FILTRO FIR

Juan Siverio Rojas

PRÁCTICA ARQUITECTURAS AVANZADAS

Contenido

Objetivo

Implementar un filtro tipo FIR (filtro de respuesta finita) para un DSP (procesador de señales digitales) de la marca Texas Instruments, más concretamente el TMS320C6000 y en su posterior comprobación de los ciclos de reloj que tarda diferentes versiones del código, a la que se han de aplicar diferentes optimizaciones.

Previo

La aplicación se ha programado utilizando C, y se ha utilizado un emulador de compilación y ejecución para tal efecto, el Code Composer Studio en su versión 5.5

- Instalación Code Composer Studio 5.5
- **Coeficientes.csv**: Fichero que contendrá los coeficientes del filtro FIR. *(opcional)*
- **Musica4.csv**: Contendrá los valores de la señal. *(opcional)*
- **Ejemplo_comienzo.c:** Fichero principal de la aplicación, contiene todas las funciones.

Aplicación

La aplicación obtiene la información de los coeficientes y valores de la señal leyendo dos archivos *.CSV (archivo de texto separado por comas), **Coeficientes.csv y Musica4.csv**.
**Estos ficheros deben de encontrarse en el mismo directorio de ejecución de la apliación.*

- La estructura de los ficheros es muy sencilla, simplemente un valor detrás de otro separado por comas. No hace falta un primer valor indicando el tamaño del fichero, ni el número de elementos, **la aplicación irá leyendo valores hasta llegar al final del texto.**

- En el caso de que alguno de los ficheros no exista, la aplicación usará unos valores por defecto que son:

  Coeficientes: 0.33, 0.33, 0.33

  Datos: 1,3,2,5,7,11,4,2,1,3

- Para establecer un límite máximo de datos, existe la constante **#define nvMax**, cuyo valor por defecto es de **3.500**

Variables principales

```
float v_datos_default [] = {1,3,2,5,7,11,4,2,1,3};
float v_coef_default [] = {0.33,0.33,0.33};
float salida,valorEntrada,borradorFloat;
int indiceValor,indiceCoeficiente,tmp,nv,nc;
char borradorCaracter;
float * restrict v_coef; ///puntero hacia array con los coeficientes leidos desde fichero csv
float * restrict v_datos; ///puntero hacia array con los datos leidos desde fichero csv
```

Función obtenerValores()

Esta función se encarga de la carga de ficheros y obtención de valores. Se muestra el código utilizado para cargar los ficheros de datos, o en caso de error, utilizar los valores por defecto.

Esta función **NO** se tiene en cuenta en la medición de tiempos.

```
void obtenerValores()
{

	///
	///Primero intento abrir fichero de coeficientes
	///
	FILE *fich_num = fopen("Coeficientes.csv","r");

	if (fich_num == NULL){
		///
		///si no se encontró o pudo abrir, lo indico y utilizao el array por defecto.
		///
		printf("\nNo se puede abrir fichero: Coeficientes.csv");
		printf("\nUsando valores por defecto");

		nc=3;
		v_coef =  v_coef_default;
	}

	else
	{

		nc=0;
		while (!feof(fich_num))
		{
			fscanf(fich_num,"%f",&borradorFloat); ///leo el valor en flotante
			nc++; ///numero de coeficientes que he leido
			if (!feof(fich_num))
				fscanf(fich_num,"%c",&borradorCaracter); ///leo la coma.

		}

		v_coef = (float *)  malloc(nc * sizeof(float));
		fseek(fich_num,0,SEEK_SET); ///me vuelvo a colocar al principio del fichero

		nc=0;
		while (!feof(fich_num))
		{
			fscanf(fich_num,"%f",&v_coef[nc]); ///leo el valor en flotante
			nc++; ///numero de coeficientes que he leido
			if (!feof(fich_num))
				fscanf(fich_num,"%c",&borradorCaracter); ///leo la coma.

		}

		fclose(fich_num);
	}


	///Imprimo los coeficientes leidos.
    for ( tmp = 0; tmp < nc; tmp++)
	printf("\nCoeficiente %d: %lf",tmp,v_coef[tmp]);

	///
	///Ahora intento lo mismo con fichero de datos
	///
	fich_num = fopen("musica4.csv","r");
```

```
        if (fich_num == NULL){
                ///
                ///si no se encontró o pudo abrir, lo indico y utilizao el array por defecto.
                ///
                printf("\nNo se puede abrir fichero: musica4.csv");
                printf("\nUsando valores por defecto");

                nv=10;
                v_datos =  v_datos_default;
        }

        else
        {
 ///aqui leo el fichero y voy contando para saber cuantos memoria tengo que asignar
realmente.Así
                //no dependo de un tamaño prefijado, sino depende del tamaño del fichero.
                nv=0;
                while ((!feof(fich_num)) && (nv < nvMax)) ///limite de valores a leer.
                {
                        fscanf(fich_num,"%f",&borradorFloat); ///leo el valor en flotante
                        nv++; ///numero de coeficientes que he leido
                        if (!feof(fich_num))
                                fscanf(fich_num,"%c",&borradorCaracter); ///leo la coma.

                }

                v_datos = (float *)  malloc(nv * sizeof(float));
                fseek(fich_num,0,SEEK_SET); ///me vuelvo a colocar al principio del fichero

                nv=0;
                while ((!feof(fich_num)) && (nv < nvMax)) ///establezco un límite de valores en
3500
                {
                        fscanf(fich_num,"%f",&v_datos[nv]); ///leo el valor en flotante
                        nv++; ///numero de coeficientes que he leido
                        if (!feof(fich_num))
                                fscanf(fich_num,"%c",&borradorCaracter); ///leo la coma.

                }

                fclose(fich_num);
        }

///Imprimo los valores leidos
            for ( tmp = 0; tmp < nv; tmp++)
                printf("\nValor %d: %lf",tmp,v_datos[tmp]);
}
```

Nota: Algo interesante en el código anterior, es la siguiente porción, en la que se recorre los ficheros incrementando un contador por cada nuevo valor, para saber exactamente la cantidad de memoria a reservar después, evitando así depender de una cantidad fija o teniendo que especificar el número de elementos en el fichero.

Como contrapartida, esta operación requiere de la lectura del fichero dos veces, una para calcular el número de elementos y otra para realmente leerlo y cargarlo en memoria.

```
///aqui leo el fichero y voy contando para saber cuantos memoria tengo que asignar realmente.Así
                //no dependo de un tamaño prefijado, sino depende del tamaño del fichero.
                nv=0;
                while ((!feof(fich_num)) && (nv < nvMax)) ///limite de valores a leer.
                {
                        fscanf(fich_num,"%f",&borradorFloat); ///leo el valor en flotante
                        nv++; ///numero de coeficientes que he leido
                        if (!feof(fich_num))
                                fscanf(fich_num,"%c",&borradorCaracter); ///leo la coma.
```

```
}
```

Función filtro_fir().

Implementa el código principal del filtro FIR implementado, el encargado del cálculo de valores, sus características son las siguientes:

- Acceso a variables globales sin ninguna otra optimización de código.
- Usa doble bucle sin ninguna optimización a nivel de Pragmas o intrínsicos

Codigo filtro_fir()

```
void filtro_fir()
{
 ///Una vez rellenados los vectores de coeficientes y de datos, procedo al cáculo de la salida
///bucle que comprobará todas las entradas posibles.
1:  for (indiceValor = 0; indiceValor < nv; indiceValor++)
    {
2:              salida=0.0; ///inicialmente salida = 0

          //bucle que realiza el cáculo para una entrada dada
 3:  for (indiceCoeficiente=0; indiceCoeficiente < nc;indiceCoeficiente++)
     ///tengo que sumar si o sí, mientras no recorra todos los coeficientes.
    {
  4:       tmp=indiceValor-indiceCoeficiente;
  5:            if ((tmp >= 0) && (tmp < nv))  ///controlo que el índice de datos esté dentro
del rango, si no es así, pongo el valor a cero, para que al multiplicar de cero y no aumente
nada su valor en la suma.

              ///esto puede pasar cuando el valor del índice es más pequeño que el número de
coeficientes que hay, por ejemplo.

   6:                   valorEntrada= v_datos[tmp];
   7:           else
   8:                   valorEntrada = 0.0;

   9:           salida = salida + (v_coef[indiceCoeficiente]*valorEntrada);

      }
   10:      printf("\nIteración : %d Valor: %lf",indiceValor,salida);
      }

}
```

Sobre la función filtro_fir():

- La línea 10, se elimina de las pruebas de tiempo de ejecución para no influir en los tiempos reales.
- La línea 5, controla que el índice apunte a una posición existente del vector de datos, para evitar acceder a direcciones del vector que no existan o con valores desconocidos. Esto realmente no es lo más óptimo, ya que conlleva una comparación en cada iteración, pero si ayuda a que la función sea más portable y adaptable a situaciones de tamaños de coeficientes o datos cambiantes.

(Otra posible implementación de este algoritmo, es a los valores que se salen de rango tanto al principio como al final, operarlos antes y después del bucle, evitando así la comprobación interna, pero siendo menos portable.

He optado por la implementación descrita como primera opción.

Función filtro_fir_intrinsico()

Esta función es igual a la **filtro_fir()** pero utilizando intrínsicos propios del procesador. Se detallan a continuación:

Intrínsico _cmpgt2():

Sustituimos: `if ((tmp >= 0) && (tmp < nv))`

Por: `if (_cmpgt2(tmp,-1) && _cmpgt2(nv,tmp))`

El resultado anterior, realiza la comprobación interna de control de los índices con intrínsicos en lugar de las operaciones de comparación normales.

Instrínsicos _lsadd() y _smpy():

Sustituimos: `salida = salida + v_coef[indiceCoeficiente]*valorEntrada);`

Por: `salida=_sadd(_smpy(v_coef[indiceCoeficiente],valorEntrada),salida);`

Hacemos la multiplicación y suma, con intrínsicos, aunque este último intrínseco no es una buena prueba, puesto que los valores con los que se trabaja en este filtro FIR, son de tipo FLOAT, pero no parece haber intrínsicos de tipo float para la suma y la multiplicación, usando redondeos.

Código filtro_fir_intrinsicos()

```
void filtro_fir_intrinsicos()
{
///Una vez rellenados los vectores de coeficientes y de datos, procedo al cáculo de la salida

        int borrador;

              ///bucle que comprobará todas las entradas posibles.
          for (indiceValor = 0; indiceValor < nv; indiceValor++)
          {
                        salida=0.0; ///inicialmente salida = 0

                  //bucle que realiza el cáculo para una entrada dada
                    for (indiceCoeficiente=0; indiceCoeficiente < nc;indiceCoeficiente++)
///tengo que sumar si o sí, mientras no recorra todos los coeficientes.
                    {
                      tmp=indiceValor-indiceCoeficiente;

                        if (_cmpgt2(tmp,-1) && _cmpgt2(nv,tmp)) ///controlo que el índice de
datos esté dentro del rango, si no es así, pongo el valor a cero, para que al multiplicar de
cero y no aumente nada su valor en la suma.

                ///esto puede pasar cuando el valor del índice es más pequeño que el número de
coeficientes que hay, por ejemplo.

                                valorEntrada= v_datos[tmp];
```

```
                        else
                                valorEntrada = 0.0;


                        salida= _lsadd(_smpy(v_coef[indiceCoeficiente],valorEntrada),salida);


                    }

                    //printf("\nIteración : %d Valor: %lf",indiceValor,salida);
            }
}
```

Función filtro_fir_unroll()

Esta función mejora la función **filtro_fir_intrinsico().**

Agrega estas funcionalidades nuevas:

- Agrega el **pragma UNROLL** (directiva predefinida del preprocesador), la cual intentará ejecutar varias iteraciones del bucle de forma paralela.

- Se ha aplicado un desenrollado de bucle de forma manual, reduciendo en número de iteraciones de los dos bucles a la mitad.

Código filtro_fir_unroll()

```
//
//Versión filtro fir con desenrollado de bucle manual y pragma.
//
void filtro_fir_unroll()
{
        ///Una vez rellenados los vectores de coeficientes y de datos, procedo al cáculo de la
salida
 #pragma UNROLL(2);

              ///bucle que comprobará todas las entradas posibles.
          for (indiceValor = 0; indiceValor < nv; indiceValor+=2)
          {
                        salida=0.0; ///inicialmente salida = 0

                  //bucle que realiza el cáculo para una entrada dada
                    for (indiceCoeficiente=0; indiceCoeficiente < nc;indiceCoeficiente+=2)
///tengo que sumar si o sí, mientras no recorra todos los coeficientes.
                    {
                      tmp=indiceValor-indiceCoeficiente;
                      if (_cmpgt2(tmp,-1) && _cmpgt2(nv,tmp))  ///controlo que el índice de
datos esté dentro del rango, si no es así, pongo el valor a cero, para que al multiplicar de
cero y no aumente nada su valor en la suma.
                                        ///esto puede pasar cuando el valor del índice es más pequeño
que el número de coeficientes que hay, por ejemplo.

                                valorEntrada= v_datos[tmp];
                      else
                                valorEntrada = 0.0;

                    salida= _lsadd(_smpy(v_coef[indiceCoeficiente],valorEntrada),salida);


                    tmp=indiceValor-indiceCoeficiente+1;
                                if (_cmpgt2(tmp,-1) && _cmpgt2(nv,tmp))  ///controlo que el
índice de datos esté dentro del rango, si no es así, pongo el valor a cero, para que al
multiplicar de cero y no aumente nada su valor en la suma.
                                        ///esto puede pasar cuando el valor del índice es más
pequeño que el número de coeficientes que hay, por ejemplo.

                                        valorEntrada= v_datos[tmp];
```

```
                                else
                                        valorEntrada = 0.0;

                                salida=
_lsadd(_smpy(v_coef[indiceCoeficiente+1],valorEntrada),salida);


                    }

                //   printf("\nIteración : %d Valor: %lf",indiceValor,salida);

                    salida=0.0; ///inicialmente salida = 0

                                //bucle que realiza el cáculo para una entrada dada
                                  for (indiceCoeficiente=0; indiceCoeficiente <
nc;indiceCoeficiente+=2) ///tengo que sumar si o sí, mientras no recorra todos los coeficientes.
                                  {
                                    tmp=indiceValor+1-indiceCoeficiente;
                                    if (_cmpgt2(tmp,-1) && _cmpgt2(nv,tmp))  ///controlo que
el índice de datos esté dentro del rango, si no es así, pongo el valor a cero, para que al
multiplicar de cero y no aumente nada su valor en la suma.
                                                ///esto puede pasar cuando el valor del índice
es más pequeño que el número de coeficientes que hay, por ejemplo.

                                                valorEntrada= v_datos[tmp];
                                        else
                                                valorEntrada = 0.0;

                                        salida=
_lsadd(_smpy(v_coef[indiceCoeficiente],valorEntrada),salida);

                                                tmp=indiceValor+1-indiceCoeficiente+1;
                                                        if (_cmpgt2(tmp,-1) && _cmpgt2(nv,tmp))
///controlo que el índice de datos esté dentro del rango, si no es así, pongo el valor a cero,
para que al multiplicar de cero y no aumente nada su valor en la suma.
                                                                ///esto puede pasar cuando el
valor del índice es más pequeño que el número de coeficientes que hay, por ejemplo.

                                                                valorEntrada= v_datos[tmp];
                                                        else
                                                                valorEntrada = 0.0;
                                                        salida=
_lsadd(_smpy(v_coef[indiceCoeficiente+1],valorEntrada),salida);

                                  }
                //      printf("\nIteración : %d Valor: %lf",indiceValor+1,salida);
          }

}
```

Pruebas

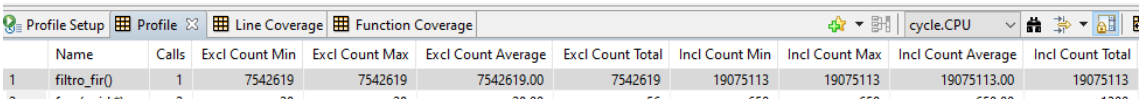
Consideraciones

- Se han realizado pruebas y diferentes optimizaciones para 21 Coeficientes y 3500 datos. Las pruebas corresponden a los tiempos obtenidos tras modificaciones en el código. Los tiempos de la carga de datos se descartan, o sea, la función **obtenerValores()**, no se tiene en cuenta en la suma de tiempos.

- Además, la impresión de los resultados por pantalla, también se ha deshabilitado para ser más precisos en los resultados.

- Se han utilizado en todo momento variables globales en todas las prueba, para centrarnos únicamente en los mejoras obtenidas en la ejecución del algoritmo.

  Versión 1: Solo versión principal, función filtro_fir() llamada desde el main() , sin ninguna optimización a nivel de compilación, o sea, **--opt_level=off.**


Profile Setup | Profile | Line Coverage | Function Coverage | cycle.CPU

	Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
1	filtro_fir()	1	7542619	7542619	7542619.00	7542619	19075113	19075113	19075113.00	19075113

  Observamos que tarda 19.075.113 cycles de CPU.

  Versión 2: Igual a la versión anterior pero con nivel de optimización --opt_level = 2.

	Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
1	filtro_fir()	1	4207049	4207049	4207049.00	4207049	15739543	15739543	15739543.00	15739543

  Observamos una mejora bastante importante de un **18% más rápido** tan solo con la optimización realizada por el compilador para este procesador.

  Versión 3: Utilizamos la función **filtro_fir_intrínsicos()** con –opt-level=2.

  Puede verse como los tiempos con respecto a la función filtro_fir() --opt_level=2 (versión 2) .son mucho menores, produciendo una **mejora de casi un 50%**

	Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
1	filtro_fir()	1	4207049	4207049	4207049.00	4207049	15739543	15739543	15739543.00	15739543
2	filtro_fir_int...	1	4826549	4826549	4826549.00	4826549	10060074	10060074	10060074.00	10060074

  Versión 4: Utilizamos la función filtro_fir_unroll() con –opt-level = 2.

  Puede verse como solo con los desenrollados de bucle, sobre todo el manual, se ha conseguido otra mejora respecto a la versión anterior, de casi otro **50 % de mejora.**

	Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total
1	filtro_fir()	1	4207049	4207049	4207049.00	4207049	15739543	15739543	15739543.00	15739543
2	filtro_fir_int...	1	4826549	4826549	4826549.00	4826549	10060074	10060074	10060074.00	10060074
3	filtro_fir_un...	1	2842054	2842054	2842054.00	2842054	5458947	5458947	5458947.00	5458947

Conclusión

La función f**iltro_fir_unroll()** ha conseguido obtener un 78 % de mejora de rendimiento con respecto al algoritmo base **filtro_fir().**

Optimizando el código, como por ejemplo con el “desenrollado manual de bucles”, se consigue una optimización bastante grande de los DSP, pero realmente eso puede ser algo muy genérico en todos los ámbitos y procesadores, o sea, a mayor optimización de código mayor eficiencia.

Lo que realmente creo que es destacable, son las opciones asociadas al compilador de los DSP, que son los que realmente sacan partido a las arquitectura específica del DSP, a su juego de instrucciones específico ISA, me refieros a los pragmas , intrínsicos, y niveles de optimización, que consiguen sin prácticamente modificar la programación, mejoras de rendimiento nada despreciables.