HPPS
Informe 1 - Optmización FIR

Juan Braun

26 de abril de 2013

Introducción

El objetivo de este informe es estudiar diferentes implementaciones de un filtro FIR analizando ventajas y desventajas. Luego se elije una implementación particular y se estudian diferentes maneras para optimizarla. Finalmente se presenta el filtro FIR optimizado.

1. FIR: versión 1

Esta implementación asume que se conoce la señal de entrada antes de realizar el filtrado. Para realizar esta implementación se modificó la interfaz de la funcion `fir_NC`, se pasa como parámetro un puntero a una muestra:

```
sample_t fir_NC(sample_t* muestra);
```

De esta manera se puede calcular la muestra de salida de la siguiente manera:

```
for(i = 0; i<TAP_LENGTH; i++)
   out+=coefsLocalesNC[i] * *(muestra-i);
```

Se puede ver que para calcular cada muestra de salida se realizan `TAP_LENGTH` sumas, para calcular el índice de la muestra de entrada, `TAP_LENGTH` multiplicaciones y `TAP_LENGTH` sumas. Ademas se realizan $2 \cdot$ `TAP_LENGTH` accesos a memoria. Si se le da el mismo costo a todas las operaciones se tiene un total de $5 \cdot$ `TAP_LENGTH` operaciones por muestra.

La ventaja de esta versión es que es fácil de implementar y es la que realiza la menor cantidad de operaciones. La desventaja es que no puede ser utilizada para trabajar en tiempo real y se necesita memoria para almacenar toda la seãl de entrada.

Para las pruebas se utilizaron diferentes combinaciones (`SIGNAL_LENGTH`, `TAP_LENGTH`).

En la Tabla 2 se ven los ciclos de reloj del procesador que se necesitaron para correr esta implementación para las diferentes señales de entrada.

A	$(10, 4)$
B	$(1024, 128)$
C	$(44100, 1024)$

Cuadro 1: Combinaciones

	`float`	`double`
A	27025	25767
B	3723250	3749336
C	1340053072	1340053456

Cuadro 2: Ciclos de reloj

	`float`	`double`
A	1.01 %	0.96 %
B	57.85 %	58.25 %
C	99.69 %	99.69 %

Cuadro 3: Tiempo relativo

En la Tabla 3 se puede ver el tiempo relativo en cada caso. Cuando se tiene una señal pequeña, el tiempo de procesamiento de es mucho menor al tiempo que se utiliza en el programa para ejecutar otras funciones que se utilizan siempre que se ejecuta un programa. Estas funciones representan el costo fijo de la ejecución. Mientras más chica la señal, el costo de procesamiento es comparable al costo fijo, por lo tanto el tiempo de procesamiento es comparable con el tiempo que lleva ejecutar las funciones correspondientes al costo fijo.

Otro parámetro utilizado para estudiar el funcionamiento de la implementación es `#ciclos/SIGNAL_LENGTH/TAP_LENGTH`. Este valor da una idea de la eficiencia de la implementación. Estos valores se pueden ver en la Tabla 4

	`float`	`double`
A	675.6	644.2
B	28.4	28.6
C	29.7	29.7

Cuadro 4: Eficiencia

En todas las medidas anteriores se discriminaron los casos en los que se usan datos tipo `float` y tipo `double`. Se puede ver que los resultados son similares. Esto se debe a que el procesador es de 64 bits, tiene el mismo costo realizar operaciones con `float`(32 bits) que con `double`(64 bits). De aqui en más se presentas resultados para tipo de dato `double`.

2. FIR: versión 2

Esta versión procesa las muestras de entrada a medida que le llegan al filtro. Puede ser utilizada para procesamiento en tiempo real. Se llena un buffer a medida que llegan muestras nuevas y se calcula la señal de salida. Cuando el buffer se llena y entra una nueva muestra, se desplazan todas las muestras en el buffer haciendo lugar para la nueva muestra. El buffer se controla de la siguiente manera:

```
for(i = TAP_LENGTH-1; i>0; i--)
   buffer[i]=buffer[i-1];
buffer[0]=muestra;
```

La muestra de salida se calcula de la siguiente manera:

```
for(i = 0; i<TAP_LENGTH; i++)
   out+=coefsLocales[i] * buffer[i];
```

En esta versión se tiene $5 \cdot$ `TAP_LENGTH` manipulaciones de memoria para el buffer, para el caculo de la salida se tienen $2 \cdot$ `TAP_LENGTH` accesos a memoria, `TAP_LENGTH` multiplicaciones y `TAP_LENGTH` sumas. Se tienen $9 \cdot$ `TAP_LENGTH`.

Esta versión es fácil de implementar, usa menos memoria que la versión anterior ya que solo se necesitan `TAP_LENGTH` valores para el buffer y una sola muestra de entrada.

Los resultados obtenidos fueron los siguientes:

	Ciclos de reloj	Tiempo relativo	Eficiencia
A	27727	1.04 %	693.2
B	6321319	70.18 %	48.2
C	2167805367	99.82 %	48.0

Cuadro 5: Resultados FIRv2

Como se mencionó anteriormente esta versión requiere $9 \cdot$ `TAP_LENGTH` operaciones mientras que la anterior requiere $5 \cdot$ `TAP_LENGTH`, se puede ver que la relación entre los ciclos de reloj de las diferentes versiones cumplen esta relación:

$$\frac{6321319}{3723250} \cong 1{,}7$$

$$\frac{2167805367}{1340053072} \cong 1{,}6$$

$$\frac{9}{5} = 1{,}8$$

3. FIR: versión 3

Esta versión utiliza un buffer circular. La ventaja de usar un buffer circular es que no hace falta mover todas las muestras en el buffer sino que agrega la nueva muestra

en el lugar en que estaba la muestra mas vieja del buffer y se guarda el indice. Para calcular el indice en el buffer circular a medida que evoluciona el tiempo de la señal se utiliza la función `void update_p(int paso);`

Para esta versión se utilizó la implementación propuesta en la solución. Los resultados obtenidos se pueden ver en la Tabla 6

	Ciclos de reloj	Tiempo relativo	Eficiencia
A	28421	1.06 %	710.5
B	6899895	71.98 %	52.6
C	2350776353	99.83 %	52.1

Cuadro 6: Resultados FIRv3

Desde el punto de vista de la memoria utilizada, esta versión utiliza aproximadamente la misma cantidad de memoria que la versión 2.

En la función `update_p` se realizan 5 sumas y 2 comparaciones en el peor de los casos(que ocurre 1 cada `TAP_LENGTH` veces). Es razonable suponer que la esta función tiene 4 sumas y 2 comparaciones. En la función `fir(sample_t muestra)` se realiza un acceso a memoria, y se realiza `TAP_LENGTH` sumas, `TAP_LENGTH` multiplicaciones y $2 \cdot$ `TAP_LENGTH` accesos a memoria. Ademas se llama a la función `update_p(int paso)` `TAP_LENGTH` $+ 1$ veces. La cantidad de operaciones es aproximadamente $10 \cdot$ `TAP_LENGTH` por muestra. Esto es coherente con los valores de tiempo obtenidos para las versiones 2 y 3.

Si bien esta implementación dio resultados similares a la versión 2, se observó que la función `update_p` consume gran parte del tiempo de procesamiento. Esto se muestra en la Tabla 7.

	Ciclos de reloj	#Ciclos update_p	%Tiempo update_p
B	6899895	2908176	30.34 %
C	2350776353	994543286	42.23 %

Cuadro 7: Costo update_p

4. Optimización

Se analizaron diferentes maneras de optimizar el filtro FIR versión 3.

4.1. Opciones del compilador de C

Al compilador de C se le puede introducir una bandera para indicarle que se desea optimizar al performance del programa. La bandera para que el compilador optimice el código es `-O`. Hay diferentes niveles de optimización, del 1 al 3, dependiendo cual se elije el compilador habilita o deshabilita banderas, el nivel 3 es el nivel en el que realiza

la mayor optimización. La optimización se realiza a costas de tiempo de compilación y de información en debugger.

Se compiló el con los tres niveles de optimización, los resultados obtenidos se pueden ver en la Tabla 8, se utilizó `TAP_LENGTH` = 128 y `SIGNAL_LENGTH` = 1024.

	-O1	-O2	-O3
Eficiencia	27.5	13.4	13.4

Cuadro 8: Optimzación compilador

4.2. Optimización buffer circular

Aqui se presentan los resultados de diferentes optimizaciones que se realizaron para la función `update_p`.

4.2.1. Implementación con la operación Módulo

Se modificó el código de la función para que calcule el indice del buffer circular utilizando la función módulo, el código de la función es el siguiente:

```
void update_p(int paso){
  if(p!=0){
    p = (p+paso) % p;
  }else{
    p+=paso;
  }
}
```

Se puede ver que para esta implementación la cantidad de operaciones es una comparación, una suma y la función módulo. En la implementación original de la función se tenia que la cantidad de operaciones era de 2 comparaciones y 4 sumas. En la Tabla 9 se pueden ver los resultados obtenidos.

	update_p	update_p_mod
#Ciclos update_p	2908176	2377728
Eficiencia	52.6	48.6

Cuadro 9: Optimzación con función módulo

4.2.2. Implementación sin parámetro de entrada

Como siempre se necesita incrementar en uno el índice del buffer circular, resulta innecesario que la función `update_p` tenga un parámetro de entrada `paso`. La nueva implementación queda:

```
void update_p(){
    if (p+1<0){
        p = TAP_LENGTH + p + 1;
    } else if (p+1>TAP_LENGTH-1){
        p = p + 1 - TAP_LENGTH;
    } else {
        p = p + 1;
    }
}
```

Los resultados obtenidos se muestran en la Tabla 10.

	update_p(paso)	update_p()
#Ciclos update_p	2908176	2378760
Eficiencia	52.6	50.6

Cuadro 10: Optimzación por no parámetro de entrada

4.2.3. Función update_p como *inline*

Hacer que una función sea *inline* sirve para decirle al compilador que se quiere que la función sea ejecutada los mas rápido posible. La manera de hacer esto es sustiyendo la llamada a la función, por el código de la mismo. La manera de hacer que una función sea *inline* es escribiendo la etiqueta `inline` antes de la declaración de la función. En la práctica se tuvieron problemas para realizar esto correctamente, por lo que se muestra parte del código para mostrar la manera en la que funciono según lo esperado. En el archivo `fir.h` se declara la función `update_p` como sigue:

```
static inline void update_p(int paso);
```

La etiqueta *static* se utilizó porque la función `update_p` solo se utiliza dentro del módulo `fir`.

Además se linkeo el programa de la siguiente manera:

```
gcc -std=c99 -finline-small-functions testfir.o fir.o -o fir_exe
```

La bandera `-finline-small-functions` es necesaria para decirle al compilador que aunque la función sea chica, la trate como *inline*. Sin esta bandera el compilador mira la función y decide no llamarla como *inline*.

Los resultados obtenidos se pueden ver en la Tabla 11, se mejora muy poco.

	update_p	inline update_p
#Ciclos	6899895	6874128
Eficiencia	52.6	52.4

Cuadro 11: Optimzación por inline de update_p

4.2.4. Función update_p como macro

Escribir la función como un macro sirve para intercalar en el programa el código de una función en particular, a diferencia de cuando se llama a una función, que hay que hacer un salto en memoria hasta donde está la función guardada, correr la función y volver a la dirección de memoria en que se estaba antes de llamar a la función. El comportamiento es similar al que se obtiene haciendo que una función sea *inline*. La diferencia es que el macro se procesa en la etapa de precompilación. Cuando se precompila, el compilador mira todas as ocurrencias del macro y sustituye en ese lugar el código que tiene adentro. La ventaja es que el programa no tiene que dar saltos de direcciones para ir a buscar una función en memoria, la desventaja es que el programa queda mas grande.

El macro para la función es:

```
#define update_p(p){\
    if (p+1<0){\
        p = TAP_LENGTH + p + 1;\
    } else if (p+1>TAP_LENGTH-1){\
        p = p + 1 - TAP_LENGTH;\
    } else {\
        p = p + 1;\
    }\
}
```

Los resultados obtenidos se pueden ver en la Tabla 13, se mejora mas que en el caso en que se usa la función como *inline*, esto lleva a pensar que tal vez la implemtación de la función inline está del todo bien.

	update_p	macro update_p
#Ciclos	6899895	532687
Eficiencia	52.6	40.5

Cuadro 12: Optimzación por macro de update_p

4.2.5. Optimizaciones fallidas

Se probó optimizar utilizando las variables como *register* sin tener exito. Lo mismo sucedió con utilizar la función `fir` como *inline*.

4.3. Filtro optimizado

Se combinaron todas las optimizaciones para llegar al filtro optimizado. Se usa el compilador con la bandera de optimización, la función `update_p` se usa como macro, tiene un paso fijo de uno y usa la función módulo.

```
#define update_p(p){ \
```

```
    if(p!=0){        \
      p = (p+1) % p; \
    }else{           \
      p += 1;        \
    }                \
}
```

Se obtiene el siguiente resultado

	FIR	FIR optimizado
#Ciclos	6899895	1749671
Eficiencia	52.6	13.3

Cuadro 13: Optimzación por macro de update_p