DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

subtitulo del trabajo

Organización del Computador II
Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Borgna, Agustín	079/15	aborgna@dc.uba.ar
Corleto, Alan	790/14	corletoalan@gmail.com
Lancioni, Franco	234/15	glancioni@dc.uba.ar

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 el presente trabajo se describe la problemática de diferencias de rendimiento entre procesamiento común de datos y procesamiento paralelo mediante vectores, aplicado a filtrado de imágenes. Se evalúan diferentes métodos para llegar a las mismas soluciones y se comparan sus velocidades de ejecución para determinar ventajas y desventajas con respecto a cada una de las implementaciones.

Índice

1. Introducción

1.1. Objetivos generales

El objetivo de este trabajo práctico es experimentar con sets de instrucciones SIMD de Intel ASM x86 sobre una serie de implementaciones de filtros en imágenes de mapas de bits (RGBA) con el propósito de entender mejor el rol del paralelismo de datos en este tipo de implementaciones y qué tan remunerador es. A partir de cada filtro, se plantearán implementaciones en C, SSE [1] (set de instrucciones sobre registros de 128 bits), AVX [2] (también sobre 128 bits pero permitiendo operaciones en formato 'no-destructivo' [3]) y su expansión de 256 bits AVX2, Acompañadas todas de un análisis para poder contrastar y discutir su rendimiento.

1.2. Notación de filtros

Dado que la temática de la aplicación de este trabajo práctico es el procesamiento de imágenes, es preciso especificar con qué tipo de imágenes vamos a trabajar y cómo vamos a describir matemáticamente las funciones que apliquen los filtros.

La imágenes sobre las que trabajamos son consideradas como matrices de píxeles. Cada pixel posee cuatro componentes: rojo (r), verde (g), azul (b) y transparencia (a). Cada una de estas componentes tiene un largo de 8 bits, por lo cual su rango de representación es $[0, 255]$.

Se define $I^k_{i,j}$ como la componente $k \in \{r, g, b, a\}$ en la fila i y la columna j de la imagen, donde i crece de abajo hacia arriba y j crece de izquierda a derecha. Por ejemplo, el pixel de la esquina inferior izquierda correspondería a la fila 0 y a la columna 0.

Se llama $O^k_{i,j}$ a la imagen de salida generada por el filtro especificado. Por ejemplo, el siguiente filtro asigna a todas las componentes de cada pixel el valor de la componente roja que se encuentra en su respectivo pixel.

$$\forall k \in \{r, g, b, a\} \qquad O^k_{i,j} = I^r_{i,j}$$

1.3. Mediciones y gráficos

Para realizar mediciones implementamos un script de benchmarking automático[4] que nos facilite la tarea y reduzca el error humano.

Para cada combinación de filtro, implementación, imagen, tamaño y parámetros adicionales corremos el tp2 un mínimo de 100 iteraciones y 2 segundos. Empezando con el mínimo de iteraciones realizamos una búsqueda exponencial hasta encontrar una cantidad que sobrepase el límite de tiempo. Usando la cantidad de iteraciones encontrada calculamos los percentiles y el promedio del tiempo con una precisión de 1uS y cantidad de ciclos por ejecución. Además, guardamos el porcentaje promedio de misses a la caché y el porcentaje promedio de mispredicciones de salto.

Nuestra medida de ciclos de ejecución es provista por la instrucción `RDTSC` . En todos los procesadores donde realizamos las mediciones el `TSC` se incrementa a una frecuencia fija e independiente, y se encuentra sincronizado entre núcleos del procesador[5] [2, Volume 3B, Chapter 17.15], por lo que resulta una medida válida de comparación entre ejecuciones en el mismo procesador.

Para las mediciones de hits a cache y misses del branch predictor utilizamos la herramienta $perf$[6] que nos devuelve los valores totales sobre la ejecución del programa a travéz de los registros contadores de performance de la cpu.

[1] Streaming SIMD Extensions
[2] Advanced Vector Extensions
[3] Preservan los operandos fuente
[4] codigo/benchmark/benchmark.py
[5] Se puede verificar esta característica buscando el flag $constant\_tsc$ en $/proc/cpuid$
[6] https://perf.wiki.kernel.org/index.php/Main_Page

En los gráficos presentados usaremos siempre la media a menos que se indique lo contrario. Y en los gráficos de barras representamos además los percentiles 10 y 90 como lineas de error.

1.4. Aclaración sobre el redondeo de punto flotante

Por defecto las conversiones de punto flotante a números enteros se realizan redondeando al número par mas cercano [2, Volume 1, Chapter 10.2.3], mientras que las operaciones de división de enteros siempre redondean hacia abajo. Como nuestros filtros, si usan operaciones de punto flotante, intentan emular operaciones de enteros nos conviene configurar el redondeo de flotantes hacia cero.

Esto se logra seteando los bits 13 y 14 del registro `MXCSR` con instrucciones. Como el registro es calee-saved deberemos guardarlo antes de modificarlo. Le cargaremos el valor por defecto, 0x1F80, mas la máscara que queremos.

```
; Guardamos el registro
SUB RSP, 8
STMXCSR [RSP]
; Cargamos nuestro valor
LDMXCSR [MXCSR_RZ] ; MXCSR_RZ tiene el valor x7F80
```

Y al finalizar la función volvemos a cargar el valor original.

Esto lo haremos en todas las implementaciones que trabajen con números flotantes.

2. Cropflip

2.1. Descripción

La imagen destino del filtro consiste en invertir verticalmente (flip) un recorte (crop) de la imagen fuente a partir de offsets dados como parámetro. El ancho y alto en píxeles de la imagen destino también se pasa como parámetros. La descripción matemática está dada por la fórmula:

$$O^k_{i,\ j} = I^k_{tamy+offsety-i-1,\ \ offsetx+j}$$

Donde el 'crop' de la imagen input corresponde con los píxeles del tipo:

$$I^k_{offsety+i,\ \ offsetx+j} \qquad \text{con} \quad 0 \leq i < tamy \quad 0 \leq j < tamx$$

MEM

I_{30}	I_{31}	I_{32}	I_{33}	I_{34}	I_{35}
I_{24}	I_{25}	I_{26}	I_{27}	I_{28}	I_{29}
I_{18}	I_{19}	I_{20}	I_{21}	I_{22}	I_{23}
I_{12}	I_{13}	I_{14}	I_{15}	I_{16}	I_{17}
I_6	I_7	I_8	I_9	I_{10}	I_{11}
I_0	I_1	I_2	I_3	I_4	I_5

Cuadro 1: Ilustración de la imagen fuente en memoria. En rojo los píxeles del crop (offsetx = 2, offsety = 2, tamx = 4, tamy = 4)

MEM

I_{14}	I_{15}	I_{16}	I_{17}
I_{20}	I_{21}	I_{22}	I_{23}
I_{26}	I_{27}	I_{28}	I_{29}
I_{32}	I_{33}	I_{34}	I_{35}

Cuadro 2: Ilustración de la imagen destino en memoria

No es difícil notar que las filas del crop en la fuente no constituyen una tira contigua de píxeles en memoria, si no que se encuentran distanciadas por el tamaño en bytes de offsetx lo que implica que necesitaremos iterar en dos ciclos anidados para poder procesar el crop.

2.2. Implementaciones

Al no involucrar operaciones aritméticas entre componentes de la imagen, las implementaciones del filtro se centran en accesos a memoria. Por lo tanto, las implementaciones solamente difieren en el modo en que se copian y asignan píxeles a la imagen output, radicando en esos métodos nuestro foco de análisis.

2.2.1. Implementaciones C y SSE

La implementación C simplemente se trata de recorrer la imagen con una sola variable aplicando pixel a pixel la transformación dada por la fórmula matemática previamente mencionada. Mientras que la implementación correspondiente a SSE recorre la imagen fuente desde la fila superior del recuadro (I_{32} en el cuadro 1) del crop hacia abajo de a 4 píxeles por iteración como indica el siguiente pseudo-código:

```
temp = src + offsetx*4 +srcRowSize*(offsety+tamy-1)
{Apunta al primer píxel de la esquina superior izq. del crop}
for i = 0 to tamy:
--> for j = 0 to tamx:
-----> xmm0 = [temp]
-----> [dst] = xmm0
-----> dst = dst  + 16
-----> temp =  temp + 16
-----> temp = temp - (dstRowSize + srcRowSize)
{Decrece el ancho de la imagen destino y el de la fuente para apuntar primer pixel de la fila de
abajo a la que procesó}
--> end for
end for
```

De esta manera, las lecturas y escrituras en memoria representan $\frac{1}{16} = 0,0625 = 6,25\,\%$ del total de los accesos a memoria de la implementación C.

2.2.2. Implementaciones SIMD paralelo en 128 y 256 bits

Se diferencian de la implementación anterior de SSE en el uso de la mayor cantidad posible de registros `XMM` (`YMM` de 256 bits en el caso de las implementaciones de AVX) para las operaciones de transferencia de bloques de píxeles de la imagen fuente a la destino.

Dado que las lecturas y escrituras en memoria de cada registro son independientes entre sí, la ejecución fuera de orden del procesador hace que la transferencia por bloque sea más rápida que la implementación individual (que recorre la imagen con un único registro `XMM`). Por cuestiones de vecindad espacial de los píxeles en la memoria, los accesos a memoria de cada registro tienen chances particularmente altas de hit en caché, no demorando así las lecturas del resto de los registros.

Recordando que los registros de AVX tienen 32 bytes de capacidad y cada pixel en nuestro formato ocupa 4 bytes, cada registro `YMM` tendrá capacidad para:

$$\frac{32\ bytes}{4\ \frac{bytes}{px}} = 8\ px$$

Por lo cual, suponiendo $tamx \equiv 0\ (mod\ 8)$ (como es el caso de una imagen de salida de 512x512), tendríamos $\frac{1}{32} = 0,03125 = 3,125\,\%$ del total de accesos a memoria de la implementación en C.

2.2.3. Copiado paralelo de vectores

La asignación de píxeles se hace llamando a las funciones externas 'copyN_sse' y 'copyN_avx2'[7] que copian tiras de píxeles de una imagen a otra por bloques de 64/4/1 ó 128/8/1 píxeles (sse y avx2 respectivamente) según sea posible en cada iteración. A modo de ejemplo (los otros casos son análogos), para copiar bloques de 64px (256B) desde la posición indicada por rsi a la indicada por rdi se cargan los valores correspondientes de la siguiente manera:

$\texttt{XMM}_0 \leftarrow$ [rsi]
$\texttt{XMM}_1 \leftarrow$ [rsi+16]
...
$\texttt{XMM}_{14} \leftarrow$ [rsi+224]
$\texttt{XMM}_{15} \leftarrow$ [rsi+240]

[7] ../entregable/tp2-bundle.v1/codigo/lib

[rdi] ← $\texttt{XMM}_0$
[rdi+16] ← $\texttt{XMM}_1$
...
[rdi+224] ← $\texttt{XMM}_{14}$
[rdi+240] ← $\texttt{XMM}_{15}$

Esta técnica de aplicar por cada iteración operaciones que, comúnmente, se harían en varias se conoce como 'loop unrolling'. Como desventaja frente a la optimización por paralelismo y minimización de branch mispredictions, en códigos de mucho mayor tamaño podría aumentar tan significativamente el tamaño del binario que aumente fuertemente el número de 'cache misses'. En nuestro caso copyN.o (que incluye a 'copyN_sse' y 'copyN_avx2') pesa 3,3 kB y es bastante probable que entre enteramente en caché.

En el caso del filtro Cropflip, las tiras de píxeles corresponden a tiras de tamaño 'tamx' (es decir, al ancho del crop).

2.3. Experimentos - Rendimiento y análisis

2.3.1. Comparación entre implementaciones

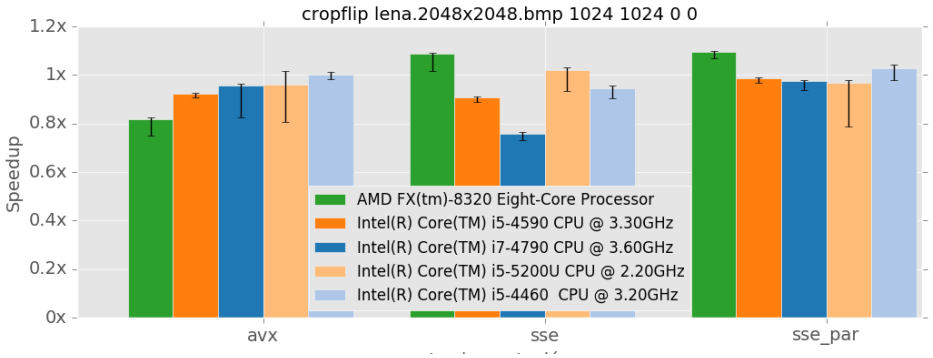


Figura 1: Tiempo de ejecución relativo de implementaciones ASM de cropflip contra la implementación C compilada con -O3

Algo interesante que nos sucedió fue que, originalmente, al hacer benchmarks con imágenes de 124x124 px, el filtro corría tan rápido que comprometía la fidelidad de la medición en los benchmarks. Así que finalmente optamos por expandir el tamaño de las imágenes de prueba.

A diferencia de los otros dos filtros, donde el procesamiento de imágenes involucra operaciones aritméticas paralelas, la implementación AVX de Cropflip no presenta 'speedup' respecto de las implementaciones de SSE con registros de 128 bits en paralelo. No sólo eso sino que demuestra menor rendimiento que su contraparte de SSE.

Investigando un poco sobre por qué sucedía esto, encontramos que en el Manual de Optimización de Sofware de Intel [8] se sugiere que, para accesos vectorizados a memoria desalineada, el rendimiento de AVX podría ser menor que el de SSE. Lo cual coincide exactamente con nuestro caso.

Sí se presentan entre las dos implementaciones de SSE, donde la diferencia es un poco más fuerte a favor de SSE 'paralela'. Dados los parámetros tales que el recuadro de crop mide 1024x1024 px, en la implementación con loop unrolling alcanza con dieciséis iteraciones procesando 64px en cada una para cubrir cada fila. Mientras que para la versión SSE común hay que recorrer: $\frac{1024\ \frac{px}{fila}}{8\ \frac{px}{iteracion}} = 128\ \frac{iteraciones}{fila}$. Sin embargo, la cantidad de iteraciones no refleja el hecho de que aún en ejecución fuera de orden los accesos a memoria siguen siendo individuales, teniendo que 'resecuencializar' las instrucciones para acceder ordenadamente. Podría suponerse que es debido a esto que el speedup no es del orden de magnitud

[8] Sección 11.6.2 - "DATA ALIGNMENT FOR INTEL® AVX - Consider 16-Byte Memory Access when Memory is Unaligned"

que sugerirían las respectivas iteraciones por fila.
Otra de las razones por las cuales el loop unrolling para copias de memoria no muestra una diferencia tan significativa es porque la microarquitectura Intel Core ejecuta hasta un $load$ y un $store$ de 128 bits por ciclo [9]. Por lo tanto cargar un `YMM` toma dos ciclos de ejecución, y termina siendo lo mismo fetchear dos cargas/descargas de 128b a una de 256b si se stallea el pipeline un ciclo más hasta liberar el puerto. Aún así, sigue entre las bondades del loop unrolling el poder coordinar por ejecución fuera de orden el $store$ anterior con el último $load$ en un único ciclo, siguiendo las características mencionadas anteriormente de Intel Core.
A esto último adicionamos el hecho de que el direccionamiento con $base\text{-}indice\text{-}offset$ tiene una latencia de 7 ciclos en AVX2 versus los 5 de SSE (si $offset + base > 2048$, 6 ciclos)[10], lo cual tiene especial importancia en nuestro TP porque es el tipo de direccionamiento habitual en el procesamiento de imágenes.

Como habíamos anticipado al hablar del loop unrolling en SSE, el tamaño del binario no es lo suficientemente grande como para no poder ser cacheado eficientemente. La vecindad espacial de las tiras de píxeles también colaboran para que el hitrate se mantenga alto aún cuando las filas del crop no son contiguas en memoria como comentamos en la descripción.

Es notable como el procesador de AMD es el que corre mas lento la implementación AVX (comportamiento que no vimos en las implementaciones AVX de otros filtros). Suponemos que puede deberse a un mal rendimiento de la implementación de copia a los registros `YMM` .

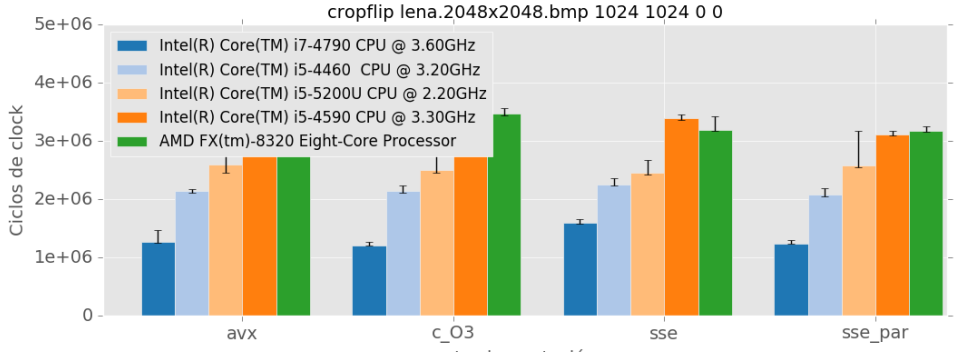


Figura 2: Ciclos de clock del timestamp counter durante la ejecución del filtro cropflip

En este gráfico se nota un poco cómo el techo de los 128 bits por ciclo hacen que el filtro Cropflip, que se basa casi completamente en operaciones $memcpy$, tenga rendimientos homogéneos entre sus implementaciones (salvando algunas diferencias puntuales ya mencionadas). Cuando testeábamos con imágenes más pequelas, curiosamente, el hitrate alto del loop unrolling sobre tiras cortas hacía que resalten un poco más sus puntos fuertes (particularmente paralelismo, no tanto minimizar branch mispredictions como veremos más adelante) a pesar del hecho de que los accesos a memoria son individuales y no pueden solaparse. Al agrandar los tamaños de las imágenes, este efecto fue considerablemente menos masivo.

Pudimos corroborar que no es casualidad que el rendimiento de la implementación en C@-$O3$ sea similar al de las implementaciones SSE utilizando GDB para constrastar dumps del código compilado y comprobando que vectorizaba con operaciones SSE como se muestra a continuación:

```
>Version usada de GDB: 7.7.1
>Version usada de GCC: 4.8.4 (cflags: -O3 -Wall -ggdb -std=c99 -pedantic -m64)


Dump of assembler code for function cropflip_c_o3:
```

[9] Intel Optimization Manual - 2.4.4.1 - Loads and Stores
[10] Intel Optimization Manual - Table 2-19. Effect of Addressing Modes on Load Latency

```
[...]
0x0000000000403eb7 <+231>: movdqu xmm0,XMMWORD PTR [rcx+rax*1]
0x0000000000403ebc <+236>: add    edx,0x1
0x0000000000403ebf <+239>: movdqu XMMWORD PTR [rsi+rax*1],xmm0
0x0000000000403ec4 <+244>: add    rax,0x10
0x0000000000403ec8 <+248>: cmp    edx,ebx
0x0000000000403eca <+250>: jb     0x403eb7 <cropflip_c_o3+231>
[...]
```

Para testear sobre las versiones actuales (al momento de realizar los tests) de GDB/GCC, corrimos con las versiones 7.11 y 5.3.1 respectivamente de cada herramienta. Al margen de haber diferencias en el resto del código (quizás optimizaciones de compilador que no estén en nuestra versión de SSE, justificando la muy pequeña diferencia vista anteriormente), el dump desensablaba el mismo ciclo.

A modo comparativo también incluimos el análisis de rendimiento entre las diversas optimizaciones para la implementación de C, donde vemos que los estimativos que hicimos anteriormente realmente (respecto de la cantidad de accesos a memoria para C) eran más bien una simplificación 'grosso modo' sin tener en cuenta que la relación C(O0)/compilado en ASM no es tan directa como pretendíamos.

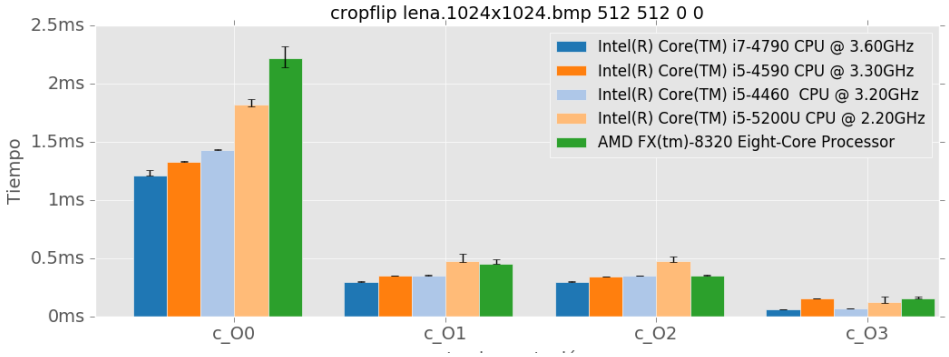


Figura 3: Tiempo de ejecución de la implementación C del filtro cropflip compilada con distintos optimizaciones

También es remarcable el salto en rendimiento se produce entre versiones de C con optimización menor a -O3, donde bajan del rango de 0.3-0.5 milisegundos para las implementaciones C-O1/C-O2 a menos de 0.1 milisegundo (100 microsegundos) para las C-03, lo cual seguramente tenga que ver con el uso de accesos vectorizados y alguna otra optimización del compilador no tan significativa como para diferenciarse de la implementación SSE. Más a simple vista, se puede ver el pobre rendimiento de C sin optimizar, entre 12 y 20 veces más lento que la implementación de 03.

2.3.2. Caché misses

Volviendo a la idea de la importancia del copiado de memoria en el filtro Cropflip, resulta inherentemente crucial entonces un funcionamiento favorable de la caché para que estos accesos no stalleen el procesamiento. Por ese motivo, en esta sección analizaremos un poco precisamente ese funcionamiento a partir de gráficos generados sobre benchmarks en un procesador con características de 8M de caché, en función del ancho y alto en píxeles de la imagen input.

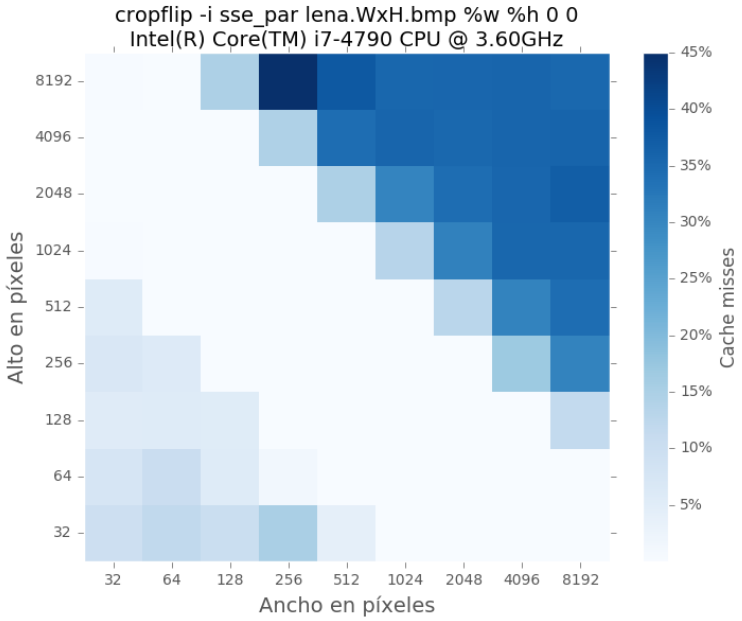


Figura 4: Porcentaje de misses de la cache en función del tamaño durante la ejecución del filtro cropflip, copiando e invirtiendo la imagen entera en el procesador mencionado anteriormente

Observamos que, como realizamos las corridas con la caché en caliente y gracias a la vecindad temporal, cuando el tamaño en bytes de la imagen es menor al de la caché el hitrate se mantiene cerca del 100 %. Cuando la imagen ya no entra en caché se termina desplazando a sí misma durante la ejecución y la cantidad de misses sube hasta cerca del 40 %.

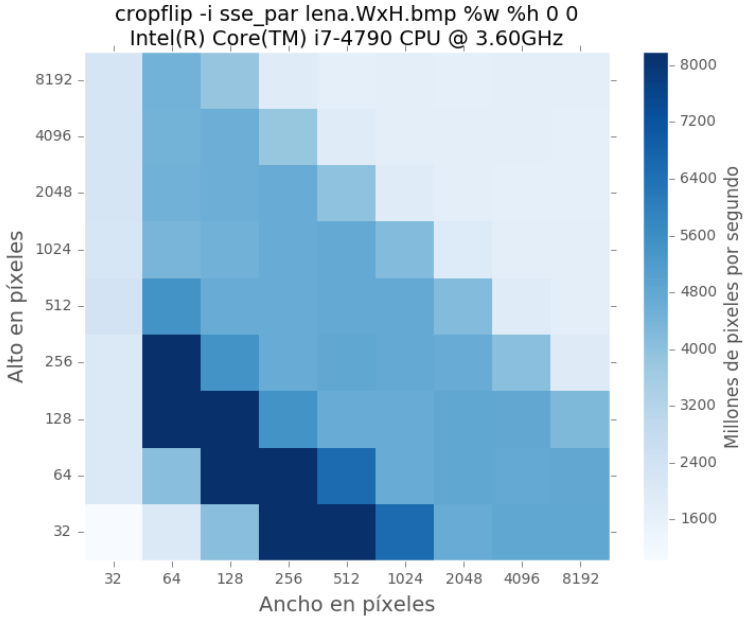


Figura 5: Millones de píxeles procesados por segundo en función del tamaño durante la ejecución del filtro cropflip, copiando e invirtiendo la imagen entera, en un procesador con 8MB de caché L3 y 256kB de caché L2 por núcleo

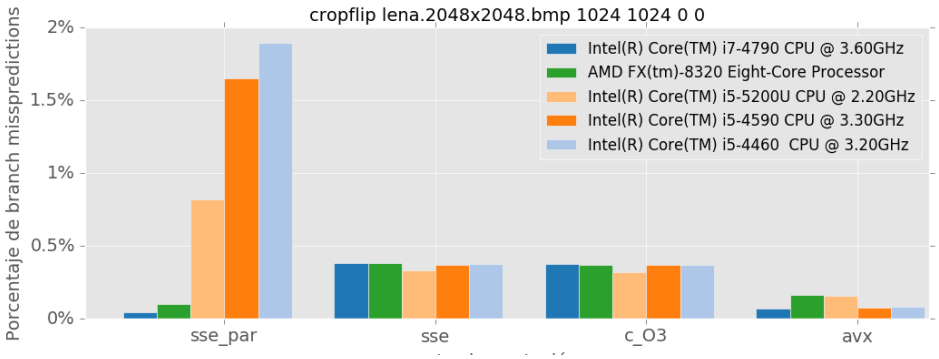
En este gráfico podemos observar cómo cuando la imagen ya no entra en caché el throughput baja a menos de la mitad, ya que el procesador se queda esperando a la memoria mas de la mitad del tiempo.

También observamos una segunda franja cuando la imagen ocupa menos de 256kB, que corresponde al tamaño de la caché L2. Es interesante ver la diferencia de rendimiento entre ambas caches, ya que

podemos procesar el doble de rápido cuando todos los accesos son a L2.

Cuando la imagen tiene poco ancho el procesamiento se ve interrumpido muy seguido por la lógica de cambio de linea, y el rendimiento se ve afectado.

2.3.3. Observación sobre las predicciones de saltos



En este gráfico se puede apreciar algo bastante curioso: mientras que para algunos procesadores el loop unrolling minimizó muy efectivamente el porcentaje de branch mispredictions en las implementaciones de AVX y de SSE en paralelo frente a las implementaciones que procesan con un único vector (ya vimos que la implementación en C con optimización O3 también usa un registro XMM para los accesos a memoria), para otros significó un retroceso.

Esto resulta particularmente contraintuitivo si tenemos en cuenta que en teoría minimizar iteraciones parecería implicar minimizar predicciones y por lo tanto penalidades. Sin embargo esto se puede deber a que, en cada ciclo, se requiere evaluar si la cantidad restante de píxeles para copiar es mayor a 64/4/1 ó 128/8/1 píxeles para poder saltar a cada subrutina. Resultando así contraproducente desde este punto de vista dicha implementación.

3. LDR (Low Dynamic Range)

3.1. Descripción

El filtro LDR modifica las zonas mas brillantes de la imagen, realzando o disminuyendo su brillo según el parámetro alpha sea positivo o negativo.

Formalmente,

$$O^k_{i,j} = ldr^k_{i,j} = I^k_{i,j} + \alpha \frac{sumargb_{i,j}}{max} I^k_{i,j} \qquad \forall k \in \{r,g,b\},\ i,j \in \mathbb{N} \text{ tq } 2 \leq i < w-2 \ \wedge\ 2 \leq j < h-2$$

donde

$$sumargb_{i,j} = \sum_{-2 \leq u,v \leq 2,\ k \in \{r,g,b\}} I^k_{i+u,j+v}$$

$$max = 5 * 5 * 255 * 3 * 255$$

y los $O^k_{i,j}$ no definidos mantienen su valor inicial.

Como para procesar cada pixel necesitamos el valor de sus vecinos no podremos aplicar el filtro en los bordes de la imagen, por lo que quedan sin modificar.

Inicialmente se nos ocurrieron tres algoritmos para realizar el filtro:

1. Iterar sobre cada pixel. Si está en el borde copiarlo directamente, sino calcular su sumargb leyendo los 25 vecinos y guardar los valores correspondientes al pixel.

   Este método tiene una implementación trivial, pero un rendimiento pésimo debido a que debe acceder 26 veces a memoria para procesar cada pixel y realiza cálculos repetidos de las sumas de cada pixel.

2. Recorrer cada pixel de la imagen calculando la suma de sus canales r, g, b, y guardar el resultado en una matriz auxiliar.

   Luego recorrer nuevamente los píxeles no-borde y realizar la función usando los valores precalculados.

   Con este método evitamos repetir los cálculos de la suma de los canales de cada pixel, pero implica usar $\mathcal{O}(ancho * altura)$ memoria adicional (como la suma de tres bytes siempre entra en un word y no necesitamos copiar el alpha, la matriz tendrá la mitad del tamaño de la imagen). Además agrega el overhead de los accesos a memoria y ocupa espacio extra en la caché.

3. Recorrer cada pixel excepto los de las filas inferiores y superiores calculando

   $$\sum_{-2 \leq v \leq 2,\ k \in \{r,g,b\}} I^k_{i,j+v}$$

   esto es, la suma de su columna ± 2. Mantener siempre los últimos cuatro resultados en memoria y al calcular el nuevo valor realizar la suma entre ellos y usarlos para procesar el pixel $(i-2, j)$, si no es un borde.

   De esta forma logramos evitar algunos cálculos repetidos pero mantenemos el uso de memoria extra constante.

De estos algoritmos decidimos implementar la primer opción en C, ya que es el método trivial contra el que queremos comparar.

Para nuestras implementaciones en assembler utilizamos el tercer método, ya que consideramos que era el que ofrecía mas potencial para paralelizar con instrucciones SSE.

3.2. Implementaciones

3.2.1. C

El código de C es bastante simple, recorre cada pixel de la imagen y:

- Si es un borde, lo copiamos directamente.
- Si no es un borde, recorremos los 25 vecinos acumulando la suma de sus componentes, y así obtenemos sumargb. Luego calculamos el resto de la fórmula y guardamos el valor final.

3.2.2. Asm - SSE

Para implementar el algoritmo (3) recorremos la imagen procesando de a 4 píxeles, manteniendo la suma de las 4 ultimas columnas y calculando 4 nuevas en cada loop.

	$I_{i-2,j+2}$	$I_{i-1,j+2}$	$I_{i,j+2}$	$I_{i+1,j+2}$	$I_{i+2,j+2}$	$I_{i+3,j+2}$	$I_{i+4,j+2}$	$I_{i+5,j+2}$
	$I_{i-2,j+1}$	$I_{i-1,j+1}$	$I_{i,j+1}$	$I_{i+1,j+1}$	$I_{i+2,j+1}$	$I_{i+3,j+1}$	$I_{i+4,j+1}$	$I_{i+5,j+1}$
MEMORIA	$I_{i-2,j}$	$I_{i-1,j}$	$I_{i,j}$	$I_{i+1,j}$	$I_{i+2,j}$	$I_{i+3,j}$	$I_{i+4,j}$	$I_{i+5,j}$
	$I_{i-2,j-1}$	$I_{i-1,j-1}$	$I_{i,j-1}$	$I_{i+1,j-1}$	$I_{i+2,j-1}$	$I_{i+3,j-1}$	$I_{i+4,j-1}$	$I_{i+5,j-1}$
	$I_{i-2,j-2}$	$I_{i-1,j-2}$	$I_{i,j-2}$	$I_{i+1,j-2}$	$I_{i+2,j-2}$	$I_{i+3,j-2}$	$I_{i+4,j-2}$	$I_{i+5,j-2}$

Cuadro 3: Ilustración de la memoria en el ciclo de ldr. En gris los píxeles que queremos procesar, en verde las columnas de las cuales ya tenemos la suma guardada y en naranja las columnas que debemos calcular.

Obviaremos escribir las variables i y j en las siguientes ilustraciones para mantener la claridad.

El proceso del loop comienza con la suma de las columnas anteriores en $\texttt{XMM}_0$, guardadas como word ya que su valor máximo es $5 * 3 * 255 < 2^{16}$:

$\texttt{XMM}_0$	0	0	0	0	$sumC_1$	$sumC_0$	$sumC_{-1}$	$sumC_{-2}$

Como cada pixel ocupa 32 bits podemos cargar los cuatro píxeles que vamos a necesitar de cada fila en 5 registros.
Luego descomprimimos cada componente a tamaño word (usando 5 registros mas) y realizamos las sumas de los componentes para obtener la suma de cada pixel, cuidándonos de borrar el alpha.

$\texttt{XMM}_N$	$sum_{5,v}$	$sum_{4,v}$	$sum_{3,v}$	$sum_{2,v}$	0	0	0	0

para $-2 \leq v \leq 2$ y $N = v + 3$.

A continuación sumamos todos los píxeles de la columna entre sí y combinamos el resultado con $\texttt{XMM}_0$, obteniendo el vector de sumas.

$\texttt{XMM}_0$	$sumC_5$	$sumC_4$	$sumC_3$	$sumC_2$	$sumC_1$	$sumC_0$	$sumC_{-1}$	$sumC_{-2}$

Luego procedemos a calcular $sumargb$. Para ello copiamos el contenido de $\texttt{XMM}_0$ a $\texttt{XMM}_5$ y luego vamos shifteando $\texttt{XMM}_0$ de a una palabra por vez mientras sumamos su valor a $\texttt{XMM}_5$ hasta que en $\texttt{XMM}_0$ queden solo las 4 sumas que acabamos de calcular y en la parte baja de $\texttt{XMM}_5$ se encuentre la suma de las 5 columnas vecinas (esto es, ya tenemos $sumargb$).

```
Repetir 4 veces:
    PSRLDQ XMM1, 2
    PADDW XMM5, XMM1
```

$\texttt{XMM}_0$	0	0	0	0	$sumC_5$	$sumC_4$	$sumC_3$	$sumC_2$

$\texttt{XMM}_5$	X	X	X	X	$sumrgb_3$	$sumrgb_2$	$sumrgb_1$	$sumrgb_0$

Observar que en $\texttt{XMM}_0$ ya quedaron los valores de las columnas listas para el siguiente loop.

Ahora convertimos cada $sumargb$ a punto flotante, la multiplicamos por el valor de α que habíamos almacenado en $\texttt{MM}_2$ y movemos replicamos cada una en un registro diferente.

$\texttt{XMM}_N$	$sumargb_u * \alpha$	$sumargb_u * \alpha$	$sumargb_u * \alpha$	$sumargb_u * \alpha$

para $0 \leq u \leq 3$ y $N = u + 5$

Mientras tanto cargamos los valores originales de los píxeles a procesar y los convertimos a punto flotante, ocupando cada pixel un registro entero.

$\texttt{XMM}_M$	A_u	R_u	G_u	B_u

para $0 \leq u \leq 3$ y $M = u + 9$

Multiplicamos las $sumargb$ anteriores por los canales de los píxeles y limpiamos la correspondiente al canal alpha con una máscara que nos armamos en el momento.

$\texttt{XMM}_N$	0	$R_u * sumargb_u * \alpha$	$G_u * sumargb_u * \alpha$	$B_u * sumargb_u * \alpha$

para $0 \leq u \leq 3$ y $N = u + 5$

Finalmente multiplicamos estos registros por el recíproco de MAX que teníamos previamente guardado y los sumamos a los registros con los valores de los píxeles.

$\texttt{XMM}_M$	A_u	ldr_u^r	ldr_u^g	ldr_u^b

para $0 \leq u \leq 3$ y $M = u + 9$

Solo resta aplicar max y min para saturar los valores calculados, reconvertirlos a byte y guardarlos en la imagen de salida.

Como la imagen que procesamos nunca tienen padding en la linea (ya que los píxeles ocupan 32b cada uno) podemos considerarla aplanada como una secuencia lineal de filas concatenadas y procesar desde el comienzo de la tercer linea hasta el final de la antepenúltima checkeos de casos especiales. De esta manera evitamos el uso de condicionales dentro del loop y el riesgo de mispredicciones de salto.
Procesar de este modo genera que en los bordes laterales se grabe basura, por lo que luego del loop debemos copiar los píxeles originales.

Para copiar las dos primeras y últimas filas usamos función $copyN\_sse$, compartida con el filtro cropflip y detallada mas arriba (2.2.3).

3.2.3. Asm - SSE con cálculos de enteros

Cuando vimos que al hacer las operaciones en punto flotante los resultados no eran perfectamente iguales debido a errores de precisión, decidimos codear una implementación realizando todos los cálculos con enteros.

La mayoría de los cambios son directos, usando por ejemplo `PMULLD` en vez de `MULPS` . Como el valor máximo que al que pueden llegar es $25 * 3 * 255 * 255 * 255 > 2^{16}$ seguimos necesitando $32b$ para almacenar los valores, esta vez como double words en vez de punto flotante de precisión simple.

El primer problema que nos encontramos fue que no existe una instrucción de división de enteros en SIMD. Recurrimos a un método de división de enteros con signo usando multiplicaciones y shifts presentado por H.S. Warren [3, Chapter 10].
Siguiendo el método descripto llegamos a la fórmula:

$$\frac{x}{max} = x * magic >> s \qquad \forall -2^{31} \leq x < 2^{31}$$

$$magic = 0x6e15c447$$

$$s = 53$$

El siguiente problema una vez resuelto fue que al estar trabajando con números enteros el resultado de una multiplicación de valores de $32b$ ocupa $64b$ y por mas que solo necesitamos la parte alta de la multiplicación (ya que la parte baja se desecha con el shift de 53 lugares), la ISA no provee una instrucción que calcule solo lo que nos importa (cuando si tiene `PMULDW` para realizar la misma operación con words).

Para solucionarlo tuvimos que duplicar cada registro con valores, shifteando la copia para quedarse con la parte alta en la parte baja, y usar la instrucción `PMULDQ` que realiza la multiplicación completa sobre dos double words y las guarda como quad words. Luego realizamos los shifts y combinamos los resultados de nuevo en un registro por pixel.

```
MOVDQA XMM_M , XMM_N
PSRLQ XMM_M , 32
```

XMM_N	0	$R_u * sumargb_u * \alpha$	$G_u * sumargb_u * \alpha$	$B_u * sumargb_u * \alpha$

XMM_M	0	0	0	$R_u * sumargb_u * \alpha$

para $0 \leq u \leq 3$, $N = u + 5$ y $M = u + 9$.

```
PMULDQ XMM_N , {registro con magic replicado 4 veces}
PMULDQ XMM_M , {registro con magic replicado 4 veces}
```

XMM_N	$R_u * sumargb_u * \alpha * magic$	$B_u * sumargb_u * \alpha * magic$

XMM_M	$G_u * sumargb_u * \alpha * magic$	0

para $0 \leq u \leq 3$, $N = u + 5$ y $M = u + 9$.

```
PSRLQ XMM_N , 53
PSRLQ XMM_M , 21
PAND XMM_M , {registro con 0s en la double word mas baja y 1s en lo demas}
```

XMM_N	0	$\delta\ ldr_u^r$	0	$\delta\ ldr_u^b$

XMM_M	0	0	$\delta\ ldr_u^g$	0

para $0 \leq u \leq 3$, $N = u + 5$ y $M = u + 9$. Donde $\delta\ ldr_u^k$ es lo que hay que sumarle al valor original del pixel en ldr_u^k.

```
POR XMM_N , XMM_M
PADDD XMM_M , {registro con los valores originales del pixel}
```

XMM_N	A_u	ldr_u^r	ldr_u^g	ldr_u^b

para $0 \leq u \leq 3$ y $N = u + 9$

Y las operaciones restantes se convierten a punto fijo trivialmente.

3.2.4. Asm - AVX/FMA

En procesadores que soportan la extensión AVX podemos aprovechar las instrucciones no destructivas para reemplazar, por ejemplo,

```
MOVDQA XMM10, XMM5
PUNPCKLBW XMM5, XMM15
PUNPCKHBW XMM10, XMM15
```

por

```
PUNPCKHBW XMM10, XMM5, XMM15
PUNPCKLBW XMM5, XMM5, XMM15
```

y ahorrarnos un par de ciclos.

Además con la extensión FMA (fused multiply & add) podemos combinar la multiplicación por el inverso de max y la suma de los valores originales del pixel, reemplazando

```
MULPS XMM5, XMM14
ADDPS XMM5, XMM9
```

por

```
VFMADD132PS XMM5, XMM9, XMM14
```

3.2.5. Asm - AVX2

Con AVX2 podemos realizar todas las operaciones que realizábamos con AVX ahora sobre los registros ymm de $256b$, por lo que ahora conservamos ocho sumas de columnas al iniciar, calculamos ocho sumas nuevas y procesamos de a ocho píxeles por ciclo.

La conversión del algoritmo es directa cuando trabajamos con cada valor independientemente, pero debemos tener cuidado en un par de casos, cuando usamos sumas horizontales o shifts.

Cuando calculamos la suma de cada pixel, si traducimos directamente las operaciones desde AVX terminamos teniendo

$\texttt{YMM}_N$								
	$sum_{13,v}$	$sum_{12,v}$	$sum_{11,v}$	$sum_{10,v}$	0	0	0	0
	$sum_{9,v}$	$sum_{8,v}$	$sum_{7,v}$	$sum_{6,v}$	0	0	0	0

para $-2 \leq v \leq 2$ y $N = v + 3$.

Pero nosotros necesitamos tener todos los resultados en la parte alta, por lo que debemos hacer una permutación:

```
VPERMQ YMM_N , YMM_N , 0b11010000
```

$\texttt{YMM}_N$								
	$sum_{13,v}$	$sum_{12,v}$	$sum_{11,v}$	$sum_{10,v}$	$sum_{9,v}$	$sum_{8,v}$	$sum_{7,v}$	$sum_{6,v}$
	0	0	0	0	0	0	0	0

Una vez que tenemos el vector con las 16 sumas de columnas combinadas en $\texttt{YMM}_0$ y queremos calcular $sumargb$ para cada uno de los 8 píxeles no podemos usar el mismo algoritmo que en SSE ya que no hay shifts del registro de $256b$ completo, por lo que debemos proceder de otra forma.

Primero, copiamos $\texttt{YMM}_0$ a $\texttt{YMM}_1$ y aplicando `VPSHUFB` con una constante que cargamos de memoria borramos las primeras 4 words de la parte alta e invertimos el orden de las siguientes 4. Luego volvemos a copiar $\texttt{YMM}_1$ a $\texttt{YMM}_5$.

$\texttt{YMM}_0$								
	$sumC_{13}$	$sumC_{12}$	$sumC_{11}$	$sumC_{10}$	$sumC_9$	$sumC_8$	$sumC_7$	$sumC_6$
	$sumC_5$	$sumC_4$	$sumC_3$	$sumC_2$	$sumC_1$	$sumC_0$	$sumC_{-1}$	$sumC_{-2}$

$\texttt{YMM}_k$								
	0	0	0	0	$sumC_6$	$sumC_7$	$sumC_8$	$sumC_9$
	$sumC_5$	$sumC_4$	$sumC_3$	$sumC_2$	$sumC_1$	$sumC_0$	$sumC_{-1}$	$sumC_{-2}$

con $k \in \{1, 5\}$

Ahora sí realizamos los shifts (de double quadword) y las sumas

```
Repetir 4 veces:
    VPSRLDQ YMM1, YMM1, 2
    VPADDW YMM5, YMM5, YMM1
```

$\texttt{YMM}_5$								
	0	0	0	0	$sumrgb_4^a$	$sumrgb_5^a$	$sumrgb_6^a$	$sumrgb_7^a$
	$sumrgb_7^b$	$sumrgb_6^b$	$sumrgb_5^b$	$sumrgb_4^b$	$sumrgb_3$	$sumrgb_2$	$sumrgb_1$	$sumrgb_0$

donde $sumrgb_u^a + sumrgb_u^b = sumargb_u$.

Ahora hacemos un shuffle con la misma constante que antes y permutamos.

```
VPSHUFB YMM1, YMM5, {registro ymm con constante pre-cargada}
VPERMQ YMM5, YMM5, 0b11011111
```

YMM_1	0	0	0	0	$sumrgb_7^a$	$sumrgb_6^a$	$sumrgb_5^a$	$sumrgb_4^a$
	$sumrgb_7^b$	$sumrgb_6^b$	$sumrgb_5^b$	$sumrgb_4^b$	$sumrgb_3$	$sumrgb_2$	$sumrgb_1$	$sumrgb_0$

YMM_5	0	0	0	0	$sumrgb_7^b$	$sumrgb_6^b$	$sumrgb_5^b$	$sumrgb_4^b$
	0	0	0	0	0	0	0	0

Y nos si hacemos la suma nos queda el registro listo para continuar.

YMM_5	0	0	0	0	$sumrgb_7$	$sumrgb_6$	$sumrgb_5$	$sumrgb_4$
	X	X	X	X	$sumrgb_3$	$sumrgb_2$	$sumrgb_1$	$sumrgb_0$

Las demás instrucciones se traducen directamente desde AVX.

3.3. Experimentos

3.3.1. Comparación entre implementaciones

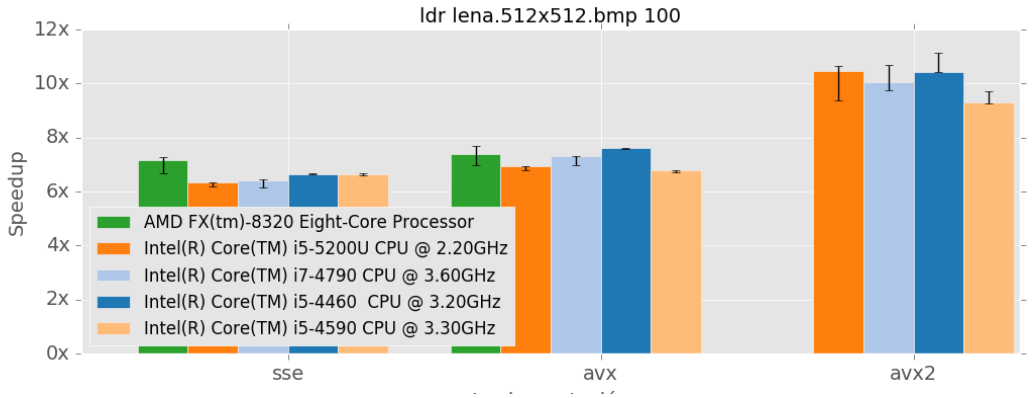


Figura 6: Speedup relativo respecto a la implementación en C compilada con O3 del filtro ldr sobre lena.bmp 512x512.

Encontramos una diferencia de performance muy grande entre la implementación de C y las implementaciones en assembler. Esto se puede deber a que el compilador de C no logra vectorizar el loop, como podemos ver en el dump de gdb.

```
0x404662 <ldr_c+273>    movzx  ecx,BYTE PTR [rbp-0x8]
0x404666 <ldr_c+277>    add    eax,ecx
0x404668 <ldr_c+279>    movzx  ecx,BYTE PTR [rbp-0x2]
0x40466c <ldr_c+283>    add    eax,ecx
0x40466e <ldr_c+285>    movzx  ecx,BYTE PTR [rbp-0x3]
0x404672 <ldr_c+289>    add    eax,ecx
0x404674 <ldr_c+291>    movzx  ecx,BYTE PTR [rbp-0x4]
0x404678 <ldr_c+295>    add    eax,ecx
0x40467a <ldr_c+297>    movzx  ecx,BYTE PTR [rbp+0x2]
0x40467e <ldr_c+301>    add    eax,ecx
0x404680 <ldr_c+303>    movzx  ecx,BYTE PTR [rbp+0x1]
0x404684 <ldr_c+307>    add    eax,ecx
0x404686 <ldr_c+309>    movzx  ecx,BYTE PTR [rbp+0x0]
```

También vemos que el uso de instrucciones no destructivas de la extensión AVX y la operación de FMA logran un ligero aumento de performance, reduciendo el tiempo de ejecución en aproximadamente 10 % en los procesadores que probamos.

Por otro lado la implementación AVX2 corre un 50 % mas rápido que SSE, mostrando que los beneficios de procesar aún mas datos en paralelo sobrepasan el problema del aumento de la complejidad del código.

3.3.2. Diferencias entre cálculos con enteros y con punto flotante

Implementación de C

Implementación SSE con punto flotante

Figura 7: Perdida de precisión en la implementación con cálculos de punto flotante. Ambas imágenes son ligeramente diferentes.

Aún con el registro `MXCSR` configurado para redondear siempre hacia cero al convertir números de punto flotante a enteros encontramos diferencias en las imágenes de salida. En una batería de imágenes de prueba encontramos que nunca un canal de un pixel difería en mas de 1 de la imagen original, y esta diferencia se encontraba en menos del 0.02 % de los píxeles.

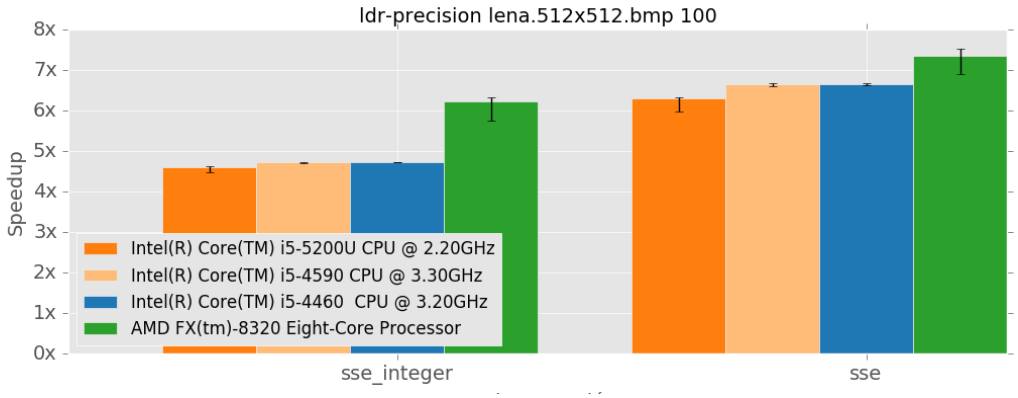


Figura 8: Comparación de tiempo de ejecución usando cálculos enteros contra cálculos de punto flotante respecto a la implementación en C compilada con O3 del filtro ldr sobre lena.bmp 512x512

Considerando que la diferencia es tan mínima y que la implementación exacta usando cálculos enteros resultó ser un 25 % mas lenta que la de punto flotante, como se muestra en la figura 8, decidimos que para un uso general, tratándose de un filtro de imágenes que serán captadas por un ojo humano, resulta mucho mas conveniente usar cálculos de punto flotante.

3.3.3. Throughput de píxeles en función del tamaño de imagen

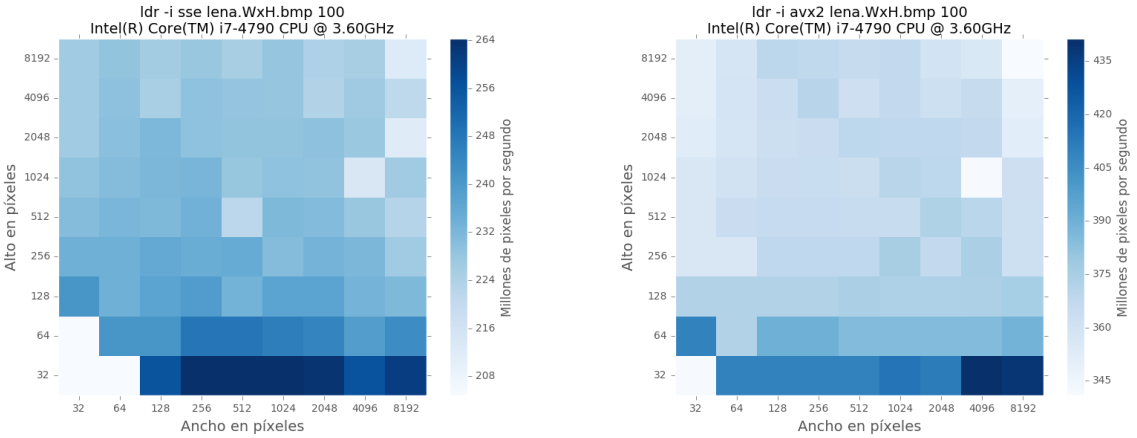


Figura 9: Millones de píxeles por segundo en implementación SSE y AVX2 en un procesador con 8MB de caché

Observamos en la figura 9 que el rendimiento se mantiene casi constante para la mayoría de los tamaños de imagen aún cuando el tamaño de la imagen sobrepasa el de la caché. Esto se debe a que el controlador de caché identifica perfectamente el patrón de accesos del algoritmo, y la cantidad de operaciones que se realizan en el ciclo le dan tiempo suficiente para prefetchear los datos necesarios de memoria.

Los valores se ven distorsionados cuando la imagen tiene poca altura ya que en ese caso la copia de las primeras y últimas dos filas, operación mucho mas rápida que los cálculos del filtro, representa un porcentaje mas alto de las operaciones. Como para la copia de los bordes se deben realizar accesos no secuenciales a memoria no vemos esta distorsión para imágenes con poco ancho.

4. Sepia

4.1. Descripción

El filtro sepia consiste en cambiar los colores de cada pixel de la siguiente manera:

$$O^r_{i,j} = 0,5 \cdot suma_{i,j}$$
$$O^g_{i,j} = 0,3 \cdot suma_{i,j}$$
$$O^b_{i,j} = 0,2 \cdot suma_{i,j}$$
$$O^a_{i,j} = I^a_{i,j}$$

donde $suma_{i,j} = I^r_{i,j} + I^g_{i,j} + I^b_{i,j}$

Como puede verse, este filtro no recibe ningún parámetro y trabaja directamente con los datos mismos de la imagen.

4.2. Implementaciones

Antes de entrar en detalle a cada una de las implementaciones hechas, es importante notar que

$$suma_{i,j} = I^r_{i,j} + I^g_{i,j} + I^b_{i,j} \leq 255 + 255 + 255 = 3 \cdot 255$$

Lo cual excede el rango de representación de las componentes de un pixel. Esto puede resultar en un problema o no dependiendo de cada caso, ya que es necesario calcular $suma_{i,j}$ para resultados intermedios. Como el resultado final de aplicar el filtro en cada componente es multiplicar $suma_{i,j}$ por un número menor o igual a 1, se consideró aplicar la ley distributiva para poder salvar este problema, ya que, por ejemplo, en el caso de la componente azul

$$O^b_{i,j} = 0,2 \cdot suma_{i,j} = 0,2 \cdot (I^r_{i,j} + I^g_{i,j} + I^b_{i,j}) = 0,2 \cdot I^r_{i,j} + 0,2 \cdot I^g_{i,j} + 0,2 \cdot I^b_{i,j} \leq 0,2 \cdot 255 + 0,2 \cdot 255 + 0,2 \cdot 255 = 0,6 \cdot 255 \leq 255$$

y en el caso de la componente verde

$$O^b_{i,j} = 0,3 \cdot suma_{i,j} = 0,3 \cdot (I^r_{i,j} + I^g_{i,j} + I^b_{i,j}) = 0,3 \cdot I^r_{i,j} + 0,3 \cdot I^g_{i,j} + 0,3 \cdot I^b_{i,j} \leq 0,3 \cdot 255 + 0,3 \cdot 255 + 0,3 \cdot 255 = 0,9 \cdot 255 \leq 255$$

Cabe resaltar que realizando las operaciones de este modo, nunca se llega a un resultado intermedio el cual exceda el límite de representación de las componentes.

Sin embargo, esta alternativa fue descartada ya que implica hacer el triple de multiplicaciones en punto flotante, lo cual reduce drásticamente la performance.

Además, se debe notar que en el caso de la componente roja ocurre lo siguiente:

$$O^b_{i,j} = 0,5 \cdot suma_{i,j} \leq 0,5 \cdot 3 \cdot 255 = 1,5 \cdot 255$$

Lo cual resulta un problema a la hora de almacenar el resultado final ya que el mismo podría exceder el rango de representación de la componente roja. En cada una de las implementaciones se detalla cómo fue resuelta esta cuestión.

4.2.1. C

A grandes rasgos, la implementación en C es intuitiva, ya que recorre cada uno de los píxeles uno por uno, por cada iteración crea una variable auxiliar llamada $suma$ (la cual, como puede esperarse, contiene el valor de $suma_{i,j}$) y asigna a cada componente el valor de dicha suma multiplicada por su respectivo factor. Debido a las dos problemáticas que se plantearon anteriormente, se hicieron algunos ajustes con respecto a la implementación. La variable auxiliar $suma$ es una variable del tipo $unsignedshort(16bits)$ para no perder precisión en las operaciones. Por la misma razón, se creó también una nueva variable auxiliar $suma_r$, la cual contiene el valor de $0,5 \cdot suma_{i,j} = I^r_{i,j}$. Para volver a $char(8bits)$ se resuelve distinto en cada caso.

- En el caso de $suma$ con respecto a las componentes azul y verde, solamente se les asigna a cada una el valor de $suma$ multiplicado por su respectivo factor ya que, como se demostró anteriormente, $O^b_{i,j}$ y $O^g_{i,j}$ son siempre menores a 255 y por lo tanto no hay que tener ningún recaudo extra. Luego de ejecutar la multiplicación, el lenguaje C simplemente convierte el dato a $char$ y lo asigna.
- En el caso de $suma_r$, se pregunta primero si el resultado final dio mayor o igual a 255. Si eso es cierto, lo satura a 255. Caso contrario, lo deja como está. En esencia se está ejecutando un $min(I^r_{i,j}, 255)$.

La cantidad de operaciones de punto flotante por pixel en esta implementación es de $\frac{3op}{px}$.

4.2.2. Asm - SSE

Cada componente ocupa 1 byte y cada pixel tiene 4 componentes, por lo cual un pixel ocupa 4 bytes. Como los registros `XMM` son de 16B, es posible procesar 4 píxeles en un solo registro simultáneamente. Al inicio del loop, se copian 4 píxeles de la imagen fuente a $\texttt{XMM}_0$ para empezar a procesar. Para simplificar la notación, se llamará K_i a la componente K del pixel $i \in 0, 1, 2, 3$

$\texttt{XMM}_0$	A_3	R_3	G_3	B_3	A_2	R_2	G_2	B_2	A_1	R_1	G_1	B_1	A_0	R_0	G_0	B_0

Luego, se copia a $\texttt{XMM}_1$ y $\texttt{XMM}_3$ el contenido de $\texttt{XMM}_0$ y se borran los alpha de los mismos

$\texttt{XMM}_1$	0	R_3	G_3	B_3	0	R_2	G_2	B_2	0	R_1	G_1	B_1	0	R_0	G_0	B_0

$\texttt{XMM}_3$	0	R_3	G_3	B_3	0	R_2	G_2	B_2	0	R_1	G_1	B_1	0	R_0	G_0	B_0

Se desempaquetan $\texttt{XMM}_1$ y $\texttt{XMM}_3$ de tal manera que $\texttt{XMM}_1$ tenga la parte baja y $\texttt{XMM}_3$ la parte alta. Ambos terminarían convirtiéndose en registros de words empaquetados. Esto se hace con el fin de poder luego ejecutar una suma horizontal de a word sin el riesgo de que el resultado quede fuera del rango de representación.

$\texttt{XMM}_1$	0	R_1	G_1	B_1	0	R_0	G_0	B_0

$\texttt{XMM}_3$	0	R_3	G_3	B_3	0	R_2	G_2	B_2

Se ejecutan las sumas horizontales necesarias para que $\texttt{XMM}_1$ tenga el resultado

$\texttt{XMM}_1$	0	0	0	0	S_3	S_2	S_1	S_0

donde $S_i = R_i + G_i + B_i$

Se convierte $\texttt{XMM}_1$ a float para poder realizar la multiplicación por los factores correspondientes, y luego se ejecutan varios shuffle de tal manera que haya un registro completo por cada pixel

$\texttt{XMM}_1$	S_0	S_0	S_0	S_0

$\texttt{XMM}_2$	S_1	S_1	S_1	S_1

$\texttt{XMM}_3$	S_2	S_2	S_2	S_2

$\texttt{XMM}_4$	S_3	S_3	S_3	S_3

Se multiplica a los cuatro registros por el siguiente vector de factores que se encuentra alojado en $\texttt{XMM}_{15}$

$\texttt{XMM}_{15}$	0	0,5	0,3	0,2

Lo cual da como resultado

$\texttt{XMM}_1$	0	$0,5 \cdot S_0$	$0,3 \cdot S_0$	$0,2 \cdot S_0$
$\texttt{XMM}_2$	0	$0,5 \cdot S_1$	$0,3 \cdot S_1$	$0,2 \cdot S_1$
$\texttt{XMM}_3$	0	$0,5 \cdot S_2$	$0,3 \cdot S_2$	$0,2 \cdot S_2$
$\texttt{XMM}_4$	0	$0,5 \cdot S_3$	$0,3 \cdot S_3$	$0,2 \cdot S_3$

Es decir

$\texttt{XMM}_1$	0	O_0^r	O_0^g	O_0^b
$\texttt{XMM}_2$	0	O_1^r	O_1^g	O_1^b
$\texttt{XMM}_3$	0	O_2^r	O_2^g	O_2^b
$\texttt{XMM}_4$	0	O_3^r	O_3^g	O_3^b

Se reconvierte cada registro a int y se empaquetan de manera saturada los datos en $\texttt{XMM}_1$. La saturación es la solución al overflow de la componente roja, ya que la instrucción misma la transforma en 255 en caso de ser mayor.

$\texttt{XMM}_1$	0	O_3^r	O_3^g	O_3^b	0	O_2^r	O_2^g	O_2^b	0	O_1^r	O_1^g	O_1^b	0	O_0^r	O_0^g	O_0^b

Por el lado de $\texttt{XMM}_0$, se eliminan los datos de los colores y se deja solo el alpha para poder luego ”fusionar”los datos con $\texttt{XMM}_1$

$\texttt{XMM}_0$	A_3	0	0	0	A_2	0	0	0	A_1	0	0	0	A_0	0	0	0

Por último, se unen los datos de $\texttt{XMM}_0$ y $\texttt{XMM}_1$ y se almacena en la imagen destino.

$\texttt{XMM}_0$	O_3^a	O_3^r	O_3^g	O_3^b	O_2^a	O_2^r	O_2^g	O_2^b	O_1^a	O_1^r	O_1^g	O_1^b	O_0^a	O_0^r	O_0^g	O_0^b

La cantidad de operaciones de punto flotante por pixel en esta implementación es de $\frac{4op}{4px} = \frac{1op}{px}$, La cual es menor comparado a la cantidad de operaciones por pixel de la implementación en C (la cual es $\frac{3op}{px}$).

Con la finalidad de aumentar el rendimiento de esta implementación, se decidió utilizar más registros para procesar más cantidad de memoria en un mismo loop, de modo tal que la ejecución fuera de orden fuera más efectiva y se realice la menor cantidad de saltos condicionales posible.

Como se pudo ver en el desarrollo del algoritmo, solo cinco registros fueron utilizados para realizar todo el proceso ($\texttt{XMM}_0$ ..$\texttt{XMM}_4$), esto nos permite procesar de a 12 píxeles por ciclo, utilizando 15 de los 16 registros xmm, dejando espacio libre para el registro extra que contiene los factores por los que hay que multiplicar.

También es necesario utilizar un registro lleno de ceros para desempaquetar los datos, lo cual haría necesario contar con 17 registros o acceder a memoria, pero como ese registro de ceros solo es necesario durante una sola parte del algoritmo (en la cual aún no están en uso todos los registros) se puede utilizar algún registro que aún no se haya procesado.

La problemática que surge a partir de procesar de a 12 píxeles, es que no se sabe si se puede estar procesando píxeles de más, ya que solo se tiene como hipótesis que el ancho de la imagen multiplicado por su altura es un número que es múltiplo de 8, y por ende no siempre tendremos una imagen cuyo tamaño total sea múltiplo de 12. Para solucionar esto, se procesa de a 12 píxeles todas las veces que se

pueda, y cuando ya no se pueda más, se procesa de a 4 como originalmente se hacía, hasta terminar de procesar toda la imagen. Notar que no puede llegar a ocurrir que el algoritmo haya tenido que procesar de a 4 píxeles más de 2 veces.

4.2.3. Asm - AVX2

En AVX2 podemos utilizar los registros extendidos `YMM` de 256 bits, por lo cual podemos procesar el doble de píxeles que en la implementación SSE. También contamos con instrucciones no destructivas que nos ahorran un par de pasos a la hora de implementar. Por ejemplo, podemos pasar de

```
MOVDQA XMM1, XMM0
PSLLD XMM1, 8
PSRLD XMM1, 8
```

a

```
VPSLLD XMM1, XMM0, 8
VPSRLD XMM1, XMM1, 8
```

La conversión del algoritmo es directa con respecto a SSE. La estrategia es la misma, con la salvedad de que hay que tener cuidado con algunas instrucciones que no funcionan exactamente igual a su instrucción análoga de SSE.

Por ejemplo, a la hora de convertir los registros a word ejecutando VPUNPCKLBW y VPUNPCKHBW, estos quedarían de la siguiente manera

```
VPUNPCKLBW XMM3, XMM1, XMM7
VPUNPCKHBW XMM1, XMM1, XMM7
```

$\texttt{YMM}_1$	0	R_5	G_5	B_5	0	R_4	G_4	B_4
	0	R_1	G_1	B_1	0	R_0	G_0	B_0

$\texttt{YMM}_3$	0	R_7	G_7	B_7	0	R_6	G_6	B_6
	0	R_3	G_3	B_3	0	R_2	G_2	B_2

Por lo general, en las instrucciones de AVX2 se mantiene la misma estructura que se vio en el reciente ejemplo: Primero se procesa la parte baja de los dos registros fuente, y luego se procesa la parte alta de cada uno, en vez de procesar cada uno por completo a la vez.

Esta implementación ejecuta la misma cantidad de operaciones en punto flotante que SSE, pero procesa el doble de píxeles. Por lo tanto, por cada loop obtenemos un rendimiento de $\frac{4op}{8px} = \frac{1op}{2px}$.

4.3. Experimentos

4.3.1. Comparación entre implementaciones

A continuación se muestran una serie de experimentos que permiten corroborar las hipótesis planteadas anteriormente en el desarrollo de las implementaciones en cuanto a tiempo de ejecución.

Repasando rápidamente lo que se planteó hasta ahora, se contaron las operaciones en punto flotante que realiza cada una de las implementaciones para hacer una aproximación de qué tan rápido corren y compararlas entre sí. Se llegó a lo siguiente:

Implementación	Cant. op. punto flotante
C	$\frac{3op}{px}$
SSE	$\frac{1op}{px}$
AVX2	$\frac{1op}{2px}$

Viendo la tabla, es de esperarse que la implementación en SSE presente un speedup de 300 % con respecto a C y AVX2 presente un speedup de 600 % con respecto a la misma.

Luego de realizar varias corridas de cada una de las implementaciones, se llegaron a los siguientes resultados en cuanto al tiempo total de ejecución:

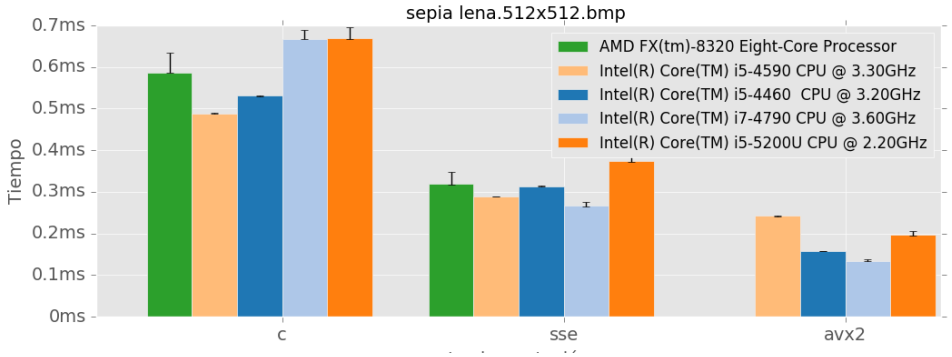


Figura 10: Tiempo de ejecución del filtro sepia sobre lena.bmp 512x512

Se puede inferir entonces que algunas de las suposiciones anteriores en cuanto a mejoras de performance resultaron ser verdaderas. La implementación en SSE corre más rápido que la misma en C y la implementación en AVX2 corre aún más rápido que SSE. Veamos con más detalle qué tanto representa esa mejora.

En el siguiente gráfico se explicita con más detalle qué tanto más rápidas son las dos implementaciones en ASM con respecto a C compilado con la optimización O3:

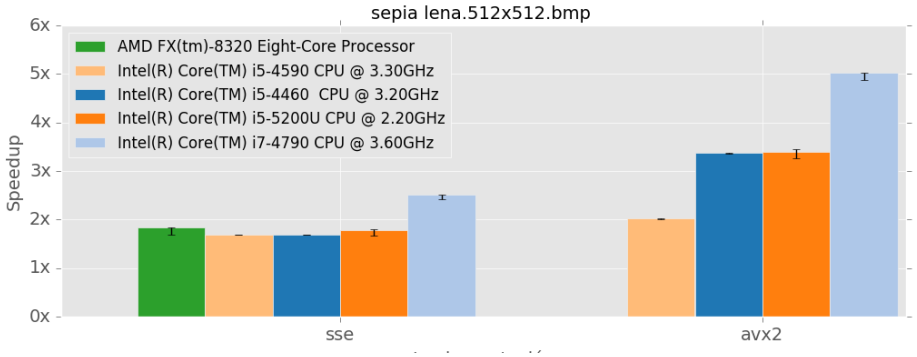


Figura 11: Tiempo de ejecución relativo de implementaciones ASM de sepia contra la implementación C compilada con -O3

Se puede ver que SSE presenta un speedup de poco menos de 200 % y AVX2 presenta un speedup de casi 350 %. Esto resulta distinto de lo que se predijo anteriormente (speedups esperados de 300 % y 600 % respectivamente). Esto puede deberse a varias causas:

- Se compiló la implementación en C con la opción de optimización automática O3, que utiliza operaciones de SSE, con lo cual podría llegar a ser que hace menos operaciones de punto flotante por pixel que las esperadas.
- Por la misma razón, es probable que al compilar se hayan ordenado las operaciones de tal manera de optimizar la performance de pipe-lining del procesador, pudiendo ejecutar más operaciones en la misma cantidad de ciclos.
- Además de operaciones en punto flotante, se realizan -en todas las implementaciones- operaciones de conversión de datos de entero a punto flotante y viceversa, lo cual también puede afectar en la performance ya que son operaciones pesadas.

4.3.2. Comparación entre optimizaciones de C

Las diferencias de performance entre la implementación C sin optimizar (O0) y las distintas optimizaciones se puede apreciar en el siguiente gráfico:

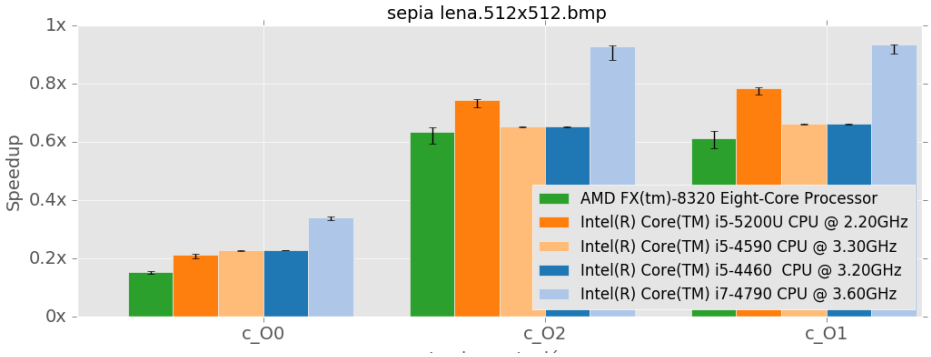


Figura 12: Tiempo de ejecución relativo de diferentes optimizaciones de la implementación en C respecto a lo optimización O3

Como puede verse, la versión sin optimizar es muchas veces más lenta que la versión completamente optimizada. Es decir que la diferencia entre la implementación en C sin optimizar y las implementaciones en SSE y AVX2 es aún más grande en cuanto a performance.

4.3.3. Caché misses en función del tamaño de imagen

Como siguiente cuestión, nos planteamos la pregunta sobre qué tanto porcentaje de Cache Misses se presentan al correr los códigos. Para encontrar una respuesta, decidimos hacer un experimento en el cual surgieron los siguientes resultados para la implementación en SSE:

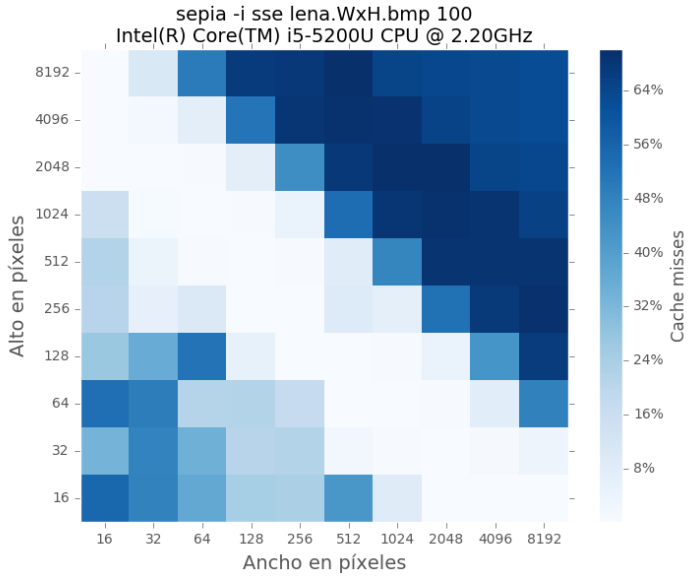


Figura 13: Porcentaje de caché misses en función del tamaño de imagen al correr el filtro sepia en un procesador con 3MB de caché

Los casos que se encuentran alrededor de 128x128 tienen un miss rate ligeramente menor debido a que varios píxeles ya se encontraban en cache a la hora de volver a testear la misma imagen pero con distinto tamaño. Luego, al aumentar el ancho o el alto de la imagen, el miss rate va aumentando progresivamente ya que cada vez aumenta más la cantidad de datos que no estaban anteriormente en cache.

Es importante resaltar cómo se produce un cambio brusco de color en la diagonal que va de 4096x128 a 128x4096. Para poder explicar este fenómeno es necesario recalcar que el procesador con el cual se realizó este experimento posee 3MB de memoria cache. También hay que recordar que cada pixel ocupa

4B en memoria. Si realizamos algunas cuentas con cualquiera de los elementos de la diagonal antes mencionada:

$$(4096 \cdot 128)px = 2^{19}px = 2^{19} \cdot 4B = 2 \cdot 2^{20}B = 2MB$$

Ahora que sabemos esto, tiene más sentido ver por qué a partir de que el tamaño total de la imagen es mayor o igual a 2^{19}, comienza a haber un aumento considerable de cache misses, ya que la imagen misma comienza a no entrar en la cache entera. Si nos movemos a la siguiente diagonal (es decir, multiplicamos por 2 el tamaño de la imagen) estaríamos trabajando ya con casos que ocupan 4MB, lo cual es incluso mayor al tamaño de la cache con la que estamos trabajando. Por eso a partir de ese punto, todas las imágenes que tengan un tamaño mayor o igual poseen un miss rate bastante similar.

Esta misma estructura se repite para los casos de las implementaciones en C y en AVX2. Esa es la razón por la cual no incluimos también los gráficos para esos dos casos ya que presentan gráficos muy similares. La única diferencia es que el máximo miss rate en C es de $\sim 80\,\%$ y en AVX es de $\sim 60\,\%$. No sabemos bien por qué ocurre esto, pero suponemos que puede ser debido al tamaño de las lecturas. A medida que se procesan más píxeles por registro, menos lecturas hay que ejecutar.

5. Conclusiones y trabajo futuro

Considerando los resultados de nuestros experimentos, creemos que la implementación de partes de un programa en assembler vale el esfuerzo cuando nuestro performance está limitado por una rutina que puede ser paralelizada. Ya que el tiempo que lleva programar en assembler es bastante mayor a lenguajes de mas alto nivel debemos considerar nuestros recursos antes de realizar todo el programa en asm.

En los filtros que realizaban cálculos logramos una gran mejora de performance al implementar usando la extensión AVX2 contra la versión SSE. Hay que tener mucho cuidado al usarla ya que varias instrucciones resultaron tener comportamientos diferentes a los que uno esperaría, sobre todo cuando se quiere hacer operaciones entre valores como un shuffle o una suma horizontal. De todos modos, leyendo cuidadosamente el manual y documentando bien cada paso se logran muy buenos resultados.

Nos quedó en el tintero experimentar con flotantes de doble precisión y ver si podemos lograr precisiones exactas sin pagar la penalidad de las cuentas con enteros.

También querríamos constatar, ya que este año salen al mercado procesadores con la extensión AVX-512, si el aumento de rendimiento se mantiene mientras mas suba el tamaño de los registros, o nos encontraremos con demasiado overhead de los cálculos auxiliares.

Referencias

[1] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Number 248966-032. January 2016.

[2] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-058US. April 2016.

[3] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2002.