



DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

28 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

1. Introducción	2
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
2. Hipotesis: Asm maneja mejor la cache	6
2.1. Hipotesis 1:	6
2.2. Hipotesis: Assembler maneja mejor la cache	6
2.3. Resultados:	6
2.4. Graficos:	8
2.5. Conclusion:	8

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 (píxeles con distancia menor o igual al radio) y realiza una multiplicación entre cada elemento de ambas, los suma, y almacena el resultado en el píxel de salida correspondiente



Figura 4: Imagen entrada

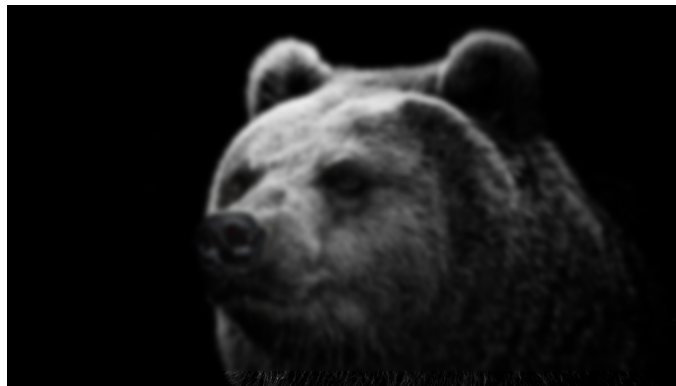


Figura 5: Imagen salida, con radio 15 y sigma 5

Además, vamos a explicar el funcionamiento de las 4 implementaciones y compararlas, 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 están hechas con el objetivo de maximizar la eficiencia en términos de tiempo. Realizaremos experimentos para corroborarlas o no, para, de esta forma, lograr un mayor entendimiento sobre el comportamiento de los filtros y las implementaciones. Y de aquellas conclusiones, vamos a deducir cuáles son las implementaciones más convenientes

1.2. Desarrollo

1.3. Diff en c

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

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

cuenta se escriben estos valores en la imagen que se devuelve. Si bien esta implementación tiene la ventaja de ser sencilla, fácil de leer y fácil de codear, solo procesa un pixel por iteración, y por ende accede una vez a la memoria por pixel procesado. Por lo tanto con imágenes muy grandes este problema se cuantifica. Para arreglar este problema, simplemente se pueden procesar mas pixeles por iteracion, pero habría que crear mas variables temporales, por lo que el uso de memoria seria mayor

1.4. Diff en asm

Esta implementación del filtro ya requiere mas tiempo y cuidado así como una explicación mas detallada de su funcionamiento. Comenzamos creando 3 mascarar, las cuales se utilizaran para conseguir todas las permutaciones de las 3 componentes del pixel.

Se procede crear el stack frame y pushear los registros que usaremos. Esta implementación utiliza solo un ciclo, ya que se recorreremos la imagen como si fuera un vector. Este ciclo itera filas \times columnas/4 ya que se procesaran 4 pixeles a la vez, (se podrían procesar mas y realizar menos iteraciones) esto ya muestra una ventaja con respecto a la implementación 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 xmmx, luego avanza los punteros en 16 para la próxima iteración. Como hay que quedarse con el modulo de la resta, realizamos la resta de las dos formas diferentes, para eso creamos en otro registro xmmx 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 realizamos todas las permutaciones posibles de las 3 componentes y las guardamos en 3 registros xmmmx. Para ello utilizamos la instrucción pshufb con las mascarar 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, aumentamos el contador y llamamos al ciclo devuelta.

Se ve a simple vista que esta implementación es mas eficiente, pues no utiliza variables temporales y realiza más trabajo por iteración que su implementación 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 implementación del blur en c tampoco es muy complicada. La realizamos en dos pasos, primero creamos la matriz de combolución 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 precisión. 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 combolución. En resumen, cuando nos situamos en un pixel, creamos 3 floats que utilizaremos como acumuladores, en los cuales vamos guardando el resultado de multiplicar las 3 componentes de los vecinos del pixel a procesar por cada una de los elementos de la matriz. Para ello se utilizan dos ciclos. Aquí tambien se utilizan

floats para no perder precisión en las cuentas.

Luego de obtener el resultado, es necesario transformamos 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 ningún 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 resolución 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. Aquí juega un papel fundamental el tamaño de la cache, ya que mientras mas grande sea, se podría 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 muchísimo 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 xmmx 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 posición del primer vecino teniendo en cuenta el pixel en donde estoy parado.

Para facilitar un poco el trabajo, calculamos la matriz de combolucion en c, por eso desde assembler hacemos un llamado a esta función, 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 y los guarda, luego trae los primeros 4 vecinos de los respectivos pixeles, osea, el primer vecino de cada uno y también los guarda. Por ultimo trae el primer valor de la matriz de combolucion y con un pshufd ubica en todo el registro el primer valor de la matriz de combolucion. Después extiende los vecinos, los pasamos a floats, de esta forma no perdemos precisión al ir multiplicando estos vecinos por registro que contiene el primer valor de la matriz de combolucion. Luego de multiplicar, guardamos el resultado y traemos los segundos vecinos de estos 4 pixeles junto con el segundo valor de la matriz de combolucion. Esto continua hasta haber procesado todos los vecinos. Una vez terminado este proceso, procedemos a transformar los floats en char, y guardarlos en la imagen destino. Notar que si el numero de pixeles a procesar por fila no es múltiplo de 4 podrían quedar pixeles a los que no se les aplicaría el filtro. Por eso en la ultima iteracion avanzamos este resto y luego retrocedemos 4 posiciones y volvemos a realizar una iteracion mas. Así 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 asi hasta procesar las filas que sean necesarias. Por ultimo al terminar libera la memoria utilizada por la matriz de combolucion.

Si bien devuelta es mas eficiente que su implemntacion en c por realizar mas trabajo en cada iteracion, el tiempo que requiere implementarlo en assembler es muchísimo 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 código, 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.

```
4k icc 700650945 698951916 701532051 701931744 702355236 700911300 700502040 702606924
700385130 700735110 700535523
```

```
4k, 4k C 699764238 699677205 699531477 699760737 699416400 699628125 699825747 699960990
699487806 699381753 699574566
```

```
2k icc 209924214 209909565 213800136 209907222 209953314 209960391 210005364 209978727
```

```
210006525 211156770 209889078
2k, 2k C 210949614 210763089 210926190 210518687 210695397 210688527 210809301 210674454
211244199 210758946 210681990
1k icc 157478697 157427610 157393875 157480062 157825200 157469811 157606969 157396743
157562616 157441539 157530222
1k, 1k en C 157432458 159177300 157445877 157799580 157509957 157411335 158674482
157469229 157480584 157480368
lena24 icc 25707793 31001416 28583751 28433815 34995588 26074746 26174735 29561400 31057908
26161341 28636590
lena24 gcc 27013662 29066746 31620973 26833249 31983549 31391802 30543990 33710355
33368767 24181408 39922867 33880031
```

2. Hipotesis: Asm maneja mejor la cache

2.1. Hipotesis 1:

El compilador icc es mas rápido que el compilador gcc: Probaremos compilar el mismo código de c con ambos compiladores y mediremos tiempos, el uso de la cache, para ver cual de los dos es mas eficiente, probando con imágenes de distinto tamaño e imágenes no cuadradas.

Resultados:

A continuacion procederemos a ejecutar el codigo en c con dos compiladores diferentes, icc, desarrollado por intel y gcc. Intel dice que su compilador es mas rapido así que veremos si los resultados lo acompañan.

Imágenes chicas:

Imágenes grandes:

2.2. Hipotesis: Assembler maneja mejor la cache

Para imágenes suficientemente grandes, donde la cache no es suficiente, los filtros en asm con simd son muuucho mas rápido que los hechos en c: Analizaremos a partir de que tamaño de imagen la cache no es suficiente. Luego veremos como se comportan los filtros

2.3. Resultados:

Nuestra hipotesis se basa en que assembler consigue un mejor uso de la cache, y le atribuimos a eso su velocidad, para comprobar eso observaremos los miss/hit de los diferentes filtros. Usaremos imagenes chicas y muy pesadas , el objetivo 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. 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 assembler 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 así, 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 relizamos los experimentos iremos incrementado 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 y tiene mas ciclos consideramos que le exige mas al predictor de saltos, ademas al no ser lineal el contenido de la cache cambiara con mas frecuencia, y así las diferencias en el uso de cache, si es que las hay, serán mas 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 aumento, se utilice una parte de la imagen completamente diferente e inesperada, por eso decidimos hacer las pruebas con este filtro.

Para ver los 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 incrementabamos mas, porque el valgrind demoraba demasiado en emular el comportamiento de la cache. A continuacion mostraremos algunos resultados obtenidos con las diferentes imagenes, mostrando su peso y porcentaje de miss.

2560x1080 8,3mb

Asm

Radio 10 instruccion miss 0.0 % data cache miss 4.1 % miss jumps 4.5 %
Radio 20 instruccion miss 0.0 % data cache miss 3.9 % miss jumps 2.3 %
Radio 30 instruccion miss 0.0 % data cache miss 2.6 % miss jumps 1.5 %

C:

Radio 10 instruccion miss 0.0 % data cache miss 0.2 % miss jumps 3.9 %
Radio 20 instruccion miss 0.0 % data cache miss 0.2 % miss jumps 1.9 %
Radio 30 instruccion miss 0.0 % data cache miss 0.2 % miss jumps 1.2 %

1920x1080 6,2mb

Asm

Radio 10 instruccion miss 0.0 % data cache miss 0.5 % miss jumps 4.5 %
Radio 20 instruccion miss 0.0 % data cache miss 4.5 % miss jumps 2.4 %
Radio 30 instruccion miss 0.0 % data cache miss 4.6 % miss jumps 1.6 %

C:

Radio 10 instruccion miss 0.0 % data cache miss 0.0 % miss jumps 4.4 %
Radio 20 instruccion miss 0.0 % data cache miss 0.2 % miss jumps 2.3 %
Radio 30 instruccion miss 0.0 % data cache miss 0.2 % miss jumps 1.5 %

1280x720 2,8mb

Asm

Radio 10 instruccion miss 0.0 % data cache miss 0.5 % miss jumps 4.5 %
Radio 20 instruccion miss 0.0 % data cache miss 4.5 % miss jumps 2.4 %
Radio 30 instruccion miss 0.0 % data cache miss 4.6 % miss jumps 1.6 %

C:

Radio 10 instruccion miss 0.0 % data cache miss 0.0 % miss jumps 4.4 %
Radio 20 instruccion miss 0.0 % data cache miss 0.2 % miss jumps 2.3 %
Radio 30 instruccion miss 0.0 % data cache miss 0.2 % miss jumps 1.5 %

1024x600 1,8mb

Asm:

Radio 10 600 instruccion miss 0.0 % data cache miss 4.1 % miss jumps 4.5 %
Radio 20 600 instruccion miss 0.0 % data cache miss 4.0 % miss jumps 2.4 %
Radio 30 600 instruccion miss 0.0 % data cache miss 3.9 % miss jumps 1.6 %

C:

Radio 10 600 instruccion miss 0.0 % data cache miss 0.2 % miss jumps 4.4 %
Radio 20 600 instruccion miss 0.0 % data cache miss 0.1 % miss jumps 2.2 %

Radio 30 600 instruccion miss 0.0 % data cache miss 0.1 % miss jumps 1.5 %

320x480 600k

Asm:

Radio 10 480 instruccion miss 0.0 % data cache miss 0.2 % miss jumps 4.5 %

Radio 20 480 instruccion miss 0.0 % data cache miss 0.3 % miss jumps 2.4 %

Radio 30 480 instruccion miss 0.0 % data cache miss 1.9 % miss jumps 1.6 %

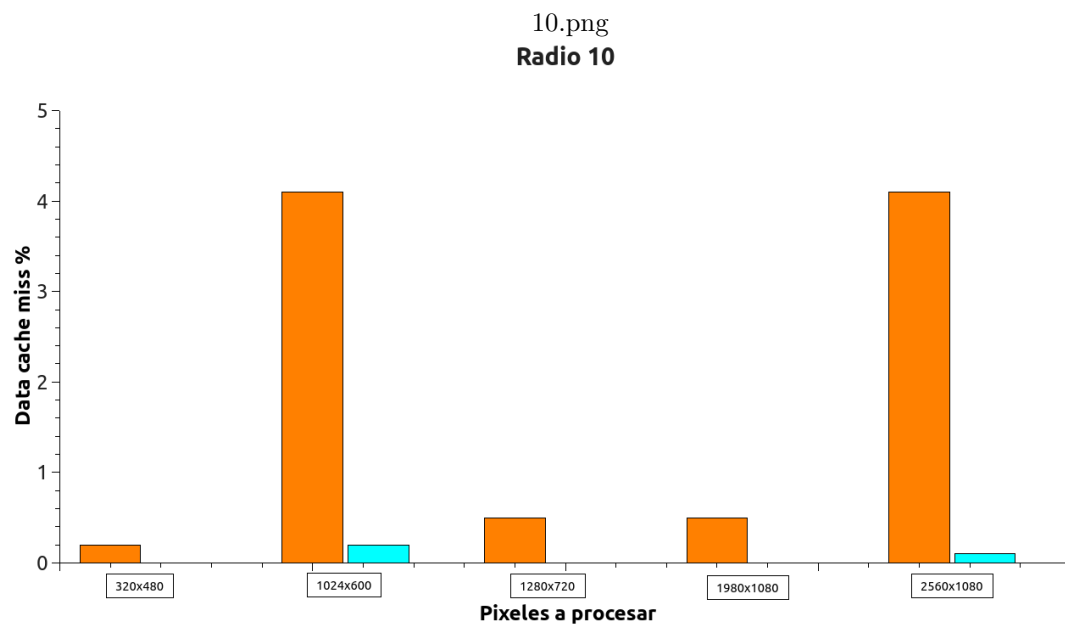
C:

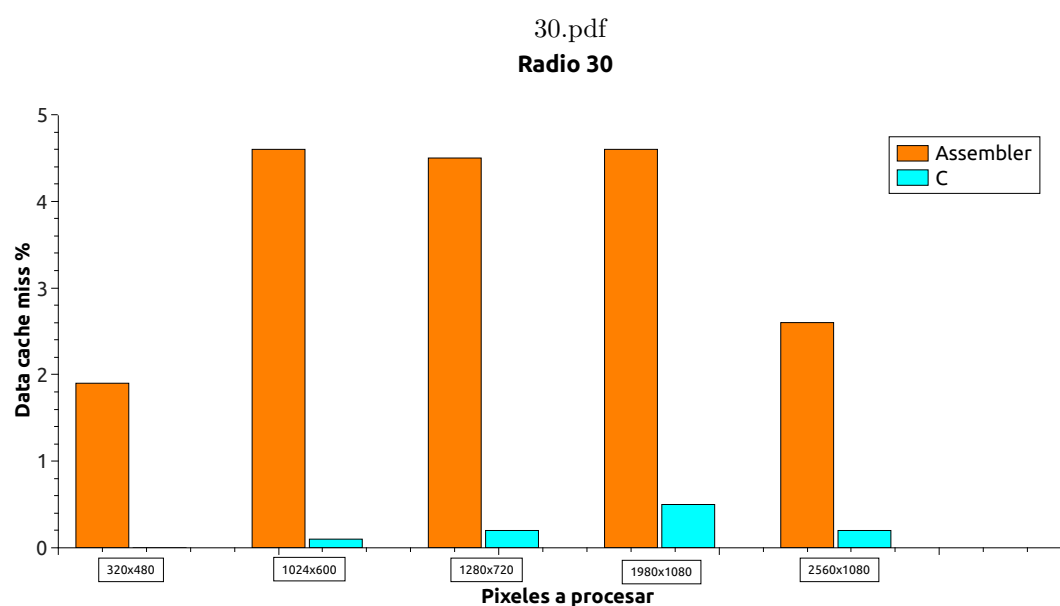
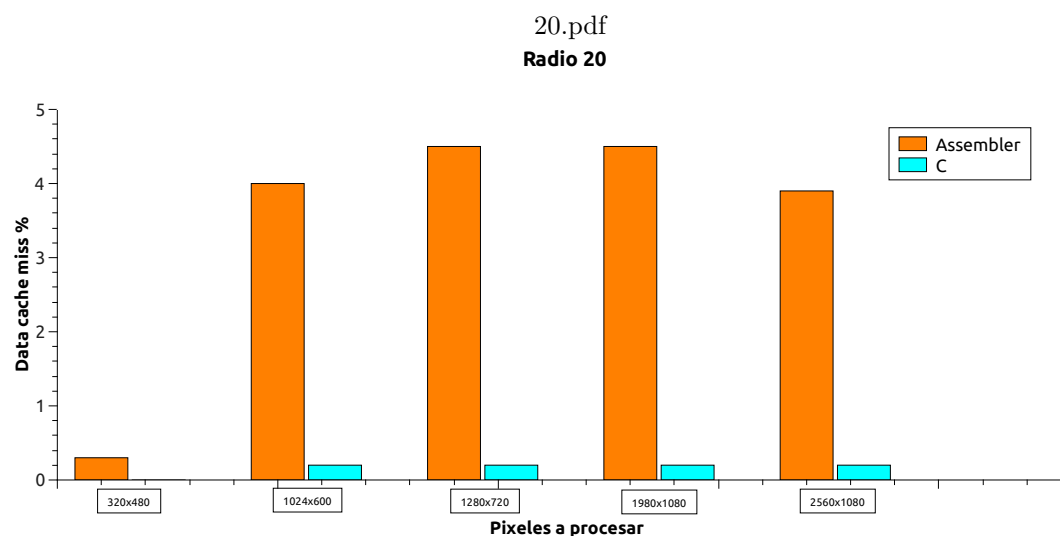
Radio 10 480 instruccion miss 0.0 % data cache miss 0.0 % miss jumps 4.4 %

Radio 20 480 instruccion miss 0.0 % data cache miss 0.0 % miss jumps 2.3 %

Radio 30 480 instruccion miss 0.0 % data cache miss 0.0 % miss jumps 1.5 %

2.4. Graficos:





2.5. Conclusion:

A partir de los graficos 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 hipotesis. Sin embargo concluimos que esto pasa porque el compilador hace un muy buen trabajo a la hora de pasar el codigo en c a lenguaje de maquina, y que lo hace de una manera que aprovecha al maximo el uso la cache. Esta conclusion sale de que sin importar demasiado el tamaño de la imagen , el porcentaje de miss de nuestra implementacion en assembler varia muy poco a medida que aumentamos el radio y el tamaño de la imagen , mientras que la de c es siempre muy similar. Por lo tanto el tamaño de imagen y el radio no afectan el radio miss/hit de la cache. Sin embargo notamos que si bien el porcentaje de miss jumps de ambas implementaciones es muy similar, la cantidad de saltos que realiza c es muchisimo mas alta que la de asm. No inclumos estos numeros en el infome porque con un radio de 20 valgrind no terminaba de procezar 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.