lo que hacen es ponernos los 5 píxeles rojos adelante y llenar con 0, los 5 píxeles azules y llenar con cero y los 5 píxeles verdes y llenar con ceros.

Realizamos la operación `Pshufb` con los tres xmm y las máscaras (un xmm con cada máscara) y luego desempaquetamos de byte a word, usando el xmm9 que habíamos llenado de 0.

El desempaqueado 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
PSRLDQXMM9, 2d
paddw xmm3, xmm9
                                movdqu xmm9, xmm3
PSRLDQXMM9, 2d
PSLLDQXMM9, 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 XMM0 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; XMM1 = XMM0
movdquXMM8, XMM0; XMM2 = XMM0

cvtdq2pd xmm3, xmm3

XORPDXMM9, XMM9; XMM7 = 0
; multiplicoblue
pshufbXMM0, [MASK_1_COLOR]
movdquXMM10, XMM0
punpcklbwXMM10, XMM9
movdqu xmm13, XMM10
XORPD xmm9, xmm9
punpcklwd xmm13, xmm9
cvtdq2pd xmm13, xmm13
mulpd xmm13, xmm3
movdquXMM1, XMM13

```



```

; en xmm6 me quede con el actual
movdqu xmm15, xmm6
pshufb xmm15, [MEQUEDOCONELDELMEDIO]
movdqu xmm8, xmm15
movdqu xmm9, xmm15
movdqu xmm10, xmm15
pshufb xmm8, [MASKCOLORROJO]; dicerojoperoseriaelblue
paddw xmm8, xmm1
pshufb xmm9, [MASKCOLORVERDE]
paddw xmm9, xmm5
pshufb xmm10, [MASKCOLORAZUL]
paddw xmm10, xmm7

```

En esta sección realizamos la división por max, a diferencia de las otras conversiones, en esta convertimos truncado a double para no perder precisión en la división. Luego de dividir realizamos distintos shifteos, y próximo a esto, shifteos con máscaras para ir quedándonos definitivamente con cada color final y sumándolos entre sí para proceder a chequear máximos y mínimos. 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 máscara 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 comparación y lo guardamos para que esos casos no vuelvan a ser chequeados. Luego, nuestro resultado nos queda en XMM1.

Segundo, si el número es menor a 255:

```

orpd xmm1, xmm2
movdqu xmm0, xmm1
movdqu XMM14, [M255]
movdqu xmm15, xmm0
pcmpgtw XMM14, XMM15

```

Donde, M255 es una máscara con todos 255.

Procedemos a:

```

movdqu XMM13, 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:

```
addr15,1
orps xmm3,xmm4
movdqu xmm0,xmm3
cmpr15,1
je.verdes
cmpr15,2
je.rojos
cmpr15,3
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:

```
xorRAX, RAX
movEAX, EBX
addEAX, R14D
xorR15, R15
movdR15D, XMM0
mov[RSI + RAX], R15D
```

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

### 2.4.3 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 diversas resoluciones:

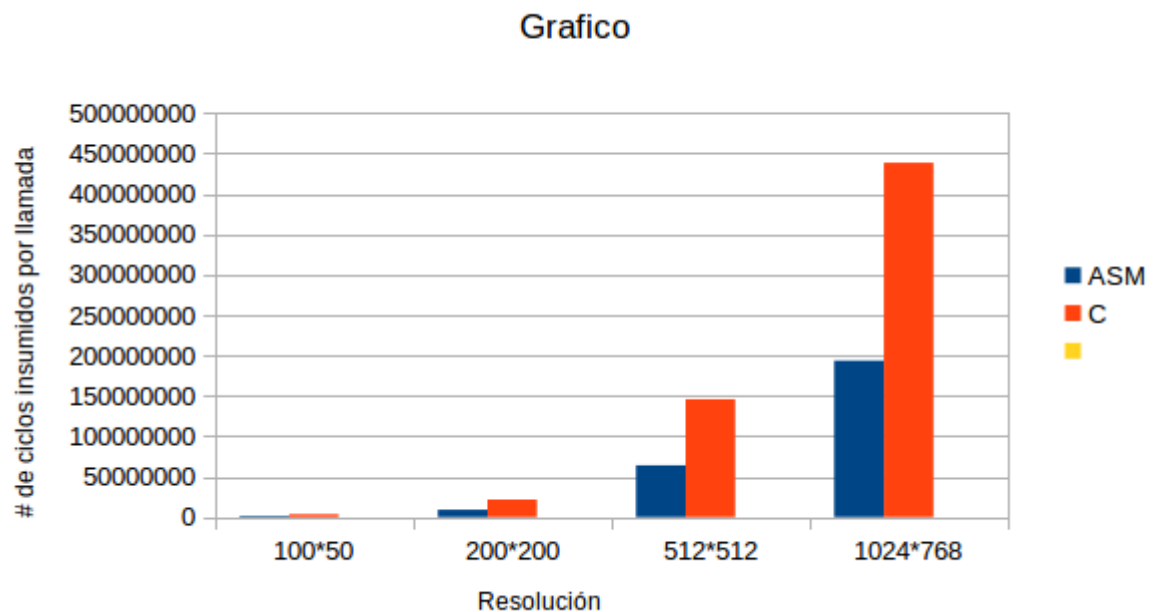
120x56  
En ASM: 1460376  
En C:3431854

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

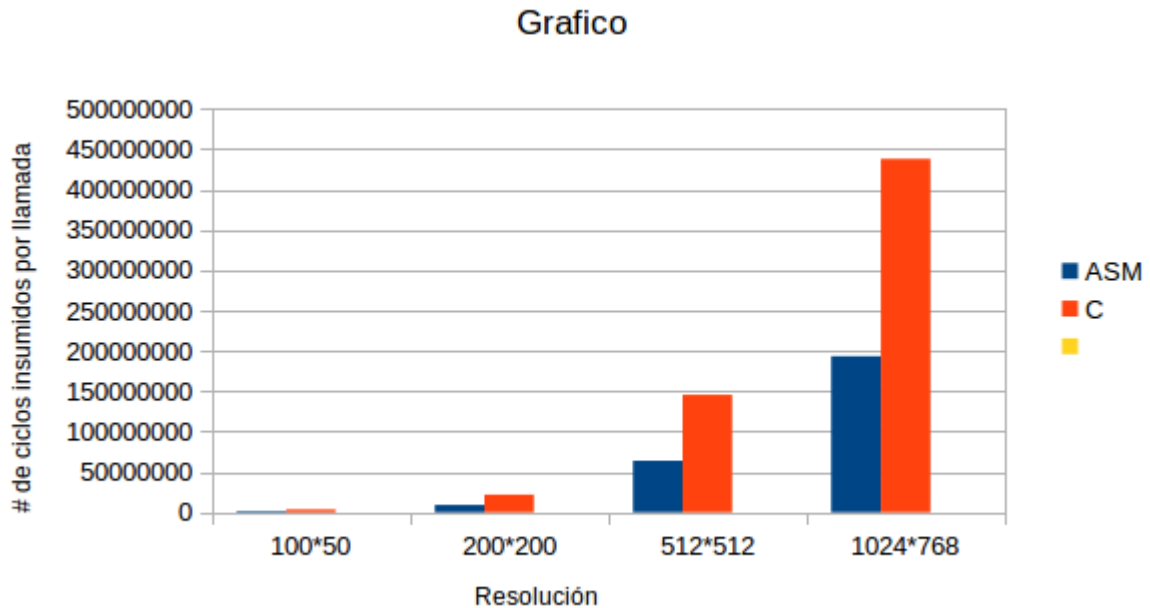
En C:145574048

1023x767

En ASM: 193388816

En C:437996160

Para una mejor representación se realizó un gráfico mostrando a simple vista las diferencias entre cada tipo de implementación 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 simultáneamente durante el mismo ciclo. En nuestra implementación de ASM en una sola iteración ya obtenemos la suma correspondiente de los píxeles vecinos, mientras que en la implementación del C por cada iteración se realizan dentro de esa misma varias iteraciones más sumando los píxeles 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 dieciseisava 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 leyéramos 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 prerequisite 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.