DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico final

Vectorización aplicada a filtros de señales digitales

Organización del computador II

Marzo de 2017

Alumno	LU	Correo electrónico
Matias Vargas Telles	318/12	mvargas@dc.uba.ar

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. Abstract

En el presente trabajo fueron optimizados 5 filtros de señales digitales involucrados en el procesamiento de datos provenientes de sensores distribuidos de fibra óptica. Utilizando los conocimientos de vectorización adquiridos durante el dictado de la materia se identificaron, diseñaron e implementaron los algoritmos correspondientes a cada filtro en assembler de Intel. Dicha implementación fue comparada en su desempeño temporal con su contraparte en lenguaje C (optimizado por el compilador GCC) obteniendo mejores tiempos de cómputo en casi todos los casos (entre 1 y 16 veces mejor performance dependiendo el caso).

Palabras clave: SIMD - DSP - ASM - AVX / SSE

2. Introducción

El objetivo de este trabajo consiste en la optimización de 5 filtros de señales digitales mediante su implementación en assembler de Intel utilizando la vectorización que permiten las extensiones SSE y AVX. Para ello fueron analizados los algoritmos involucrados, se propuso una alternativa vectorizada de los mismos, se implementó y se comparó en tiempo de ejecución con su versión escalar en lenguaje C optimizado mediante las opciones del compilador GCC.

3. Contexto y condiciones del problema

La aplicación que genera da origen a la este trabajo consiste en el monitoreo de sensores distribuídos de fibra óptica. Estos sensores permiten monitorear alteraciones mecánicas (vibraciones) o de temperatura a lo largo de grandes distancias. Debido a las distancias que cubren generan gran cantidad de datos que deben ser debidamente procesados para obtener datos en tiempo real que permitan, por ejemplo, detectar fugas o roturas en tuberías o cambios considerables en la temperatura.

Para este fin se monitorea continuamente la intensidad de una señal conocida cuando atraviesa n puntos equiespaciados sobre la fibra. Esta señal proviene del scattering raman o rayleigh de la fibra dependiendo la aplicación. Cada scattering provee información sobre un parámetro a sensar como puede ser la temperatura o el nivel de vibración de una zona de la fibra. Este monitoreo se realiza utilizando una placa digitalizadora que entregará un valor de intensidad sobre n puntos equiespaciados sobre la fibra a frecuencias entre 40kHz y 100kHz. Esto genera una matriz de n columnas (donde n puede variar entre 100 y 40000 dependiendo la distancia a cubrir y la resolución espacia que se desee obtener de la medición) y una fila por cada medición temporal. Las columnas de la matriz se denominan **bins** o bines y las filas **shots**.

La digitalizadora entrega una de estas matrices cada cierto intervalo de tiempo configurable. Estas matrices se acumulan en memoria como vectores unidimensionales para su procesamiento.

El dato crudo se representa con shorts de 16 bits aunque los valores significativos son 14 bits ya que esa es la resolución de la placa de adquisición y digitalización utilizada (http://www.adlinktech.com/PD/web/PD_-detail.php?pid=1287). El manual de dicha placa establece que los 2 bits menos significativos de cada dato serán siempre 0 (se shiftean antes procesar cada dato) y que **cada fila de la matriz** (i.e. cada adquisición temporal) debe tener **al menos 80 puntos y su tamaño debe ser divisible por 8** para optimizar transferencias a la memoria principal. Estos datos son aprovechados para la implementación de los filtros que se ve ligeramente simplificada con esta condición. Sin embargo se aclara en la sección correspondiente cómo resolver el caso en el que no existiera esta situación.

3.1. Filtros

Uno de los primeros filtros a implementar consiste en obtener el valor medio de intensidad para cada punto de la fibra (bin) durante un período de tiempo dado en shots (filas de la matriz de dato crudo). Este filtro es el más básico y permite desahacerse de ruido gaussiano de la medición. A este filtro usualmente se le agrega también el desvío estándar de la señal para conocer su disperción en un tiempo dado. Estos dos filtros parten de una matriz de bins por shots datos y devuelven un vector con un dato por cada bin de la fibra.

Hay situaciones en las que resulta necesario normalizar la señal recibida en estas aplicaciones con otra señal conocida para filtrar posibles variaciones espurias provenientes de anormalidades en el equipo. Para estos casos se agrega un filtro que calcula el cociente de estas señales (por cada punto de la fibra hay 2 señales) y los promedia y calcula su desvío. Este filtro se ejecuta junto con los otros 2 para monitorear el estado de ambas señales además del resultado de la normalización.

En otro tipo de aplicaciones se necesita más discrecionalidad a la hora de eliminar ruido ya que las variaciones a detectar suceden en intervalos cortos de tiempo. Para ello se implementarán dos filtros más: moving average (o media móvil) y Butterworth pasa alto.

Para este tipo de casos no pueden promediarse todos los valores de cada punto porque se perderían variaciones cortas. Sin embargo es necesario eliminar un poco de ruido. Para este fin, en lugar de prome-

diar todos los valores de cada punto de la fibra se realiza un moving average con una ventana de tamaño fijo w. De esta manera cada punto de la matriz original es reemplazado por el promedio entre sí mismo y los próximos w-1 valores. Así, de la matriz original se pasa a otra matriz con menos filas y variaciones muy rápidas suavizadas.

Una vez reducida la matriz original se puede calcular: Desvío estándar por columnas o Filtro Butterworth pasa altoy luego desvío estándar. El filtro pasa alto permite detectar las variaciones pero a partir de una frecuencia de corte. Consiste en aplicar un polinomio lineal sobre las columnas de la matriz utilizando coeficientes conocidos (tantos coeficientes como orden tenga el filtro). Para más datos sobre y ejemplos sobre el filtro, mirar el anexo.

4. Implementación de filtros

En esta sección serán descriptos los filtros y se presentan las nociones clave para su vectorización. De cada filtro se muestra el código vectorizado loop principal solamente. En el código adjuntado se puede ver cómo se adecúan los punteros y los índices entre iteraciones para recorrer las entradas según corresponda.

4.1. Promedio por columnas

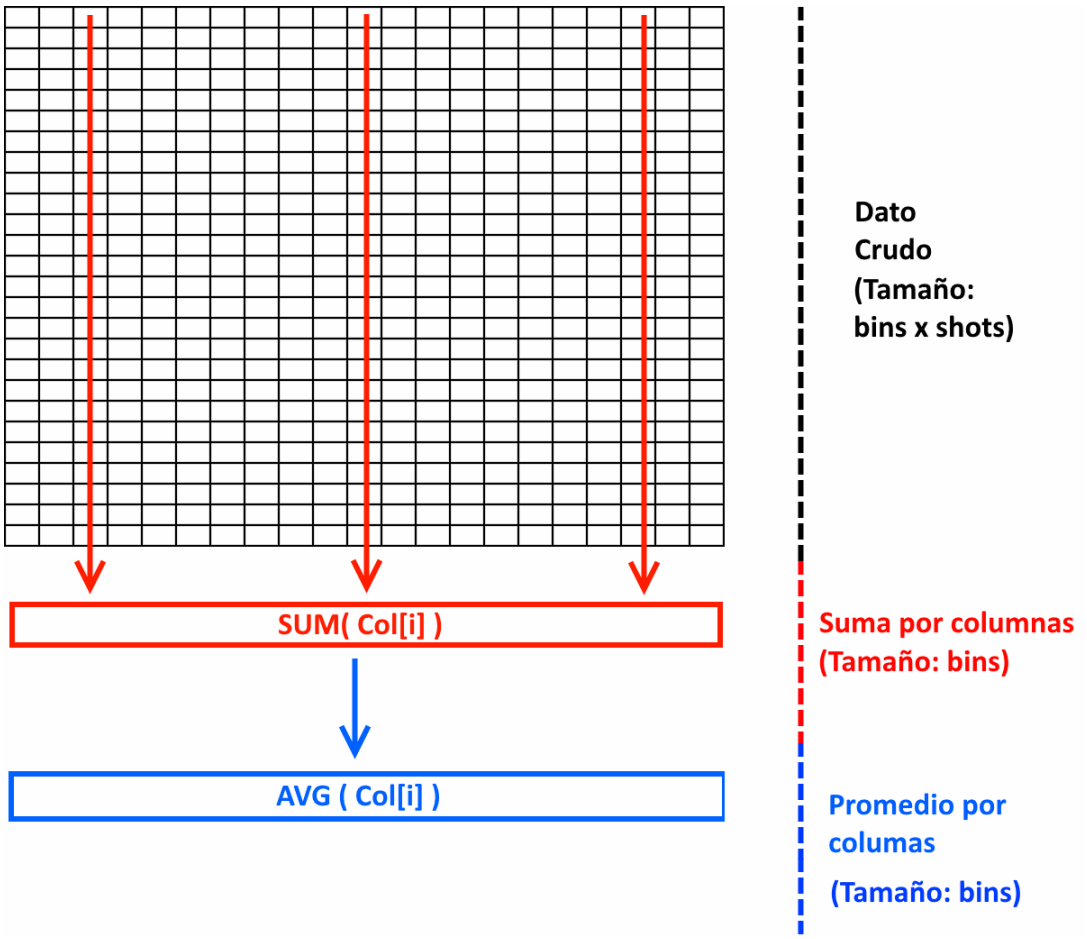

Figura 1: Representación gráfica del filtro para promedios por columnas

Para computar este filtro basta con acumular en un vector los valores correspondientes a cada columna para todas las filas de la matriz involucrada. Una vez obtenida la suma de cada columna y conociendo la cantidad de filas, el promedio se obtiene diviendo cada valor del vector con la acumulación por la cantidad de filas en la matriz. Notar que la operación que domina el costo de este filtro es la suma ya que

se computa sobre la totalidad de los elementos de la matriz mientras que la división se realiza sobre un conjunto muy reducido de valores en comparación.

4.1.1. Vectorización propuesta

Para vectorizar este algoritmo se decidió vectorizar la parte de la suma ya que domina ampliamente el costo total del filtrado. Si bien la operación sobre el vector de sumas es apta para ser vectorizada (ya que implica aplicar la misma operación básica sobre elementos distintos), la ventaja de *performance*que aporta respecto del código en C es despreciable (consume menos del 1 % en términos del tiempo computacional total del filtro).

Para acumular los datos es necesario prestar atención al tamaño de los mismos ya que, de acumular en el mismo tipo de datos que el que viene puede resultar muy rápidamente en un *overflow*. Para ello se decidió acumular, en lugar de en shorts, en enteros. Esto provee un espacio más que suficiente para acumular una cantidad considerable de valores. Dado que el máximo valor positivo de los shorts de 16bits con signo es 32768 pero que, al ser de 14 bits cada dato, el máximo valor después de hacer shift resulta 8192 y que los enteros con signo pueden acumular hasta 2^31, esto permite, en el peor caso, acumular 262144 valores por cada enterlo, lo cual es más que suficiente ya que la placa de adquisición envía cantidades considerablemente menores por cada intervalo.

Por este motivo el algoritmo vectorizado extiende a entero con signo la máxima cantidad de elementos que entra en un registro (4 en registros XMM, 8 en YMM), elimina los 2 bits menos significativos con un shift y procede a acumular en el vector resultado. Una vez que se alcanza el final de una fila, el puntero que recorre la matriz de dato crudo sigue avanzando mientras que el que recorre el vector resultado vuelve al principio para seguir acumulando.

```
;RAX: Dato crudo. RBX: Vector resultado

    VPMOVSXWD xmm2,[rax];pasa de short a int extendiendo signo
    ADD rax,r14;desplazo puntero dato crudo
    VPSRAD xmm2,2;divido por 4 todos los datos por precision placa adquisicion
    VMOVDQA xmm0,[rbx] ;cargo valores del vector resultado
    VPADDD xmm0,xmm2 ;xmm0+=xmm2
    VMOVDQA [rbx],xmm0 ;actualizo vector resultado con la suma
    ADD rbx,rdx ;muevo puntero en vector resultado
```

Listing 1: Código del loop para la suma

Notar que la versión implementada asume que la cantidad de columnas de la matriz (puntos del espacio) es divisible por 8 ya que la placa digitalizadora impone esa condición. De no ser así, en la última iteración de cada fila habrá que reposicionar los punteros que avanzan para que no lean datos incorrectos, repitiendo así los cálculos que se solapen. Esta repetición resulta en un *overhead* menor pero necesario para no romper la vectorización.

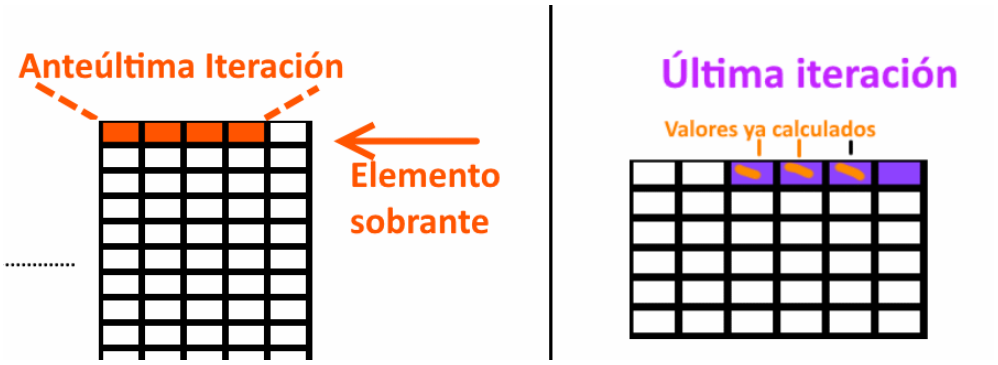

Figura 2: Representación gráfica de las 2 últimas iteraciones del loop de suma cuando los elementos no son múltiplo de la cantidad de valores que entran en un registro XMM o YMM. Sombreados se encuentran los elementos procesados en cada iteración.

4.2. Desvío estándar por columnas

Para calcular este filtro se siguió una filosofía similar a la empleada en el filtro de promedios. Se calcula, para cada columna, además de la suma, la suma de sus elementos elevados al cuadrado. De esta manera, al finalizar el filtro quedan 2 vectores que se pueden utilizar para obtener el promedio y el desvío estándar utilizando directamente su fórmula: $\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \mu)^2}{n}}$ Nuevamente esta operación sobre los 2 vectores resultado representa considerablemente menos del 1 % del tiempo de ejecución como para justificar su vectorización ya que opera sobre los datos filtrados y no sobre la matriz completa.

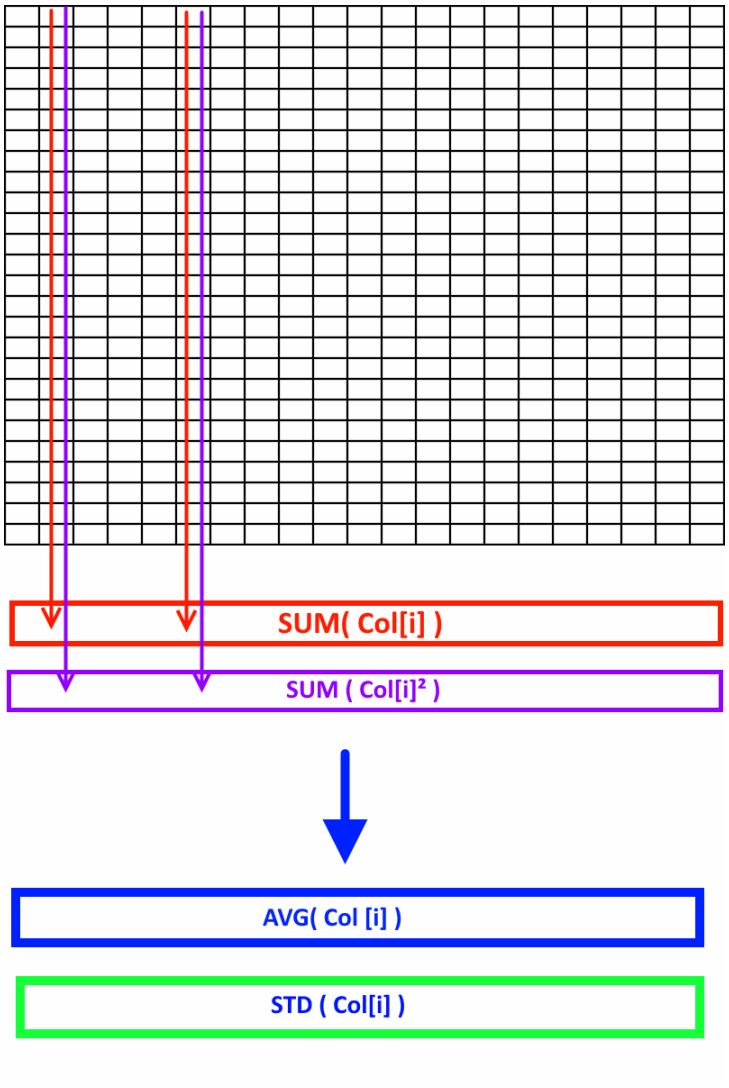

Figura 3: Representación gráfica del filtro para std por columnas

Aquí también hay que tener presente el tamaño de los datos a acumular ya que elevar múltiples shorts al cuadrado puede saturar muy rápido incluso a los ints de 4 bytes. Para ello es que se dedicidió acumular en long ints.

4.2.1. Vectorización propuesta

La forma de leer los datos, recorrer la matriz y actualizar el vector resultado es idéntica a la descripta en el filtro de los promedios. Lo único que cambia es la necesidad de realizar 2 escrituras por iteración ya que el tamaño de los valores de resultado ocupan el doble de espacio en memoria.

```
;RAX: Dato crudo. RBX: Vector resultado

    VPMOVSXWD xmm2,[rax];pasa de short a int extendiendo signo
    ADD rax,r14;desplazo puntero dato crudo
    VPSRAD xmm2,2;divido por 4 todos los datos por precision placa adquisicion
    VMOVDQA xmm0,[rbx] ;cargo valores del vector resultado
    VPADDD xmm0,xmm2 ;xmm0+=xmm2
    VMOVDQA [rbx],xmm0 ;actualizo vector resultado con la suma
    ADD rbx,rdx ;muevo puntero en vector resultado
```

```
;suma al cuadrado. R10 : vector resultado (long)
    VPMULLD xmm2,xmm2;elevo al cuadrado en int
    VMOVQ xmm4,xmm2;muevo parte baja
    VPUNPCKLDQ xmm4,xmm3;extiendo parte baja de int a long.
    VMOVDQA xmm6,[r10] ;cargo valores resultado
    VPADDQ xmm6,xmm4; acumulo
    VMOVDQA [r10],xmm6;actualizo resultado
    ADD r10,rdx ; muevo puntero

    VPUNPCKHDQ xmm2,xmm3;;extiendo parte alta de int a long
    VMOVDQA xmm1,[r10]
    VPADDQ xmm1,xmm2; xxm1+=high
    VMOVDQA [r10],xmm1
    ADD r10,rdx
```

Listing 2: Código del loop para la suma y la suma al cuadrado

4.3. Filtro Normalizador

Para calcular este filtro es necesario dividir las columnas consecutivas de a pares (columna 1 dividido columna 2, columna 3 dividido columna 4 y así sucesivamente) para luego calcular promedio y desvío estándar del cociente. Para promedio y STD se opera como en los filtros correspondientes (calculando suma y suma al cuadrado). Si bien parece similar a los anteriores, este filtro impone nuevas restricciones. En primer lugar los valores sobre los que se promedia o calcula el desvío son valores de punto flotante (de precisición doble) por lo que resulta necesario convertir los valores enteros sobre los que se estaba trabajando. A su vez para realizar los cocientes de forma vectorial es necesario reordenar los datos dentro de los registros. Finalmente la última diferencia consiste en que, si bien por diseño de los sistemas, las señales a medir nunca son nulas,al tratarse de dispositivos experimentales es necesario contemplar el caso de la división por cero puesto que, de no hacerlo, si algún denominador fuera cero, el valor de todos los cálculos que dependan de él se verían anulados (al dar **inf** la división, la acumulación siempre resultaría en **inf**). Este aspecto del filtro introduce una característica que se busca evitar en los algoritmos vectorizados: el *branching*. Sin embargo, mediante el uso de máscaras, ha sido posible.

4.3.1. Vectorización propuesta

Para computar vectorialmente este filtro es necesario convertir los valores a *double*. Sin embargo previamente hace falta ordenar los valores crudos leídos para que, al realizar la conversión, queden en las mitades del vector XMM o YMM los elementos del denominador de un lado y los del denominador del otro para, de esta manera, al moverlos a 2 vectores separados, poder dividir elemento a elemento. Finalmente es necesario comparar cada elemento del denominador con 0 para, luego de la división, usar la máscara de la comparación para eliminar el posible resultado espurio de la cuenta posterior.

```
;EN XMM12 est n los datos crudos en int.
:R15 y R13 tienen los resultados de suma y suma cuadrado del cociente.

        VPSHUFD xmm12,xmm12, 11011000b ;Reordeno parte baja y alta
        VCVTDQ2PD xmm13,xmm12 ;convierto numerador
        VPSHUFD xmm12,xmm12, 11101110b
        VCVTDQ2PD xmm8,xmm12  ;convierto denominador
        VPXOR xmm10,xmm10
        VCMPPD xmm10,xmm8,100b ;comparo por 0
        VDIVPD xmm13,xmm8
        VANDPD xmm13,xmm10 ;anulo con la m scara el resultado del denominador con 0
        VMOVAPD xmm7,[r15]
```

```
        VADDPD xmm7,xmm13
        VMOVAPD [r15],xmm7  ;sumo y actualizo
        ADD r15,rdx
        Vmulpd xmm13,xmm13;cociente^2
        VMOVAPD xmm11,[r13]
        VADDPD xmm11,xmm13
        VMOVAPD [r13],xmm11 ;sumo y actualizo
        ADD r13,rdx
```

Listing 3: Código del loop para la sección del cociente

4.4. Moving average

Para calcular este filtro se debe, en primer lugar, inicializar la primera fila del resultado con los promedios por columnas de todos los valores desde la fila 0 hasta la fila *tamaño_ventana-1* donde *tamaño_ventana* es el tamaño de ventana móvil elegido para suavizar esa cantidad de filas.

Asi si el tamaño de ventana es 5, para inicializar la primera fila del resultado se deben promediar por columnas las primeras 5 filas. Una vez hecho esto se debe desplazar la ventana una fila hacia abajo y volver a calcular el promedio por columnas pero ahora desde la fila 1 hasta la 6 y así hasta llegar al final de la matriz original. Hacer esto de esa forma seria ineficiente ya que no hace falta volver a calcular todo el promedio si no que basta con sumar al promedio ya calculado la diferencia entre el valor nuevo que se incorpora a la ventana y el valor que se va dividido el tamaño de la ventana. De esta manera, para calcular la fila 1 a 6 teniendo el valor de promedios de 0 a 5 se le debe sumar a este valor la diferencia entre las filas 1 y 6 dividido 5 (elemento a elemento de toda la fila).

La vectorización de esta operación resulta directamente así:

```
.loop:

    vpmovsxwd  ymm0,[rdi];ymm0=valor que sale
    vpmovsxwd  ymm1,[rax];ymm1=valor que entra
    ;ymm1 y ymm0 son enteros porque vienen de la matriz de datos crudos
    add rax,16
    add rdi,16;avanzo punteros

    vpsrad ymm0,2
    vpsrad ymm1,2;divido por 4

    VPSUBD  ymm0,ymm1,ymm0;calculo la diferencia old-new

    vcvtdq2pd ymm3,xmm0;convertido a double parte baja
    vextracti128 xmm1,ymm0,1b
    vcvtdq2pd ymm1,xmm1;convertido a double parte alta
    vdivpd ymm3,ymm3,ymm15
    vdivpd ymm1,ymm1,ymm15;en ymm15 esta el valor de la ventana
    vaddpd ymm3,ymm3,[rbx] ;sumo a lo que habia en la fila anterior
    add rbx,32  ;avanzo punteros
    vaddpd ymm1,ymm1,[rbx] ;repito para parte alta
    add rbx,32

    vmovapd [rsi],ymm3 ;escribo en la fila actual
    add rsi,32 ;avanzo punteros
    vmovapd [rsi],ymm1 ;repito para parte alta
```

```
    add rsi,32

    sub rdx,1
    jnz .loop
```

Listing 4: Código del loop para la sección del cociente

4.5. Butterworth pasa alto

Este filtro toma como parámetros dos listas de coeficientes (una para la matriz original y otra para la resultado) y una matriz para filtrar devolviendo una matriz de igual tamaño filtrada. Para obtener una fila de la matriz procesada se debe primero multiplicar la fila correspondiente de la matriz original por el primer coeficiente de la lista de coeficientes correspodiente a esta matriz. Luego se prosigue sumando las filas anteriores por cada coeficiente hasta agotar la lista de coeficientes. Finalmente se hace lo mismo pero restando en lugar de sumarndo cada fila de la matriz resultado por los coeficientes de la lista de coeficientes adecuada. Para las primeras filas se realiza el procedimiento utilizando la máxima cantidad de coeficientes posibles.

Así, por ejemplo, si se cuenta con 3 coeficientes, para calcular la primera fila de la matriz resultado se usa el primer coeficiente con la primera fila de la matriz original. Luego, para la segunda fila se utiliza el primer coeficiente con la segunda fila de la matriz original y el segundo coeficiente con la rimera fila de la matriz original y el segundo coeficiente de la lista de coeficientes de la matriz resultado con la primera fila de dicha matriz. Luego se prosigue como indica el párrafo anterior hasta el final del proceso donde se obtiene una matriz resultado de iguales dimensiones que la original.

Notar que tanto los coeficientes como la matriz original y la resultante están compuestos por valores de punto flotante con precisión doble. Esto se debe a que la entrada de este filtro es la salida del moving average.

Vale la pena destacar de este filtro que, si bien todas las operaciones son vectorizables, para conseguir un dato hay que acceder a tantos datos como cantidad de coeficientes, lo cual puede resultar en una carga de tiempo no optimizable mediante la vectorización.

Dado que para este filtro la inicializacion del mismo (generación de las primeras cantidad de coeficientes filas) puede resultar considerable en tiempo respecto al resto del filtro se decidió implementarla en assembler. En la práctica se utilizan entre 5 y 8 coeficicentes dependiendo del orden del filtro por lo cual resulta despreciable el tiempo de la inicialización. No obstante vale la pena notar que en lineas de código la incialización y la complejidad en cuanto a su claridad es similar a la del resto del algoritmo. Esto puede resultar un problema ya que mantener o desarrollar te código, si no redunda en un beneficio claro de performance, no provee ventajas respecto de su equivalente en C.

Se propone la siguiente vectorización:

```
;RAX apunta a matriz original. RBX al resultado.
        vmulpd ymm0,ymm1,[rax] ;en ymm1 estan los coeficientes apropiadaos
        vmovapd [rbx],ymm0 ;carga para despu s sumar y restar
        add rax,32
        add rbx,32
        sub r14,1
```

Listing 5: Código del loop para calcular el primer paso en cada fila (solo toma datos de la matriz original)

```
;R8 apunta a matriz filtrada. RAX a matriz original
        vmulpd ymm0, ymm2,[r8];  ymm2: coeficientes matriz filtrada
        VFMSUB231PD ymm0, ymm1,[rax]; en ymm1 coeficientes matriz original
        vaddpd ymm0,ymm0,[rbx];acumulo en rbx donde esta el resultado
        vmovapd [rbx],ymm0 ;muevo resultado
        add r8,32
        add rax,32
        add rbx,32
```

```
        sub r14,1
```

Listing 6: Código del loop para computar cada fila una vez inicializada (toma datos de ambas matrices usando los coeficientes adecuados)

5. Experimentación

Para realizar las comparativas temporales fueron utilizadas las computadoras del laboratorio 5 del departamento de computación de la FCEyN que cuentan con un microprocesador intel i5 4460 con soporte para set de instrucciones SSE4 y AVX2. En todos los casos se adjuntan los archivos utilizados para las pruebas con sus correspondientes makefiles y los parametros de los filtros utilizados en cada caso. Estos valores pueden ser modificados en cada archivo C para correr las pruebas que hagan falta.

5.1. Resultados y análisis

5.1.1. Filtros AVG, STD y Normalizador

Estos filtros fueron implementados con registros SSE y también empleando los registros AVX. Ambas versiones fueron comparadas entre sí y contra una implementación en C del mismo filtro optimizado con la opción -O3 de GCC (versión 4.8.4).

Para la **suma** se obuvo que las versiones SSE y AVX mejoraron por igual la performance del código correspondiente de C independientemente del tamaño de registro utilizado. Resultaron entre 10 y 16 veces más rápidos dependiendo del tamaño de la entrada (Ver gráfico). Si bien las sumas se realizan sobre datos de tipo int (Entran 4 int por registro xmm y 8 en los ymm), posiblemente el procesador utilizado ejecute el filtro en varias unidades de procesamiento de aritmética de enteros lo cual explicaría que la mejora obtenida sea mayor a la esperada por el tamaño de los registros empleados en la vectorización. A su vez que el código SSE tarde lo mismo que el AVX en este caso puede deberse a que el procesador, al tener capacidad para vectorizar la operación de suma sobre AVX, pueda utilizar dos registros xmm para cargar un ymm y realizar la operación con las ventajas de estos últimos.

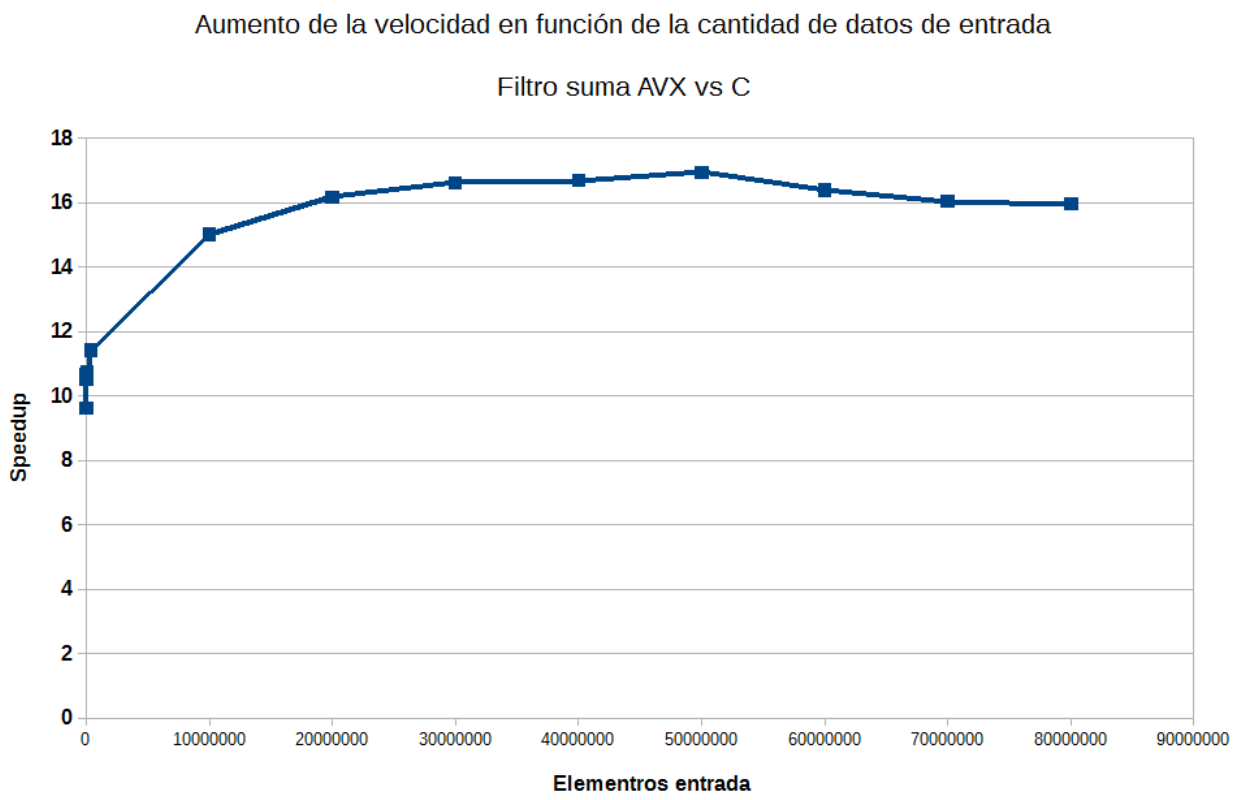

Figura 4: Gráfico de speedup temporal en función del tamaño de entrada para filtro de suma(AVX)

Para el caso de las sumas al cuadrado, la versión de SSE y AVX también mejoraron la performance del código C (Fig 7) .En este caso, además, el filtro AVX logró mejorar a su contraparte SSE aunque no alcanzó a duplicar su desempeño temporal (Fig 8). Esto puede deberse posiblemente a que la performance esté en gran parte afectada por la lectura de los datos de la memoria y entonces poder procesar el doble de datos por vez no se translade directamente al tiempo final de ejecución (que incluye lectura de datos y operaciones).

En este caso donde las sumas al cuadrado se realizan en datos de tipo long (4 por cada registro ymm) también se puede ver que la mejora respecto de C duplica al factor de vectorización lo cual posiblemente también se deba a que haya más de una unidad de aritmética entera ejecutando este filtro.

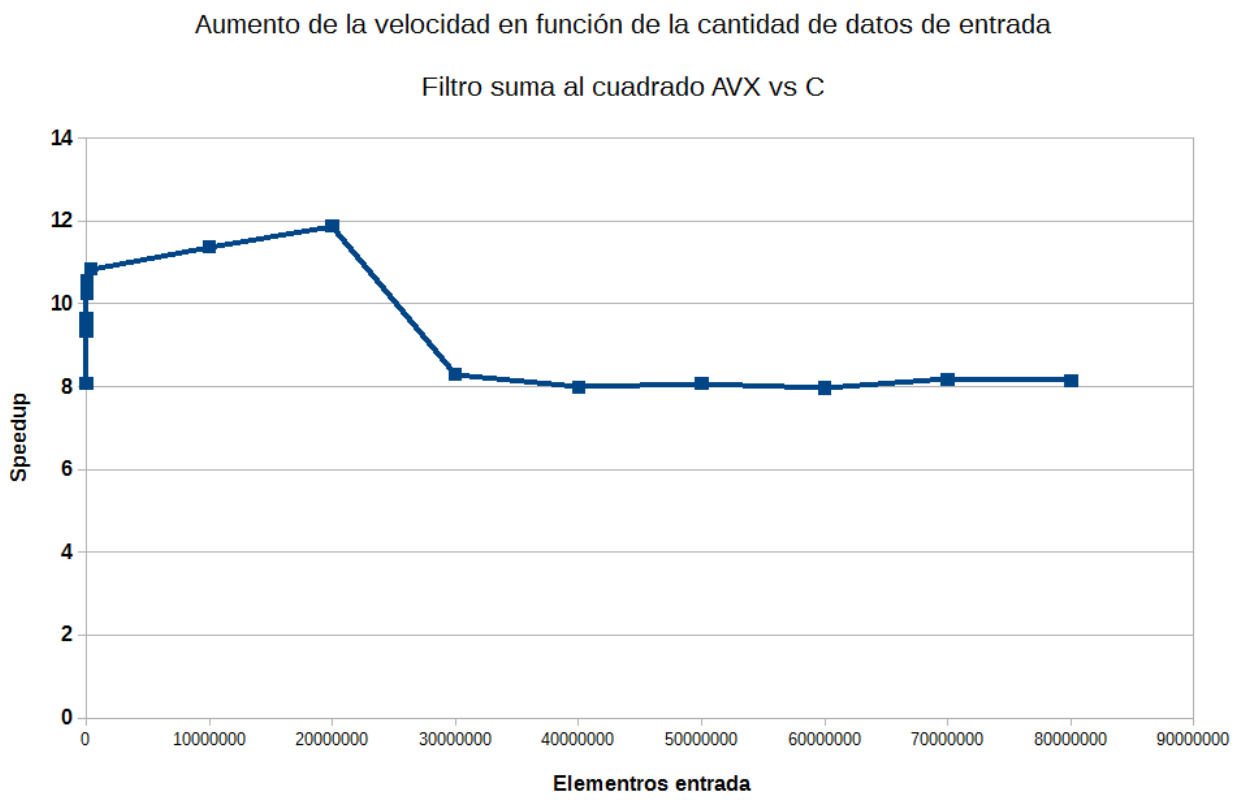

Figura 5: Gráfico de speedup temporal en función del tamaño de entrada para filtro de suma al cuadrado(AVX)

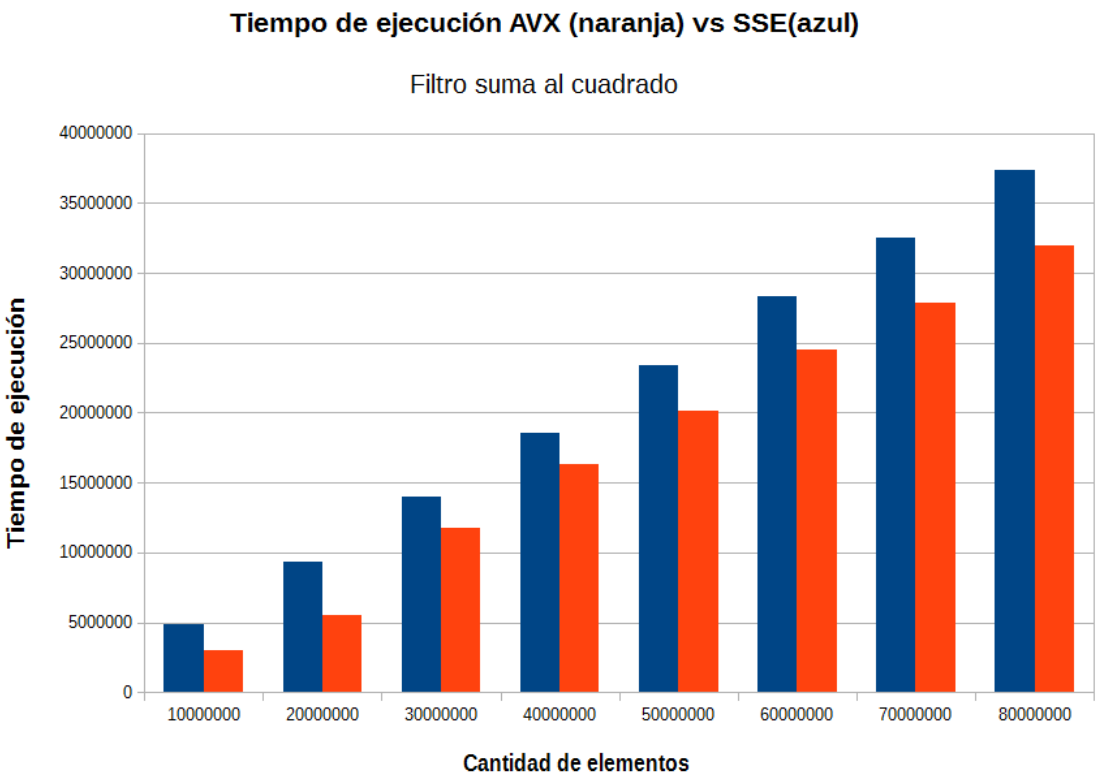

Figura 6: Gráfico comparativo de desempeño temporal en función del tamaño de entrada para filtro de suma al cuadrado

Finalmente para el caso de el procesamiento completo con la normalización usando los **cocientes** SSE y AVX volvieron a desempeñarse de igual manera y ambos mejorando el equivalente de C. Si bien esto parecería contradecir lo dicho en el párrafo anterior de que aumentar la carga de las operaciones vs la lectura en memoria serviría para mostrar el potencial de los registros AVX, hay que tener en cuenta que las operaciones que se agregaron fueron divisiones con *floats* de precisión doble. Esta operación, entre otras, pareciera no funcionar al doble de velocidad en registros AVX que en SSE [1]. Por lo tanto y como esta es la operación que domina el tiempo de ejecución, no es posible apreciar la mejora de los registros AVX.

En este último caso vale la pena mencionar que al incrementar la cantidad de datos de la entrada los tiempos se estabilizan por encima del factor 4 esperado por las dimensiones de los registros ymm donde entran 4 doubles. Esta leve (pero notoria) mejora podría deberse a que el código está vectorizado completamente, es decir, evita el branching del código (al verificar si el denominador es 0 el código en C utiliza un if mientras que el código assembler usa una máscara), lo cual puede ser el motivo que explica esa mejora porcentual.

[1] `http://www.agner.org/optimize/instruction_tables.pdf` en la sección de Haswell puede verse que la instrucción DIVPD presenta el doble de latencia para registros ymm que para los xmm resultando equivalente una division en ymm a dos en xmm y por lo tanto no sacando provecho de los datos extra

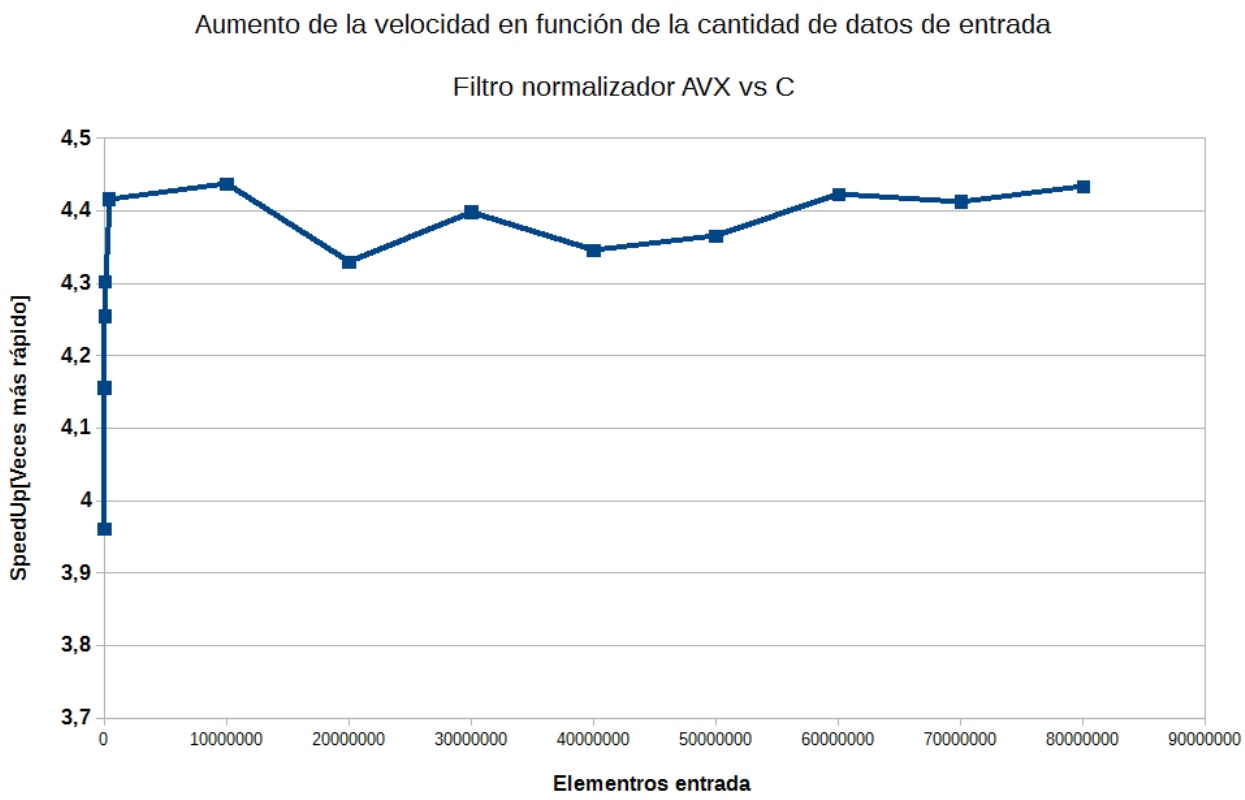

Figura 7: Gráfico de speedup temporal en función del tamaño de entrada para filtro normalizador(AVX)

5.2. Moving average

Este filtro se implementó solamente con registros AVX. Compilando con la opción -O3 el código en C y el de assembler proveen igual en performance. Compilando con -O2 el código assembler mejora en un factor 4 el tiempo de ejecución. Esta es la mejora esperada por el factor de vectorización dado 4 son los valores que procesa por iteración. Dado que el loop involucrado es relativamente sencillo y muestra pocas dependencias sin branches, sumado a que la optimización -O3 permite al compilador vectorizar el código (mientras que -O2 no), es posible pensar que GCC logró vectorizar el loop y a eso se debe la nula mejoría. Dado que este filtro tiene una incialización más compleja que el resto, también se midió el tiempo de esta respecto del código en assembler para ver si valía la pena optimizarla y resultó que consume menos del 0.001 % del tiempo de ejecución dado que el tamaño de la ventana siempre es de varios órdenes de magnitud menor que la totalidad de las filas de la matriz (la ventana no suele ser mayor a 20 puntos mientras que las filas totales de la matriz puede ser varias decenas de miles).

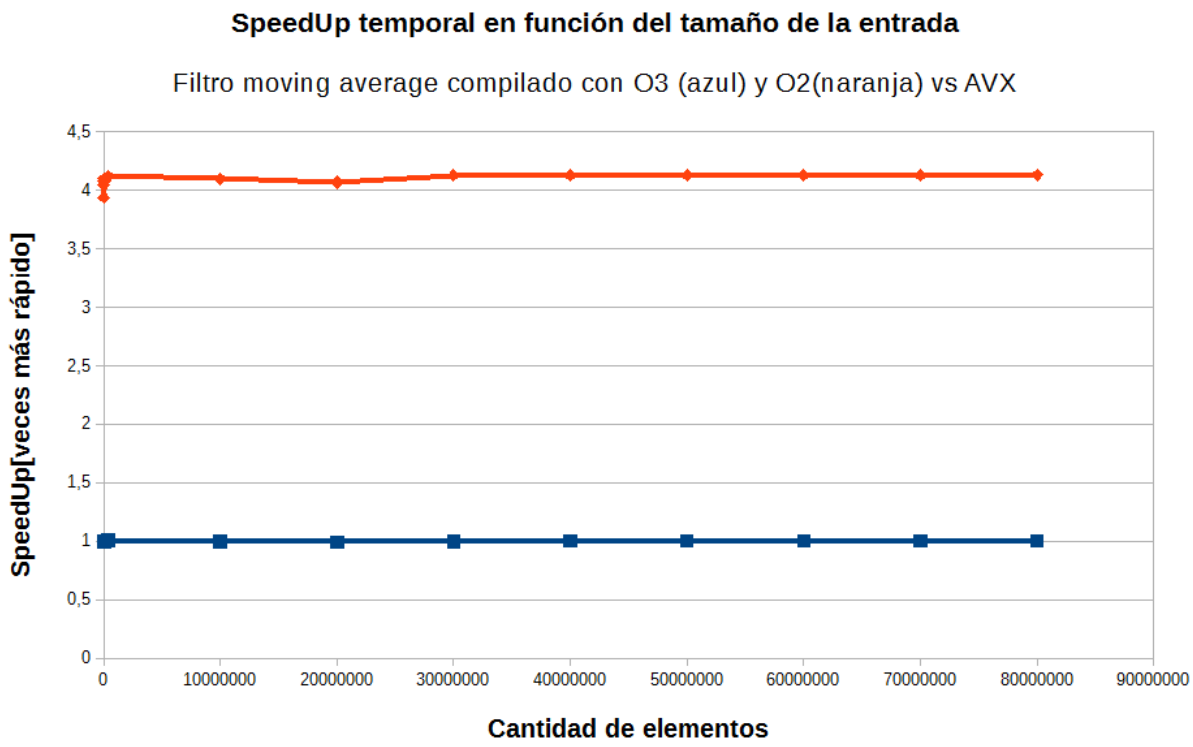

Figura 8: Gráfico comparativo de speedup temporal en función del tamaño de entrada para filtro moving average compilado con O2 y O3

5.3. Butterworth pasa alto

Al correr las pruebas para este filtro se notó que los tiempos de ejecución para tamaños pequeños de entrada resultaban mucho mejor de lo esperados. Al incrementar la cantidad de elementos de la entrada, la mejora caía notoriamente por debajo del factor 4 de vectorización (aunque se mantenía mejor que el código en C). Se puede suponer que la mejora excesiva en los casos pequeños se debe a la acción de la caché dado que este algoritmo se encuentra fuertemente limitado por los accesos a memoria (para cada dato hace falta consultar tantos datos en memoria como orden (cantidad de coeficientes) tenga el filtro). Entonces, al ser más pequeña la entrada los datos pueden permanecer accesibles en caché para agilizar el tiempo de acceso permitiendo así valerse de todas las ventajas en las operaciones mientras que, al crecer la cantidad de datos, el acceso a memoria empieza a limitar las ganancias de esta implementación.

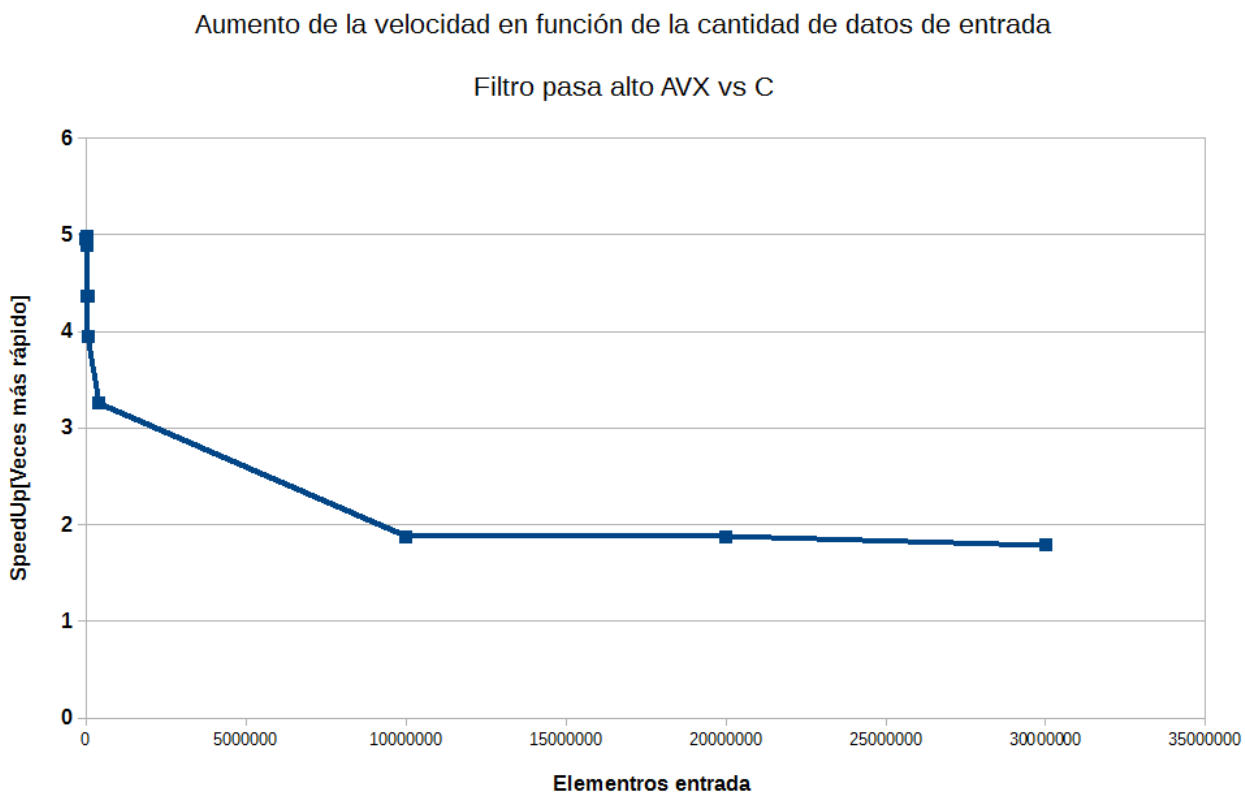

Figura 9: Gráfico comparativo de speedup temporal en función del tamaño de entrada para filtro Buterworth pasa alto

Para poner a prueba esta hipótesis se modificó ligeramente el filtro agregando el loop central una raiz cuadrada por iteración. Al hacer esto el tiempo de ejecución pasó a ser 25 % del tiempo de C para cualquier tamaño de entrada, mostrando que este tipo de optimizaciones son realmente valiosas cuando la cantidad de operaciones a realizar ocupa un tramo importante del tiempo de ejecución. En la carpeta buterSqrt se encuentra un archivo con un caso de ejemplo que permite ver esta situación.

El cálculo de error respecto de la implementación de C puede resultar mayor 0 debido a la utilización de instrucciones FMA en el código assembler ya que estas realizan un solo redondeo de punto flotante por cada multiplicación-suma mientras que, si el compilador de C no las utiliza, realiza un redondeo por cada operación (uno para suma y otro para multiplicación).

6. Conclusión

Este trabajo tenía como objetivo la optimización en assembler de filtros de señales digitales mediante la vectorización de la versión escalar de los mismos. Este objetivo fue conseguido para 4 de los 5 filtros con diferentes grados de éxito (entre 1 y 16 veces de mejora temporal). Para ello fue necesario diseñar, implementar, comparar e interpretar los resultados de los algoritmos vectorizados con sus versiones escalares. Mediante este proceso se puso de manifiesto la dificultad, complejidad y baja mantenibilidad que presenta el código en lenguaje ensamblador aunque también por otro lado se pudo apreciar la mejora de performance que es capaz de brindar. De este modo es posible juzgar qué segmentos de código optimizar y en cuáles no merece la pena detenerse. Por otro lado respecto de la vectorización se apreció su potencial tanto en registros XMM como en los YMM. Se pudo ver que no es un proceso que garantiza mejorías absolutamente ya que depende de las operaciones involucradas, de los accesos a memoria que haga el algoritmo o de la estructura del código que se le entrega al compilador varia su porcentaje de mejora. Así, un algoritmo en donde dominan las divisiones por punto flotante no se ve beneficiado del

aumento del tamaño de registros de SSE a AVX mientras que uno con operaciones entre enteros, como el caso de sumas al cuadrado, sí experimenta la mejora. O por ejemplo en el caso del filtro de media móvil (moving average), al tratarse de un loop sencillo, las optimizaciones más agresivas de GCC logran generar, de forma automática, un código igualmente eficiente que la vectorización planteada en este trabajo. Finalmente en el caso del fitro pasa altos, al ser un factor importante en el tiempo total del algoritmo los accesos a memoria, la mejora de la vectorización se ve reducida en su potencial. Es posible concluir entonces a modo de cierre que la vectorización de algoritmos es una herramienta poderosa que presenta sus limitaciones en cuanto al código fuente (legibilidad, mantenibilidad,etc(y al diseño de los algoritmos (no branches, aplicación de operaciones reiteradas sobre un conjunto de datos, poco impacto en el tiempo de ejecución de los accesos a memoria, entre otros factores) pero que, en los casos en los que estas limitaciones no son un problema, la mejora que brindan excede ampliamente sus desventajas.

7. Anexo 1: Ejemplos y uso de filtros

Cada ejecutable de filtro recibe como parámetros obligatorios la cantidad de columnas (bines), filas(shots) y un archivo binario con los datos de entrada (una matriz de shots*bines de elementos de tipo short para los filtros de suma, suma al cuadrado, cocientes y moving average y una matriz de elementos de tipo double para el filtro pasa alto). Como parámetro opcional se puede agregar uno o 2 archivos de salida (según corresponda para cada filtro: moving average,buterworth y suma devuelven 1 archivo de salida mientras que suma2 y cocientes devuelven 2). Los ejecutables corren las versiones de C y assembler, imprimen por pantalla la comparativa de tiempos para 5 iteraciones y almacenan en el o los archivos de salida los resultados obtenidos de las versiones de assembler.

El archivo ejemplos.py (se ejecuta como *python ejemplos.py*) permite ejecutar los ejemplos mostrados a continuación (salvo el pasa alto). Ese script muestra como genera cada archivo de entrada y grafica dicha entrada y la o las salidas filtradas. Requiere que los ejecutables compilados estén en el directorio de trabajo donde se ejecuta con el nombre que le da el makefile. Ignorar los resultados que imprime.

Para los ejemplos de uso de los filtros de promedio (suma), desvío estándar (suma2) y cocientes se le pasa una matriz que tiene una distribución conocida en cada columna. De esta manera se puede ver fácilmente que el promedio, desvío estándar y cociente que retornan los filtros son correctos.
Para el caso de los promedios la distribución que se genera es uniforme con media 200 y debe devolver como resultado valores cercanos a 50 (ya que recordemos que los filtros dividen por 4 los valores de la entrada porque estos están pensados para valores de 14 bits). En el caso del desvío estándar, este filtro genera 2 archivos de salida, uno para calcular la media y otro para el desvío. En este caso se optó por una distribución normal con media 4000 y desvío 100 por lo que el filtro debe devolver una media de 1000 y desvío de 25 (a través de la suma y la suma al cuadrado que luego son transformados en esos valores). En el caso de los cocientes se genera una matriz en la que cada par de valores a normalizar está compuesto por un valor y su doble, por lo que al realizar el promedio de los cocientes entre valores consecutivos, el promedio da 0.5 y el desvío 0.

Estos ejemplos se pueden ejecutar en la carpeta de cada filtro como:

./suma 80 750 80_750.in sumas.out
./suma2 80 1000 80_1000.in sumas.out sumas2.out
./cociente 80 1000 80_1000.in sumasCociente.out sumas2Cociente.out

para promedio, desvio estandar y normalizador respectivamente. En cada carpeta se encuentra la imagen de la salida para los ejemplos (como la genera ejemplos.py).

Para el caso de moving average se genera una entrada en franjas de tamaño mayor a la ventana (10 shots temporales es la vetana en este caso. Cada franja mide 20 shots) y se observa que la salida presenta un patrón similar aunque suavizado.

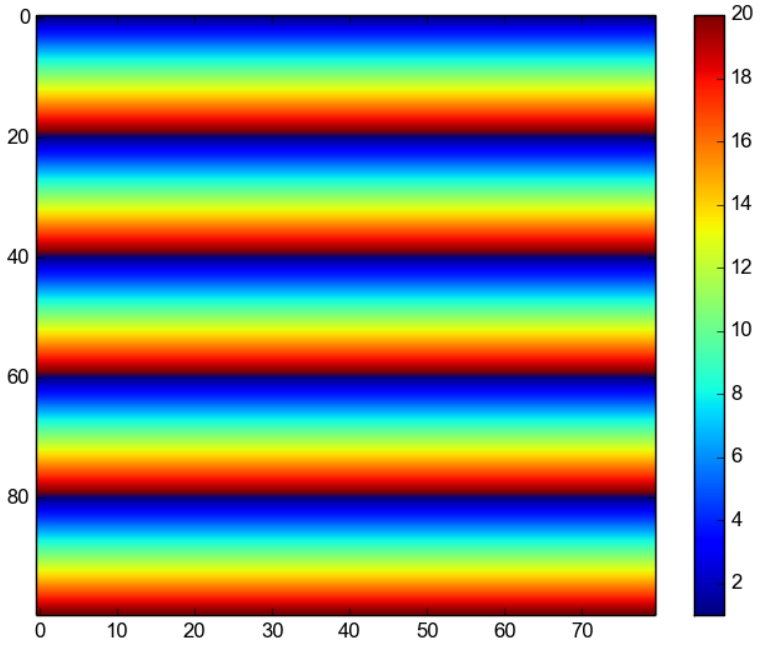

Figura 10: Entrada moving average

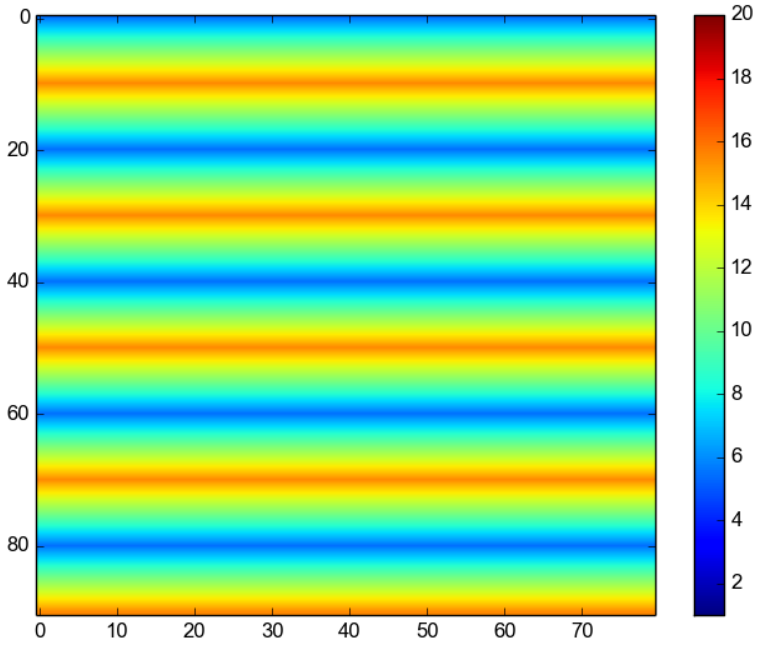

Figura 11: Salida moving average

Este se puede ejecutar como:
./mov_avg 80 100 80_100.in movAvg.out

Para el caso del pasa alto se genera una matriz donde las columnas son senos de frecuencia creciente en un rango en el cual actúa la frecuencia de corte. Para este caso se usó una frecuencia de corte de 20Hz

y los coeficientes se obtuvieron de la función buter de matlab (se adjunta script).

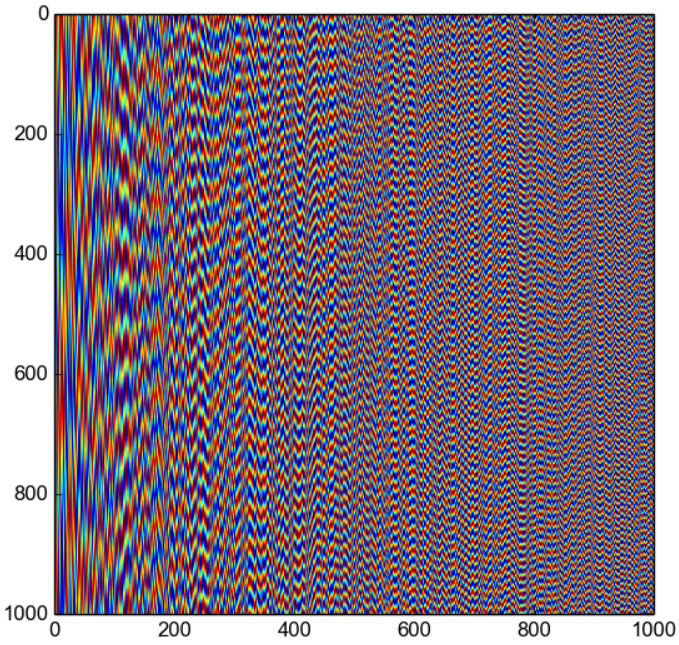

Figura 12: Entrada buterworth con frecuencia creciente.

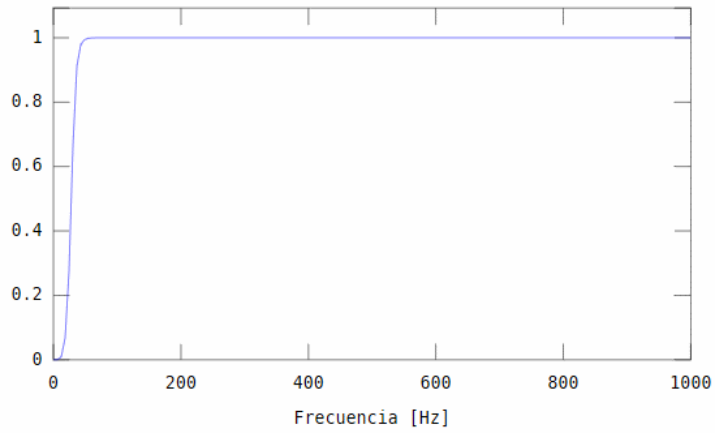

Figura 13: Respuesta de filtro buterworth pasa alto con frecuencia de corte 20hz

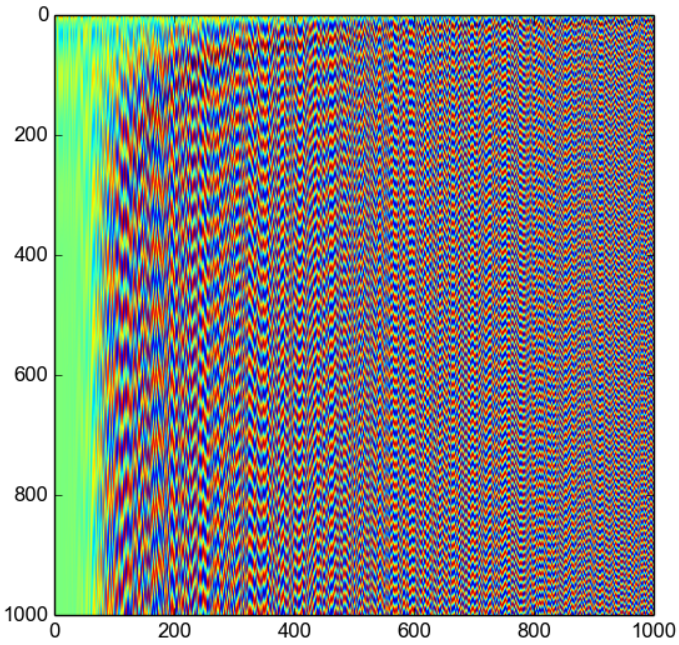

Figura 14: Salida buterworth

Se observa como se atenúan notoriamente todas las posiciones con frecuencias bajas hasta llegar a la posición en la cual se supera la frecuencia de corte y se recuperan los valores originales con cierta atenuación en las posiciones cercanas al corte. Este ejemplo puede ejecutarse como:
./buter 1000 1000 1000_1000.in buter.out