INFORME TAREA FILTRO FIR CODE COMPOSER

Rebeca Maestro López

INTRODUCCIÓN:

Este es un informe sobre el trabajo realizado en el modulo 2: DSP de la asignatura de Arquitecturas Avanzadas y de Propósito Específico cuenta con dos partes:
la implementación y optimización de un filtro FIR, y la optimización de un código previamente dado.
FIR es un acrónimo en inglés para Finite Impulse Response o Respuesta finita al impulso. Se trata de un tipo de filtros digitales cuya respuesta a una señal o impulso como entrada tendrá un número finito de términos no nulos.
Su expresión en el dominio nn es:

$$y_n = \sum_{k=0}^{N-1} b_k x_{n-k}$$yn = k=0∑N−1 bkxn−k

donde $y_n$yn es el valor de salida y $b_k$bk son los coeficientes y x_{n-k}xn−k la señal de entrada.

Primero vamos a ver unos ejemplos del código realizado y las optimizaciones implementadas.Despúes comentaremos los resultados obtenidos.

CARACTERÍSTICAS DEL PROYECTO:

El trabajo se divide en 6 ficheros con las distintas versiones del código.

- implementacion1.c
- version1.c
- version2.c
- version3.c
- version4.c
- version5.c
  Como indican los nombres de los ficheros, en implementacion1.c se encuentra la primera parte del trabajo que incluye la implementación de la función de filtro FIR y la lectura de los valores de los coeficientes (archivo coeficientes.csv) y de los valores del archivo musica4.csv.
  En el resto se incluye la funcion `practica()` en el main y las modificaciones correspondientes respecto a las distintas versiones de las funciones del fichero practica.c y del filtro FIR.
  También se lleva a cabo el profiling en cada uno de los main con

```
            // Code to be profiled
            start_time = _itoll(TSCH, TSCL);
            //**************************
            //----------INTRODUCIR FUNCION--------
            //**************************
            end_time = _itoll(TSCH, TSCL);
```

Midiendo ambas funciones a la vez, `filtrofir` y `practica()`.

EXPLICACIÓN DEL CÓDIGO:

Funcion del filtro FIR

Como vemos consta de dos bucles anidados, en los que el vector de salida toma el valor de la acumulacion del vector de entrada por los coeficientes correspondientes.

```
float* filtrofir(float* vector_coef, float* vector_datos){
           float* vector_out = (float *) malloc(NData * sizeof(float));
            int i;
            int j;
            for(i=0; i<NData; i++){
                vector_out[i] = 0;
                for(j=0; j<NCoef ; j++)
                    vector_out[i] += vector_coef[j] * vector_datos[i-j];
            }
          return  vector_out;

}
```

Lectura de coeficientes

Debido a que los coeficientes se encuentra en formato de cadena char separados por comas, necesitamos separarlos y tokenizarlos para poder pasarlos al vector como float .

```
float* inicializacion_coeficientes()
{
    float* vector_coeficientes = (float*) malloc(NCoef*sizeof(float));

    int i=0;
    FILE *fich_coef = fopen ("../Coeficientes.csv", "r");
    if(fich_coef == NULL){
                printf("No se pudo abrir archivo de coeficientes");

        }
    const size_t line_size = 3000;
    char* line = malloc(line_size);
    fgets(line, line_size, fich_coef);
    char delim[] = ",";
    char* token;
    for (token = strtok(line, delim); token; token = strtok(NULL, delim) )
     {
        if( i>=NCoef)break;
        else{
         vector_coeficientes[i]=atof(token);
         printf("coef=%s\n", token);
         i++;
        }
     }

    fclose(fich_coef);

    return vector_coeficientes;
}
```

Lectura de Datos

El fichero musica4.csv tiene los datos bien dispuestos por ello no hace falta tokenizar y con fscanf se pasan al vector de entrada el número de datos que indique NData.

```
float* inicializacion_vector_in()
{

    float* vector_entrada = (float*) malloc((NData)*sizeof(float));

    int i=0;
    FILE *fich_datos = fopen("../musica4.csv","r");
    if(fich_datos == NULL){
        printf("No se pudo abrir archivo de datos");

    }
    while(fscanf(fich_datos, "%f", &vector_entrada[i]) != EOF && i<NData)
       i++;
    fclose(fich_datos);
    //free(v_fich_num_csv );

    return vector_entrada;

}
```

Versión 1

La version 1 unicamente se diferencia en que se llama a la función practica desde el main y se miden los ciclos de CPU

```
void main()
{
            float* vector_in;
            float* vector_coef;
            float* vector_out ; //Inicializacion de Vector Out

            vector_coef = inicializacion_coeficientes();
            vector_in = inicializacion_vector_in();

            uint64_t start_time, end_time, overhead, cyclecount;

                   // In the initialization portion of the code:
                   start_time = _itoll(TSCH, TSCL);
                   end_time = _itoll(TSCH, TSCL);
                   overhead = end_time-start_time; //Calculating the overhead of the method.
                   // Code to be profiled
                   start_time = _itoll(TSCH, TSCL);
                   //*************************
                   //----------------
                    practica();
                    vector_out = filtrofir(vector_coef,vector_in);


                   //*************************
                   end_time = _itoll(TSCH, TSCL);
                   cyclecount = end_time-start_time-overhead;
                   printf("The code cyclecount: %lld CPU cycles\n", cyclecount);

                int i;

            printf("\t\tVECTOR SALIDA:\n");
                for(i=0;i<NData;i++)
                    printf("%f\n",vector_out[i]);
                printf("================================================\n");



}
```

Versión 2

En esta versión se realizan pequeñas modificaciones haciendo uso de las keywords CONST y RESTRICT.

```
float* filtrofir( float* const restrict vector_coef,  float* const restrict vector_datos){
           float * restrict vector_out = (float *) malloc(NData * sizeof(float));

    //main
         float* restrict vector_in;
            float* restrict vector_coef;
            float* restrict vector_out ;
     //practica

        int * restrict arr1 = malloc(Size1 * sizeof(int));
        int * restrict arr2 = malloc(Size2 * sizeof(int));
        int * restrict arr3 = malloc(Size2 * sizeof(int));
        int * restrict c = malloc(Size2 * sizeof(int));
        int * restrict d = malloc(Size2 * sizeof(int));
    //coeficientes
    float*  restrict vector_coeficientes = (float*) malloc(NCoef*sizeof(float));
    //entrada
     float* restrict  vector_entrada = (float*) malloc((NData)*sizeof(float));
```

Versión 3

En esta versión se mantienen las modificaciones anteriores y se realiza el desenrollado manual y optimizaciones de bucles y condicionales como se indica a continuación.

```
// DESENROLLADO MANUAL DE PROLOGO Y EPILOGO
float* filtrofir(float* const restrict vector_coef, float* const restrict vector_datos){
           float* restrict vector_out = (float *) malloc(NData * sizeof(float));
            int i;
```

```
            for(i=0; i<NData; i++){
                vector_out[i] = 0;
                int indice=0;
                int iteraciones = (NCoef/BLOQUE);
                int resto = (NCoef%BLOQUE);
                while(iteraciones-- > 0){

                    vector_out[i] += vector_coef[0+indice] * vector_datos[i-0+indice];
                                vector_out[i] += vector_coef[1+indice] * vector_datos[i-1+indice];
                                vector_out[i] += vector_coef[2+indice] * vector_datos[i-2+indice];
                                vector_out[i] += vector_coef[3+indice] * vector_datos[i-3+indice];
                                vector_out[i] += vector_coef[4+indice] * vector_datos[i-4+indice];
                                vector_out[i] += vector_coef[5+indice] * vector_datos[i-5+indice];
                                vector_out[i] += vector_coef[6+indice] * vector_datos[i-6+indice];
                                vector_out[i] += vector_coef[7+indice] * vector_datos[i-7+indice];
                                vector_out[i] += vector_coef[8+indice] * vector_datos[i-8+indice];
                                vector_out[i] += vector_coef[9+indice] * vector_datos[i-9+indice];
                                indice +=BLOQUE;
                }
                switch(resto) {

                                case 1: vector_out[i] += vector_coef[1+indice] * vector_datos[i-1+indice];
                                case 2: vector_out[i] += vector_coef[2+indice] * vector_datos[i-2+indice];
                                case 3: vector_out[i] += vector_coef[3+indice] * vector_datos[i-3+indice];
                                case 4: vector_out[i] += vector_coef[4+indice] * vector_datos[i-4+indice];
                                case 5: vector_out[i] += vector_coef[5+indice] * vector_datos[i-5+indice];
                                case 6: vector_out[i] += vector_coef[6+indice] * vector_datos[i-6+indice];
                                case 7: vector_out[i] += vector_coef[7+indice] * vector_datos[i-7+indice];
                                case 8: vector_out[i] += vector_coef[8+indice] * vector_datos[i-8+indice];
                                case 9: vector_out[i] += vector_coef[9+indice] * vector_datos[i-9+indice];

                                case 0: ;
                }
            }
          return  vector_out;

}
// OPTIMIZACIÓN DE LA CONDICIÓN
int funcion_22(int val){
        int res;
        res = val+20;
        if(val > 10){
                res = val+5;
        }
        return res;
}
// REDISTRIBUCIÓN DE LOS BUCLES Y CONDICIONES
void practica2(){
        // Assign memory
        int *arr1 = malloc(Size1 * sizeof(int));
        int *arr2 = malloc(Size2 * sizeof(int));
        int *arr3 = malloc(Size2 * sizeof(int));
        int *c = malloc(Size2 * sizeof(int));
        int *d = malloc(Size2 * sizeof(int));

        int sum1, sum2, sum3;

        int i;
        i=0;

        for(i; i<50; i++){
                int a, b;
                a = funcion_22(i);
                b = funcion_22(3);
                arr1[i]=(b*a) + i;
        }
        i=0;
        for(i; i<300; i++){
                int a;
                a = funcion_1(i);
                d[i] = (i+25 * 2)+a;
                d[i] = funcion_22(d[i]);
                c[i]=i+45;
```

```
        }

        minV(arr2, d, c);
        vDotV(arr3, arr2, c);
        sum1 = sumVelements(arr1, 25);
        sum2 = sumVelements(arr2, 25);
        sum3 = sumVelements(arr3, 25);

        //printf("sum1: %d, sum2: %d, sum3: %d\n",sum1, sum2, sum3);


        free(arr1);
        free(arr2);
        free(arr3);
        free(c);
        free(d);
}
```

Versión 4

Aquí se revierte el desenrollado manual y se incluyen Pragmas y desenrollados automáticos.

```
//desenrollado con Pragma UNROLL
float* filtrofir(float* const restrict vector_coef, float* const restrict vector_datos){
           float* vector_out = (float *) malloc(NData * sizeof(float));

                int i;
                int j;

                #pragma MUST_ITERATE(1000)
                for(i=0; i<NData; i++){
                        vector_out[i] = 0;
                        #pragma UNROLL(3)
                        for(j=0; j<NCoef ; j++)
                                vector_out[i] += vector_coef[j] * vector_datos[i-j];
                }

          return  vector_out;

}
//main
#pragma DATA_ALIGN(8)
            float* restrict vector_in;
            float* restrict vector_coef;
            float* restrict vector_out ;
// practica
#pragma MUST_ITERATE(50)
        for(i; i<50; i++){
                int a, b;
                a = funcion_22(i);
                b = funcion_22(3);
                arr1[i]=(b*a) + i;
        }
        i=0;
        #pragma MUST_ITERATE(300)
        for(i; i<300; i++){
                int a;
                a = funcion_1(i);
                d[i] = (i+25 * 2)+a;
                d[i] = funcion_22(d[i]);
                c[i]=i+45;

        }
// funcion minV optimizada
    void minV2(int *minV, int *in1, int *in2){
        int i;
    #pragma MUST_ITERATE(200)
        for (i = 0; i < Size2; i++) {
                minV[i] = in2[i];
                if(in1[i] < in2[i]){
                        minV[i] = in1[i];
                }


        }
}
```

Versión 5

En esta continuamos con los cambios anteriores y añadimos distintos intrínsecos.

```
// USO DE n_assert , y _amem4_const
float* filtrofir(float* const restrict vector_coef, float* const restrict vector_datos){

                _nassert(((int)vector_datos & 0x3) == 0);
                _nassert(((int)vector_coef & 0x3) == 0);
           float* vector_out = (float *) malloc(NData * sizeof(float));

                int i;
                int j;

                #pragma MUST_ITERATE (1000)
                for(i=0; i<NData; i++){
                        vector_out[i] = 0;
                        #pragma UNROLL(21)
                        for(j=0; j<NCoef ; j++)
                                vector_out[i] += _amem4_const(&vector_coef[j]) * _amem4_const(&vector_datos[i-j]);
                }

          return  vector_out;

}
// USO de _min2
void minV2(int *minV, int *in1, int *in2){
        int i;
#pragma MUST_ITERATE(200,200)
        for (i = 0; i < Size2; i++) {
                minV[i] =_min2( in1[i],in2[i]);
        }
}
```

TABLAS COMPARATIVAS:

Después de debugear todas las versiones con los distintos niveles de optimización que nos provee Code Composer Studio 5.5 , nos dan los siguientes resultados :

Versión\ Nivel de Optimización	OFF	0	1	2	3
1	4144251	3857952	3804124	3876346	3859559
2	4135220	3849238	3795410	3733908	3733908
3	4999809	4705460	4614247	4652621	4548177
4	4131292	3844715	3791712	3665088	3660646
5	2724939	2502562	2408549	2257963	2253496

En ciclos de CPU (teniendo en cuenta que se imprimian todos los coeficientes)

VALORACIÓN PERSONAL:

Como podemos observar en el gráfico se notan mucho las optimizaciones realizadas por el compilador, con el nivel 0 de optimización ya se consigue una caida de los ciclos en todas las versiones, en este nivel se asignan variables a registros y se simplifican expresiones entre otras cosas. En los niveles sucesivos los cambios no son tan drásticos, ya que con el nivel 1, se eliminan asignaciones no utilizadas y expresiones comunes locales. Además podemos ver que comparando la versión 1 con la 2 , en la cúal solo se añadieron las keywords const y restrict, no hay grandes diferencias y los resultados son muy similares.
Por otro lado tenemos la versión 3 que destaca porque tiene el desenrollado manual que no parece tener muy buenos resultados, con ninguno de los niveles nos acercamos a las cifras anteriores, lo cuál indica que hacemos trabajar más a la maquina en asignaciones, lecturas,..que lo que le ahorramos desenrollando el bucle del filtro fir. En la versión 4 se mejora en todos los niveles aunque tampoco de manera drástica y es en la versión 5 en la que vemos los frutos de las optimizaciones y se consiguen unos resultados muy buenos con notables cambios entre cada nivel de optimización.