Filtro FIR

Arquitecturas Avanzadas y de Propósito Específico

Cheuk Kelly Ng Pante (alu0101364544@ull.edu.es)

28 de junio de 2024

Índice general

1. Introducción

FIR es un acrónimo en inglés para *Finite Impulse Response* o Respuesta Finita al Impulso. Se trata de un tipo de filtro digital que si su entrada es un impulso (una delta de Kronecker), su salida es un número limitado de términos no nulos. Este tipo de filtro se utiliza comúnmente en aplicaciones de procesamiento de señales, como en la industria de las telecomunicaciones, el procesamiento de audio y la ingeniería de control.

Su expresión en el dominio n es la siguiente:

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

donde x_n es la entrada, y_n es la salida, b_k son los coeficientes del filtro y N es el orden del filtro.

La estructura básica de un filtro FIR es la siguiente:

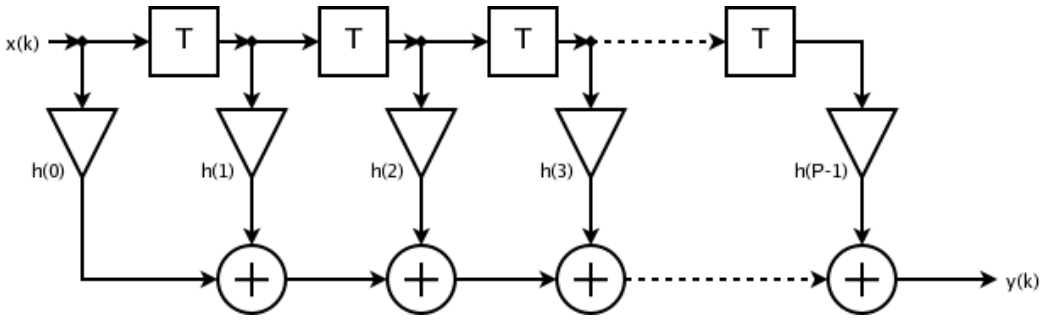

Figura 1.1: Estructura básica de un filtro FIR

2. Características del proyecto

El proyecto se divide en 5 versiones de filtros FIR:

- Versión 0: Filtro FIR base
- Versión 1: Filtro FIR mejorado
- Versión 2: Incluye el uso de keywords, como ***const*** y ***restrict***
- Versión 3: Desenrrollado manual y optimizacion de bucles, condicionales, etc.
- Versión 4: Utilización de ***pragmas***
- Versión 5: Uso de ***intrinsics***

En cada versión incluye una implementación de un filtro FIR, dependiendo de la versión, y la lectura de los valores de los coeficientes (Coeficientes.csv) y los valores de los datos de entrada (musica4.csv).

En cuanto al profiling se optado por usar *clock()*, que es una función que mide el tiempo de ejecución de un programa. En principio se pensó en usar *gprof*, que es una herramienta de análisis de rendimiento para

aplicaciones UNIX, pero no se pudo ya que los tiempos de ejecución eran muy pequeños y no se podía medir correctamente.

Además, se ha calculado los ciclos de reloj de cada versión del filtro FIR. Para ello, se ha creado una función utilizando la instrucción *rdtsc* (Real Time Stamp Counter) que mide el número de ciclos de reloj desde el último reinicio. La función es la siguiente:

```
uint64_t rdtsc(){
  unsigned int lo, hi;
  __asm__ __volatile__("rdtsc" : "=a"(lo), "=d"(hi));
  return ((uint64_t)hi << 32) | lo;
}
```

Lo que hace la función es utiliza la instrucción de ensamblador *rdtsc* para leer el contador de tiempo de la CPU, y devuelve este valor como un número de 64 bits pero dividido en dos partes de 32 bits, *lo* es la parte baja y *hi* es la parte alta. Entonces, se ejecuta la instrucción *__asm__ __volatile__* que es una instrucción de ensamblador en línea que se utiliza para incrustar código ensamblador en el código C.

El *profiling* se ha hecho de la siguiente forma:

```
start_cycle = rdtsc();
start = clock();
// --- Codigo a medir ---
end = clock();
end_cycle = rdtsc();
```

Por otra parte, para calcular el tiempo de ejecución y los ciclos de reloj se ha repetido N veces, para obtener un promedio de los valores. En este caso, se ha decidido hacer 100 repeticiones, pero se puede cambiar el valor de N ya que se define como una constante en cada versión del filtro FIR.

```
#define REPETITIONS 100

// Aplicacion del filtro FIR
for (i = 0; i < REPETITIONS; i++) {
  start_cycle = rdtsc();
  start = clock();
  firfilter(vector_coef, vector_in, result);
  end = clock();
  end_cycle = rdtsc();

  elapsed = (double)(end - start) * 1000.0 / CLOCKS_PER_SEC;
  mean_time[i] = elapsed;
  mean_cycles[i] = end_cycle - start_cycle;
}
```

3. Implementaciones

Lectura de los coeficientes

Para leer los coeficientes del filtro FIR se ha creado la siguiente función:

```
float* init_coefficients() {
  float* array_coeff = (float*)malloc(COEF * sizeof(float));
  int i = 0;
  FILE* file_coeff = fopen("../data/Coeficientes.csv", "r");
  if (file_coeff == NULL) {
    printf("Error al abrir el archivo\n");
    exit(1);
  }

  while (fscanf(file_coeff, "%f", &array_coeff[i]) != EOF && i < COEF) {
    i++;
  }
  fclose(file_coeff);

  return array_coeff;
}
```

Lo que se hace es abrir el archivo *Coeficientes.csv* que contiene los valores de los coeficientes del filtro FIR, y se lee cada valor hasta que se llegue al final del archivo o se haya leído todos los valores. Luego, se cierra el archivo y se devuelve el vector de coeficientes.

Lectura de los datos de entrada

Para leer los datos de entrada del filtro FIR se ha creado la siguiente función:

```
float* init_data() {
  float* array_data = (float*)malloc(N * sizeof(float));

  int i = 0;
  FILE* file_data = fopen("../data/musica4.csv", "r");
  if (file_data == NULL) {
    printf("Error al abrir el archivo\n");
    exit(1);
  }

  while (fscanf(file_data, "%f", &array_data[i]) != EOF && i < N) {
    i++;
  }

  fclose(file_data);

  return array_data;
}
```

Para la lectura de los datos de entrada se hace lo mismo que en la lectura de los coeficientes, se abre el archivo *musica4.csv* que contiene los valores de los datos de entrada, se leen los valores hasta que se llegue al final del archivo o se haya leído todos los valores y se cierra el archivo.

3.1. Versión base

Para la implementación base se ha hecho la función *firfilter()* que recibe los coeficientes del filtro FIR y los datos de entrada como punteros a float, y devuelve un puntero a float con los valores de la salida del filtro FIR.

```
float* firfilter(float* vector_coef, float* vector_data) {
  float* result = (float*)malloc((N + COEF - 1) * sizeof(float));
  int i, j;
  for (i = 0; i < N; i++) {
    result[i] = 0;
    for (j = 0; j < COEF; j++) {
      if (i - j >= 0 && i - j < N) {
        result[i] += vector_coef[j] * vector_data[i - j];
      }
    }
  }

  return result;
}
```

Lo que hace la función es inicializar un vector de float con el tamaño de la suma de los tamaños de los vectores de coeficientes y de los datos de entrada. Luego, se inicializa un bucle que recorre el vector de salida, y dentro de este bucle se inicializa otro bucle que recorre el vector de coeficientes. Dentro de este segundo bucle se comprueba si el índice de la salida menos el índice del bucle interno es mayor o igual a 0 y menor que el tamaño del vector de datos de entrada. Si se cumple la condición, se multiplica el valor del coeficiente por el valor del dato de entrada y se suma al valor de la salida.

3.2. Versión 1 - Mejora de la implementación base

Para la versión 1 se ha mejorado la implementación base del filtro FIR. Lo que se ha hecho es trabajar directamente con los punteros de los vectores de coeficientes, de los datos de entrada y de la salida. Usar punteros directamente y no devolver un puntero a float con los valores de la salida del filtro FIR es más eficiente. Además, se ha eliminado la condición del bucle interno, ya que no es necesario.

```
void firfilter(float* vector_coef, float* vector_data, float* result) {
  int i, j;
  for (i = 0; i < N; i++) {
    result[i] = 0;
    for (j = 0; j < COEF; j++) {
      result[i] += vector_coef[j] * vector_data[i - j];
    }
  }
}
```

3.3. Versión 2 - Uso de keywords

En la segunda implementación se ha incluido el uso de keywords, como *const* y *restrict*. La keyword ***const*** se utiliza para declarar una variable como constante, es decir, que no se puede modificar su valor. La keyword ***restrict*** se utiliza para indicar que dos punteros no se solapan, es decir, que no apuntan a la misma dirección de memoria. Esto permite al compilador realizar optimizaciones en el código.

```
void firfilter(const float* restrict const vector_coef,
               const float* restrict const vector_data,
               float* restrict const result) {
  int i, j;
  for (i = 0; i < N; i++) {
    result[i] = 0;
    for (j = 0; j < COEF; j++) {
      result[i] += vector_coef[j] * vector_data[i - j];
    }
  }
}
```

3.4. Versión 3 - Desenrrollado manual

En la tercera implementación se ha desenrrollado manualmente el bucle interior. Esta es un técnica de optimización que consiste en incrementar el número de instrucciones dentro de un bucle para disminuir el número de iteraciones del bucle. En este caso, se ha desenrrollado el bucle interior expandiendo cada iteración de j manualmente, en este caso se ha desenrrollado 25 veces. Además, se agrega una condición para evitar que se acceda a límites fuera del vector de datos de entrada. Por último, se multiplica por los elementos correspondientes del vector de datos y se acumulan en el resultado.

```
void firfilter(const float* restrict const vector_coef,
               const float* restrict const vector_data,
               float* restrict const result) {
  int i;
  for (i = 0; i < N; i++) {
    result[i] = 0;

    if (i >= 24) result[i] += vector_coef[24] * vector_data[i - 24];
    if (i >= 23) result[i] += vector_coef[23] * vector_data[i - 23];
    if (i >= 22) result[i] += vector_coef[22] * vector_data[i - 22];
    if (i >= 21) result[i] += vector_coef[21] * vector_data[i - 21];
    if (i >= 20) result[i] += vector_coef[20] * vector_data[i - 20];
    if (i >= 19) result[i] += vector_coef[19] * vector_data[i - 19];
    if (i >= 18) result[i] += vector_coef[18] * vector_data[i - 18];
    if (i >= 17) result[i] += vector_coef[17] * vector_data[i - 17];
    if (i >= 16) result[i] += vector_coef[16] * vector_data[i - 16];
    if (i >= 15) result[i] += vector_coef[15] * vector_data[i - 15];
    if (i >= 14) result[i] += vector_coef[14] * vector_data[i - 14];
    if (i >= 13) result[i] += vector_coef[13] * vector_data[i - 13];
    if (i >= 12) result[i] += vector_coef[12] * vector_data[i - 12];
    if (i >= 11) result[i] += vector_coef[11] * vector_data[i - 11];
    if (i >= 10) result[i] += vector_coef[10] * vector_data[i - 10];
    if (i >= 9) result[i] += vector_coef[9] * vector_data[i - 9];
    if (i >= 8) result[i] += vector_coef[8] * vector_data[i - 8];
    if (i >= 7) result[i] += vector_coef[7] * vector_data[i - 7];
    if (i >= 6) result[i] += vector_coef[6] * vector_data[i - 6];
    if (i >= 5) result[i] += vector_coef[5] * vector_data[i - 5];
    if (i >= 4) result[i] += vector_coef[4] * vector_data[i - 4];
    if (i >= 3) result[i] += vector_coef[3] * vector_data[i - 3];
    if (i >= 2) result[i] += vector_coef[2] * vector_data[i - 2];
    if (i >= 1) result[i] += vector_coef[1] * vector_data[i - 1];
    result[i] += vector_coef[0] * vector_data[i - 0];
  }
}
```

3.5. Versión 4 - Uso de pragmas

En la cuarta implementación se ha utilizado *pragmas* para optimizar el código. Los *pragmas* son directivas que se utilizan para indicar al compilador cómo debe compilar el código. En este caso, se ha utilizado el pragma *#pragma omp parallel for* para paralelizar el bucle exterior del filtro FIR. Y se ha utilizado el pragma *#pragma unroll 25* para desenrrollar el bucle interior. El código es el siguiente:

```
void firfilter(const float* restrict const vector_coef,
               const float* restrict const vector_data,
               float* restrict const result) {
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    result[i] = 0;
    #pragma unroll 25
    for (int j = 0; j < COEF; j++) {
      result[i] += vector_coef[j] * vector_data[i - j];
    }
  }
}
```

3.6. Versión 5 - Uso de intrínsecos

Por último, en la quinta implementación se ha utilizado intrínsecos SIMD (Single Instruction, Multiple Data) 128 bits para optimizar el código. Los intrínsecos son funciones que se utilizan para acceder a instrucciones específicas de la CPU. En este caso, se ha utilizado la instrucción *__m128* para acceder a los registros SIMD de 128 bits. Además, se ha utilizado la función *_mm_loadu_ps()* para cargar los valores de los coeficientes y los datos de entrada en registros. La función *_mm_mul_ps()* para multiplicar los valores de los coeficientes por los valores de los datos de entrada y la función *_mm_add_ps()* para sumar los resultados. Por último, se ha utilizado la función *_mm_storeu_ps()* para almacenar los resultados en el vector de salida. Haciendo uso de los *intrinsics* se consigue una mejora en el rendimiento del filtro FIR. El código es el siguiente:

```
void firfilter(const float* restrict const vector_coef,
               const float* restrict const vector_data,
               float* restrict const result) {
  int i, j;
  __m128 coef_reg = _mm_loadu_ps(vector_coef);
  __m128 data_reg;
  int simd_size = 8;

  for (i = 0; i < N; i += simd_size) {
    __m128 result_reg = _mm_setzero_ps();
    for (j = 0; j < COEF; j++) {
      if (i >= j) {
        data_reg = _mm_loadu_ps(&vector_data[i - j]);
        __m128 coef_mul_data = _mm_mul_ps(coef_reg, data_reg);
        result_reg = _mm_add_ps(result_reg, coef_mul_data);
      }
    }
    _mm_storeu_ps(&result[i], result_reg);
  }
}
```

4. Resultados

En esta sección se presentan los resultados de las implementaciones del filtro FIR. Se ha medido el tiempo de ejecución (en milisegundos) y los ciclos de reloj de cada versión del filtro FIR. Se ha hecho 100 repeticiones para obtener un promedio de los valores. Los resultados se presentan en la siguiente tabla:

Versión \ Nivel Optimización	**Off**		**0**	
	Tiempo ejecución	Ciclo reloj	Tiempo ejecución	Ciclo reloj
Base	0.686350	2336651	0.653410	2224720
1	0.542620	1849069	0.539160	1836593
2	0.590532	2011620	0.587926	2002169
3	0.516320	1758842	0.541410	1846202
4	0.687710	2343170	0.591850	2016569
5	0.134570	460487	0.129370	442618

Cuadro 4.1: Resultados de las implementaciones del filtro FIR

Versión \ Nivel Optimización	**1**		**2**	
	Tiempo ejecución	Ciclo reloj	Tiempo ejecución	Ciclo reloj
Base	0.355950	1212921	0.322680	1099958
1	0.294340	1003215	0.102240	349864
2	0.120157	411047	0.097767	334884
3	0.430710	1469602	0.081840	281132
4	0.159950	547177	0.107720	369235
5	0.023110	81309	0.020160	72086

Cuadro 4.2: Resultados de las implementaciones del filtro FIR

Versión \ Nivel Optimización	**3**	
	Tiempo ejecución	Ciclo reloj
Base	0.321940	1097628
1	0.028110	97467
2	0.031861	110578
3	0.078820	270689
4	0.029960	104292
5	0.019010	66657

Cuadro 4.3: Resultados de las implementaciones del filtro FIR

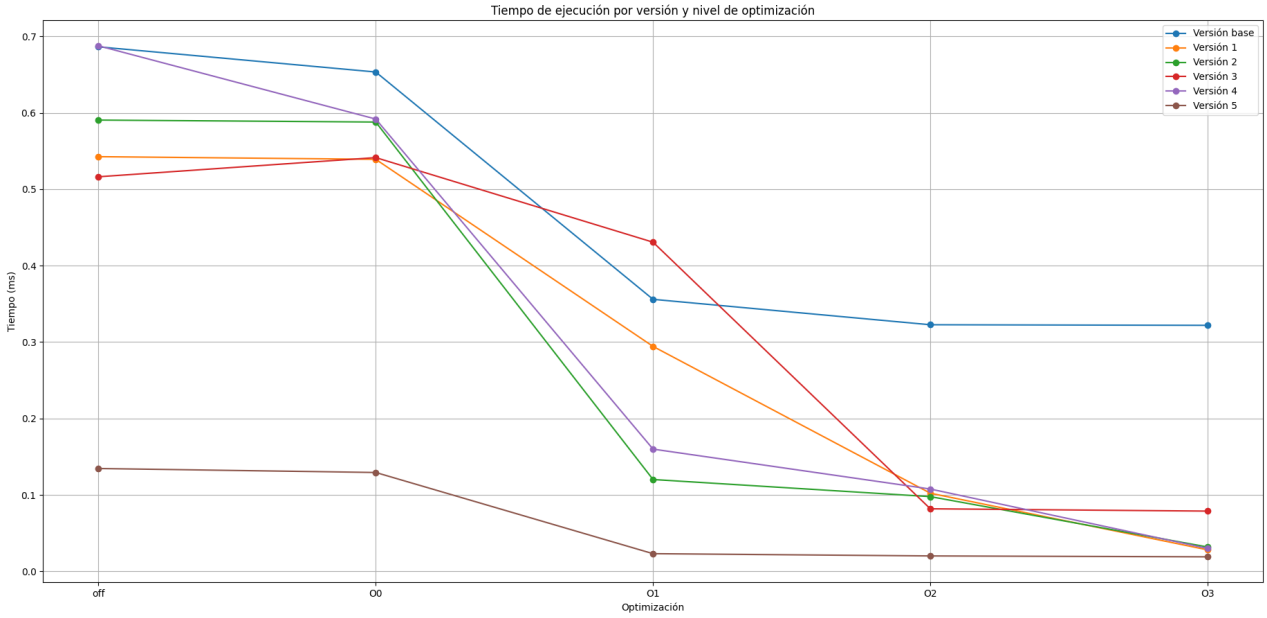

Figura 4.1: Gráfica de los resultados de las implementaciones del filtro FIR

Como podemos observar en la tabla y en la gráfica, la versión 5 es la que tiene un mejor rendimiento en cuanto a tiempo de ejecución y ciclos de reloj. Además, al usar las optimizaciones *O0* hasta *O3* se puede observar que el rendimiento mejora en todas las versiones del filtro FIR. Por otro lado, la versión 5 es la más eficiente en cuanto a tiempo de ejecución y ciclos de reloj, ya que se ha utilizado intrínsecos SIMD 128 bits.

5. Conclusiones

Este proyecto ha servido para ver cómo se puede mejorar el rendimiento de un filtro FIR utilizando diferentes técnicas de optimización. Además, implementar diferentes versiones del filtro y comparar su rendimiento. Se ha podido observar que al aplicar optimizaciones en el código, como el uso de keywords, pragmas e intrínsecos, se consigue una mejora en el rendimiento del filtro FIR. Además, se ha podido medir el tiempo de ejecución y los ciclos de reloj de cada versión del filtro FIR, y se ha podido observar que la versión 5 es la más eficiente en cuanto a tiempo de ejecución y ciclos de reloj. Por último, se ha podido observar que al aplicar las optimizaciones *O0* hasta *O3* se consigue una mejora en el rendimiento de todas las versiones del filtro FIR.