TP Final

Organizacion del Computador II
Segundo Cuatrimestre de 2014

Alumno	LU	Correo electrónico
De Carli, Nicolás	164/13	nikodecarli@gmail.com

Docente	Nota

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

Índice

1. Acerca del proyecto

Durante la cursada de la materia, para el segundo trabajo práctico se implementaron 3 filtros de imágenes en assembler de la arquitectura AMD64, haciendo uso de las instrucciones de SIMD que proveen los distintos sets de instrucciones SSE. Para este trabajo práctico final se reescribieron las funciones del tp original utilizando AVX, que es un set de instrucciones SIMD de intel más moderno. Tambien se implementaron en assembler de ARMv7-A utilizando NEON, el set de instrucciones SIMD de la empresa inglesa. Luego se escribieron en assembler de cada una de las dos arquitecturas 3 filtros de imágenes nuevos y 3 funciones nuevas. Por último se procedió a hacer benchmarks para medir la cantidad de ciclos de procesador que consume cada una de las 18 funciones escritas. El fin del proyecto es poder ver en cada una de las 9 funciones cómo se compara la performance de la implementación para AMD64, con la version de assembler de ARMv7-A.

2. Sobre las arquitectura ARM

En 1983, cuando ingenieros de Acorn Computers empezaron a pensar el diseño del primer chip ARM, no habia una empresa que domine el mercado de microprocesadores como lo hace Intel hoy en las computadoras de escritorio. Se podía ver mucha creatividad en una gran diversidad de arquitecturas de procesadores, parecía haber muchas más oportunidades de crear un procesador nuevo. En Acorn simplemente querían diseñar una arquitectura que sirva para sus computadoras, y veían con escepticismo la idea de llegar a algo que no se haya hecho antes. Sin embargo, tenían la firme idea que la performance de cómputo de una computadora estaba determinada principalmente por el bandwidth de memoria del procesador. Por ejemplo el chip 32016 de National Semiconductor tenía un buen conjunto de instrucciones de 32 bits, sin embargo su performance escalaba con respecto al ancho de banda de memoria. En esa época los procesadores de 16 bits no podían usar en su totalidad el bandwidth que proveía la memoria de los sistemas, la gente de Acorn creía que esto era un grave error de arquitectura. Mientras pensaban cómo solucionar este problema, uno de los ingenieros trajo un paper de David Patterson titulado ”The Case for the Reduced Instruction Set Computer”, este trabajo describía un nuevo tipo de procesador diseñado por estudiantes. En aquella época los procesadores eran lo que ahora se conoce como CISCs, ya que muchas de sus instrucciones ejecutaban más de una óperación atómica. El problema con este modelo es que en la mayoría de los casos no había una biyección entre lenguajes e instrucciones, entonces muchas veces el compilador traducía una linea de codigo en un conjunto de directivas de procesador que hacían más de lo que se deseaba, porque no había forma de hacer solo lo necesitado. El enfoque RISC era tener un conjunto chico de instrucciones simples y atómicas, esta idea implicaba ir en dirección contraria a dónde iba la industria, y además la gente de Acorn dudaba de que una idea tan obvia no haya sido probada por los fabricantes de procesadores, sin embargo decidieron diseñar y crear un chip RISC. Asi nació en 1985 el primer procesador ARM. Los primeros benchmarks mostraron que la performance del chip era aproximadamente 25 veces mayor a la de la ”BBC Micro”, exactamente lo que predecían los calculos de ancho de banda de memoria. Como los procesadores eran diseñados con el fin de funcionar en computadoras de escritorio, originalmente ser de bajo consumo no era un objetivo, pero sí lo era ser de bajo costo. Por lo tanto se queria lograr que la cubierta del procesador sea de plástico, ya que era 100 veces más barata que una hecha con cerámica. Para que pueda andar correctamente en plástico, el consumo no debía superar 1 watt. Como en la época las herramientas de ingeniería no eran tan precisas, los ingenieros apuntaron a un consumo mucho menor para tener un margen de error grande. El prototipo final consumía 0.1 watts. Las claves del bajo consumo del chip ARM eran su simpleza, y que utilizaba solamente 25 mil transistores, el procesador 80386 de Intel lanzado también en 1985 contenía 275 mil. Las propiedades de bajo consumo, un tamaño de chip chico y buena performance fueron las claves del éxito de los procesadores ARM.

En 1990 se fundó ARM Holdings, la empresa que se dedicó a seguir desarrollando procesadores de tipo RISC. Hasta mediados de 2016 se vendieron mas de 86 billones de procesadores ARM. Se usan principalmente en dispositivos que se benefician de su bajo consumo, como celulares y tablets. Tambien se encuentran en aparatos que no necesitan poder de cómputo como routers, lavadoras o microondas.

2.1. ARMv7-A

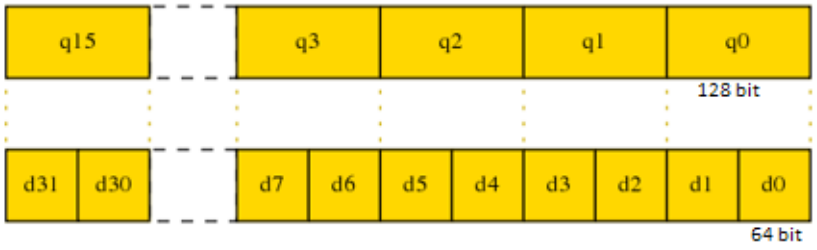
Esta arquitectura incorpora algunas caracteristicas de tipo RISC. Por ejemplo tiene un solo gran arreglo uniforme de 16 registros de 32 bits. El Stack Pointer y Link Register son almacenados entre llamados de funciones en los registros R13 y R14 respectivamente (siendo R0 a R15 el rango). El registro R15 almacena el program counter. Las Operaciones de procesamiento de datos solo trabajan con contenidos de registros y parámetros de instrucciones, no es posible realizarlas directamente sobre memoria. Modos de direccionamiento simples, todas las direcciones de load y store son determinadas por contenidos de registros y parámetros de instrucciones.

Además, esta arquitectura de ARM provee algunas características que no son propias de un RISC. Por ejemplo instrucciones que combinan un shift con una operacion lógica o aritmética. Otra cosa interesante es que se pueden lograr múltiples loads y stores con una sola instrucción, esto sirve para tratar de maximizar el throughput de datos. Con el fin de maximizar el throughput de ejecución, se pueden agregar condiciones a muchas instrucciones, que finalmente son ejecutadas solo cuando la condición se cumple. También se proveen modos de direccionamiento auto incrementales o decrementales para optimizar ciclos. Estas mejoras a un diseño RISC básico hacen que la arquitectura tenga un buen balance de performance, tamaño de programas chico, bajo consumo de energía y un área de silicona chica.

La arquitectura ARMv7-A esta orientada a aplicaciones de uso diario, es por esto que soporta un Virtual Memory System Architecture (VMSA), basado en un Memory Management Unit (MMU), que es una característica requerida por varios sistemas operativos.

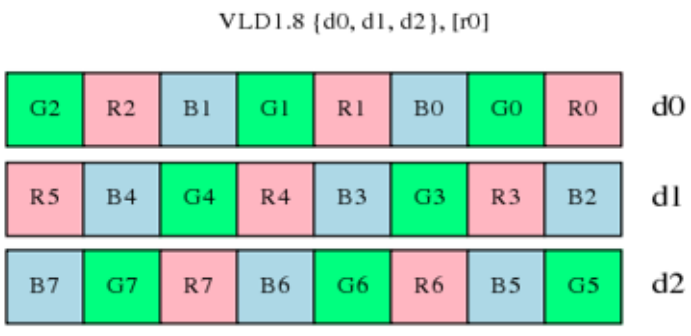
2.2. NEON

Opcionalmente, algunos chips basados en la arquitectura ARMv7-A también implementan un set de instrucciones SIMD llamado NEON. Este provee un banco de registros híbrido, que puede ser usado como 16 registros de 128 bits llamados de q0 a q15, o como 32 registros de 64 bits enumerados de d0 a d31. En la imagen de abajo se puede ver ilustrado el concepto.



The NEON register bank.

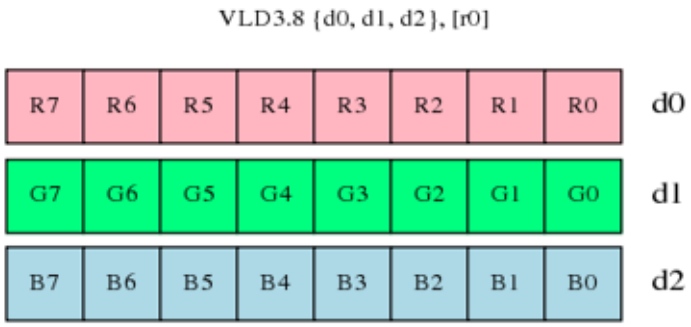
Neon fue diseñado para acelerar la performance de aplicaciones multimedia y procesamiento de señales, es por esto que nos provee con diversas formas de cargar datos de memoria a los registros anteriormente mencionados. Supongamos que tenemos una imagen RGB de 24 bits, en la cual los píxeles están ordenados en memoria de esta manera: R, G, B, R, G, B... Si usamos un load que lleva datos RGB de manera linear desde memoria a los registros, nos quedaría así:



Loading RGB data with a linear load.

Si tuviesemos que intercambiar los colores rojo y azul, utilizando el input de esa manera nos quedaría

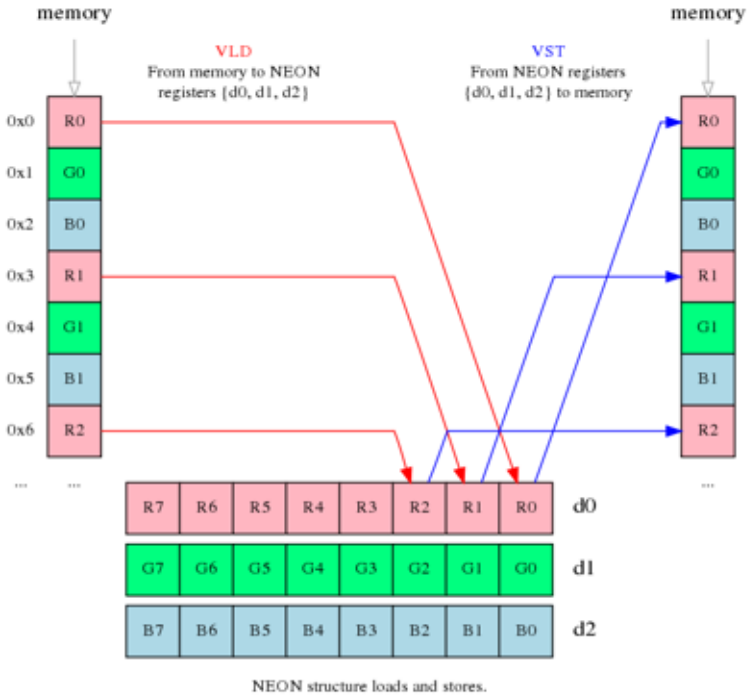
codigo poco elegante, ya que tendríamos que utilizar máscaras, shifts y combinaciones. Probablemente no sería eficiente. NEON provee instrucciones de load y store que nos ayudan en esta situación, que traen datos desde memoria de manera separada. Por ejemplo en este caso se podría usar $VLD3$ para separar rojo, verde y azul mientras son cargados, la imagen de abajo muestra cómo quedan los registros d0, d1 y d2 luego de ejecutar la operacion $VLD3,8d0, d1, d2, [r0]$.



Loading RGB data with a structure load.

Como se puede observar, 24 bytes son traidos desde la dirección de memoria que está almacenada en r0, y son guardados en los registros de manera intercalada, el primer byte queda en d0, el segundo en d1, el tercero en d2, el cuarto en d0...

Ahora podemos intercambiar el registro de pixeles rojos con el de azules, utilizando solo la instrucción $VSWPd0, d2$. Luego si escribimos en memoria de nuevo con intercalado de datos, usando la instrucción $VST3$, vamos a haber canjeado los canales azul y rojo usando solo 3 operaciones. Debajo se puede ver con más de detalle cómo operan el load y store utilizados.



NEON structure loads and stores.

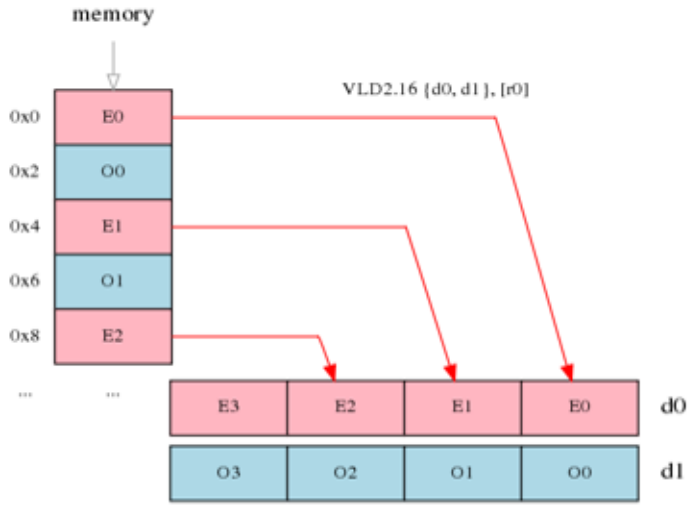
NEON nos provee con 4 diferentes tipos de loads, los elementos intercalados pueden ser de 8, 16 o 32 bits:

- VLD1: hace un load linear, carga 1 a 4 registros de 64 bits con datos desde memoria, sin usar intercalado.
- VLD2: carga 2 o 4 registros de 64 bits con datos desde memoria, intercalando elementos pares e impares de los primeros 128 bits en los primeros 2 registros, e haciendo lo mismo para los últimos 128 bits con los registros 3 y 4, si es que fueron especificados.

- VLD3: carga 192 bits desde memoria a 3 registros de 64 bits, intercalando como vimos en el ejemplo anterior con elementos de 8 bits.
- VLD4: carga 256 bits almacenados en memoria a 4 registros de 64 bits, se comporta de la misma manera que VLD3, pero intercalando de a 4 elementos.

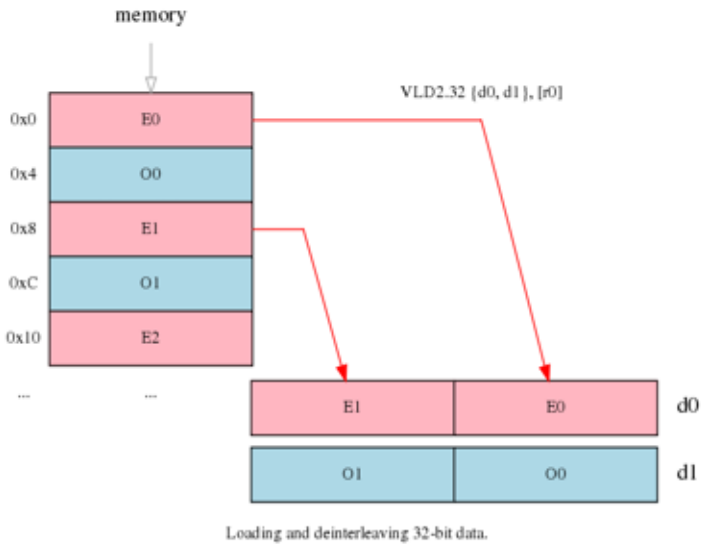
Los stores son capaces de realizar los mismos intercalados, pero llevando la información desde los registros hacia la memoria.

Veamos ahora cómo se comporta un mismo tipo de intercalado con distintos tamañanos de elementos. Si utilizamos la instrucción VLD2 para cargar elementos de 16 bits en 2 registros de 64 bits, nos quedan 4 variables en el primer registro y 4 en el segundo, con pares de elementos adyacentes en diferentes registros. Esto se puede ver ilustrado en la imagen de abajo:



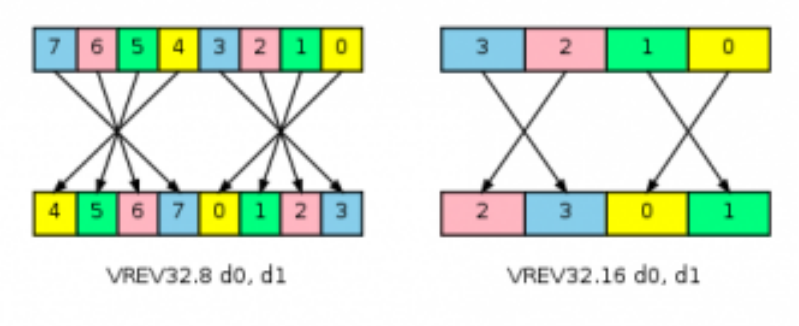
Loading and deinterleaving 16-bit data.

Si cambiamos el tamaño de elemento a 32 bits, se carga desde memoria la misma cantidad de datos, pero ahora cada registro contiene 2 variables, las cuales están de nuevo separadas entre pares e impares. Abajo podemos ver la operación:

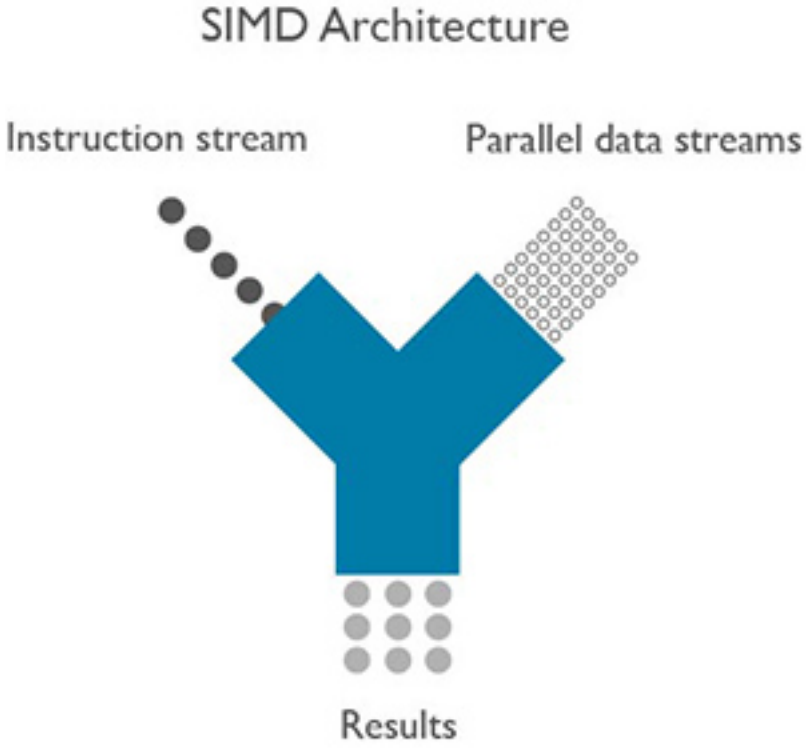


Loading and deinterleaving 32-bit data.

Esta particularidad de poder operar sobre distintos tamaños de elementos se ve en casi todas las instrucciones que provee NEON. Por ejemplo, la instrucción VREV32 revierte el orden de los elementos dentro de cada bloque de 32 bits, si utilizamos 8 bits como tamaño de variable vamos a estar invirtiendo 4 elementos por bloque, mientras que si especificamos 16 bits como medida de cada dato se van a ver invertidos solo 2 elementos por cada 32 bits. Esta diferencia se puede ver ilustrada abajo:



Como es esperado, con NEON también es posible realizar una operación aritmética a muchas variables con una sola instruccion. En estos casos especificar el tamaño de elemento nos va a cambiar cosas como overflow y saturación. Combinando los stores y loads vistos con instrucciones que operan sobre muchas variables a la vez, podemos lograr una increible mejora de performance. Esta situación se puede ver graficada en la imagen de abajo:



Hoy en día algunos compiladores soportan Auto Vectorización, es decir intentan explotar las funcionalidades SIMD de NEON para maximizar la performance de la aplicación.

3. Metodología

Para poder medir la performance, se implementó en C un programa que levanta en RAM una imagen del disco, le aplica los filtros que uno desee, y la vuelve a almacenar en memoria secundaria. Todas las funciones miden el reloj del procesador antes y despues de realizar los cálculos correspondientes, y se lo pasan como valor de retorno al programa. Dentro del programa, habia un ciclo que corría alguno de los filtros 10 veces sin parar e imprimía la cantidad de ciclos consumida en cada iteración. Este proceso se repitió 10 veces para probar cada función, al final el promedio de las 100 iteraciones se utilizaba como resultado. En algunos pocos casos se obtuvieron mediciones anormalmente altas, se consideró que algo del sistema operativo había ocurrido en el medio de la ejecución programa que causó una baja de la performance, estos outliers no fueron tenidos en cuenta a la hora de calcular el promedio.

Los benchmarks de AMD64 fueron corridos en un procesador Intel Core i7-3610QM, el cual es una implementación de la arquitectura conocida como Ivy Bridge. Por otro lado los tests de ARMV7-a fueron realizados en un chip ARM Cortex-A15 r3, el cual estaba dentro de un procesador Tegra K1 de nvidia. Este último procesador se puede obtener de manera embebida en una placa que bootea Ubuntu, motivo conveniente por el cual fue elegido para el trabajo. Se decidió utilizar el CPU de Intel mencionado anteriormente porque salió al mercado el mismo año que el de ARM. Las plataformas que contenían a cada procesador corrían Ubuntu 14.04 al momento de las pruebas. Frecuentemente el kernel del sistema operativo reduce la performance del sistema con el fin de bajar el consumo de energía, para prevenir que esto ocurra se ejecutaron scripts en ambos sistemas antes de correr cada medicion.

Todas las funciones operan sobre imagenes de tipo BMP. Dado que este formato almacena cada color de cada pixel de manera explícita, qué archivo se usa para cada benchmark no es relevante. Si cambia la resolución, cada función fue probada en una imagen HD, esto es 1280x720, y en una FullHD, es decir que tiene 1920x1080 pixels.

4. Funciones implementadas

4.1. Filtros Originales

4.1.1. Sierpinski

El fltro sierpinski toma una imagen fuente y genera un efecto fractálico encima, simulando los triángulos de sierpinski pero con cuadrados. El filtro toma la posición actual que esta recorriendo en la imagen y, dada cierta cuenta, genera un coeficiente entre 0 y 1, que se multiplica sobre cada componente de cada pixel.

$$\mathsf{dst}_{(i,j)} = \mathsf{src}_{(i,j)} * \mathsf{coef}_{(i,j)}$$

$$\mathsf{coef}_{(i,j)} = \frac{1}{255{,}0}(\lfloor \frac{i}{cant_{filas}} * 255{,}0 \rfloor \oplus \lfloor \frac{j}{cant_{cols}} * 255{,}0 \rfloor)$$

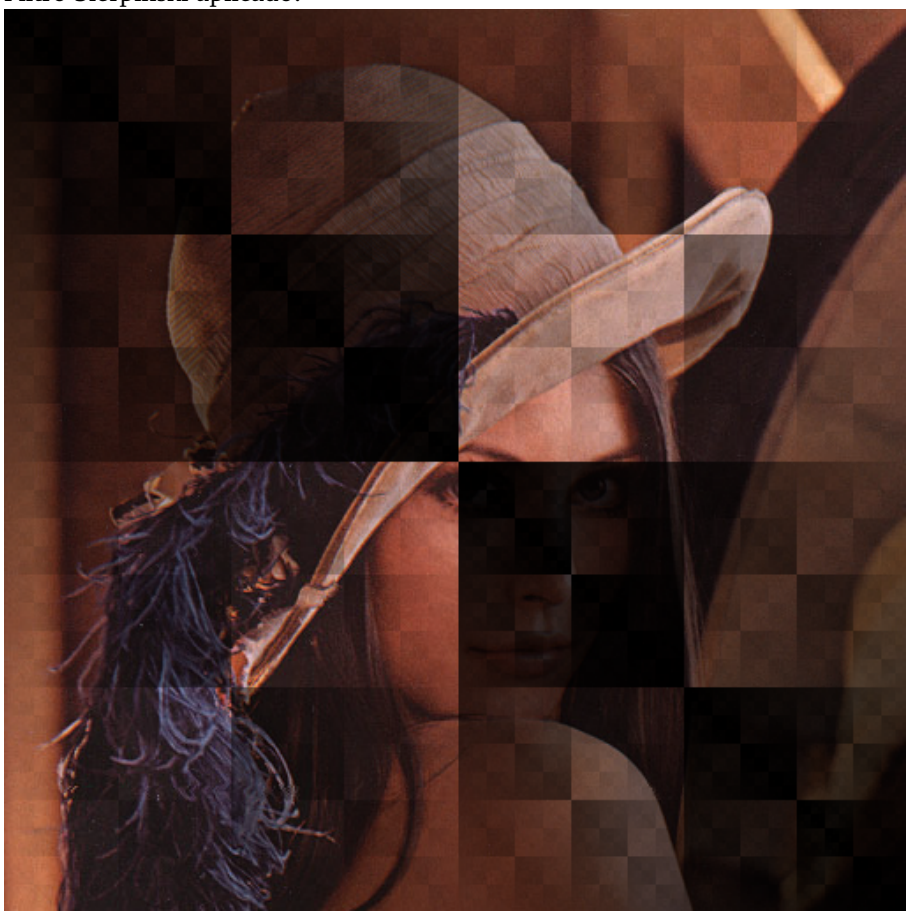
Las fracciones denotan que la operación se hace en punto otante, mientras que las partes enteras denotan que se pasan a enteros para poder hacer el XOR de los bits. Todas las conversiones de doble a entero se hacen truncando, no redondeando.

En imágenes HD, la versión de intel tarda 3 290 585 ciclos en computarse, mientras que la de ARM consume 20 623 220 ciclos de cpu.

Para archivos FullHD, el filtro sobre el cpu de Intel consume 7 702 815 ciclos, la versión de ARM 46 208 651.

Imagen original, este cuadro se llama Lena:

Filtro Sierpinski aplicado:



4.1.2. Motion Blur

Este filtro toma una imagen fuente y aplica un efecto de desenfoque de movimiento. Esto se logra tomando parte del valor del pixel original y añadiendo partes del valor de sus vecinos. En este trabajo, el desenfoque será de un movimiento de 45 grados. Para cada componente independiente del pixel (R, G y B) la fórmula matemática sería:

$dst(i,j) = 0{,}2 \times src_{(i-2,j-2)} + 0{,}2 \times src_{(i-1,j-1)} + 0{,}2 \times src_{(i,j)} + 0{,}2 \times src_{(i+1,j+1)} + 0{,}2 \times src_{(i+2,j+2)}$

Al resultado se lo satura en 255.Además, dado que en los bordes no es posible calcular mblur por la ausencia de vecinos, se escribirá en esos casos el valor 0.

Motion Blur aplicado a Lena:

En imágenes HD, la versión de intel tarda 3 490 812 ciclos en computarse, mientras que la de ARM consume 21 875 228 ciclos de cpu.

Para archivos FullHD, el filtro sobre el cpu de Intel consume 8 256 004 ciclos, la versión de ARM 41 469 436.

4.1.3. Bandas

El filtro bandas toma una imagen fuente y genera bandas en varios tonos de gris. Se suman los componentes R, G y B de cada pixel, dando como resultado un número b entre 0 y 765.

$b(i,j) = src.r(i,j) + src.g(i,j) + src.b(i,j)$

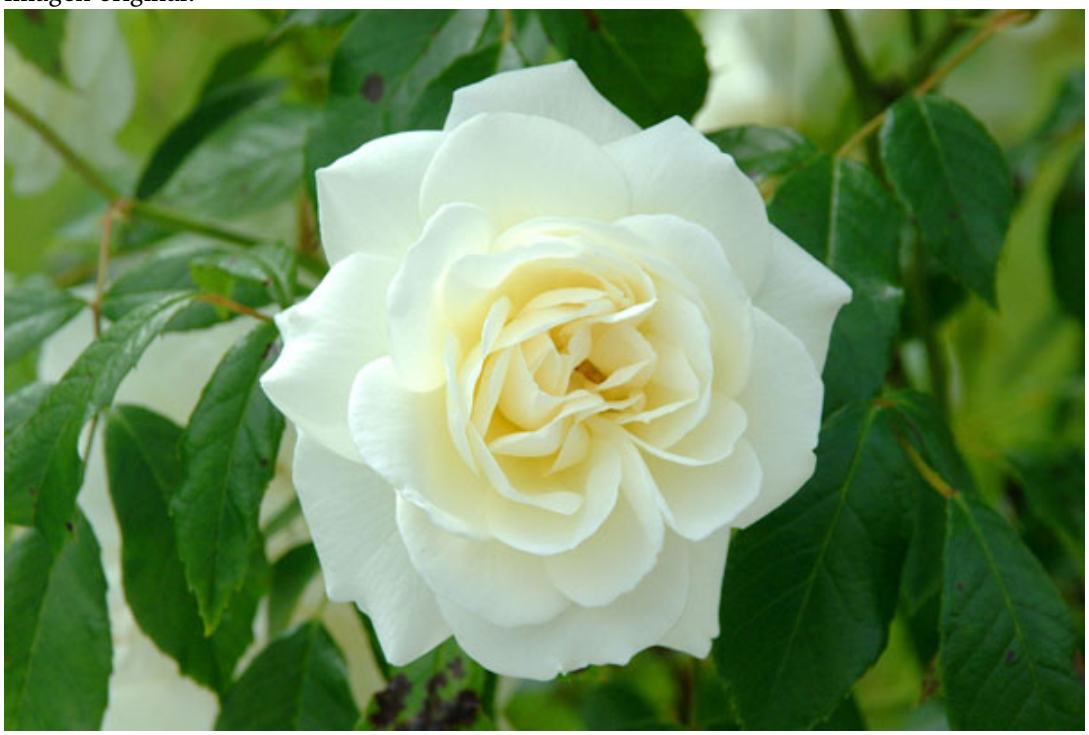
En función de b, se determina el color del pixel en la imagen destino

$$dst(i,j) < r,g,b >= \begin{cases} < 0,0,0 > & \text{si } b < 96 \\ < 64,64,64 > & \text{si } 96 \leq b < 288 \\ < 128,128,128 > & \text{si } 288 \leq b < 480 \\ < 192,192,192 > & \text{si } 480 \leq b < 672 \\ < 256,256,256 > & \text{si } 672 \leq b \end{cases} \quad (1)$$

En imágenes HD, la versión de intel tarda 1 401 296 ciclos en computarse, mientras que la de ARM consume 4 905 470 ciclos de cpu.

Para archivos FullHD, el filtro sobre el cpu de Intel consume 3 389 953 ciclos, la versión de ARM 11 299 636.

Imagen original:



Filtro blanco aplicado:

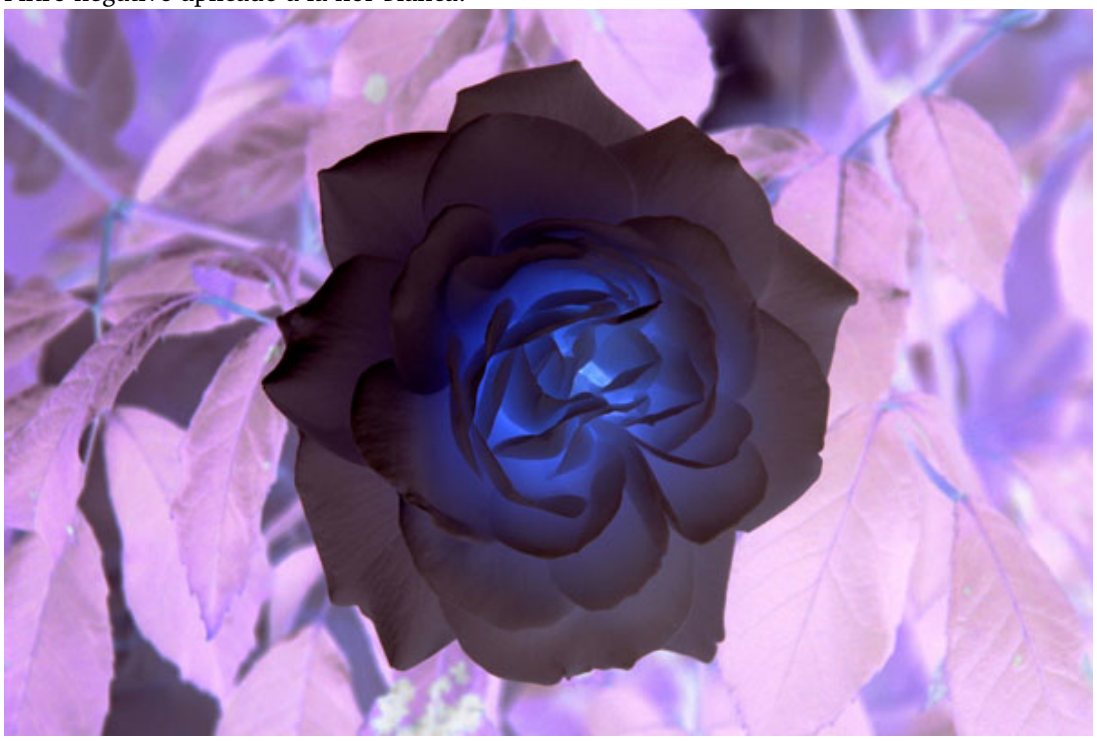
4.2. Filtros Nuevos

4.2.1. Negativo

El filtro negativo invierte el color de la imagen. Esto se logra simplemente dejando en cada color de cada pixel de la imagen destino el resultado de realizar un xor entre el color original y 255, es decir:

$dst.r(i,j) = src.r(i,j) \oplus 255$
$dst.g(i,j) = src.g(i,j) \oplus 255$
$dst.b(i,j) = src.b(i,j) \oplus 255$

Filtro negativo aplicado a la flor blanca:



En imágenes HD, la versión de intel tarda 561 113 ciclos en computarse, mientras que la de ARM consume 2 898 591 ciclos de cpu.

Para archivos FullHD, el filtro sobre el cpu de Intel consume 1 592 112 ciclos, la versión de ARM 7 651 370.

4.2.2. Sharp

Este filtro resalta los bordes dentro de la imagen. Para lograr este efecto, se realiza la siguiente cuenta:

$dst(i,j) = src(i,j) \times 10 - src_{(i-1,j-1)} - src_{(i,j-1)} - src_{(i+1,j-1)} - src_{(i-1,j)} - src_{(i+1,j)} - src_{(i-1,j+1)} - src_{(i,j+1)} - src_{(i+1,j+1)}$

Es decir, a cada pixel se lo multiplica por 10 y se le resta cada uno de sus 9 vecinos.

Para los pixeles $dst(i,j)$ con $i = 0$ o $j = 0$ o $i = cantColumnas - 1$ o $j = cantFilas - 1$ no es posible calcular la ecuación anterior, por lo tanto los vamos a dejar en negro.

En imágenes HD, la versión de intel tarda 3 150 181 ciclos en computarse, mientras que la de ARM consume 19 329 151 ciclos de cpu.

Para archivos FullHD, el filtro sobre el cpu de Intel consume 7 123 589 ciclos, la versión de ARM 40 266 345.

Filtro sharp aplicado a la flor blanca:

4.2.3. Gris

Este filtro simplemente multiplica por 0.4 al color rojo del pixel original, y replica ese resultado en todos los colores del pixel destino.

En imágenes HD, la versión de intel tarda 1 069 110 ciclos en computarse, mientras que la de ARM consume 3 480 991 ciclos de cpu.

Para archivos FullHD, el filtro sobre el cpu de Intel consume 2 538 401 ciclos, la versión de ARM 7 773 476.

Filtro Gris aplicado a Lena:

4.3. Codificación de imagen

En el formato BMP, todos los pixeles tienen 4 componentes: Rojo, Verde, Azul y Brillo. Además, al comienzo de cada archivo se especifica cuantos bits van a estar destinados a cada parte. El formato más usado en cada pixel dedica 8 bits a cada uno, ocupando 32 bits cada pixel. Todos los filtros del trabajo operan con imagenes de esta característica. Sin embargo, existe otra codificación muy usada, que es darle 8 bits a cada color y 0 bits al brillo, ya que esta componente ya casi no se usa y achica el tamaño de cada pixel a 24 bits. Para poder soportar este tipo de archivo, se implemento una función que toma una imagen de 24 bits por pixel (8:8:8:0), y la convierte a una de 32 bits (8:8:8:8), formato en el cual es posible aplicarle los filtros. También se hizo una función que realiza la inversa, convierte un archivo de 32 bits (8:8:8:8), a uno que utiliza el perfil de 24 bits (8:8:8:0).

De esta manera cuando se carga una imagen de 24 bits por pixel, automaticamente se la transforma al formato de 32 bits para poder aplicarle los filtros. Haya ocurrido esto o no, el usuario puede especificar que el archivo de salida sea de 24 bits o de 32 bit.

Si bien realizar los filtros para que operen sobre imagenes de 24 bits podría llegar a resultar en una mejora de performance, haciendo esto, al tener que convertir un archivo de 32 bits se pierde la información de brillo de cada pixel.

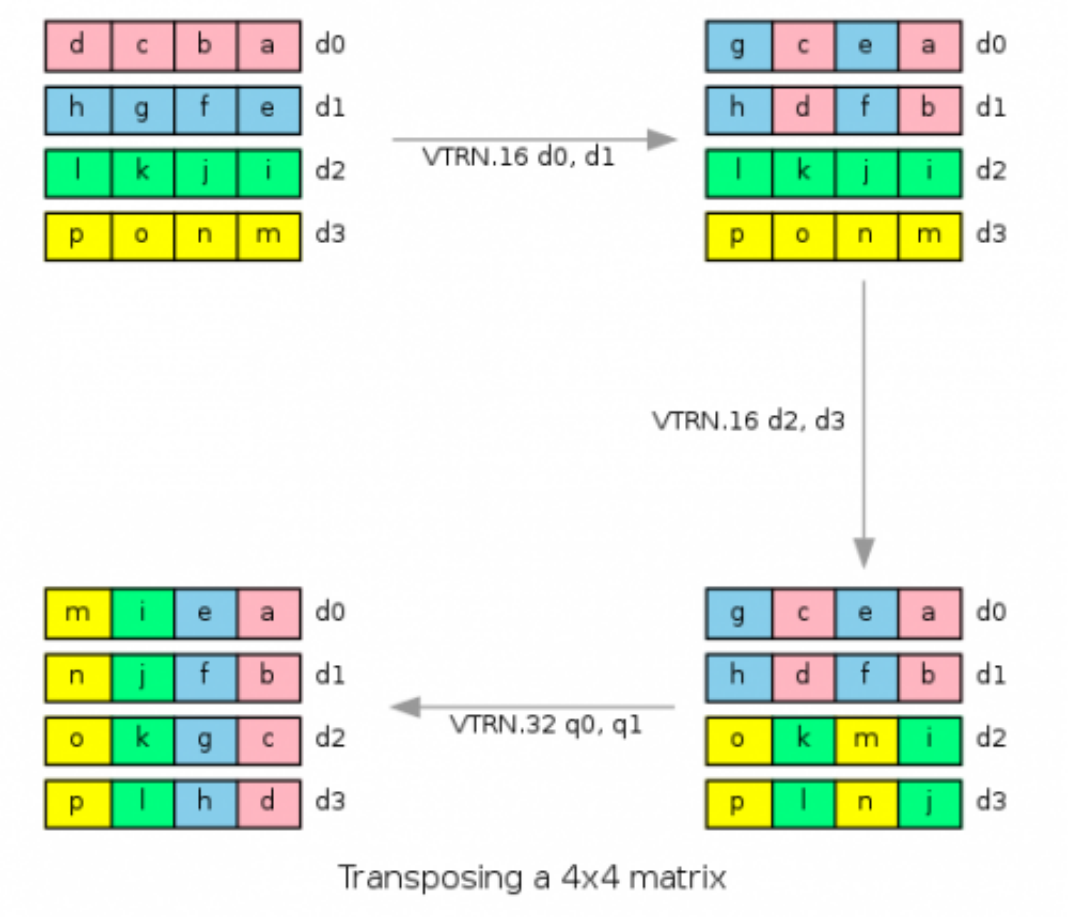
La performance de compresión y descompresión es similar (y muy buena). En imágenes HD la versión de Intel consume 675 912 ciclos, la de ARM 2 711 083. Para archivos FullHD, el procesador de Intel logra computar la función luego de 2 551 262 ciclos, al de ARM le toma 6 175 489.

4.4. Rotación de imagen

Supongamos que tenemos que rotar un bloque de 4x4 pixeles. Si lo vemos como una matriz, calculando la transpuesta y luego computando una reflexión en el eje Y de ella, nos quedan los datos iniciales rotados en 90 grados.

La idea de esta función es aplicarle ese procedimiento a cada bloque de 4x4 pixeles de la imagen original, y luego almacenarlos en la dirección de memoria correspondiente del archivo destino.

NEON provee instrucciones pensadas para ser usadas en operaciones matriciales, por ejemplo podemos computar la transpuesta de una matriz de 4x4 usando solo 3 instrucciones, de la siguiente manera:



Luego, utilizando solo la instrucción VREV vista en la sección 2.2, podemos calcular la reflexión de cada uno de los 4 vectores.

Al aplicarle esta función a Lena, nos queda la siguiente imagen:

Con respecto a tiempos, en imágenes HD Intel tarda 2 038 798 ciclos, y ARM 18 637 340. En archivos FullHD Intel consume 6 517 342 ciclos, y ARM requiere de 45 604 441.

5. Conclusión

Primero veamos en dos tablas todos los resultados de todas las funciones. El número llamado coef indica por cuanto hay que multiplicar el resultado de Intel para igualar al de ARM, en otras palabras, cuantas veces el chip de Intel es capaz de computar la función en el tiempo que ARM lo hace una vez.

720p	Intel	ARM	Coef
Sierpinski	3 290 585	20 623 220	6.27
Motion Blur	3 490 812	21 875 228	6.27
Bandas	1 401 296	4 905 470	3.50
Negativo	561 113	2 898 591	5.17
Sharp	3 150 181	19 329 151	6.14
Gris	1 069 110	3 480 991	3.26
Codificación	675 912	2 711 083	4.01
Rotación	2 083 798	18 637 340	9.14

1080p	Intel	ARM	Coef
Sierpinski	7 702 815	46 208 651	6.00
Motion Blur	8 256 004	41 469 436	5.02
Bandas	3 389 953	11 299 636	3.33
Negativo	1 592 112	7 651 370	4.81
Sharp	7 123 589	40 266 345	5.65
Gris	2 538 401	7 773 476	3.06
Codificación	2 551 262	6 175 489	2.42
Rotación	6 517 342	45 604 441	7.00

Lo primero a destacar es la impresionante performance del procesador Intel. Todas las funciones, incluso corriendo en imagenes FullHD, tardan menos de 10^7 ciclos. En algunos casos mucho menos, en imagenes HD a veces la performance es del orden de los 10^5 ciclos. Teniendo en cuenta que un procesador moderno corre a mas de $3 * 10^9$ ciclos por segundo, cualquier filtro podría ser ejecutado mas de 500 veces por segundo.

Por otro lado, podemos apreciar que las mayores diferencias de performances se dan en filtros que requieren calculos. En funciones simples ARM presenta una muy buena performance, estando debajo de la linea de los 10^7 en la mayoría de los casos. En los filtros pesados (Sierpinski, Motion Blur, Sharp y Rotación) ARM tarda alrededor de 20 millones de ciclos para imágenes HD y 40 millones en FullHD. Si bien por los resultados de Intel uno se inclinaría a pensar que esos números no son buenos, evaluando que el procesador ARM usado corre a 2.2 Ghz, podemos calcular que en un segundo cualquiera de las funciones pesadas sobre imagenes HD puede ser computada aproximadamente 110 veces. Mientras que en archivos FullHD se podría correr alrededor de 55 veces por segundo. Visto de otra manera, podemos afirmar que computar Sierpinski, Motion Blur, Sharp o Rotación en una imagen HD en el procesador ARM tarda menos de una centésima de segundo. Sobre una FullHD tarda menos que $1/50$ de segundo.

Por lo tanto podemos afirmar que ARM presenta una buena solución SIMD. Dado que el procesador consume menos de 5 Watts y que tiene pocos transistores, es perfectamente entendible que las instrucciones de cómputo le cuesten un poco más que a un Core i7. El alto ancho de banda de memoria, una de las bases teóricas de ARM, hace que filtros como Negativo que casi no manipulan datos sean muy rápidos.

Como marcamos antes, Intel presenta una excelente solución de SIMD, logrando computar funciones extremadamente rápido.

6. Bibliografía

Para empezar a programar en Assembler de ARM recomiendo leer el manual del procesador a usar. Allí se puede encontrar información sobre los registros que posee y las instrucciones que soporta.

Ademas este tutorial puede ayudar:

`http://www.davespace.co.uk/arm/introduction-to-arm/index.html`

Para aprender NEON recomiendo este tutorial:

Parte 1

Parte 2

Parte 3

Parte 4

Parte 5