



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Operaciones con SIMD

Organización del Computador II
Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Benzo, Mariano	198/14	marianobenzo@gmail.com
Martinez Quispe, Franco	025/14	francogm01@gmail.com
Travi, Fermín	234/13	fermintravi@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

Resumen

En este trabajo se presentan implementaciones sobre el procesamiento de imágenes de manera tal que se computen los datos de forma vectorizada, utilizando la tecnología SIMD de Intel para procesar varios de ellos simultáneamente y así obtener un mejor rendimiento. Luego se presentan distintas aproximaciones a los problemas, presentando hipótesis y experimentaciones en base a su rendimiento que permiten comprobarlas o refutarlas según un determinado criterio.

Índice

1. Objetivos generales	3
2. Contexto	3
3. Filtros	4
3.1. Smalltiles	4
3.1.1. Implementación	4
3.1.2. Experimentación	5
3.2. Rotar	6
3.2.1. Implementación	6
3.2.2. Experimentación	7
3.3. Pixelar	8
3.3.1. Implementación	8
3.3.2. Experimentación	9
3.4. Combinar	11
3.4.1. Implementación	11
3.4.2. Experimentación	12
3.5. Colorizar	14
3.5.1. Implementación	14
3.5.2. Experimentación	15

1. Objetivos generales

El objetivo de este Trabajo Práctico es comprender el uso de las instrucciones que aprovechan la tecnología SIMD de Intel para procesar varios datos simultáneamente, en conjunto con un análisis con respecto a las implementaciones realizadas para lograr un mayor entendimiento de su funcionamiento y a su vez de cómo plantear y analizar distintas problemáticas sobre un mismo tema.

2. Contexto

Para la implementación y uso de las instrucciones SIMD, se trabaja sobre el procesamiento de imágenes, aplicando distintos filtros sobre los mismos. Las imágenes se almacenan en memoria como una matriz con elementos de 32 bits, donde cada elemento corresponde a un pixel de la imagen. Para las implementaciones sobre C, se utiliza la siguiente estructura provisto en *tp2.h* para trabajar sobre los pixeles:

```
typedef struct bgra_t {  
    unsigned char b, g, r, a;  
} __attribute__((packed)) bgra_t;
```

Además, para las mediciones de rendimiento, se cuentan la cantidad de *ticks* del procesador que conllevó ejecutar una determinada implementación de los filtros. Los datos de entrada varían en igual cantidad en tamaño y ancho. No se analizan distintas imágenes porque sus componentes no interfieren en los cálculos realizados por los filtros; es decir, el rendimiento de los filtros es independiente de las componentes cromáticas de las imágenes (el único que trabaja en base a los valores de los colores es *Colorizar*, el por qué no lo afectan se encuentra en la explicación de su implementación).

Los filtros se corren cien veces en total y se obtiene un promedio del mismo. La carga y guardado de imágenes se ejecuta anterior y posteriormente a la corrida del filtro, por lo que no afecta a las mediciones. Para evitar la presencia de *outliers*, se calcula la desviación estándar de las mediciones. Si la misma es mayor o solo un 10% menor al promedio, se descarta la medición y se realiza devuelta.

El código utilizado para las mediciones se encuentra en el archivo *tp2.c*, método *void correr_filtro_imagen*. Los archivos con los datos de las mediciones se encuentran en la carpeta 'Mediciones' y los códigos de los experimentos se encuentran en la carpeta 'Filtros', con el sufijo 'exp' o similares.

3. Filtros

3.1. Smalltiles

3.1.1. Implementación

Este filtro implica dividir la imagen original en cuatro cuadrantes, manteniendo el tamaño original. Para generar cada cuadrante, recorreremos la imagen original saltando una fila y columna de por medio. Es decir, cuando procesamos un pixel, descartamos a su vecino y lo almacenamos en cada una de las posiciones de los cuadrantes en la imagen destino, las cuales las calculamos mediante la cantidad de columnas y filas de la imagen fuente (dividiendo por la mitad y multiplicando por el tamaño de los datos de la matriz (32 bits)). Luego, al terminar de recorrer la fila, saltamos la siguiente fila (sumándole al puntero de la imagen fuente el tamaño de la fila en bytes).

Debido a que solo recorreremos la mitad de las filas, la cantidad de iteraciones que se realizan equivale a la cantidad de columnas multiplicado por la cantidad de filas dividido dos. Ya que se itera por columnas, en la implementación en lenguaje ensamblador se mantiene un contador que aumenta en cada iteración y, una vez que se alcanzó la cantidad total de columnas, se asume que se terminó de recorrer la fila y se realizan las operaciones pertinentes para mover los punteros a la fila posterior a la siguiente en memoria (se saltea una fila, como se explicó anteriormente).

El procesamiento en lenguaje ensamblador se realiza de a cuatro pixeles por iteración, por lo que se divide la cantidad total de columnas por cuatro. No es posible procesar de a más pixeles a la vez porque cada uno ocupa 4 bytes y solo entran cuatro de ellos en los registros de SIMD. Dado que el filtro consta básicamente de reordenar datos, se utiliza una única instrucción para su procesamiento. La misma es la instrucción *shuffle*, la cual se ejecuta de manera tal que por cada cuatro pixeles en el registro, se ubican de forma subsiguiente a aquellos de la primera y tercera columna y se los almacena en memoria en cada puntero a cada cuadrante de la imagen destino, descartando a los otros dos.

Dentro de la implementación en C, se recorre la imagen fuente iterando una cantidad de veces equivalente a $\frac{filas}{2} * \frac{cols}{2}$, lo cual representa al tamaño de la imagen de cada cuadrante. De esta manera, el programa selecciona los pixeles de fila y columna par, los cuales son los que se guardarán en la imagen destino. Se itera píxel por píxel, donde en cada iteración se crean cuatro punteros a las direcciones de memoria correspondientes a cada cuadrante de la imagen destino.

Ya que se procesa una mayor cantidad de pixeles por iteración en la implementación en lenguaje ensamblador, se espera que sus tiempos de ejecución sean menores a su análogo en C.

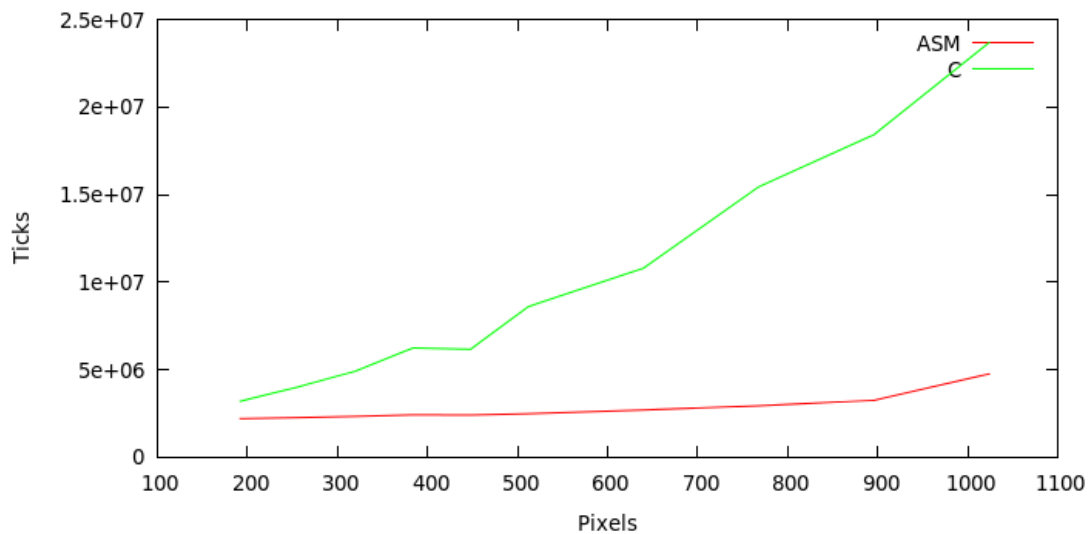


Figura 1: Comparación de los tiempos de ejecución de cada implementación

Como se puede apreciar en 1, hay una clara diferencia entre las distintas implementaciones. Como se suponía, la implementación en lenguaje ensamblador consume una cantidad de tiempo mucho menor para procesar las imágenes de gran tamaño que su análogo en C. Dado que la principal diferencia entre ambas implementaciones es la cantidad de píxeles que se procesan por iteración, suponemos que esta es la razón mayor de por qué se aprecia semejante diferencia.

3.1.2. Experimentacion

Hipótesis

La implementación en lenguaje ensamblador consiste únicamente de ejecutar la instrucción *shuffle* sobre los píxeles a procesar, utilizando un inmediato de 8 bits para reordenar los datos. Dado que el reordenamiento de los datos no es complicado, se lo puede reemplazar fácilmente utilizando *shift* y las operaciones *insert* y *extract*, de manera tal de no perder información al realizar el *shift*.

Si bien es de esperar que la operación *shift* consuma menos tiempo de ejecución que *shuffle*, ya que simplemente es un corrimiento de bits, queremos analizar si en conjunto con las operaciones *insert* y *extract* es posible obtener un mejor rendimiento. Dado que tanto *shuffle* como *insert* y *extract* utilizan bits de control para el reordenamiento de datos, suponemos que utilizan una lógica o unidades de hardware similares, y los tiempos de ejecución no deberían variar mucho.

Resultados

Para lograr analizar la diferencia entre las instrucciones, medimos los tiempos de ejecución de ambas implementaciones en lenguaje ensamblador (según lo descrito al principio del trabajo práctico) y las comparamos.

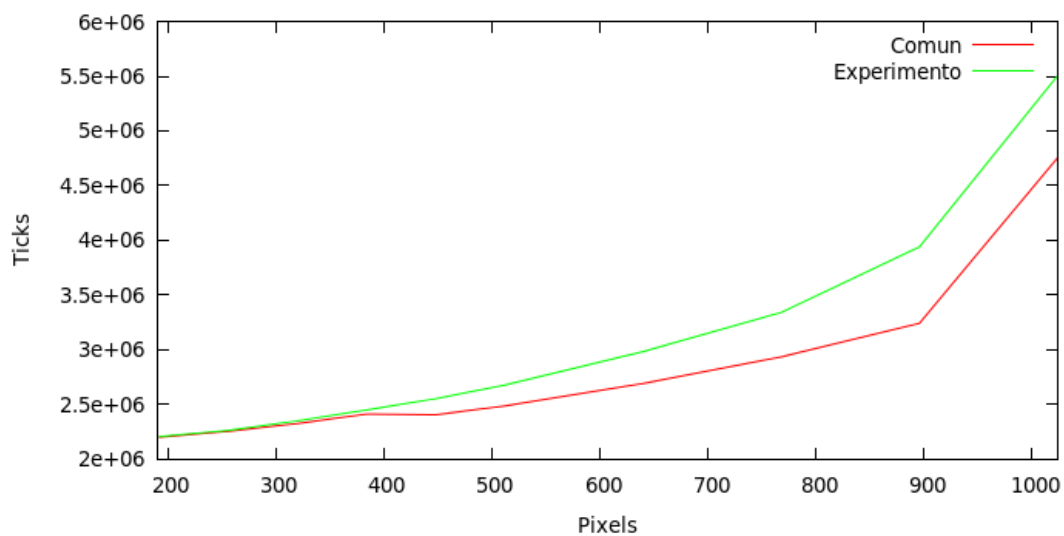


Figura 2: Comparación de los tiempos de ejecución del experimento

Tal como se muestra en el gráfico, los tiempos de ejecución son similares para datos de entrada de tamaño pequeño (a tal punto que se los podría considerar equivalentes), pero para imágenes más grandes, la diferencia se acentúa. Desconocemos por qué la diferencia entre instrucciones implica mayor costo para datos de entrada más grandes, pero podemos concluir que, en líneas generales, las operaciones de *insert* y *extract* implican un mayor costo que simplemente utilizar *shuffle* para reordenar los datos, probablemente porque implica utilizar dos instrucciones que utilizan la misma unidad de hardware, a diferencia de una.

3.2. Rotar

3.2.1. Implementación

Este filtro lo que hace es simplemente rotar los valores de los colores en los píxeles. Lo implementamos en muy pocas líneas en lenguaje ensamblador gracias a que utilizamos la instrucción *shuffle* junto a una máscara (escrita como constante en memoria) que indica la nueva posición de cada valor en el píxel de la imagen destino. Por lo tanto, lo único que se realiza en cada iteración es la carga de la imagen de memoria en un registro, ejecutar la instrucción *shuffle* y almacenar el registro en la posición de memoria del puntero a la imagen destino.

Como los valores en los píxeles no son alterados de ninguna forma, no hay pérdida de precisión alguna.

Se itera sobre la imagen columna por columna, y, una vez que se terminó de recorrer una fila, se decrementa el contador que contiene la cantidad de filas a recorrer. De esta manera, una vez que dicho contador llegó a cero, se terminó de recorrer la imagen. Como se opera de a 4 píxeles por vez, se divide la cantidad de columnas (que están en píxeles) por cuatro y los punteros avanzan de a 16 bytes. No se puede operar de a más píxeles por vez porque cada uno ocupa 32 bits, entonces solo se pueden almacenar cuatro de ellos por vez en los registros de SIMD.

La implementación realizada en C itera sobre la imagen con dos ciclos anidados (avanzando por columnas), de a un píxel por vez. Simplemente rota los valores de los píxeles en la imagen destino. Si bien las operaciones se realizan de manera muy similar a la implementación en lenguaje ensamblador, esta última recorre la imagen más rápido al procesar de a 4 píxeles por iteración, por lo que el rendimiento debería ser mayor.

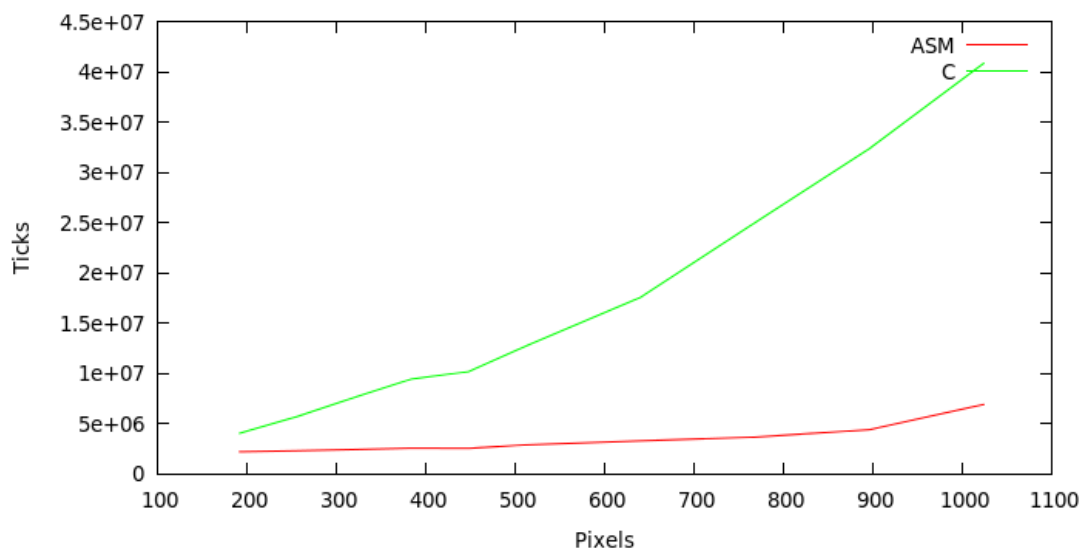


Figura 3: Comparación de los tiempos de ejecución de cada implementación

Como se puede observar en 3, la cantidad de *ticks* de reloj necesarios para ejecutar el mismo filtro en su implementación en C es considerablemente mayor a aquellos necesarios para ejecutar el filtro en su implementación en lenguaje ensamblador, como era de esperarse. La diferencia aumenta a medida que aumenta el tamaño de la imagen, donde se puede observar que la versión en C tiene una complejidad lineal, mientras que la implementación en lenguaje ensamblador lo hace en un orden menor.

Las mediciones se realizaron según lo descrito al principio del informe en Ubuntu 14.04, con 4GB de memoria RAM DDR2 y un procesador Intel Core 2 Quad Q9550 corriendo a una frecuencia de 2830MHz.

3.2.2. Experimentación

Hipótesis

La instrucción principal de este filtro, como mencionamos anteriormente, es *shuffle*. En esta experimentación queremos probar lo que ocurre si hacemos la rotación de valores de manera manual, simplemente desempaquetando y empaquetando los valores en las nuevas posiciones. Esto implica escribir más instrucciones y realizar un código más ilegible.

Como la cantidad de instrucciones ha aumentado considerablemente, creemos que el tiempo de ejecución será mayor, a pesar de que sean (probablemente) menos costosas que la operación *shuffle*. Aun así, debería ser menor a la implementación en C.

Resultados

Se realizó una medición de la cantidad de *ticks* de reloj del CPU que conllevó ejecutar el filtro para los distintos tamaños de entrada. Únicamente se varió el tamaño de las imágenes y no sus componentes porque las cuentas a realizar son independientes de los valores cromáticos de la imagen, por lo que se descartó que puedan haber variaciones apreciables dependiendo de la imagen en sí.

Las mediciones se realizaron según lo descrito al principio del informe en Ubuntu 14.04, con 4GB de memoria RAM DDR2 y un procesador Intel Core 2 Quad Q9550 corriendo a una frecuencia de 2830MHz.

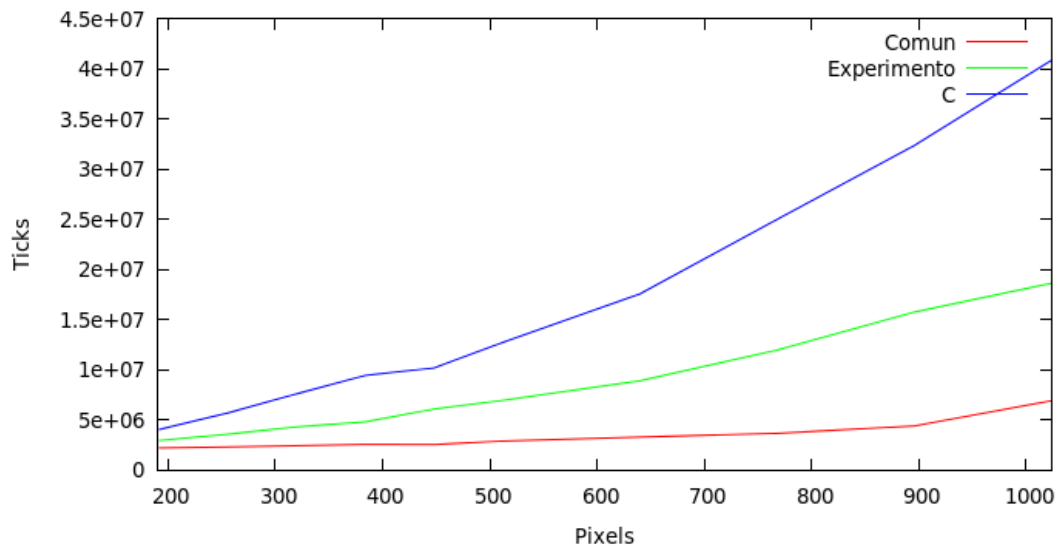


Figura 4: Comparación de los tiempos de ejecución

Los resultados coinciden con lo esperado, donde los tiempos de ejecución del filtro operando con la instrucción *shuffle* son menores al resto, y a su vez, la versión en C sigue siendo la menos eficiente, ya que opera píxel por píxel e implica más accesos a memoria. También podemos notar que las diferencias entre las implementaciones se hacen más notorias a medida que aumenta la cantidad de datos a procesar. Concluimos que el empaquetado y desempaquetado de datos conlleva un costo mayor para reordenar datos a su equivalente utilizando la instrucción *shuffle*.

3.3. Pixelar

3.3.1. Implementación

Como el filtro necesita la información de dos filas para realizar el promedio, recorreremos tanto la imagen fuente como la imagen destino con dos punteros. Estos se mueven de forma paralela, es decir, en las mismas columnas pero en filas consecutivas. Al trabajar de esta forma y utilizando instrucciones SIMD podemos aplicar los cambios a ocho píxeles a la vez (cuatro por punteros).

Para no perder precisión, todas las operaciones necesarias para obtener los promedios se realizan en enteros. Una vez calculados (uno por grupo de 4 píxeles), se reemplazan los valores originales de los píxeles en la imagen destino por el valor promedio correspondiente.

Se itera sobre la imagen columna por columna, y, una vez que se terminaron de recorrer las dos filas paralelas, se decrementa el contador que contiene la mitad de la cantidad total de filas a recorrer. De esta manera, una vez que dicho contador llegó a cero, se terminó de recorrer la imagen. Como se opera de a 4 píxeles por puntero, se divide la cantidad de columnas (que están en píxeles) por cuatro y los punteros avanzan de a 16 bytes. A su vez, cuando se terminan de recorrer las filas, los punteros avanzan la cantidad de columnas multiplicado por cuatro (bytes), saltándose cada puntero una fila. No se puede operar de a más píxeles por vez porque cada uno ocupa 32 bits, por lo que solo se pueden almacenar cuatro de ellos a la vez en los registros de SIMD.

La implementación realizada en C itera sobre la imagen con dos ciclos anidados (avanzando por columnas) de a 4 píxeles por vez, avanzando de a dos columnas y dos filas a la vez. Simplemente se calcula el promedio en cuatro píxeles y se coloca este valor en la imagen destino en las posiciones de los píxeles procesados. Si bien las cuentas se realizan de manera muy similar a la implementación en lenguaje ensamblador, esta última recorre la imagen más rápido al procesar de a 8 píxeles por iteración, por lo que el rendimiento debería ser mayor.

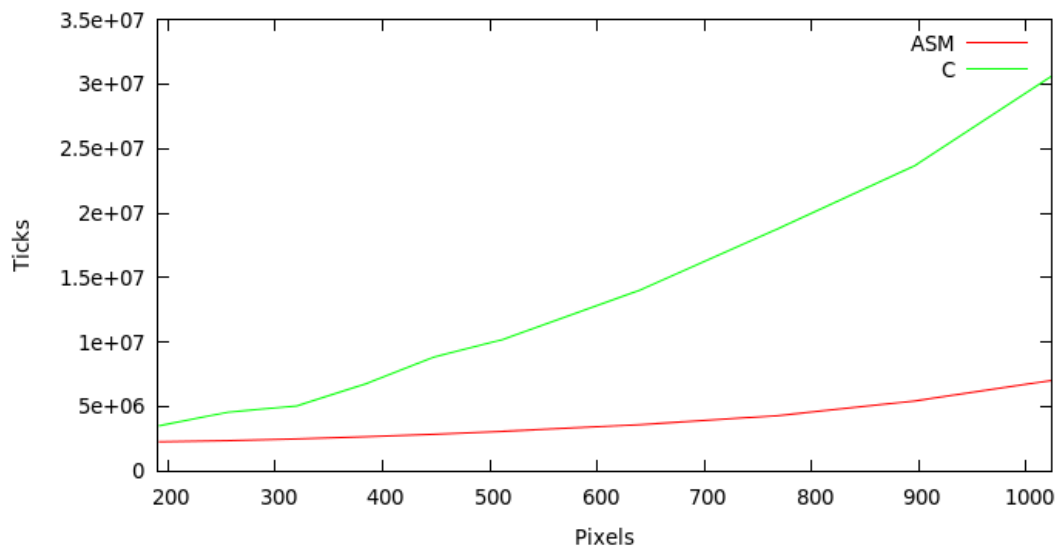


Figura 5: Comparación de los tiempos de ejecución de cada implementación

Como se puede observar en 5, la cantidad de *ticks* de reloj necesarios para ejecutar el mismo filtro en su implementación en C es considerablemente mayor a aquellos necesarios para ejecutar el filtro en su implementación en lenguaje ensamblador, como se suponía. La diferencia aumenta a medida que aumenta el tamaño de los datos de entrada, donde se puede observar que la implementación en C tiene una complejidad lineal, mientras que la implementación en lenguaje ensamblador lo hace en un orden más bajo.

Las mediciones se realizaron según lo descrito al principio del informe en Ubuntu 14.04, con 4GB de

memoria RAM DDR2 y un procesador Intel Core 2 Quad Q9550 corriendo a una frecuencia de 2830MHz.

3.3.2. Experimentación

Hipótesis

Dentro de las series de operaciones que ejecuta el filtro, existe una división entera que efectuamos con la instrucción *psrld*. En esta experimentación queremos ver la diferencia, tanto en tiempo como en precisión, al utilizar una división de punto flotante. Entendemos que en este caso no tiene sentido utilizarla, porque los decimales que perdemos al utilizar la división entera los ibamos a descartar de todas formas, ya que la información en los pixeles se encuentra en enteros. Aún así, nos interesa ver las diferencias entre ambos métodos. Suponemos que la diferencia en rendimiento no debería variar mucho, ya que la división en punto flotante se realiza a través de una tabla *hardcodeada*, y analizaremos la variación en la precisión (la cual debería ser peor para la división en punto flotante).

Resultados

Se realizó una medición de la cantidad de *ticks* de reloj del CPU que conllevó ejecutar el filtro para los distintos tamaños de entrada. Únicamente se varió el tamaño de las imágenes y no sus componentes porque las cuentas a realizar son independientes de los valores cromáticos de la imagen, por lo que se descartó que puedan haber variaciones apreciables dependiendo de la imagen en sí.

Las mediciones se realizaron según lo descrito al principio del informe en Ubuntu 14.04, con 4GB de memoria RAM DDR2 y un procesador Intel Core 2 Quad Q9550 corriendo a una frecuencia de 2830MHz.

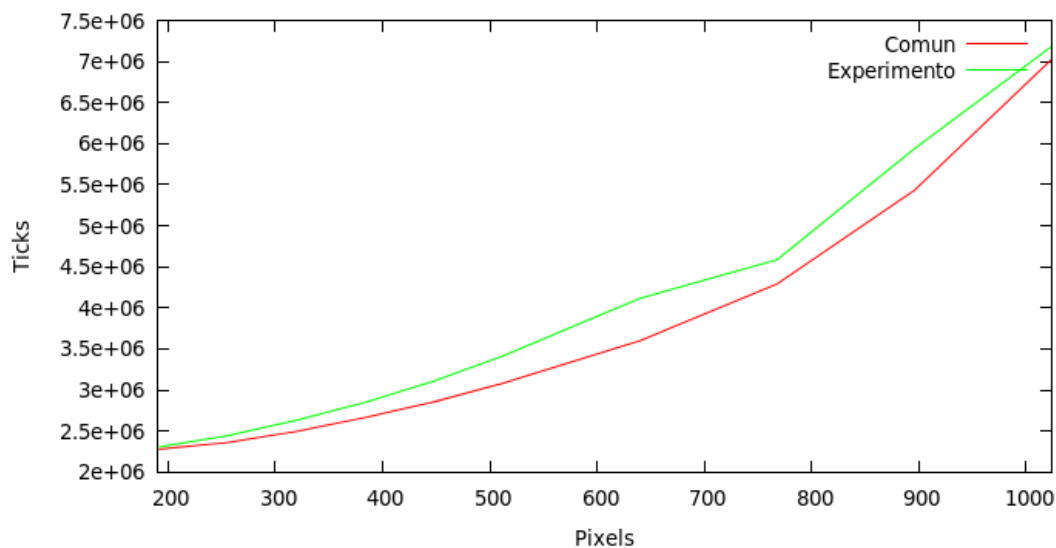


Figura 6: Comparación de los tiempos de ejecución

Los resultados indican que, si bien existe una tabla *hardcodeada* para las operaciones costosas como multiplicación y división en punto flotante, su costo sigue siendo mayor a aquél de operar la división *shifteando*, pero la diferencia no es tan notable como entre la implementación en C y en lenguaje ensamblador, denotada en la figura 5. Es decir, efectivamente el costo insumido no es tan grande, a pesar de que involucre operar con *floats*.

Una vez determinado que insume un mayor costo la operación con punto flotante, falta analizar la diferencia en la precisión.

Para analizar qué tanta diferencia hay entre un resultado y otro, utilizamos la herramienta *bmpdiff*, generando imágenes que muestran la diferencia píxel a píxel para cada componente. El parámetro epsilon de la herramienta *bmpdiff* se fijó en cero.

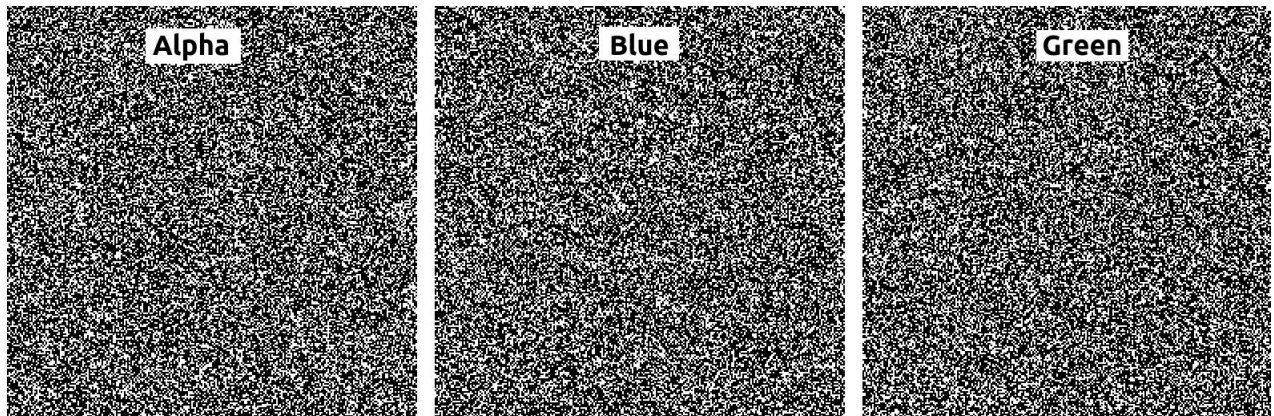


Figura 7: Diferencia píxel a píxel para cada componente

Como se puede observar, las diferencias en la imagen son uniformes, lo que confirma que el color no tuvo incidencia en las operaciones. Si bien parecen ser muchos los píxeles con distintos valores, la diferencia es prácticamente nula: al ejecutar *bmpdiff* una vez más pero con el parámetro *epsilon* en 1 (es decir, permitiendo que haya un margen de error de ± 1 para cada color) se obtuvieron imágenes en negro para todas las componentes. En otras palabras, la diferencia de precisión es mínima, ya que solo difieren en uno cada componente de cada píxel. Sabiendo esto podemos concluir que a pesar de ser operaciones aproximadas, las mismas solo difieren con el cálculo exacto por ± 1 , lo que implica que es únicamente un tema de redondeo/truncamiento.

3.4. Combinar

3.4.1. Implementación

Ya que el filtro se ejecuta sobre el reflejo vertical de la imagen fuente, en lenguaje ensamblador se mantiene como invariante un puntero a la imagen invertida. Es decir, a medida que se recorre la matriz de la imagen fuente, este puntero la recorre de atrás hacia delante. Por lo tanto, avanza restándole posiciones de memoria. Por otro lado, como recorre de atrás hacia delante, levanta los píxeles de memoria de manera invertida, por lo que en cada iteración del ciclo, luego de levantar los píxeles, se reordenan de manera que coincidan vectorialmente con aquellos contenidos en el registro que opera sobre la matriz comúnmente (es decir, de delante hacia atrás).

Para no perder precisión, antes de realizar la resta entre píxeles, se extiende cada componente del píxel a un entero de 32 bits y se hacen todas las operaciones intermedias operando como *floats*. Además, como la cuenta incluye el uso de constantes, las mismas se guardan en memoria como datos de solo lectura. Para no perder rendimiento levantándolas en cada iteración del ciclo, se guardan en un registro específico antes de empezar a recorrer la imagen.

Se itera sobre la imagen columna por columna, y, una vez que se terminó de recorrer una fila, se decrementa el contador que contiene la cantidad de filas a recorrer. De esta manera, una vez que dicho contador llegó a cero, se terminó de recorrer la imagen. Como se opera de a 4 píxeles por vez, se divide la cantidad de columnas (que están en píxeles) por cuatro y los punteros avanzan de a 16 bytes. No se puede operar de a más píxeles por vez porque cada uno ocupa 32 bits, entonces solo se pueden almacenar cuatro de ellos por vez en los registros de SIMD.

La implementación realizada en C itera sobre la imagen con dos ciclos anidados (avanzando por columnas), de a un píxel por vez. Las operaciones se realizan con una variable auxiliar de tipo *float* y luego se castea la misma al tipo de dato de la estructura correspondiente al color del píxel (*unsigned char*). Si bien las cuentas se realizan de manera muy similar a la implementación en lenguaje ensamblador, esta última recorre la imagen más rápido al procesar de a 4 píxeles por iteración, por lo que el rendimiento debería ser mayor.

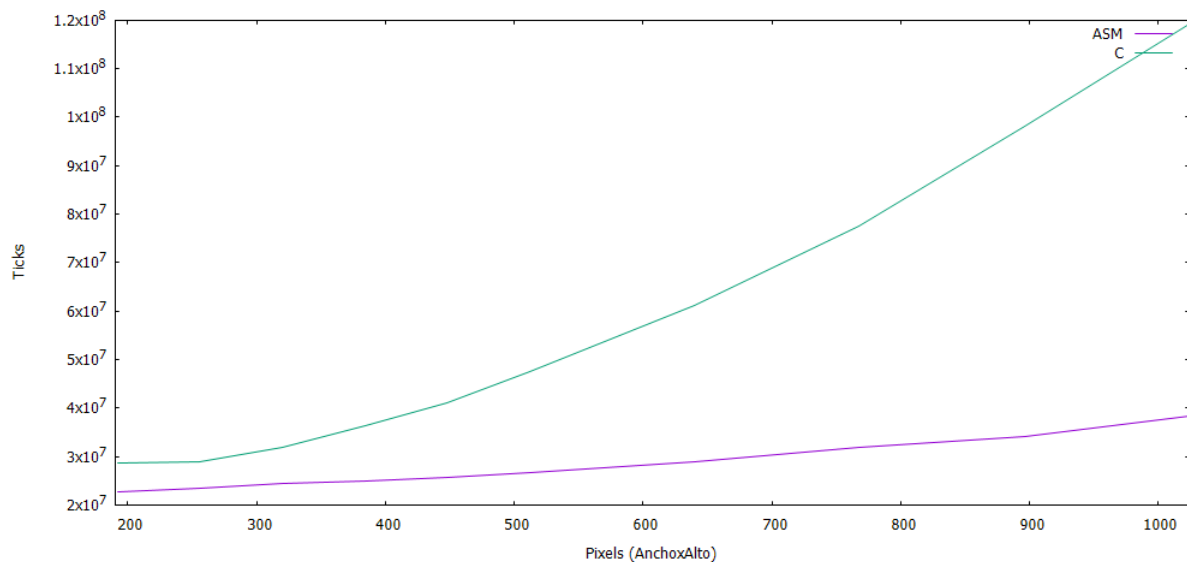


Figura 8: Comparación de los tiempos de ejecución de cada implementación

Como se puede observar en 8, la cantidad de *ticks* de reloj necesarios para ejecutar el mismo filtro en su implementación en C es considerablemente mayor a aquellos necesarios para ejecutar el filtro en su implementación en lenguaje ensamblador, como era de esperarse. La diferencia aumenta a medida que aumenta el tamaño de la imagen, donde se puede observar que la implementación en C tiene una

complejidad lineal, mientras que la implementación en lenguaje ensamblador lo hace en un orden más bajo.

Las mediciones se realizaron según lo descrito al principio del informe, con un parámetro *alpha* de 128, en Ubuntu 16.04, con 1GB de memoria RAM DDR3 y un procesador Intel Core i3-2100 corriendo a una frecuencia de 3096MHz.

3.4.2. Experimentación

Hipótesis

Ya que el filtro consta de ejecutar una serie de operaciones, la mayor complejidad del mismo se encuentra en las cuentas que hay que realizar. Si bien la suma y resta de enteros no insume prácticamente ningún costo, dentro de las operaciones a realizar se encuentra una multiplicación con valores de punto flotante (*alpha*) y la división por 255.0. Ya que *alpha* representa un valor comprendido en $[0.00;255.0]$, si se lo interpreta como un entero, el truncamiento no representa una pérdida de precisión muy importante, y el rendimiento obtenido debería compensar tal pérdida. Así mismo, se puede realizar algo similar con la división por 255.0. Si bien la división por enteros es costosa, es posible dividir el resultado parcial obtenido por 256 en lugar de 255. La operación no insume un mayor costo porque simplemente consta de *shiftear* los bits, en lugar de realizar una división propiamente dicha. Además, ya que la operación se realizaría únicamente con enteros, no es necesario hacer *casteos* intermedios entre punto flotante y enteros, los cuales son muy costosos.

Resultados

Se realizó una medición de la cantidad de *ticks* de reloj del CPU que conllevó ejecutar el filtro para los distintos tamaños de entrada. Únicamente se varió el tamaño de las imágenes y no sus componentes porque las cuentas a realizar son independientes de los valores cromáticos de la imagen, por lo que se descartó que puedan haber variaciones apreciables dependiendo de la imagen en sí. Por otro lado, el parámetro *alpha* se fijó en 128.

Las mediciones se realizaron según lo descrito al principio del informe, en Ubuntu 16.04, con 1GB de memoria RAM DDR3 y un procesador Intel Core i3-2100 corriendo a una frecuencia de 3096MHz.

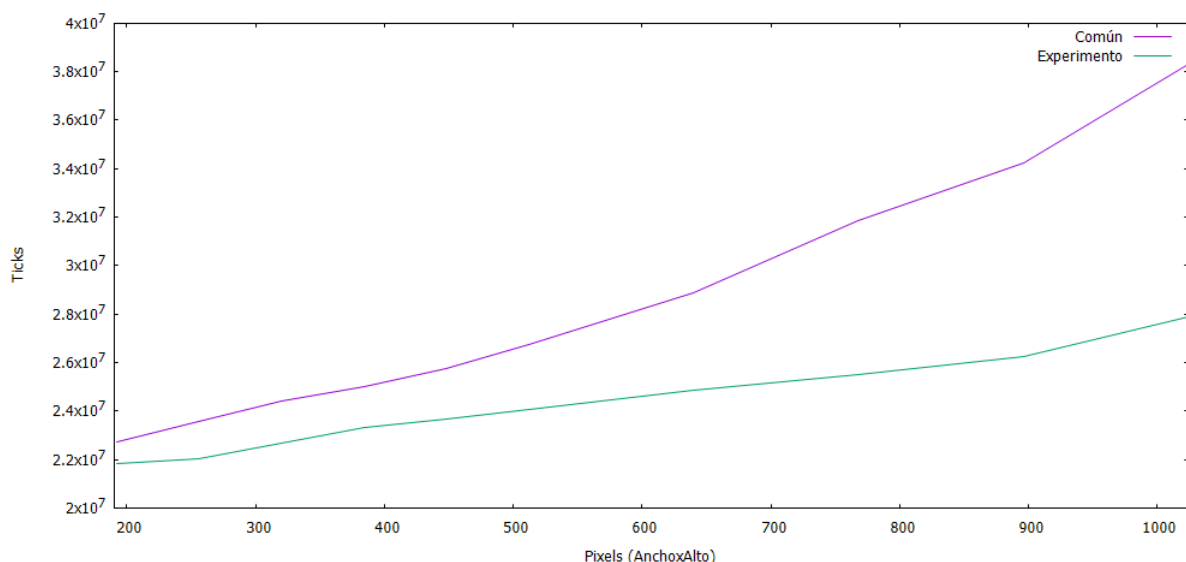


Figura 9: Comparación de los tiempos de ejecución

Los resultados concuerdan con lo esperado, donde los tiempos de ejecución del filtro operando únicamente con enteros son menores a aquellos utilizados para operar sin pérdida de precisión. La diferencia

se hace más notable a medida que aumenta el tamaño de la imagen, pero no es tan notable como la diferencia entre la implementación en C y en lenguaje ensamblador, denotada en la figura 8.

Sin embargo, obteniendo la diferencia porcentual entre los distintos tiempos de ejecución, se obtiene que para datos de entrada chicos, la mejora ronda entre el 4 y el 8 %, mientras que para los tamaños de imagen más grande, la diferencia entre operar con enteros y operar con números en punto flotante de precisión simple llega al 40 %. Queda claro que el rendimiento mejora al operar con enteros, pero falta analizar cuánta precisión se pierde.

Para analizar qué tanta diferencia hay entre un resultado y otro, utilizamos la herramienta *bmpdiff*, generando imágenes que muestran la diferencia píxel a píxel para cada componente. Las imágenes se obtuvieron corriendo el filtro con el parámetro *alpha* de valor 127.484, para realzar las diferencias por pérdida de precisión. El parámetro epsilon de la herramienta *bmpdiff* se fijó en cero.

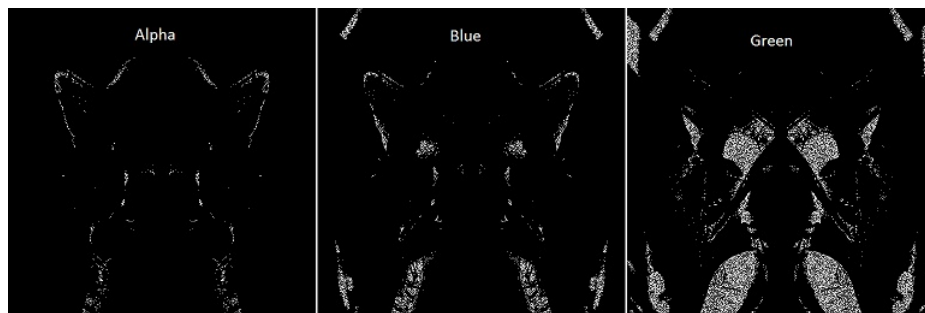


Figura 10: Diferencia píxel a píxel para cada componente

Como se puede observar, la mayor diferencia se obtiene en la componente verde, mientras que la roja (no incluida en la figura) no poseía diferencias. Si bien parecen ser varios los píxeles con distintos valores, la diferencia es prácticamente nula: al ejecutar *bmpdiff* una vez más pero con el parámetro epsilon en 1 (es decir, permitiendo que haya un margen de error de \pm uno para cada color) se obtuvo imágenes en negro para todas las componentes. En otras palabras, la pérdida de precisión es mínima, ya que solo difieren en uno cada componente de cada píxel, mientras que la mejora en rendimiento es notable, especialmente para imágenes de mayor tamaño. Por lo tanto, se concluye que es preferible utilizar la implementación con enteros si la precisión no es muy importante, especialmente teniendo en cuenta la mejora en tiempos de ejecución para imágenes grandes.

3.5. Colorizar

3.5.1. Implementación

El filtro Colorizar trabaja con los valores máximos de los colores de aquellos píxeles que se encuentren alrededor del píxel a procesar, incluyéndolo. Por esta razón, se mantuvo un puntero a la fila anterior y posterior del píxel a procesar, de manera tal de poder aprovechar las instrucciones de SIMD para realizar varias comparaciones simultáneamente. Por otro lado, como las operaciones a realizar incluyen el uso de constantes, las mismas se cargaron en memoria (en la sección de datos de solo lectura) y luego a los registros al inicio del programa. Además, por enunciado, la primera y última fila, así como la primera y última columna, no se procesan. Por lo tanto, el puntero a la imagen destino comienza una fila y una columna más adelante. Lo mismo ocurre para el puntero de la imagen fuente que contiene los píxeles a procesar.

Debido a que no se procesa la primera y última columna (siendo la cantidad total de columnas múltiplo de cuatro), y solo se pueden realizar comparaciones de a cuatro píxeles por vez, el algoritmo implementado en lenguaje ensamblador procesa de a dos píxeles por iteración.

Luego de obtener los valores máximos de cada color de los píxeles a procesar y sus vecinos, se comparan entre ellos generando máscaras. Estas mismas máscaras aprovechan el valor *true* de las comparaciones interpretándolas como un -1 y, luego de obtener su complemento, agregando el valor 1 donde haya ceros. De esta manera, al multiplicar por el registro que contiene al parámetro α , se obtiene $-\alpha$ en aquellos lugares donde la comparación haya resultado falsa y α en aquellos lugares donde la comparación haya resultado verdadera. Luego solo resta sumar por la constante 1,0 cargada previamente en un registro.

Para obtener los mínimos entre los resultados obtenidos y el valor 255, se realiza una operatoria muy similar utilizando máscaras junto con la operación lógica AND y luego sumando. El valor *alpha* original de los píxeles es copiado al resultado una vez realizadas las operaciones.

En el caso de la implementación realizada en C, la iteración sobre la matriz opera de manera muy similar, saltando la primera y última fila y la primera y última columna de cada fila, con la diferencia de que se procesa de a un píxel por vez. Además, para buscar el máximo, se realiza otro bucle iterando sobre los vecinos del píxel, por lo que se espera que la diferencia de rendimiento entre una implementación y otra sea considerable (ya que realiza bastantes más accesos a memoria que la implementación en lenguaje ensamblador).

Solo se varió el tamaño de las imágenes a probar porque, si bien los máximos dependen del valor de los colores, las comparaciones se efectúan independientemente de sus valores, y la operación a realizar (suma o resta) según los resultados de las mismas es equivalente en términos de rendimiento.

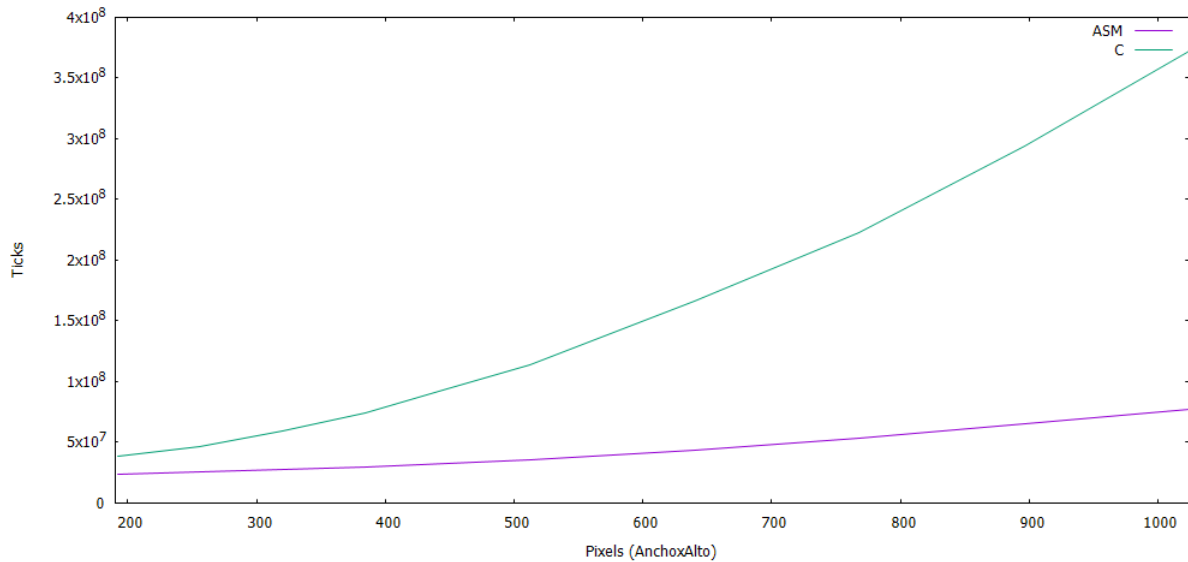


Figura 11: Comparación entre las distintas implementaciones

Como se puede observar en la figura 11, la diferencia entre la implementación en lenguaje ensamblador y aquella en C es muy considerable, donde la implementación en C consume muchos más *ticks* de reloj a medida que aumenta el tamaño de los datos de entrada y su contraparte lo hace en un orden bastante menor. Suponemos que la diferencia es más notable que en otros filtros (teniendo en cuenta que se procesa únicamente de a dos pixeles por iteración en lugar de a cuatro, como los demás) por la cantidad de accesos a memoria que requiere la implementación en C para buscar el máximo.

3.5.2. Experimentación

Hipótesis

Dado que el filtro se caracteriza por buscar los máximos en un cuadrado alrededor del píxel a procesar, muchos resultados se solapan. Es decir, los máximos de cada columna se calculan más de una vez. Si bien esto no representa un problema para la implementación en lenguaje ensamblador (ya que las comparaciones se realizan de manera vectorial y los accesos a memoria no se reducen), para la implementación en C se puede reducir considerablemente los accesos a memoria si se reusan los resultados, manteniendo los resultados parciales (como una suerte de programación dinámica). De esta manera, excepto para la primera iteración de cada fila, ya se consiguieron los máximos de dos de las tres columnas a evaluar.

Almacenando estos resultados parciales, se reducen los accesos a memoria en un orden no menor, dado que solo es necesario calcular el máximo de la columna siguiente al píxel a procesar. En total, la cantidad de accesos a memoria deberían reducirse en 2/3 (aproximadamente, ya que para la primera iteración de cada fila sigue siendo necesario calcular los máximos de las tres columnas).

Los accesos a memoria deberían conformar una parte considerable del costo insumido por la implementación en C, por lo que se espera que, al reducirlos, la diferencia de tiempos entre la implementación en ensamblador y la implementación en C sea menor a aquella de los demás filtros (dado que en este filtro se procesa de a dos en lugar de a cuatro).

Resultados

Para analizar lo dicho, se ejecutó ambas implementaciones en C de la manera descrita al principio del trabajo práctico y se comparó ambos resultados.

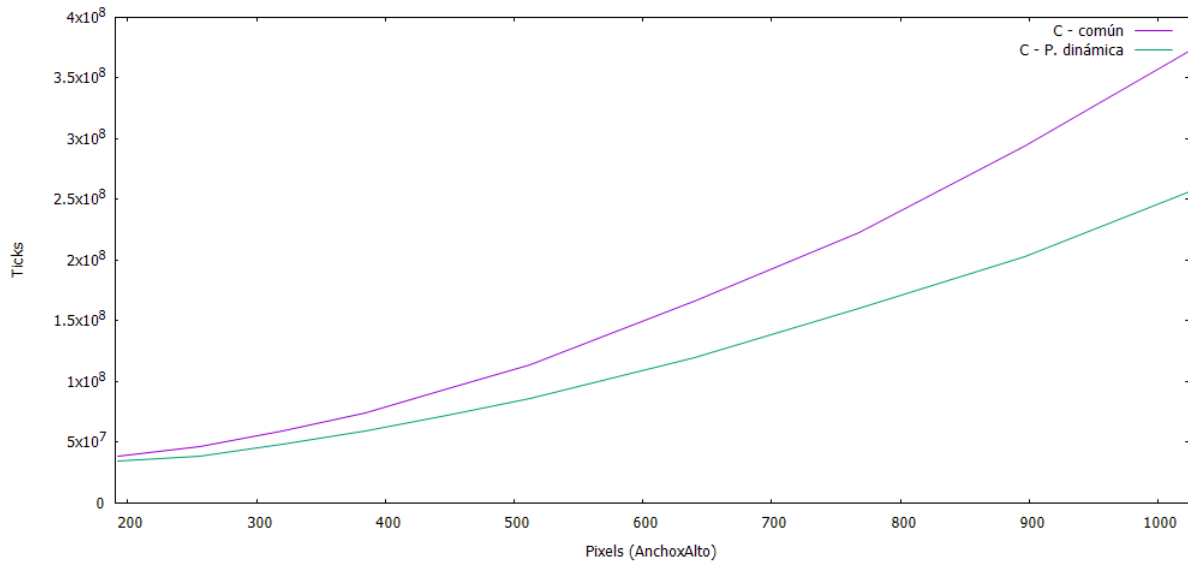


Figura 12: Comparación entre las distintas implementaciones en C

Como se puede observar en la figura 12, efectivamente se redujo el tiempo de ejecución, acentuándose la diferencia para imágenes de mayor tamaño. Calculando la diferencia porcentual entre ambas implementaciones, para imágenes de menor tamaño se obtuvo una mejora promedio del 15 % y para datos de entrada considerablemente más grandes, se mejoró en un 45 % aproximadamente. Comparando el rendimiento con la implementación en lenguaje ensamblador, se obtuvo lo siguiente:

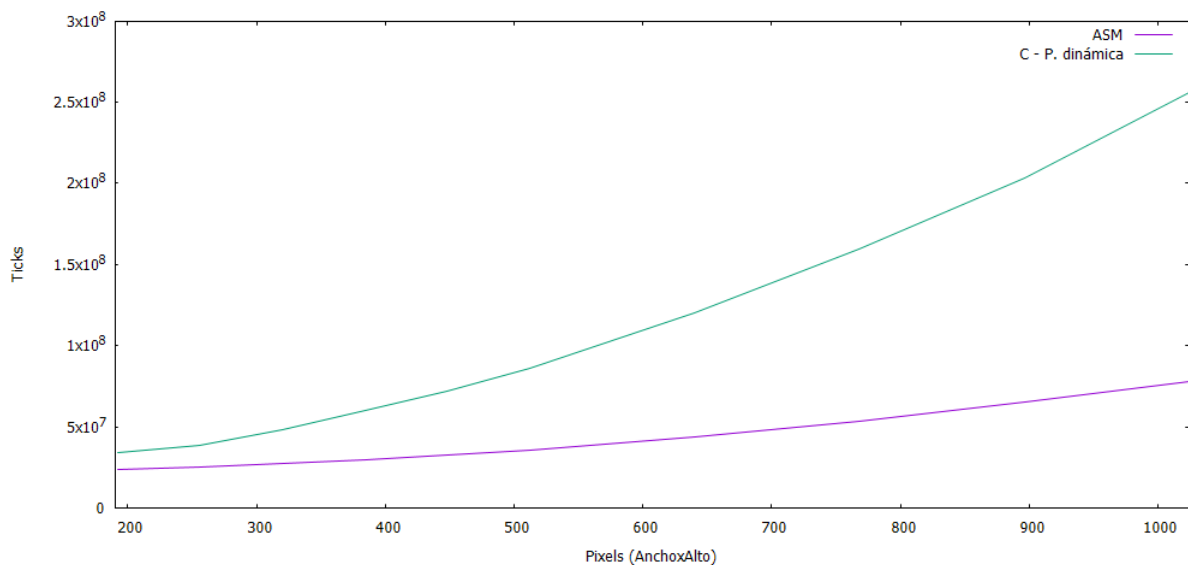


Figura 13: Comparación entre las distintas implementaciones

En esta figura es fácilmente observable que la diferencia entre la implementación en lenguaje ensamblador y su análoga en C es mucho menor a comparación de los resultados obtenidos en la figura 11. Estos resultados conciden con lo planteado anteriormente, además de denotar una diferencia entre implementaciones menor a comparación de los demás filtros, ya que estos procesan de a cuatro pixeles mientras que este solo lo hace de a dos, comprobando lo dicho en la hipótesis.