DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Filtros SIMD

Organización del Computador II

Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Jerónimo Dacunda Ratti	710/18	jero.d.r22@gmail.com
José Gutiérrez	459/00	jagj77@gmail.com
Schiavinato Mauro	299/19	maurolschiavinato@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

El presente trabajo tiene por finalidad exponer los resultados obtenidos durante la implementación de los filtros de imágenes: ImagenFantasma, ColorBordes, ReforzarBrillo y PixeladoDiferencial. Analizaremos las diferencias entre la versión en C con distintas optimizaciones contra la nuestra en unassembler con instrucciones de vectorización. Luego haremos varios experimentos sobre las implementaciones de estos filtros.

Índice

1. Introducción

El objeto de este informe es adjuntar, clarificar y complementar la implementación de filtros sobre imágenes utilizando las intrucciones SIMD de la arquitectura x64 en assembler como solicitó la cátedra. La especificación y una version en C de los filtros fue porporcionada por la cátedra. Este trabajo también contiene una comparación entre la implementación en C con distintas opciones de optimización y la versión en assembler de los mismos.

2. Desarrrollo

A continuación se delinean las funciones de filtro realizadas y su operatoria, haciendo hincapié en las técnicas que utilizamos para vectorizar el algoritmo y así aprovechar las instrucciones SIMD.

2.1. Imagen Fantasma

Figura 1: Filtro Imagen Fantasma

2.2. Color Bordes

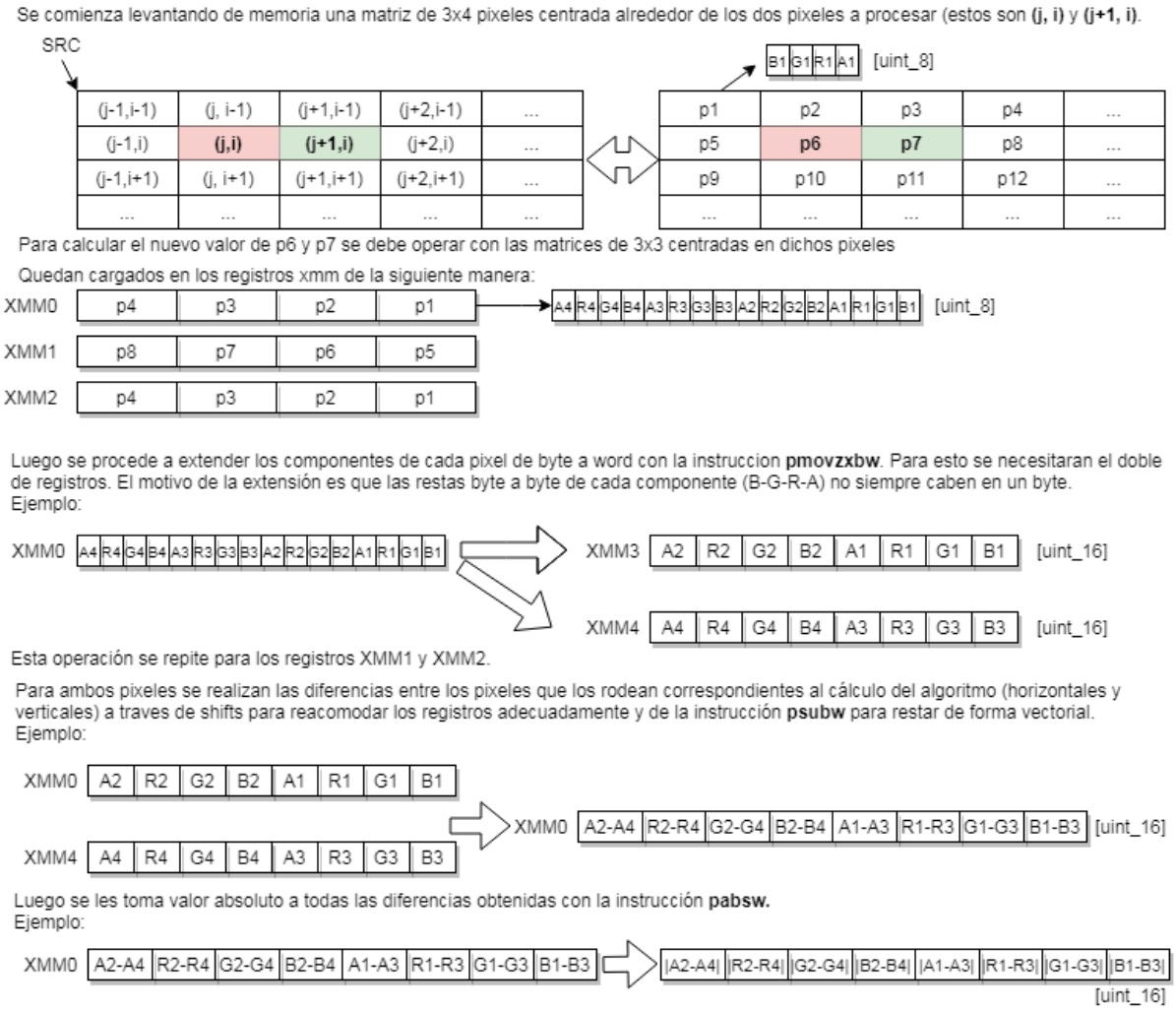
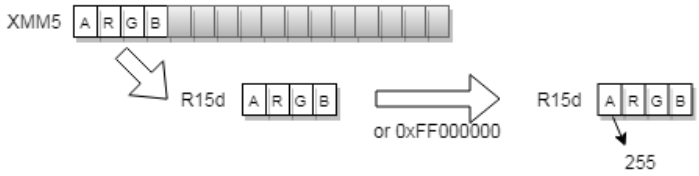


Figura 2: Filtro Color Bordes (parte 1)

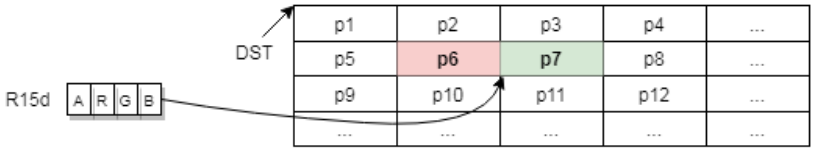
Repitiendo la instrucción **paddusw** (tras shifts para reacomodar los registros debidamente) se acumulan las sumas correspondientes a cada componente de manera saturada hasta obtener:

Donde B, G y R son los resultados finales para cada componente correspondiente al procesamiento del pixel **p7** (el segundo de los pixeles procesados). Respecto al componente Alpha, este se vio afectado por todas las operaciones al igual que los otros tres, pero su valor final siempre debe ser 255.
A continuación se empaqueta el pixel pasando sus componentes de tamaño word nuevamente a byte (de forma saturada) con la instrucción **packuswb** obteniendo:

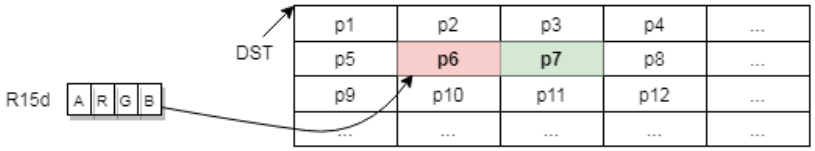
Para devolver el valor del componente A a 255, se extrae el pixel hacia un registro de propósito general con la instrucción **pextrd** y se realiza un **or** con la constante 0xFF000000.



Finalmente se mueve el pixel a memoria, a la imagen destino.



Se repiten los pasos de suma saturada, extracción y restablecimiento a 255 del componente A para el otro pixel procesado (**p6**) y se lo mueve también al destino en memoria.



Por último, al llegar al final de cada fila, se mueve a destino dos pixeles blancos (correspondientes al borde derecho de la fila actual y al borde izquierdo de la fila siguiente). Para ello se utiliza la constante 0xFFFFFFFFFFFFFFFF (dos pixeles con sus 4 comp. en 255). Tras finalizar el procesamiento de toda la imagen, se utiliza la misma constante para poner en blanco todo el borde superior e inferior.

Figura 3: Filtro Color Bordes (parte 2)

2.3. Reforzar Brillo

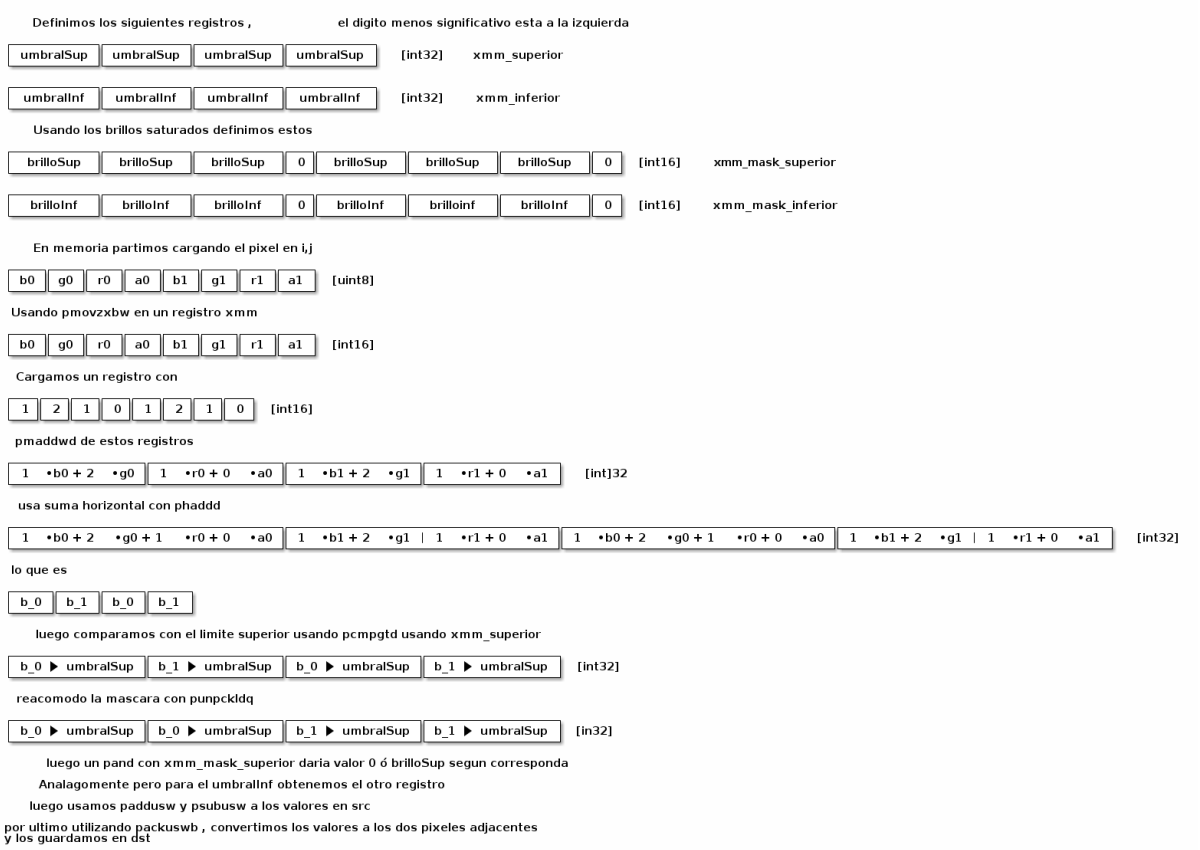


Figura 4: Filtro Reforzar Brillo

2.4. Pixelado Diferencial

En un loop tomamos las primeras cuatro líneas y vamos sumando en el registro xmm0 la suma. Luego sumamos horizontalmente copiandolo a otro registro, shifteando y sumandolos. Luego la sumatoria que nos queda en la mitad inferior del registro la copiamos al superior. Luego dividimos por 16 cada word y obtenemos el promedio en la parte inferior y superior del registro xmm0.

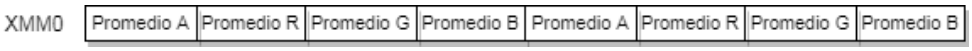


Figura 5: Pixelado Diferencial, XMM0

En un loop tomamos las primeras cuatro líneas y vamos sumando en el registro xmm1 la diferencia de la línea actual con el promedio. En cada paso tomamos valor absoluto.

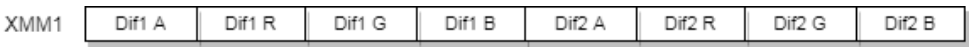


Figura 6: Pixelado Diferencial, XMM1

Luego sumamos horizontal y extraemos la diferencia a un registro.

Figura 7: Pixelado Diferencial, XMM2

Comparamos con el umbral que nos pasaron por parámetro y si es menor entonces copiamos las líneas de la imagen del source al destino. Si es mayor entonces copiamos el promedio que teníamos guardado en xmm0 en todos las líneas del destino.

3. Comparación C vs ASM

A continuación se exhiben, para cada filtro, comparaciones de performance entre la implementación en assembler y en C con diferentes niveles de optimización. Para generar las muestra y facilitar su reproducibilidad, automatizamos con archivos bash las compilaciones con distintos parametros.

Además modificamos el código principal para generar archivos log y también para mostrar el desvío estándar de todas las mediciones. Estos log son procesados en notebooks de python para realizar los graficos. En los graficos utilizamos la media aritmética sobre todas las imágenes proporcionadas por la cátedra y mostramos el intervalo de confianza a dos desviaciones estandar (suponemos que la variable a medir tiene distribución normal, y dos desviaciones estandar tomaría el 95 % de los resultados, así descontando los outliers). Aunque en realidad no es cierto que la variable aleatoria que tomamos tenga distribución normal porque el performance de un filtro depende también de la naturaleza de la imágen (por ejemplo en pixelado diferencial hay una condición dependiendo de los píxeles), observamos que se terminan comportando como una normal, ya que enrealidad la cuestión más importante en el tiempo de ejecución termina siendo la cantidad de píxeles que tiene que procesar. Al tomar el algoritmo sobre todas las imágenes y promediarlo terminamos obteniendo en el gráfico una medida unificada sobre como se comporta en general para todas las imágenes.

Para las imágenes de los gráficos utilizamos todas

3.1. Imagen Fantasma

Podemos observar en la Figura 8 que nuestro código en assembler es 3x veces más rápido que la implementación de C con optimizaciones O3.

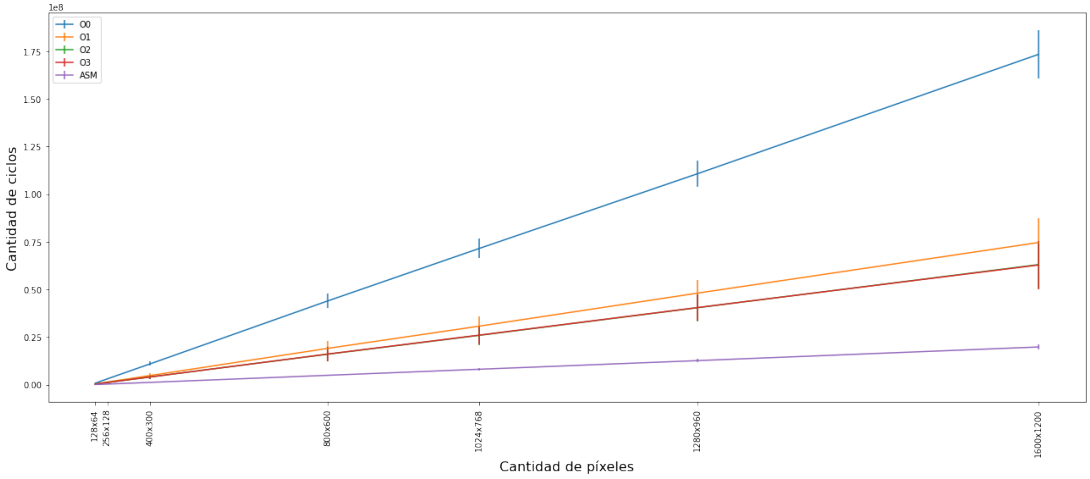


Figura 8: Gráfico ciclos vs distintas resoluciones de imágenes para filtro Imagen Fantasma
Notar que O2 está debajo de O3

3.2. Color Bordes

Podemos observar en la figura 9 que nuestro código en assembler es hasta 2x veces más rápido que la implementación en C con optimizaciones O3.

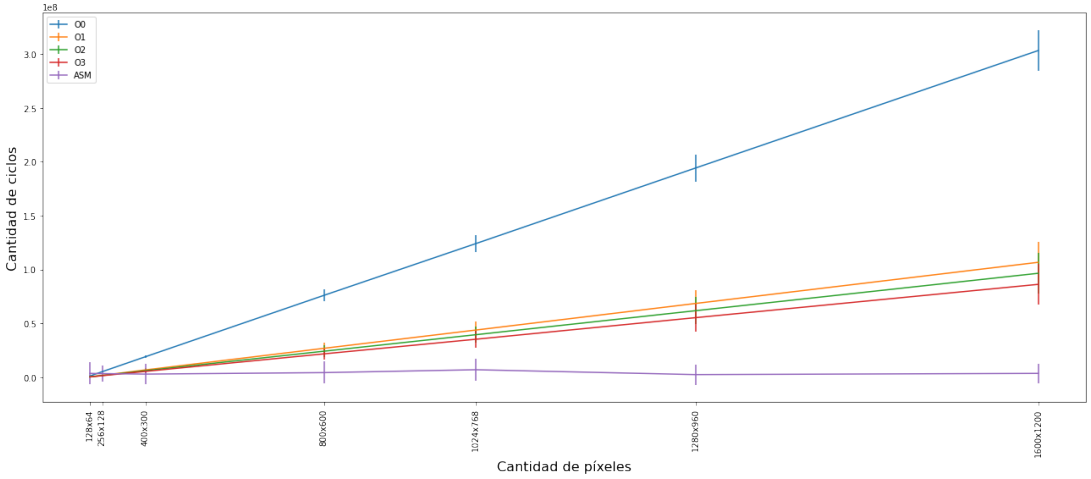


Figura 9: Gráfico ciclos vs distintas resoluciones de imágenes para filtro Color Bordes

3.3. Reforzar Brillo

Podemos observar en la figura 10 que nuestro código en assembler es 5x veces más rápido que la implementación en C con optimizaciones en O2 y hasta 10x veces más rápido que O1.

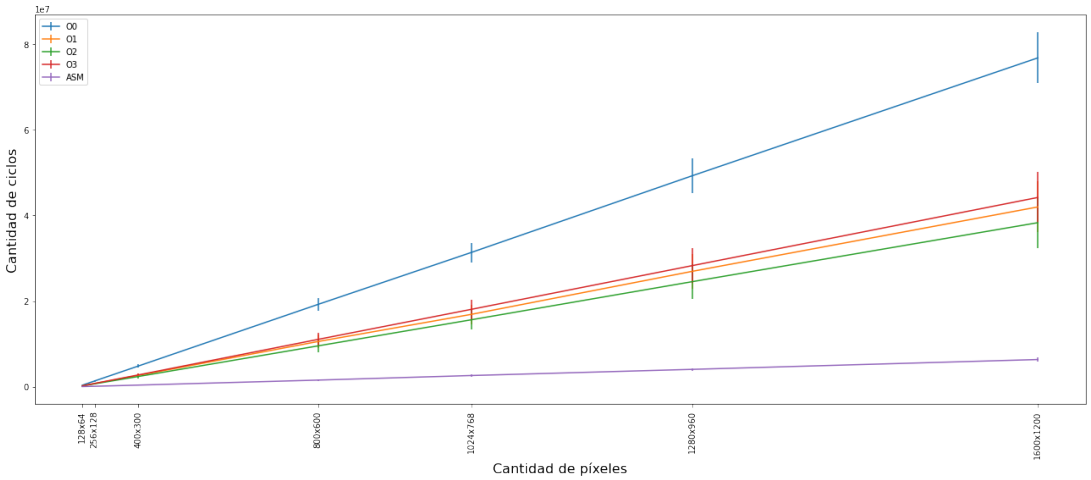


Figura 10: Gráfico ciclos vs distintas resoluciones de imágenes para filtro Reforzar Brillo

3.4. Pixelado Diferencial

Podemos observar en la figura 11 que nuestro código en assembler es 3x veces más rápido que la implementación en C con optimizaciones O3. Y si lo comparamos contra O1 entonces es 7x veces más rápido.

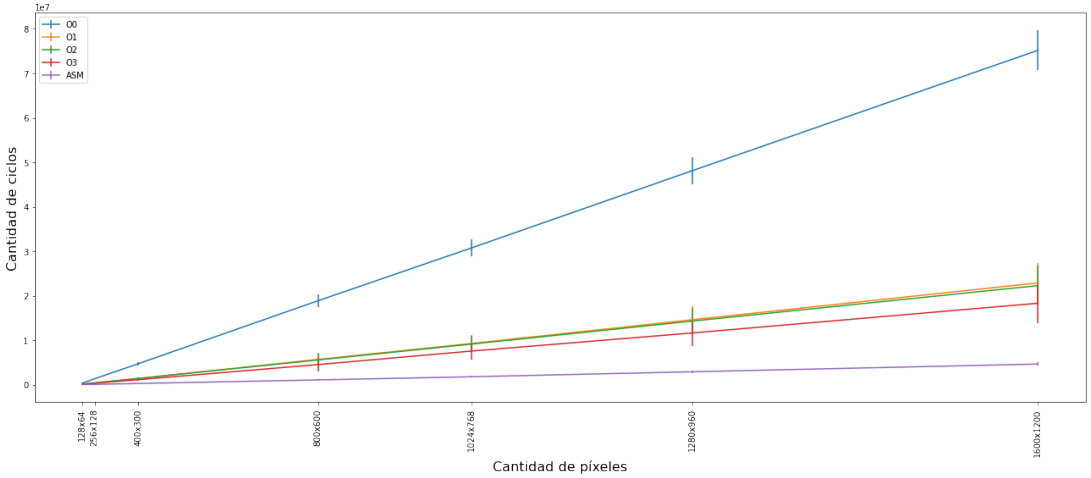


Figura 11: Gráfico ciclos vs distintas resoluciones de imágenes para filtro Pixelado Diferencial

3.5. Optimizaciones de GCC e Instrucciones XMM

Algo interesante para realizar es inspeccionar el código de los filtros generado por el gcc con nivel de optimizacion O3. Para ello utilizamos el comando objdump. Pudimos observar que en el único filtro que utiliza instrucciones xmm (instrucciones vectoriales) es en ImagenFantasma. En el resto utiliza instrucciones comunes. Nosotros creemos que esto pasa porque GCC no se dió cuenta que podía optimizar el código mediante instrucciones vectoriales. Podemos también observar que el código de C de ImagenFantasma también es el más simple, teniendo solo dos loops y realizando unas pocas operaciones adentro sin ninguna condición.

3.6. Rendimiento sobre distintas relaciones de aspecto

Nos pareció interesante medir el rendimiento de los filtros sobre distintas relaciones de aspecto de imagenes. La idea preliminar que teníamos es que en los casos donde las imagenes son mucho más largas que altas (relación aspecto mayor) se procesarian más rápidamente que las otras. Suponiamos esto porque cada vez que se cambia de fila cambia el flujo del código, rompiendo el pipeline del procesador.

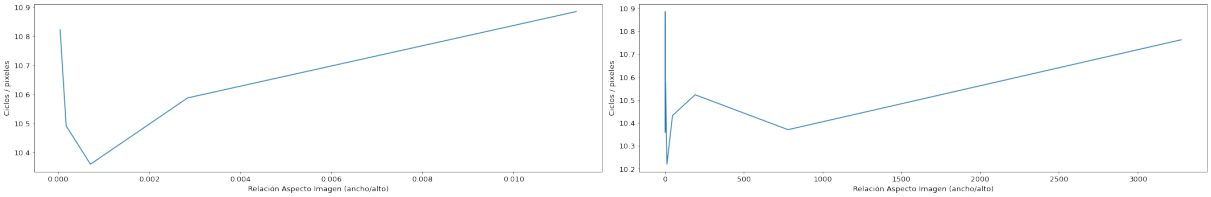
Para testear los filtros por distintas relaciones de aspecto fuimos convirtiendo una imagen de la cátedra por varios relaciones de aspecto mediante el programa imagemagick mediante el comando convert. Luego tomamos 1000 iteraciones del programa para cada imagen y tomamos el promedio. Luego este promedio lo dividimos por la cantidad de píxeles de la imagen. Para los errores utilizamos el doble de la desviación estándar y luego lo dividimos por la cantidad de píxeles.

Estas mediciones terminamos dividiendo sobre la cantidad de píxeles porque al ir cambiando la relación de aspecto de las imágenes es bastante dificil obtener imagenes con la misma cantidad de píxeles, siempre obteníamos un poco más o un poco menos. Esto sucede porque las imagenes tienen que formar con todos los píxeles un rectángulo perfecto.

Para las relaciones de aspecto fuimos agrandando el ancho por 2 en cada iteración.

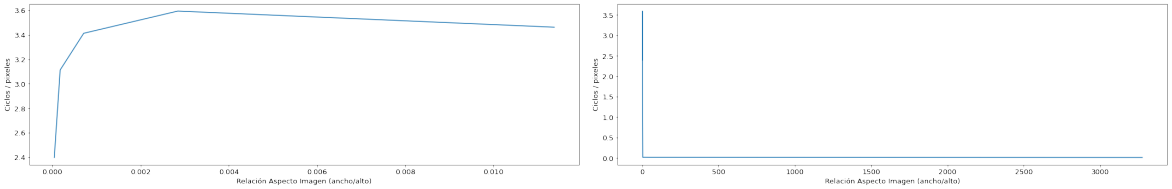
Podemos observar en las 12, 13, 14, 15 los gráficos de cada filtro donde el eje Y es los ciclos por pixel y el eje X es la relación de aspecto. Como la relación de aspecto es una división, entonces muchas valores que podemos tomar dentro de los valores de ancho y alto caen en números entre 0 y 1, y muchos valores terminan siendo muy altos (tan altos como la cantidad de píxeles de la imagen). Por eso decidimos realizar dos gráficos por cada filtro, donde en uno vemos los valores más chicos y en otro los más grandes.

Podemos notar que no hay ningún patrón muy definido, solamente que en las relaciones de aspecto más chicas generalmente el rendimiento es peor. Esto podemos creer que pasa porque es un patrón atípico tener un for dentro de otro for que tenga muy pocas iteraciones, haciendo que el pipeline del procesador no rinda bien. Igual podemos observar por las escalas, que la diferencia de rendimiento es bastante poca, lo cual hace que no nos tengamos que preocupar mucho por este tema.



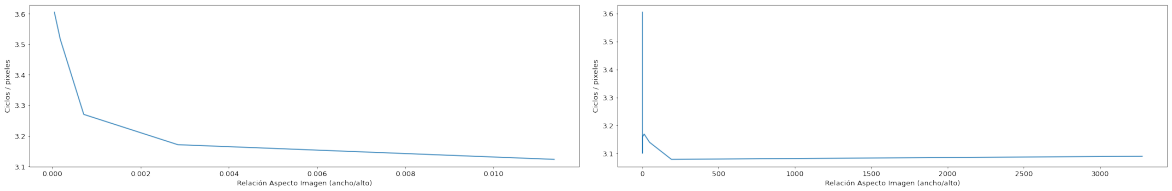
(a) ImagenFantasma (b) ImagenFantasma 2

Figura 12: Ciclos/pixeles vs relación de aspecto



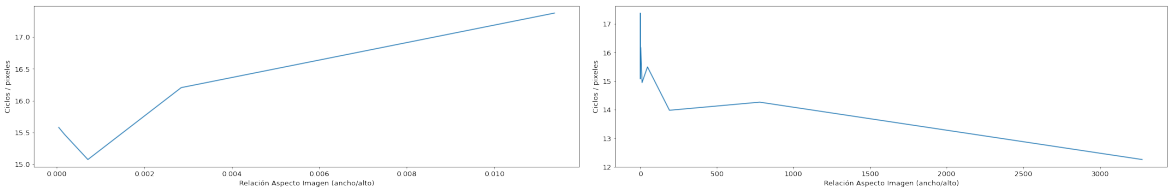
(a) ColorBordes (b) ColorBordes 2

Figura 13: Ciclos/pixeles vs relación de aspecto



(a) ReforzarBrillo (b) ReforzarBrillo 2

Figura 14: Ciclos/pixeles vs relación de aspecto



(a) PixeladoDiferencial (b) PixeladoDiferencial 2

Figura 15: Ciclos/pixeles vs relación de aspecto

4. Experimentación y análisis

Toda la experimentación fue realizada en notebooks de jupyter que dejamos al alcance del lector en la carpeta notebooks. Para la compilación de las distintas versiones del programa se utilizan scripts de shell. Los notebooks de jupyter tienen en la primer celda tienen los comandos de ejecución de los scripts para compilarlos. Para poder tener varias versiones del programa utilizamos flags con %ifdef en los archivos de assembler, por lo cual en los archivos fuente de los algoritmos se van a encontrar varias veces el código pero aplicadas las diferencias de los experimentos.

Para los gráficos de rendimiento vamos a utilizar el mismo estilo que cuando comparabamos assembler contra C.

4.1. Experimento desenrollando de ciclos

El primer experimento que vamos a realizar va a tratar sobre el desenrollado de ciclos. Definimos desenrollar una vez un ciclo a duplicar el código de forma tal de realizar dos iteraciones del ciclo dentro del ciclo y luego dividimos en la mitad la cantidad de iteraciones. Aunque ambos códigos terminan siendo la misma cantidad de instrucciones en el mismo orden, la diferencia que nosotros creemos que va a ser decisiva va a ser que uno tiene menor cantidad de saltos condicionales. Esto es importante para un procesador ya que este ejecuta instrucciones del futuro en paralelo y cuando se encuentra con una condición es problemático.

Para el experimento vamos a utilizar el filtro de ImagenFantasma y vamos a comparar una versión original donde solo se examinan dos pixeles por ciclo contra otra versión donde vamos a analizar 4 píxeles en cada ciclo.

4.1.1. Resultados

Podemos observar en Figura 16 que la recta de rendimiento que desenrollando una sola vez el loop obtenemos una mejora en rendimiento apreciable en el gráfico. Esto aunque era lo esperado, no hay tanta diferencia como la creída. Tiene una mejora de un 8 %. Notar que podríamos desenrollarlo más veces, haciendolo más rápido todavía.

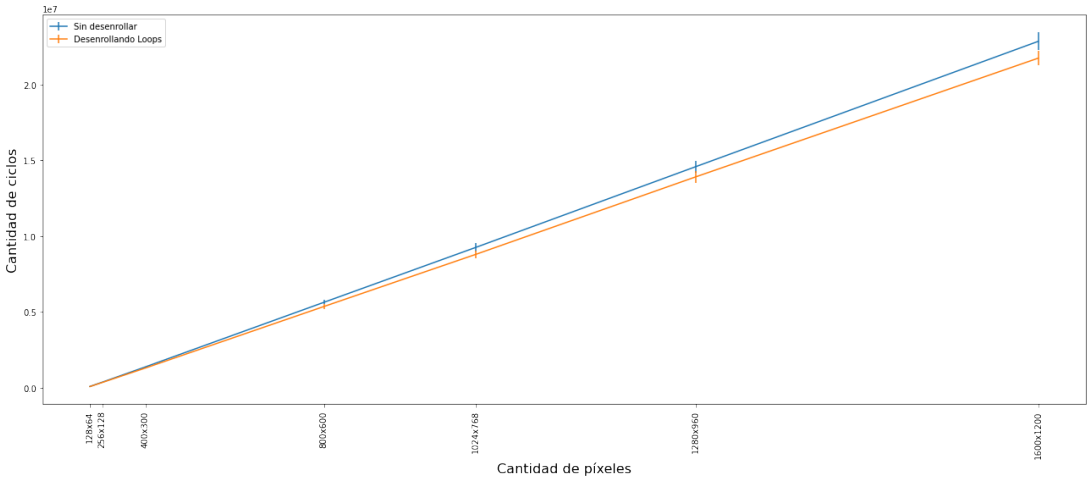


Figura 16

4.2. Experimento accesos de memoria

Vamos a analizar las diferencias de reutilizar datos de registros xmm contra ir a buscar los datos a la memoria.

Para esto vamos a utilizar el filtro de PixeladoDiferencial. En la primera versión de la implementación vamos a guardarnos una vez los datos en registros xmm y en la otra versión vamos a pedir los datos de memoria cada vez que se necesiten. Por como es la implementación del algoritmo, se llegan a pedir los datos hasta tres veces de memoria (primero para el promedio, luego para la diferencia y finalmente para el copiado). Vamos a comparar al igual que antes, ambos códigos mediante distintas resoluciones de imágenes. Nosotros esperamos que pedir los datos de la memoria sea mucho peor que pedir los datos de un registro xmm ya cargado.

4.2.1. Resultados

Podemos observar en 17 que la recta de rendimiento de ambos algoritmos es muy similar. Esto no fue lo esperado y creemos que debido a la utilizacion de la memoria cache , siendo imperceptibles pedirlos de memoria o pedirlos de un registro xmm ya cargado.

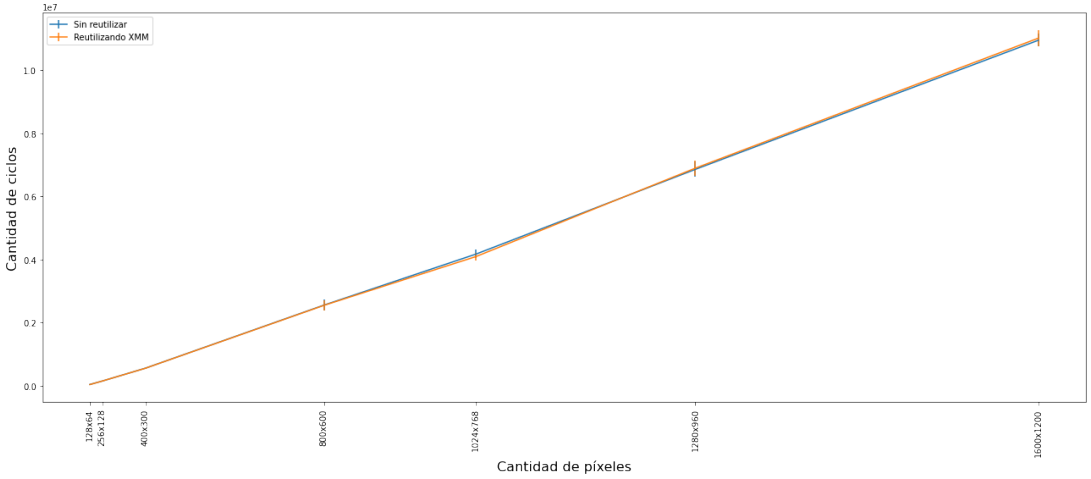


Figura 17: Rendimiento reutilizar variables vs pedir datos de memoria

A partir de esto, investigando encontramos una serie de instrucciones que nos podrían servir para tratar de que el procesador no guarde los datos en el cache. Entre ellas encontramos:

- WBINVD (Write Back and Invalidate Cache): invalida el cache pero es una instrucción privilegiada
- CLFLUSH (Flush Cache Line): invalida el cache la dirección que le pasemos.

Terminamos utilizando CLFLUSH y cada vez que vamos a pedir los datos a memoria, antes de pedirlos ejecutamos la instrucción CLFLUSH.

Podemos observar en la 18 la gran diferencia que obtenemos ahora entre la versión que reutiliza las variables y en la que no.

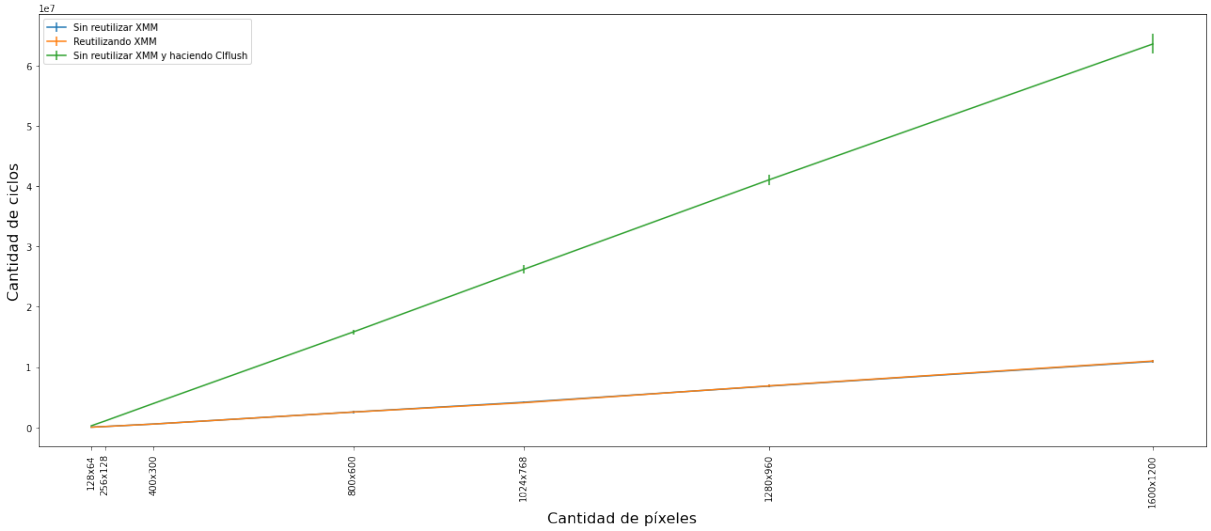


Figura 18: Comparación reutilizando xmm contra pedir memoria realizando CFLUSH

Hay que notar que no tenemos en cuenta los ciclos que consume la instrucción de CLFLUSH, lo cual puede ser bastante grande ya que no creemos que esté optimizada , no es una instrucción habitual. Para tratar de averiguar cuanta diferencia genera esta instrucción en el experimento, vamos a comparar el algoritmo utilizando CFLUSH y luego pidiendo de memoria contra utilizar CFLUSH y luego pidiendo los datos de los registros XMM que teníamos guardados los datos.

Podemos observar en la figura 19 que al comparar ambos no notamos diferencia de performance. Notar que ambos programas son mucho más costosos que sin utilizar CLFLUSH.

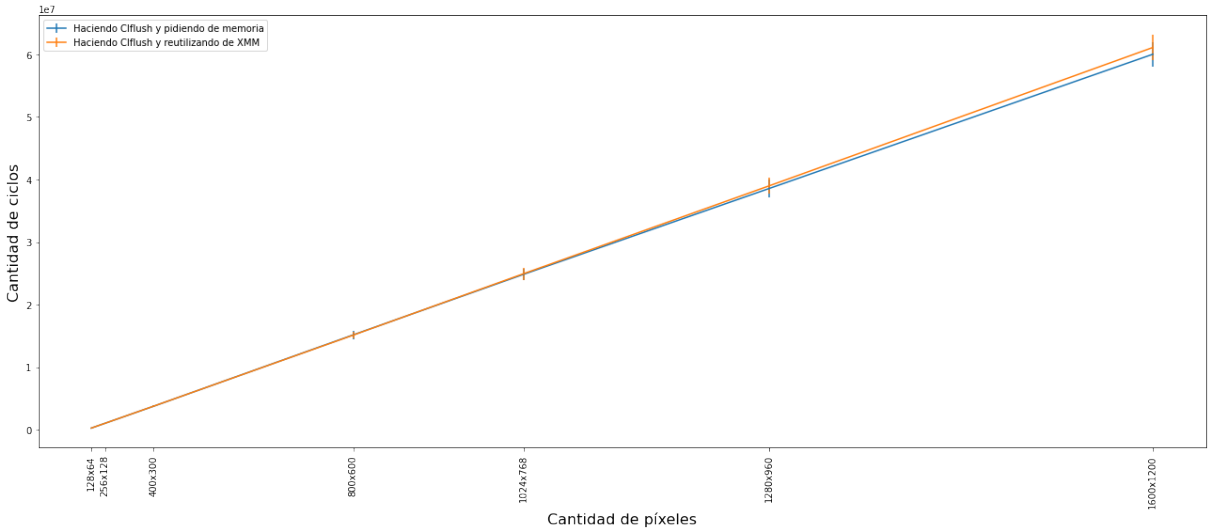


Figura 19: Comparación reutilizando xmm contra pedir memoria. Realizando CFLUSH en ambos.

Para expandir sobre el tema, puede ser que el procesador termine cambiando de lugar las instrucciones como las ejecute, haciendo que la invalidación de cache la haga después de que pida la dirección. Existen instrucciones para evitar esto, como la instrucción LFENCE, pero no desarollamos sobre esta instrucción.

5. Conclusión

La conclusión del experimento desenrollado de ciclos muestra que siempre y cuando se aumente el paralelismo a nivel de datos y se aproveche la localidad de los mismos es beneficioso como técnica de optimización. Creemos que al duplicar el código estamos mejorando el rendimiento para que pueda agarrar más instrucciones por ciclo el procesador. También otra conclusión es que siempre hay que balancear legibilidad y entendimiento del código contra rendimiento. Al duplicar el código estamos perdiendo legibilidad y incrementando la posibilidad de encontrarnos con bugs en el futuro y hemos observado que el rendimiento que ganamos es bastante poco.

La conclusión del experimento de accesos de memoria es que para este tipo de programa no cambia reutilizar variables xmm contra pedir de nuevo los datos de la memoria. Pero debe considerarse la localidad de los datos ya que la cache es responsables de una parte importante de la performance.

Como conclusión general de este Trabajo Practico, podemos asegurar que todavia el compilador de C no puede aprovechar las instrucciones SIMD de la arquitectura X64 y generar codigo vectorizado, por lo tanto una implementacion en assembler que las utilice es mas perfomante. Notar que esto no es asegurado en todos los casos y no siempre el beneficio es el mismo. Se observó el caso de ColorBordes, donde la compilación de C con optimizaciones si usó intrucciones vectoriales. En este caso observamos que hay bastante poca diferencia de la implementación de assembler contra la de C, lo cual indica que hay que balancear el mayor esfuerzo que es programar en assembler contra el beneficio performante que obtendríamos al usarlo.

También hay que notar que en la práctica, estos algoritmos de gran paralelismos, pueden ser implementados para ejecutarse por placas de videos (externas o intelhd) donde se puede aprovechar al máximo el paralelismo.