



## DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico II

29 de septiembre de 2015

Organizacion del computador II

Integrante	LU	Correo electrónico
Gonzalez Benitez, Alberto Juan	324/14	gonzalezjuan.ab@gmail.com
Lew, Axel Ariel	225/14	axel.lew@hotmail.com
Noli Villar, Juan Ignacio	174/14	juaninv@outlook.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		



**Facultad de Ciencias Exactas y Naturales**  
Universidad de Buenos Aires  
Ciudad Universitaria - (Pabellón I/Planta Baja)  
Intendente Güiraldes 2160 - C1428EGA  
Ciudad Autónoma de Buenos Aires - Rep. Argentina  
Tel/Fax: (54 11) 4576-3359  
<http://www.fcen.uba.ar>

# Índice

<b>1. Introducción</b>	<b>2</b>
1.1. Introducción . . . . .	2
1.2. Desarrollo . . . . .	3
1.3. Diff en c . . . . .	3
1.4. Diff en asm . . . . .	4
1.5. Blur en c . . . . .	4
1.6. Blur en asm . . . . .	5
<b>2. Hipotesis: Asm maneja mejor la cache</b>	<b>6</b>
2.1. Resumen . . . . .	6
2.2. Experimentaciones . . . . .	6
2.3. Datos obtenidos . . . . .	7
2.4. Gkuráficos . . . . .	7
2.5. Conclusion . . . . .	8
<b>3. Hipotesis: El compilador ICC es mas veloz que el GCC</b>	<b>9</b>
3.1. Resumen . . . . .	9
3.2. Experimentacion . . . . .	9
3.3. Resultados . . . . .	9
3.4. Conclusion . . . . .	11
<b>4. Hipotesis: El uso de la cache de ICC es diferente al de GCC</b>	<b>11</b>
4.1. Resumen . . . . .	11
4.2. Experimentaciones . . . . .	12
4.3. Datos obtenidos: . . . . .	12
4.4. Conclusion . . . . .	13
<b>5. Hipotesis: asm sin simd es mejor que C</b>	<b>14</b>
5.1. Resumen . . . . .	14
5.2. Assembler con SIMD es más rápido que la Assemller sin SIMD . . . . .	14
5.3. Gráficos SIMD vs sin SIMD . . . . .	15
5.4. La implementación en C es más rápida que la implementación en Assemller sin SIMD	15
5.5. Gráficos C vs sin SIMD . . . . .	16
<b>6. Conclusión</b>	<b>17</b>
6.1. Conclusion . . . . .	17

## 1. Introducción

### 1.1. Introducción

En este trabajo practico trabajaremos con procesamiento de imagenes. Implementaremos dos filtros, blur y diff, ambos en dos lenguajes diferentes, assembler y c.

El diff, o diferencia de imagenes, toma como parametro dos imagenes y generara una nueva en la que se resalta donde difieren, en escala de grises. Para cada par de pixeles correspondientes, se calcula la maxima diferencia entre sus componentes y la misma se devuelve en el pixel correspondiente de la imagen de salida

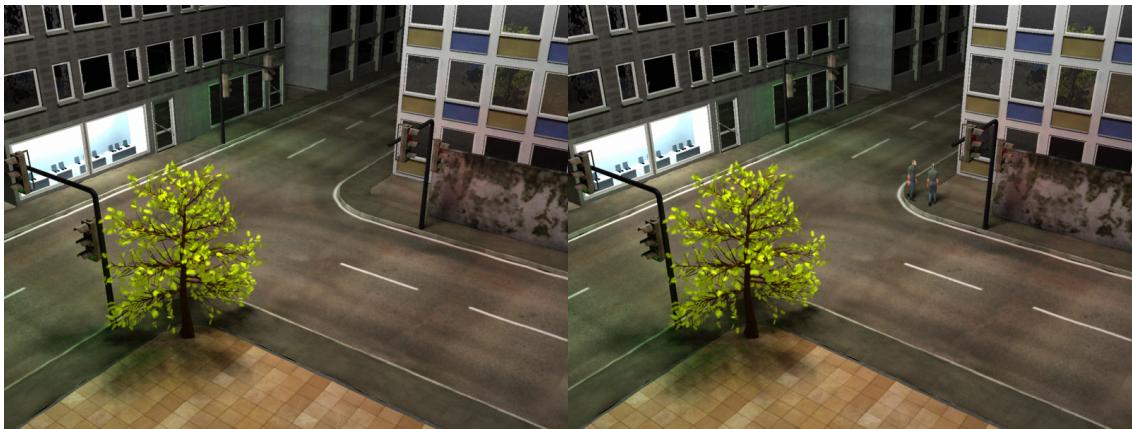


Figura 1: Imagen 1

Figura 2: Imagen 2



Figura 3: Resultado

El blur (gaussiano) consiste en un suavizado de la imagen, para darle un aspecto desenfocado a la misma. El mismo toma como parametro una imagen, un radio y un  $\sigma \in \mathbb{R}$ . El filtro se realiza de la siguiente manera. Primero se calcula, dado el  $\sigma$  y el radio, la matriz de convolucion correspondiente, utilizando la funcion gaussiana. Luego, para cada pixel, toma la matriz formada

por los vecinos (pixeles con distancia menor o igual al radio) y realiza una multiplicacion entre cada elemento de ambas, los suma, y almacena el resultado en el pixel de salida correspondiente



Figura 4: Imagen entrada



Figura 5: Imagen salida, con radio 15 y sigma 5

Ademas, vamos a explicar el funcionamiento de las 4 implementaciones y comparalas, analizando sus diferencias, tanto en tiempo de ejecución como cuanto tardan en programarse o su uso de la cache.

También planteamos un conjunto de hipótesis. Estas estan hechas con el objetivo de maximizar la eficiencia en terminos de tiempo. Realizaremos experimentos para corroborarlas o no, para , de esta forma, lograr un mayor comprendimiento sobre el comportamiento de los filtros y las implementaciones. Y de aquellas conclusiones, vamos a deducir cuales son las implementaciones mas convenientes

## 1.2. Desarrollo

### 1.3. Diff en c

Esta implementacion del diff, es bastante sencilla y se puede codear en relativamente poco tiempo (comparándola con la de assembler ). Básicamente comienza transformando el vector en un matriz para trabajar con mas comodidad.

Luego realiza dos ciclos, uno para iterar sobre las filas y otro sobre las columnas, y de esta forma se accede a cada pixel de la imagen. Se crean 3 int que representan cada color del pixel y se hace la respectiva cuenta para lograr el efecto del filtro, que en pocas palabras es hacer una resta de los dos pixeles de las imágenes y poner el máximo valor en las 3 componentes del pixel. Luego de

realizar esta cuenta se escriben estos valoren en la imagen que se devuelve (agregando ademas, la transparencia, con valor 255). Si bien esta implementacion tiene la ventaja de ser sencilla, fácil de leer y fácil de codear, solo procesa un pixel por iteración, y accede tres vez a la memoria por pixel procesado. Por lo tanto con imágenes muy grandes este problema se cuantifica.

#### 1.4. Diff en asm

Esta implementacion del filtro ya requiere mas tiempo y cuidado asi como una explicacion mas detallada de su funcionamiento. Comenzamos creando, en la section .data, 3 mascaras, las cuales utilizaremos mas adelante para conseguir todas las permutaciones de las 3 componentes del pixel. Se procede crear el stack frame y pushear los registros que usaremos y que hay que conservar. Esta implementacion utiliza solo un ciclo, ya que se recorren la imagen como si fuera un vector. Este ciclo itera filas  $\times$  columnas/4 ya que se procesaran 4 pixeles a la vez. Esto ya muestra una ventaja con respecto a la implementacion en c, ya que se realizan menos accesos a memoria.

El ciclo realiza lo siguiente, primero mueve 4 pixeles de cada una de las dos imágenes a registros dos xmm, en un acceso a memoria cada uno, y luego avanza los punteros en 16 para la próxima iteracion. Como hay que quedarse con el modulo de la resta, realizamos la resta de las dos formas diferentes (es decir, primero a-b y luego b-a), para eso creamos en otro registro xmm una copia de los pixeles de la primera imagen y procedemos a hacer una resta entre el registro que contiene a la primera imagen con el registro que contiene a la segunda, guardando el resultado en el primero, y luego hacemos la misma resta pero con los operandos en el orden inverso, utilizando el registro con la copia. De esta forma al tomar el máximo entre los dos registros que contiene las restas, nos aseguramos de tener el modulo de la resta.

Ahora hay que poner el máximo de las 3 componentes, en las 3 componentes. Para ello crearemos todas las permutaciones posibles de las 3 componentes y las guardamos en 3 registros xmm. Para ello utilizamos la instrucción pshufb con las mascaras anteriormente explicadas.

Para poner el máximo en las 3 componentes, vamos realizando un máximo entre los 3 registros con las permutaciones, de esta forma al ir aplicando máximo, la mayor componente va sobreviviendo y pisando las mas chicas, al final queda la mayor componente en los 3 lugares.

Por ultimo, ponemos en la componente de la transparencia el valor 255, movemos el resultado (de los 4 pixeles) a la imagen de salida en una sola instrucción, aumentamos el contador y llamamos al ciclo devuelta.

Se ve a simple vista que esta implementacion es mas eficiente, pues no utiliza variables temporales y realiza más trabajo por iteracion que su implementacion en c, sin embargo requiere de mucho mas conocimiento y por ende su lectura y comprendimiento deja de ser tan sencillo.

#### 1.5. Blur en c

La implementacion del blur en c tampoco es muy complicada. La realizamos en dos pasos, primero creamos la matriz de convolucion ya que como la misma no varia a lo largo del proceso, decidimos crearla al principio. Esta matriz es de floats, ya que sus valores van entre 0 y 1, y necesitan de bastante precision. Luego el segundo paso es procesar cada pixel.

Para esto el código consta de 4 ciclos, dos para situarnos en los pixeles, y otros dos para cuando se quiere aplicar el filtro al pixel, recorrer la matriz de convolucion y los vecinos del pixel. En resumen, cuando nos situamos en un pixel, creamos 3 floats que utilizaremos como acumuladores, en los cuales vamos guardando la suma del resultado de multiplicar las 3 componentes de los vecinos del pixel a procesar por cada uno de los elementos correspondientes de la matriz de convolucion. Aquí tambien se utilizan floats para no perder precision en las cuentas.

Luego de obtener el resultado, es necesario transformar los floats en numeros de 8 bits, ya que cada

componente va de 0 a 255, por eso los transformamos a unsigned char. Notar no nos preocupamos por la saturacion, ya que ningn acumulador puede pasarse de 255, esto se debe a los valores de la matriz son chicos (su suma da 1).

Por ultimo guardamos los valores en la imagen destino. Notar que la cantidad de iteraciones de esta implementacion es muy alta si el radio elegido y la resolucion de la imagen son grandes, ya que por cada pixel se itera una cantidad de veces igual a aproximadamente 4 veces el radio al cuadrado. Creemos que aqu juega un papel fundamental el tamao de la cache, ya que mientras mas grande sea, se podr usar un radio mas grande sin perder mucho rendimiento.

## 1.6. Blur en asm

La implementacion de este filtro en assembler ya pasa a ser bastante complicada, ya que se requiere muchsimo cuidado al iterar, un error puede generar un acceso a memoria no valido, un corrimiento en el efecto blur, o incluso un efecto totalmente inesperado en la imagen. Ademas hay que ser cauto con el uso de registros que no sean xmm ya se necesitan guardar varios valores y hay pocos registros.

Antes de comenzar a procesar calculamos varias cosas, filas a procesar, pixeles por fila a procesar, saltos para pasar a la siguiente fila (teniendo en cuenta el radio), pixeles a procesar, pixel en donde se comienza y como calcular la posicin del primer vecino teniendo en cuenta el pixel en donde estoy parado.

Para facilitar un poco el trabajo, calculamos la matriz de convolucion en c, por eso desde assembler hacemos un llamado a esta funcin, la cual debe pedir memoria para guardar susodicha matriz. Luego de obtener todo esto, un ciclo que pasa por todos los pixeles a procesar realizara lo siguiente. Primero agarra 4 pixeles de la imagen a procesar (en una sola instruccion) y los guarda, luego trae los primeros 4 vecinos de los respectivos pixeles, osea, el primer vecino de cada uno y tambin los guarda. Por ultimo trae el primer valor de la matriz de convolucion (en realidad toma los primeros 4, en un xmm) y con un pshufd ubica en todo el registro el primer valor de la matriz de convolucion (como es un float, lo repite cuatro veces). Despus extiende los vecinos, los pasamos a floats, para no perder precision al ir multiplicando estos vecinos por el primer valor de la matriz de convolucion que es float. Luego de multiplicar, sumamos los resultados parciales en un acumulador (un acumulador por pixel) y traemos los segundos vecinos de estos 4 pixeles junto con el segundo valor de la matriz de convolucion. Esto continua hasta haber procesado todos los vecinos. Una vez terminado este proceso, procedemos a transformar los floats de los acumuladores (ahora resultados totales) en char, y comprimirlos en un solo xmm. Luego, los guardamos en la imagen destino en una sola instruccion.

Notar que si el numero de pixeles a procesar por fila no es mltiplo de 4 podrn quedar pixeles a los que no se les aplicar el filtro o aplicarle el filtro a pixeles de mas. Por eso en la ultima iteracion por fila, avanzamos este resto y luego retrocedemos 4 posiciones y volvemos a realizar una iteracion mas. As que aunque puede pasar que algunos pixeles vuelvan a ser procesados, nos aseguramos de aplicarles el filtro a todos.

Este ciclo aplica el filtro a todos los pixeles correspondientes de una fila, luego salta lo correspondiente a la siguiente fila y as hasta procesar las filas que sean necesarias. Por ultimo al terminar libera la memoria utilizada por la matriz de convolucion.

Si bien es mas eficiente que su implementacion en c por realizar mas calculos en menos instrucciones y acceder a memoria en menos oportunidades, el tiempo que requiere implementarlo en assembler es muchsimo mas que el que se tarda en c, ya que hay que tener mucho cuidado con las cantidad de iteraciones, ademas de que es bastante facil quedarse sin registros, y perderse en el cdigo, por eso aunque sea mucho mas eficiente, a veces es preferible programar en c y que luego el compilador haga el pasaje a lenguaje de maquina.

## 2. Hipotesis: Asm maneja mejor la cache

### 2.1. Resumen

Para imágenes suficientemente grandes, donde la cache no es suficiente, los filtros en asm con simd son muuuchos mas rápido que los hechos en C

### 2.2. Experimentaciones

Nuestra hipotesis se basa en que asembler consigue un mejor uso de la cache, y le atribuimos a eso su velocidad con respecto a C, para comprobar eso observaremos los miss/hit de las diferentes implementaciones de los filtros. Usaremos imagenes chicas que no tengan problema en entrar en la cache y muy pesadas. El objetivo de esto es ver si cuando la imagen es mas grande que la cache, los miss/hit de la misma cambian mucho o no en comparacion a los de las imagenes chicas, asi como tambien ver estas diferencias entre assembler y C. En este caso haremos tests en una computadora que tiene una cache de 6mb, por lo que usaremos imagenes que tienen un peso superior al de ese valor.

Con esto esperamos corroborar que los filtros implementados en asembler con simd tienen mas hit que los de C en situaciones que incluyen tanto a procesar una imagen mas grande que la cache como con una mas chica, y por ende son mucho mas rapidos que los de C. Queremos ver asi, si implementar codigo directamente en asm resulta en un mejor uso de la cache, y entonces concluir que esto es una de las razones de lo rapido que son los filtros en asm con respecto a los de C. A medida que realizamos los experimentos iremos incrementando el tamaño de la imagen para ver si la relacion Miss/Hit de ambos filtros va variando o no. Con cada tamaño de imagen ademas variaremos los parametros del filtro.

Los siguientes experimentos los realizamos sobre el filtro blur, ya que como este es mas complicado, tiene mas ciclos y accede a posiciones de la imagen mas inesperadas que las del filtro diff, consideramos que le exige mas al predictor de saltos, y al no ser lineal el contenido de la cache cambiara con mas frecuencia. De esta manera las diferencias en el uso de cache, si es que las hay, seran más pronunciadas. Este filtro ademas recibe parametros, si bien cambiar sigma no altera el uso de la cache (pues solo afecta cuentas), cambiar el radio produciria que mientras este aumente, se utilice una parte de la imagen mas grande y completamente diferente e inesperada, por eso decidimos hacer las pruebas con este filtro.

Para ver el uso de la cache utilizaremos el programa valgrind y realizaremos cada experimento varias veces para obtener un promedio sacando outliers. Fijamos como 30 el radio maximo ya que si lo incrementamos mas, el valgrind demora demasiado en emular el comportamiento de la cache (mas de media hora). A continuacion mostraremos algunos resultados obtenidos con las diferentes imagenes, mostrando su peso y porcentaje de miss.

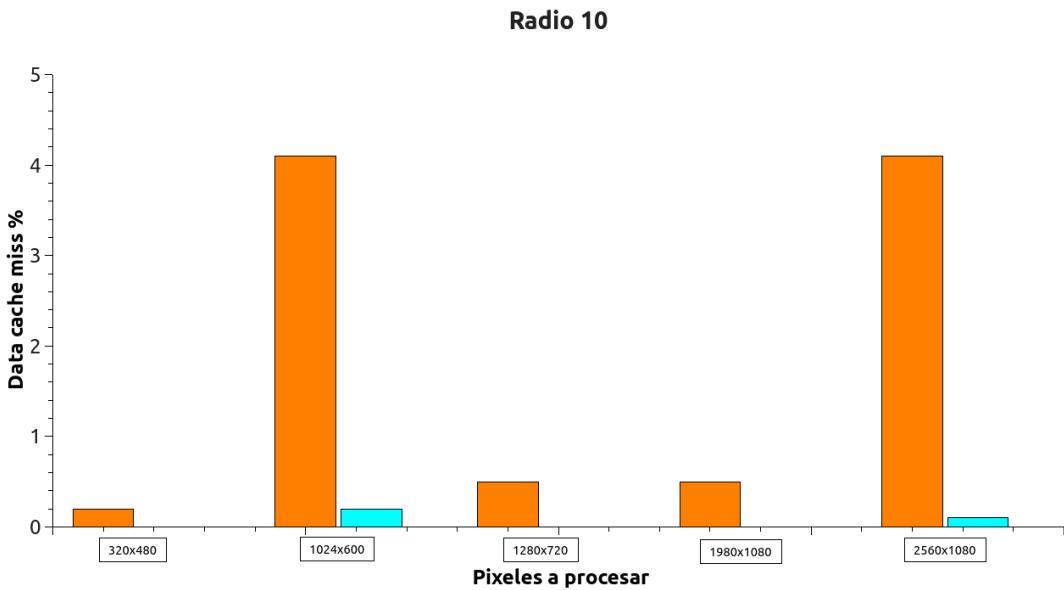
### 2.3. Datos obtenidos

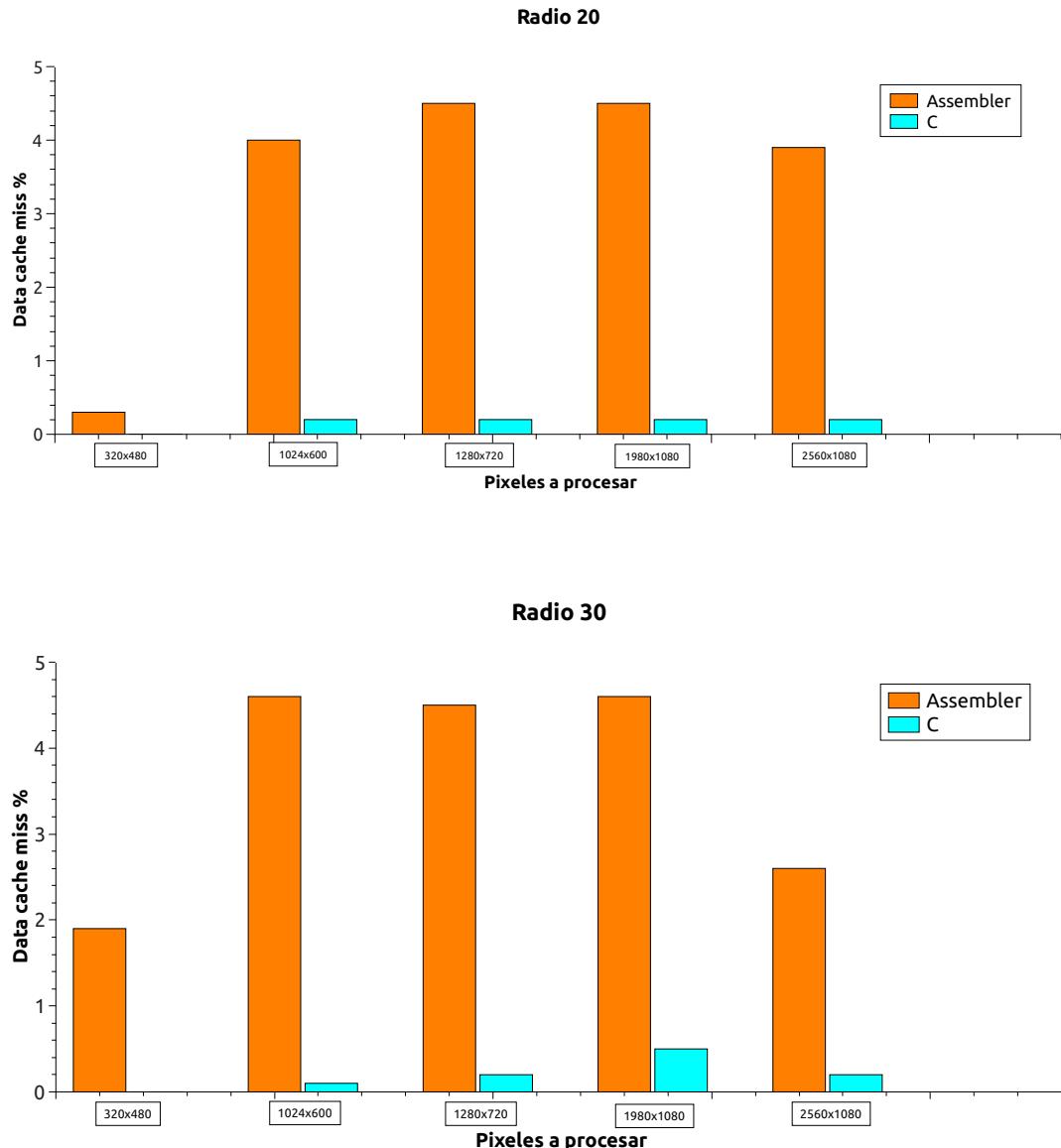
Data cache misses y Miss Jumps Blur (5,10)										
2560x1080 8,3mb 1920x1080 6,2mb 1280x720 2,8mb 1024x600 1,8mb 320x480 600k										
Asm	4.1 %	4.5 %	0.5 %	4.5 %	0.5 %	4.5 %	4.1 %	4.5 %	0.2 %	4.5 %
C	0.2 %	3.9 %	0.2 %	3.9 %	0.0 %	4.4 %	0.2 %	4.4 %	0.0 %	4.4 %

Data cache misses y Miss Jumps Blur (5,20)										
2560x1080 8,3mb 1920x1080 6,2mb 1280x720 2,8mb 1024x600 1,8mb 320x480 600k										
Asm	3.9 %	2.3 %	4.5 %	2.4 %	4.5 %	2.4 %	4.0 %	2.4 %	1.3 %	2.4 %
C	0.2 %	1.9 %	0.2 %	2.3 %	0.0 %	2.3 %	0.1 %	2.2 %	0.0 %	2.3 %

Data cache misses y Miss Jumps Blur (5,30)										
2560x1080 8,3mb 1920x1080 6,2mb 1280x720 2,8mb 1024x600 1,8mb 320x480 600k										
Asm	2.6 %	1.5 %	4.6 %	1.6 %	4.6 %	1.6 %	3.9 %	1.6 %	1.9 %	1.6 %
C	0.2 %	1.2 %	0.2 %	1.5 %	0.0 %	1.5 %	0.1 %	1.5 %	0.0 %	1.5 %

### 2.4. Gkuráficos





## 2.5. Conclusion

A partir de los gráficos y de los datos obtenidos se puede ver claramente un mejor uso de la cache por parte de los filtros en C, esto va en contra de nuestra hipótesis. Sin embargo concluimos que esto pasa porque el compilador hace un muy buen trabajo a la hora de pasar el Código en C a lenguaje de máquina y que lo hace de una manera que aprovecha al máximo el uso de la cache. Esta conclusión sale de que al ver el porcentaje de miss de los filtros en C a medida que aumentamos el radio y el tamaño de la imagen es siempre muy similar. Por lo tanto el tamaño de imagen y el radio no afectan el radio miss/hit de la cache en el filtro implementado en C. Esto no pasa con nuestros filtros en asm, en los cuales si se pueden ver variaciones más notables en el porcentaje de miss. De esto concluimos que el problema de los miss en asm es la forma en que programamos el filtro.

Por otro lado notamos que si bien el porcentaje de miss jumps de ambas implementaciones es muy similar, la cantidad de saltos que realiza C es muchísimo más alta que la de asm. No incluimos estos números en el informe porque con un radio de 20 valgrind no terminaba de procesar la imagen,

sin embargo al parar la emulacion de la cache en diferentes momentos, el porcentaje de miss era siempre el mismo, tanto en asm como en C. Con radios mas chicos la emulacion si terminaba y si se podia observar una cantidad de saltos hasta 5 veces mayor en la implementacion de C. Esto se puede atribuir a que asm utiliza simd y procesa 4 pixeles por iteracion y por ende realiza menos viajes a la memoria, y por eso es mas rapido.

### **3. Hipotesis: El compilador ICC es mas veloz que el GCC**

#### **3.1. Resumen**

Queremos corroborar que el compilador icc es mejor, en terminos temporales (mas adelante se analizara el comportamiento respecto de la cache), que el compilador gcc.

#### **3.2. Experimentacion**

Para analizar los compiladores usaremos el filtro diff. Lo aplicaremos sobre cuatro imagenes, de diferente tamaño. Sobre cada imagen, correremos el filtro varias veces, dado que los procesos de nuestra computadora pueden afectar temporalmente a los filtros. De esta manera, nos aproximaremos mas al resultado real. Calcularemos la desviacion estandar, para reflejar que solo es una aproximacion

Para cada imagen, la procesamos utilizando ambos compiladores y anotamos los tiempos (en ciclo de clocks), y a estos, los dividimos por la cantidad de pixel de la imagen. Asi, obtenemos cuanto tarda, en promedio, procesar un pixel. Luego, de estos valores, obtenemos la media y la desviacion estandar. Tomaremos el tamaño de la imagen como la cantidad de pixeles, dado que se procesan todos los pixeles en orden consecutivo, con lo cual no habria diferencia si la imagen tuviera otra largo o ancho

#### **3.3. Resultados**

Hechas las mediciones, obtenemos lo siguiente

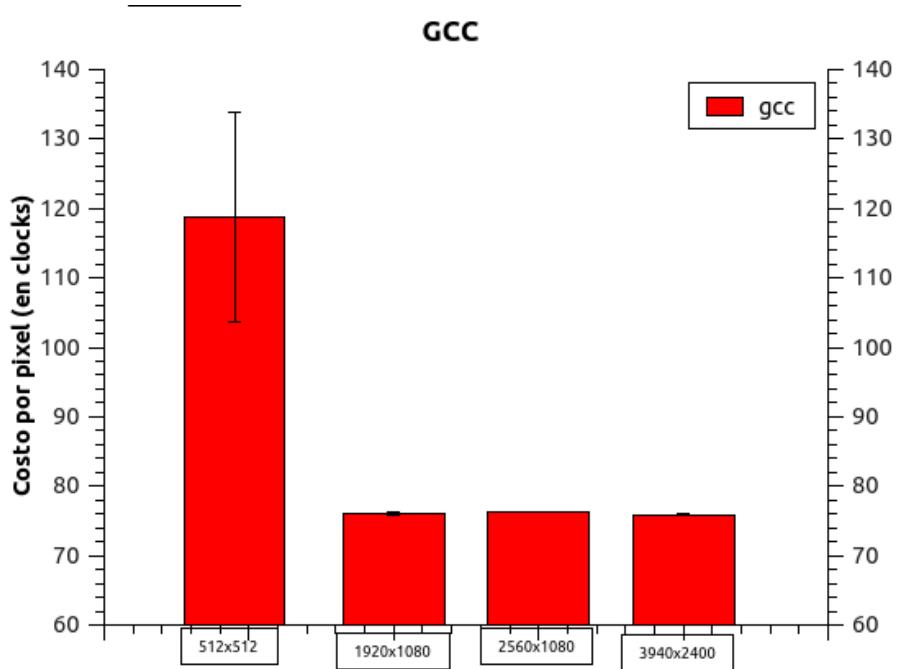


Figura 6: Medicion de gcc

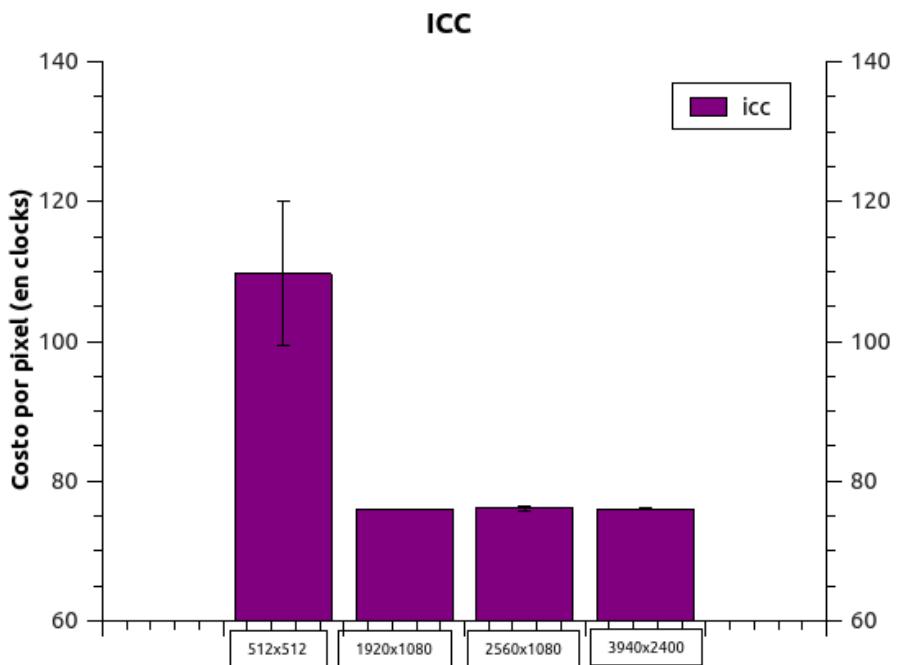


Figura 7: Medicion de icc

### 3.4 Conclusin HIPOTESIS: EL USO DE LA CACHE DE ICC ES DIFERENTE AL DE GCC

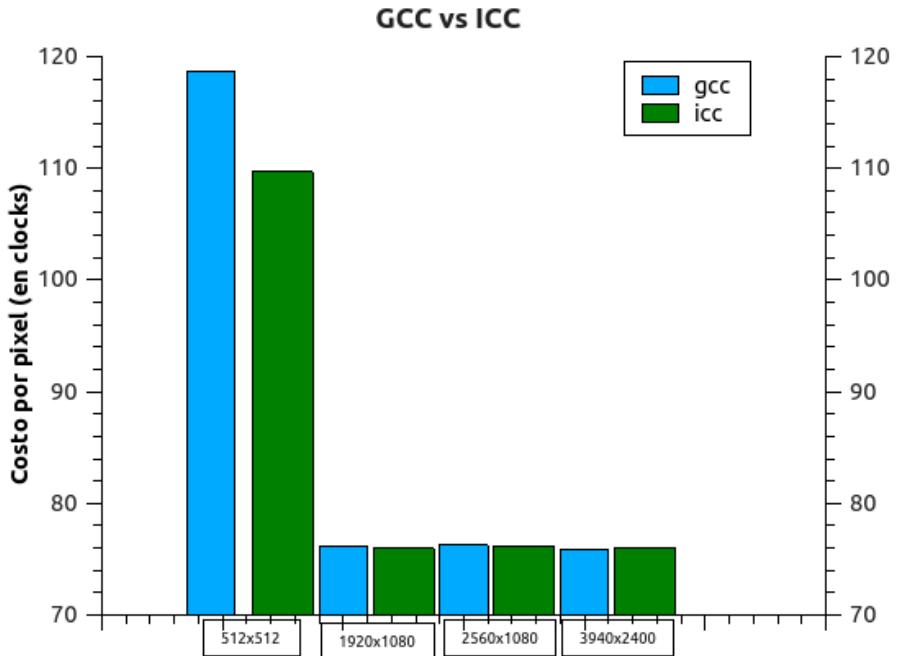


Figura 8: Comparacion entre gcc y icc

Podemos analizar las siguientes cosas de los resultados obtenidos:

Primero, en la imagen mas pequeña, el tiempo de procesamiento por pixel es bastante mayor que en las imagenes mas grandes, en las cuales el tiempo por pixel es casi el mismo. Esto sucede en ambos compiladores. Creemos que esto puede deberse a que hay alguno/algunos procesos en la computadora que no son constantes (es decir, no ocurren cada x cantidad de ciclos) y que influyen en los tiempos. Por eso para imagenes pequeñas se nota su influencia, pero para imagenes grandes no. Tiene sentido entonces, que la desviacion estandar nos de mucho mas alta en la imagen de menor tamao Segundo, si bien para la imagen mas pequeña se puede observar que el compilador icc procesa mas rapido, no existe tal diferencia para las imagenes mas grandes. Incluso, en la imagen mas grande, es mas rapido (aunque la diferencia es cercana al 1%)

#### **3.4. Conclusion**

Refutamos, tristemente, nuestra hipotesis y concluimos entonces que el compilador icc no tiene ventaja sobre gcc. Compararemos mas adelante el uso de la cache, pero, a priori, no vale la pena gastar plata en icc.

### **4. Hipotesis: El uso de la cache de ICC es diferente al de GCC**

#### **4.1. Resumen**

El uso de la cache es el mismo sin importar el compilador: Usando programas para ver el contenido y el uso del cache (como por ejemplo valgrind) procesaremos varias imgenes, de distintos tamaos, e imgenes no cuadradas, para corroborar que sin importar el compilador, el uso de la cache es el mismo.

## 4.2 Experimento HYPOTHESIS: EL USO DE LA CACHE DE ICC ES DIFERENTE AL DE GCC

### 4.2. Experimentaciones

Veremos a continuacion si el uso de la cache por ambos compiladores es igual, similar o completamente difetenete, para esto relizaremos varias pasadas con los filtros implementados en c a diferentes imagenes utilizando dos compiladores diferentes, ICC y GCC. Esperamos ver que el funcionamiento de la cache es igual en ambos.

Utilizaremos el valgrind, el cual nos muestra los misses y las prediciones de saltos fallidas, dependiendo de esos resultados podremos concluir si el uso de la cache es igual de eficiente o si varia .

Primero veremos como es el comportamiento con el filtro blur, por cuestines de cuanto se tarda en emular la cache a medida que se aumenta el radio, decidimos probar los casos con un radio chico, en este caso 10, e ir aumentando solo el tamaño de la imagen. Si hay diferencias en el uso de la cache con radios chicos, con radios mas grandes esta diferencia se va a mantener o incluso aumentar, por eso consideramos que fijar el radio no perjudica nuestro experimento.

### 4.3. Datos obtenidos:

Data cache misses y Miss Jumps Blur (5,10)				
1920x1080 6,2mb 1280x720 2,8mb 1024x600 1,8mb 320x480 600k				
ICC	Dm:0.2% Mj:3.9%	Dm:0.0% Mj:4.4%	Dm:0.2% Mj:4.4%	Dm:0.2% Mj:4.4%
GCC	Dm:0.2% Mj:3.9%	Dm:0.0% Mj:4.4%	Dm:0.2% Mj:4.4%	Dm:0.2% Mj:4.4%

Data cache misses y Miss Jumps Blur (5,10)				
1920x1080 6,2mb 1280x720 2,8mb 1024x600 1,8mb 320x480 600k				
ICC	Dm: 3,414,935	Dm: 1,555,908	Dm: 28,968,342	Dm: 121,320
GCC	Mj: 44,316,774	Mj: 19,412,416	Mj: 12,819,354	Mj: 3,044,089
ICC	Dm: 3,416,876	Dm: 1,555,552	Dm: 28,967,956	Dm: 120,936
GCC	Mj: 44,316,403	Mj: 19,411,551	Mj: 12,818,470	Mj: 3,043,229

Data cache misses y Miss Jumps Diff				
1920x1080 6,2mb 1280x720 2,8mb 1024x600 1,8mb 320x480 600k				
ICC	Dm:0.2% Mj:0.0%	Dm:0.2% Mj:0.0%	Dm:0.3% Mj:0.1%	Dm:0.3% Mj:0.3%
GCC	Dm:0.2% Mj:0.0%	Dm:0.2% Mj:0.0%	Dm:0.3% Mj:0.1%	Dm:0.3% Mj:0.3%

Data cache misses y Miss Jumps Diff				
1920x1080 6,2mb 1280x720 2,8mb 1024x600 1,8mb 320x480 600k				
ICC	Dm: 1,295,470	Dm:651,145	Dm: 490,247	Dm: 134,168
GCC	Mj: 9,349	Mj: 8,712	Mj: 8,432	Mj: 7,611
ICC	Dm: 1,295,437	Dm: 651,114	Dm: 490,202	Dm: 134,126
GCC	Mj: 8,964	Mj: 8,324	Mj: 8,069	Mj: 7,238

#### **4.4. Conclusion**

Luego de obtener los resultados y analizarlos observamos que si bien los valores de misses y miss jumps son muy parecidos , no son lo suficientemente parecidos como para concluir que el uso de la cache que hacen ambos compiladores es el mismo. Esto lo deducimos porque vimos que cuando corrimos varias veces el mismo filtro a la misma imagen utilizando siempre solo un compilador, al ver los resultados (los miss cache), estos varian entre si en muy poco (+- diez ), pero si compramos los resultados de misses obtenidos utilizando compiladores diferentes estas variaciones son suficientemente grandes como para ver que definitivamente no hacen lo mismo.

Por otro lado si miramos los porcentajes, se observa que ambos tienen exactamente los mismos, por eso concluimos que no eran necesarios graficos, pues los valores son exactamente iguales, para cualquiera de los dos filtros, en cualquier situacion. Finalmente podemos concluir que en realidad los compiladores manejan la cache de maneras distintas. Como un detalle, se puede ver que ICC tiende a tener mas misses y miss jumps que GCC. Antes habiamos visto que ICC es un poco mas rpido que GCC, bueno aqui podemos confirmar que eso no se debe a un mejor uso de la cache.

## 5. Hipotesis: asm sin simd es mejor que C

### 5.1. Resumen

Esta implementación del filtro la vamos a utilizar para corroborar una de nuestras hipótesis planteadas.

El funcionamiento de esta implementación realizada en assembler es completamente diferente a la usada en SIMD. Además de diferente lo positivo de esta implementación es su simplicidad.

Lo malo de esta implementación es la cantidad de iteraciones a realizar y la cantidad de accesos a memoria por iteración. El pseuocódigo es el siguiente:

```
//Tengo 6 registros temporales supongamos que se enumeran de R1...R6 todos de 1Byte
// El puntero a la imagen 1 lo voy a llamar Img1, el de la imagen dos Img2 y el puntero a la
//Imagen de destino lo voy a llamar Dest
//Ademas de todos estos tengo un registro contador (Cont) que tiene como valor: Cont = an-
//cho*alto que representa las iteraciones a rehalar
```

```
while(Cont != 0)           //Itero tantas veces como pixeles tenga
    R1<-[Img1];          //traigo la componente red de la imagen 1
    R2<-[Img2];          //traigo la componente red de la imagen 2
    Img1++;                //Me muevo un byte en la imagen 1
    Img2++;                //Me muevo un byte en la imagen 2
    R3<-[Img1];          //Traigo la componente Green de la imagen1
    R4<-[Img2];          //Traigo la componente Green de la imagen2
    Img1++;                //Me muevo un byte en la imagen1
    Img2++;                //Me muevo un byte en la imagen2
    R5<-[Img1];          //Traigo la componente Blue de la imagen1
    R6<-[Img2];          //Traigo la componente Blue de la imagen2
    Img1+=2;               //Apunto al siguiente pixel saltando la componente alpha
    Img2+=2;               //Apunto al siguiente pixel saltando la componente alpha
    //Ahora voy a calcular el modulo de la resta en cada uno de los componentes
    R1<- mod(R1-R2)
    R2<-mod(R3-R4)
    R3<-mod(R5-R6)
    //una vez que tengo en R1, R2, R3 el modulo de las restas de cada uno de los componentes
    //paso a calular el maximo entre esos 3
    R1<- max(R1, R2);
    R1<-max(R1, R3);
    //Finalmente los muevo a la imagen de destino cuidando de poner 255 en la componente
alpha
    [Dest]<-R1;
    Dest++;
    [Dest]<-R1;
    Dest++;
    [Dest]<-R1;
    Dest++;
    [Dest]<-255;           //muevo el 255 en byte obviamente
    Dest++;
    Cont-;
```

### 5.2. Assembler con SIMD es más rápido que la Assemller sin SIMD

¿Es esta hipótesis verdadera? La respuesta es sí. Pero... ¿por qué?  
Primero voy a hacer varios casos de prueba entre ambas implementaciones para analizar su com-

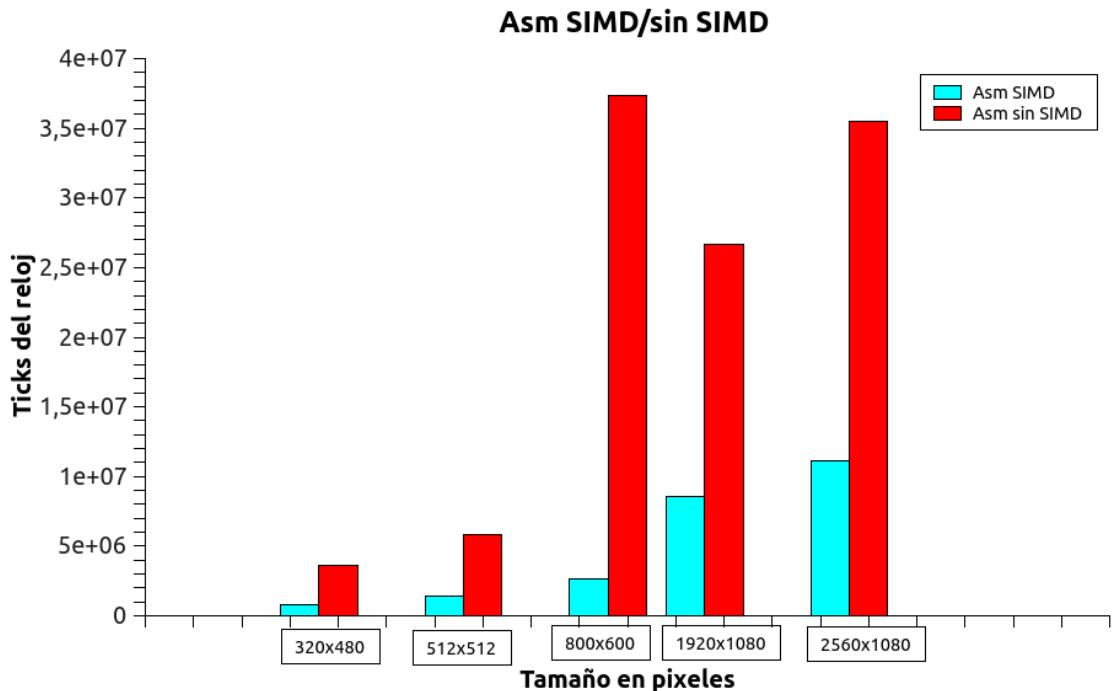
portamiento. Lo esperado a observar en esta comparación es una notable diferencia a favor del filtro realizado usando SIMD.

¿Y como medimos esta diferencia?

La métrica decidida para esta y para otras comparaciones es la métrica dada por los Ticks del reloj.

Para eso voy a realizar varios casos de prueba y voy a comparar la cantidad de ciclos de clock utilizados .

### 5.3. Gráficos SIMD vs sin SIMD



Como esperábamos, el uso del modelo SIMD es muy superior ya que tiene menor cantidad de iteraciones, menor acceso a memoria por iteración y mayor capacidad de procesamiento. No hay necesidad de examinarlo mas a fondo ya que hoy en día casi se puede realizar una demostración formal sobre la ventaja de SIMD sobre asm sin usar SIMD

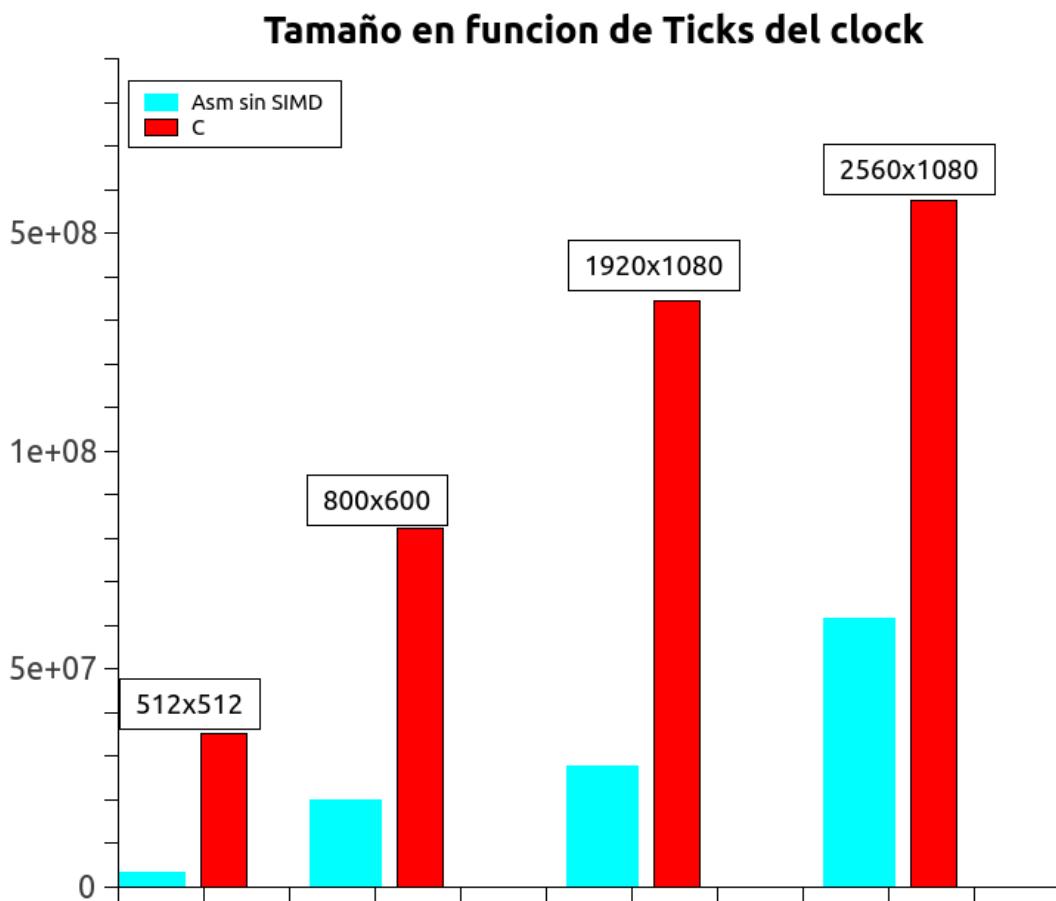
### 5.4. La implementación en C es más rápida que la implementación en Assembler sin SIMD

A continuación vamos a comparar el rendimiento entre la implementación del filtro en C y su implementación en assembler sin el uso de SIMD.

Nuestra hipótesis consiste en que la implementación en C es más rápida que la implementación realizada en Assembler. Para tratar de corroborar esto vamos primero a comparar el tiempo en Ticks del reloj utilizados para procesar una imagen en ambas implementaciones. Lo esperado en este caso es que el filtro en C demore menos.

El siguiente gráfico muestra, para varias imágenes de distinto tamaño, el tiempo de proceso dado en ticks de reloj de la implementación en C comparado a la implementación en Assembler sin SIMD.

## 5.5. Gráficos C vs sin SIMD



El gráfico anterior muestra, a diferencia de lo supuesto anteriormente, un menor tiempo de procesamiento para el filtro implementado assembler sin SIMD.

¿Y por qué sucede esto?

La diferencia consiste principalmente en que en la implementación de Assembler no hago accesos innecesarios a memoria. Es decir, en cada ciclo accedo en total nueve veces a memoria, ni mas ni menos.

Uno estaría tentado a decir que sucede lo mismo en la implementación del filtro en C, estaría tentado a decir que también accedo esa misma cantidad de veces a memoria pero a diferencia de la implementación en Assembler, accedo más veces indirectamente porque en mi función tengo variables y las variables se guardan en el Stack.

En mi implementación en C de la función diff tengo varias variables y opero con ellas en todo momento ya sea para hacer una resta, para calcular un maximo entre gamas o también incluso para guardar mi resultado en la imagen de destino. Cada una de estas operaciones realizadas con las variables demanda un acceso a memoria, mas específicamente al stack, para poder obtener su valor dando como resultado una mayor cantidad de accesos a memoria y una mayor cantidad de Ticks del reloj en cada iteración del ciclo. Por ende concluimos que nuestra hipótesis resultó ser falsa

## 6. Conclusión

### 6.1. Conclusion

Realizamos este trabajo con el fin de poder optimizar temporalmente, mediante cambios en las implementaciones o compiladores, los filtros realizados. Podemos resumir nuestros resultados en este cuadro.

"+" : Bueno ; "-" : Malo	<b>Asm</b>	<b>C</b>
<b>Tiempo en codear</b>	--	++
<b>Accesos a memoria</b>	++	-
<b>Tiempo de ejecución</b>	++	+
<b>Debuggeo</b>	--	++
<b>Uso de la cache</b>	++	++
<b>Comprensión</b>	-	+
<b>Dificultad de codeo</b>	-	++

Basicamente, como concluimos que las implementaciones en assembler con simd son las optimas. Esto, siempre y cuando se apliquen sobre imagenes muy grandes o sobre un conjunto grande de imagenes. Esto se debe a los altos costes de implementacion que tiene asm (costes en terminos temporales). Sobre las difencias temporales entre asm y C podemos concluir que no se deben a la manera de manejar la cache.