Elaborazione Sequenziale di Segnali Audio con DFT e IDFT

Contents

1 Introduzione

Questo progetto si propone di elaborare segnali audio utilizzando un approccio sequenziale. L'obiettivo è implementare trasformazioni nel dominio delle frequenze per applicare un filtro passa-basso e produrre un segnale audio modificato. Il processo comprende le seguenti fasi:

- Lettura di un file audio in formato `.wav`;
- Calcolo della Trasformata Discreta di Fourier (DFT);
- Applicazione di un filtro passa-basso nel dominio delle frequenze;
- Calcolo della Trasformata Inversa (IDFT);
- Scrittura del segnale audio modificato su file;
- Generazione di un report sui tempi di esecuzione.

1.1 Fase di test

Tutti i codici sviluppati sono testati in locale, i report di Nsight Compute sono stati generati utilizzando le GPU fornite dalla piattaforma google colab (Tesla T4).

Table 1: Caratteristiche di Macchina

	Macchina
CPU	AMD Ryzen 7 8845HS
GPU	NVIDIA GeForce RTX 4050 Mobile
O.S.	Fedora Linux 41

I programmi sono stati testati su un audio mono con 44.1 kHz di frequenza di campionamento della durata 2,5 s contenente rumore bianco: `mid_noise.wav`

I risultati della applicazione del filtro si trovano nella directory `./output`, ogni file generato dal programma possiede nel nome il timestamp dell'esecuzione nel formato `Ymd_HMS` (anno, mese, giorno, ore, minuti, secondi).

I tempi di esecuzione sono riportati nei file all'interndo della directory `./reports`. I risultati mostrano tempi di calcolo delle singole fasi del programma e il tempo di esecuzione totale.

2 Implementazione Sequenziale

Il codice è stato scritto in linguaggio `C` e utilizza le librerie standard per la manipolazione di file e la misurazione del tempo. La struttura principale del programma comprende:

2.1 Lettura e Scrittura di File Audio

I file audio sono letti e scritti in formato `.wav` standard, utilizzando un'intestazione di 44 byte per rappresentare i metadati del file. Le seguenti funzioni implementano la lettura e scrittura dei file audio:

Listing 1: Funzione per leggere file audio .wav.

```
// Funzione per leggere i campioni audio da un file .wav
void readWavFile(const char *filename, double *x, int N) {
    FILE *file = fopen(filename, "rb");
    if (file == NULL) {
        printf("Errore nell'apertura del file %s\n", filename);
        exit(1);
    }

    // Salta l'intestazione del file .wav (44 byte standard)
    fseek(file, 44, SEEK_SET);

    // Leggi i campioni audio come int16_t e convertili in double
    int16_t *buffer = (int16_t *)malloc(N * sizeof(int16_t));
    fread(buffer, sizeof(int16_t), N, file);
    for (int i = 0; i < N; i++) {
        x[i] = (double)buffer[i];
    }

    free(buffer);
    fclose(file);
}
```

Listing 2: Funzione per scrivere file audio .wav.

```
// Funzione per scrivere un file .wav con l'intestazione
void writeWavFile(const char *filename, double *x, int N) {
    FILE *file = fopen(filename, "wb");
    if (file == NULL) {
        printf("Errore nell'apertura del file %s\n", filename);
        exit(1);
    }

    // Scrivi un'intestazione standard per un file .wav a 16 bit, mono,
        44.1 kHz
    uint8_t header[44] = {
        'R', 'I', 'F', 'F',
        0, 0, 0, 0, // Placeholder per la dimensione del file
        'W', 'A', 'V', 'E',
        'f', 'm', 't', ' ',
        16, 0, 0, 0, // Dimensione del blocco fmt
        1, 0, // PCM
        1, 0, // Canali (mono)
        0x44, 0xAC, 0x00, 0x00, // Frequenza di campionamento: 44100 Hz
        0x88, 0x58, 0x01, 0x00, // Byte rate: 44100 * 2
        2, 0, // Block align: 2 byte
        16, 0, // Bit depth: 16 bit
        'd', 'a', 't', 'a',
        0, 0, 0, 0 // Placeholder per la dimensione dei dati
    };

    // Calcola la dimensione totale del file e dei dati
```

```
    int dataSize = N * sizeof(int16_t);
    int fileSize = 44 + dataSize - 8;

    // Aggiorna i campi dell'intestazione
    header[4] = (fileSize & 0xFF);
    header[5] = ((fileSize >> 8) & 0xFF);
    header[6] = ((fileSize >> 16) & 0xFF);
    header[7] = ((fileSize >> 24) & 0xFF);

    header[40] = (dataSize & 0xFF);
    header[41] = ((dataSize >> 8) & 0xFF);
    header[42] = ((dataSize >> 16) & 0xFF);
    header[43] = ((dataSize >> 24) & 0xFF);

    // Scrivi l'intestazione
    fwrite(header, sizeof(uint8_t), 44, file);

    // Scrivi i campioni audio convertiti in int16_t
    int16_t *buffer = (int16_t *)malloc(N * sizeof(int16_t));
    for (int i = 0; i < N; i++) {
        buffer[i] = (int16_t)x[i];
    }
    fwrite(buffer, sizeof(int16_t), N, file);

    free(buffer);
    fclose(file);
}
```

2.2 Trasformata Discreta di Fourier (DFT)

La DFT converte un segnale audio dal dominio del tempo a quello delle frequenze. Per il salvataggio di parte reale e complessa del segnale abbiamo utilizzato una struttura `Complesso` contenente `real` e `imag`.
La funzione `dft()` implementa la seguente formula:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot \cos\left(\frac{2\pi kn}{N}\right) - x[n] \cdot \sin\left(\frac{2\pi kn}{N}\right) \tag{1}$$

Listing 3: Implementazione della DFT.

```
// Funzione che calcola la Discrete Fourier Transform di un segnale
    audio
void dft(double *x, Complesso *X, int N) { // N = numero di campioni,
    complessit  O(N^2)
    for (int k = 0; k < N; k++) {
        X[k].real = 0;
        X[k].imag = 0;
        for (int n = 0; n < N; n++) {
            double angle = 2 * PI * k * n / N;
            X[k].real += x[n] * cos(angle);
            X[k].imag -= x[n] * sin(angle);
        }
    }
}
```

2.3 Filtro Passa-Basso

Un filtro passa-basso è applicato nel dominio delle frequenze per rimuovere le componenti indesiderate. La funzione `filtro()` azzera parte reale e immaginaria del segnale al di sopra di una soglia specificata. I nostri test sono stati fatti utilizzando 1000Hz:

Listing 4: Applicazione del filtro passa-basso.

```
// Funzione che applica un filtro passa-basso al segnale audio
void filtro(Complesso *X, int N, int fc, int fs) {
    // Calcolo dell'indice di taglio corrispondente alla frequenza fc
    int cutoffIndex = (int)((fc * N) / fs);

    for (int k = 0; k < N; k++) {
        // Applica il filtro passa-basso
        if (k > cutoffIndex && k < N - cutoffIndex) {
            X[k].real = 0;
            X[k].imag = 0;
        }
    }
}
```

2.4 Trasformata Inversa (IDFT)

La IDFT riporta il segnale audio dal dominio delle frequenze al dominio del tempo. La funzione `idft()` è basata sulla formula:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \cdot \cos\left(\frac{2\pi kn}{N}\right) - X[k] \cdot \sin\left(\frac{2\pi kn}{N}\right) \tag{2}$$

Listing 5: Implementazione della IDFT.

```
// Funzione che calcola l'Inverse Discrete Fourier Transform del
    segnale audio filtrato
void idft(Complesso *X, double *x, int N) { // N = numero di campioni,
    complessit  O(N^2)
    for (int n = 0; n < N; n++) {
        x[n] = 0;
        for (int k = 0; k < N; k++) {
            double angle = 2 * PI * k * n / N;
            x[n] += X[k].real * cos(angle) - X[k].imag * sin(angle);
        }
        x[n] /= N;
    }
}
```

2.5 Misurazione delle Prestazioni

I tempi di esecuzione delle diverse fasi del programma vengono misurati utilizzando la funzione `clock()` della libreria `time.h` per determinare start e end delle funzioni.

Listing 6: Misurazione tempi.

```
    // Calcola la DFT
    start = clock();
    dft(x, X, N);
    end = clock();
    dftTime = (double)(end - start) / CLOCKS_PER_SEC;
```

2.6 Risultati

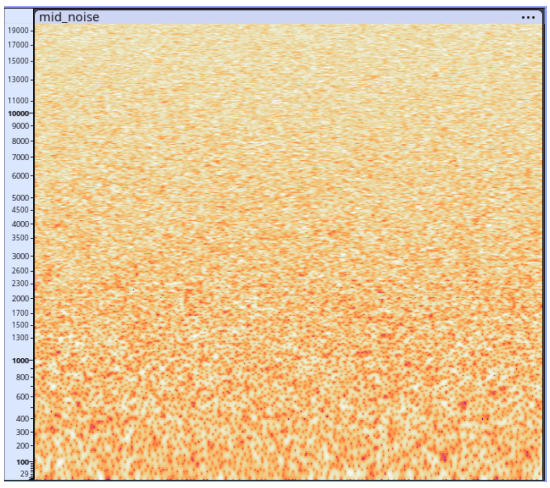
La seguente tabella riporta i tempi di esecuzione del programma sequenziale, che, questi costituiranno la base di confronto per valutare i miglioramenti introdotti nelle fasi successive.

Table 2: Esecuzione Sequenziale

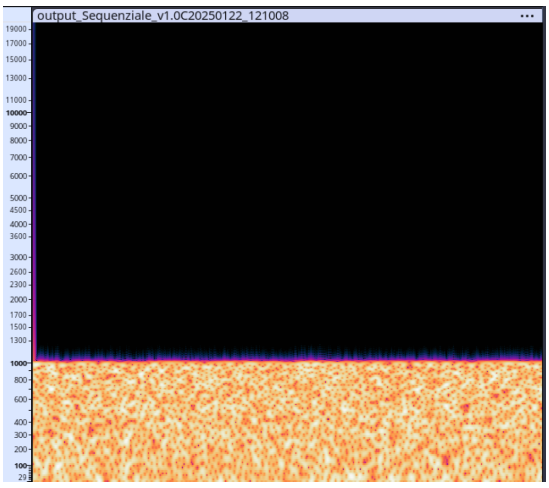
Funzione	**Tempo**
DFT	215.199623s
Filtro	0.000159s
IDFT	215.232599s
Totale	430.432381s

Abbiamo quindi visualizzato il file audio generato utilizzando *Audacity* per la sua immediata funzionalità di visualizzazione dello spettro delle frequenze, le due immagini che seguono mostrano il file `input` e `output`.

Figure 1: Confronto Input Output



(a) Spettro Input



(b) Spettro Output

3 Implementazione Cuda

Nell'implementazione parallela abbiamo scelto di assegnare a ciascun thread il calcolo di un singolo elemento della DFT e della IDFT.

Listing 7: Implementazione della IDFT.

```
__global__ void dftKernel(const double *x, Complesso *X, int N){

    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if(i < N){
        X[i].real = 0;
        X[i].imag = 0;
        for(int j = 0; j < N; j++){
            double angle = 2 * PI * i * j / N;
            X[i].real += x[j] * cos(angle);
            X[i].imag -= x[j] * sin(angle);
        }
    }

}
```

Ogni thread si occupa del calcolo di parte reale e parte immaginari di ogni elemento del segnale.

3.1 Risultati

I risultati dei tempi di elaborazione sono prodotti dal codice e riportati nella cartella `./reports`

Table 3: Esecuzione Sequenziale

Funzione	**Tempo Sequenziale**	**Tempo Parallelo**	**Speedup**
DFT	215.199623s	6.101256s	97.16%
Filtro	0.000159s	0.001747s	−998.74%
IDFT	215.232599s	6.230328s	97.10%
Totale	430.432381s	12.333331s	97.13%

4 Ottimizzazioni

Conseguentemente all'osservazione dei risultati prodotti ci siamo concentrati su possibili ottimizzazioni del software. Nei vari passaggi, sono state introdotte importanti migliorie per ottimizzare le prestazioni complessive. Il valore principale su cui ci siamo concentrati è stato il tempo di esecuzione e le ottimizzazioni sono state effettuate con l'obiettivo di ridurlo. Abbiamo osservato da Nsight Compute i suggerimenti sui report relativi ad ogni versione per poter lavorare alla successiva.

4.1 Versione 1.5, introduzione del formato Float

Nella versione 1.5 abbiamo effettuato una modifica del tipo di dato utilizzato per la rappresentazione dei dati audio e della trasformata. Abbiamo deciso di passare dal tipo `double` al tipo `float` per cercare di sfruttare l'ottimizzazione della GPU per i calcoli in virgola mobile e per il fatto che generalmente i calcoli con i float a singola precisione sono computazionalmente meno complessi.

Come da aspettative, infatti, osservando il report ottenuto da Nsight ci accorgiamo di una miglioria evidente nei tempi di esecuzione dei due Kernel rispetto alla versione 1.0.

Table 4: speedup v1.5

Funzione	**v1.0**	**v1.5**	**Speedup**
DFT	$6.101256s$	$2.037019s$	66.61%
Filtro	$0.001747s$	$0.002626s$	-49.97%
IDFT	$6.230328s$	$2.023284s$	67.53%
Totale	$12.333331s$	$4.062929s$	67.07%

Come si può notare dai valori in tabella è stato raggiunto un miglioramento di circa il 67% rispetto alla versione precedente, il tempo di esecuzione migliora di circa 8 secondi.

4.2 Versione 2.0: Aggiunta del Kernel per il Filtro

La seconda fase di ottimizzazione si concentra sull'implementazione di un kernel dedicato alla parallelizzazione del filtraggio delle frequenze. Questo approccio cerca di migliorare i tempi totali evitando il trasferimento di dati tra Device e Host e viceversa.

Listing 8: Kernel filtro.

```
__global__ void filtro(Complesso *X, int N, int fc, int fs) {
    int k = blockIdx.x * blockDim.x + threadIdx.x;

    if (k < N) {
        // Calcolo dell'indice di taglio corrispondente alla frequenza
           fc
        int cutoffIndex = (fc * N) / fs;

        // Applica il filtro passa-basso
        if (k > cutoffIndex && k < N - cutoffIndex) {
            X[k].real = 0;
            X[k].imag = 0;
        }
    }
}
```

Table 5: speedup v2.0

Funzione	**v1.5**	**v2.0**	**Speedup**
DFT	$2.037019s$	$2.029292s$	-
Filtro	$0.002626s$	$0.000407s$	84.50%
IDFT	$2.023284s$	$2.014007s$	-
Totale	$4.062929s$	$4.043706s$	-

L'implementazione del filtro come kernel porta a un miglioramento notevole rispetto alla sua controparte sequenziale, non tanto per la complessità di calcolo ma poiché i tempi delle versioni precedenti prendevano in considerazione anche i trasferimenti di memoria da Host a Device.

Nonostante un miglioramento dell'84.5% sul tempo della singola funzione, questa influisce molto poco sul tempo totale, portando a un tempo totale pressoché identico alla versione precedente.

Dai report di *Nsight Compute*, il kernel del filtro mostra un throughput di memoria elevato, che ci suggerisce che lo streaming multiprocessor sta accedendo in maniera corretta alla memoria. Un compute throughput basso ma dovuto alla ridotta complessità computazionale.

L'ottimizzazione introdotta è probabilmente apprezzabile su file audio di grandi dimensioni in quanto hanno un numero elevato di campioni e quindi vengono evitati lunghi trasferimenti di dati in memoria.

4.3 Versione 3.0 Utilizzo Alignment per indirizzi coalesenti

Nella fase 3.0, ci siamo concentrati sull'ottimizzazione dell'accesso alla memoria tramite l'introduzione dell'allineamento della struttura `Complesso`. In particolare, abbiamo utilizzato la direttiva `__align__(8)` per allineare la struttura a 8 byte. Questa istruzione in fase di compilazione forzerà il compilatore ad allineare gli indirizzi di memoria della struct.

I tempi di esecuzione non migliorano visibilmente, il throughput di memoria per il kernel DFT è leggermente migliorato nella versione 3.0, da 84.07% a 84.10%. Questo suggerisce che l'allineamento della struttura ha migliorato l'efficienza nell'accesso alla memoria.

4.4 Versione 3.5 Riduzione Numero Istruzioni

Nella versione 3.5, il codice è stato ottimizzato introducendo l'uso delle istruzioni `fmaf` e `__sincosf`. Queste funzioni intrinseche permettono di ridurre sostanzialmente il numero di operazioni, la prima svolgendo in una unica istruzione le operazioni di somma e prodotto, la seconda calcolando in una sola istruzione sia il seno che il coseno dell'angolo.

Il kernel risulta quindi come segue:

Listing 9: Kernel DFT

```
__global__ void dftKernel(const float *x, Complesso *X, int N) {
    // Indice globale del thread
    int i = blockIdx.x * blockDim.x + threadIdx.x;

    if (i < N) {
        // Inizializza le parti reale e immaginaria
        float real = 0.0f;
        float imag = 0.0f;

        // Angolo base
        float angleFactor = 2.0f * PI * i / N;
        float angle, cosAngle, sinAngle;

        // Iterazione sui campioni
        for (int j = 0; j < N; j++) {
            angle = angleFactor * j;
            // Calcola il coseno e il seno dell'angolo
            __sincosf(angle, &sinAngle, &cosAngle);
            // Usa fmaf per calcolare le parti reale e immaginaria
            real = fmaf(x[j], cosAngle, real);
            imag = fmaf(-x[j], sinAngle, imag);
        }
        // Salva il risultato nei valori complessi di output
        X[i].real = real;
        X[i].imag = imag;
    }
}
```

Le modifiche apportate nei Kernel DFT e IDFT hanno riscontrato miglioramenti significativamente visibili sia nei tempi di esecuzione che nel report di *Nsight Compute*.

Con un miglioramento di 97.35% l'implementazione di queste istruzioni si è rivelata la chiave per diminuire i tempi di esecuzione.

Table 6: speedup v3.5

Funzione	**v3.0**	**v3.5**	**Speedup**
DFT	2.047402s	0.030865s	98.53%
Filtro	0.000542s	0.000598s	-
IDFT	2.028341s	0.077342s	96.20%
Totale	4.076285s	0.108226s	97.35%

4.5 Versione 4.0 Loop Unrolling dei cicli per la DFT e la IDFT

Abbiamo inoltre tentato di effettuare un loop unrolling dei cicli interni della DFT e della IDFT per cercare di eseguire meno iterazioni, riducendo l'overhead.

Listing 10: Loop Unrolling su Kernel DFT

```
__global__ void dftKernel(const float *x, Complesso *X, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    float angle, cosAngle, sinAngle;
    float angleFactor = 2.0f * PI * i / N;
    float real = 0;
    float imag = 0;
    if (i < N) {
        X[i].real = 0;
        X[i].imag = 0;
        for (int j = 0; j < N; j+=10) {

            angle = angleFactor * j;
            __sincosf(angle, &sinAngle, &cosAngle);
            real = fmaf(x[j], cosAngle, real);
            imag = fmaf(-x[j], sinAngle, imag);

            angle = angleFactor * (j+1);
            __sincosf(angle, &sinAngle, &cosAngle);
            real = fmaf(x[j+1], cosAngle, real);
            imag = fmaf(-x[j+1], sinAngle, imag);

            ...

            angle = angleFactor * (j+9);
            __sincosf(angle, &sinAngle, &cosAngle);
            real = fmaf(x[j+9], cosAngle, real);
            imag = fmaf(-x[j+9], sinAngle, imag);
        }
        // Salva il risultato nei valori complessi di output
        X[i].real = real;
        X[i].imag = imag;
    }
}
```

Tuttavia questo tentativo di miglioramento non ha portato risultati positivi, abbiamo infatti notato un leggero peggioramento delle prestazioni da 0.108s della versione `3.5` a 0.114 s della `4.0`.

4.6 Versione 5.0 Shared Memory

Per cercare di velocizzare ulteriormente gli accessi in memoria, abbiamo provato ad implementare l'utilizzo da parte dei kernel della `shared memory`.

Listing 11: Utilizzo della Shared Memory nel Kernel DFT

```
__global__ void dftKernel(const float *x, Complesso *X, int N) {
    __shared__ float shared_x[BLOCK_SIZE]; // Memoria condivisa per i campioni audio

    int tid = threadIdx.x;
    int i = blockIdx.x * blockDim.x + tid;
    int globalIdx;

    float angle, cosAngle, sinAngle;
    float angleFactor = 2.0f * PI * i / N;
    float real = 0;
    float imag = 0;

    for (int blockOffset = 0; blockOffset < N; blockOffset += blockDim.x) {
        // Copia i dati nella memoria condivisa (shared memory)
        int idx = blockOffset + tid;

        if (idx < N) {
            shared_x[tid] = x[idx];
        } else {
            shared_x[tid] = 0.0f; // Padding per i thread fuori dai limiti
        }
        __syncthreads(); // Sincronizzazione tra i thread del blocco

        // Calcolo della DFT usando i dati nella shared memory
        for (int j = 0; j < blockDim.x; j+=10) {

            globalIdx = blockOffset + j;
            if (globalIdx < N) {
                angle = angleFactor * globalIdx;
                __sincosf(angle, &sinAngle, &cosAngle);
                real = fmaf(shared_x[j], cosAngle, real);
                imag = fmaf(-shared_x[j], sinAngle, imag);
            }

           ---

        }
        __syncthreads(); // Assicura che tutti i thread abbiano completato il ciclo
    }

    // Scrittura del risultato nella memoria globale
    if (i < N) {
        X[i].real = real;
        X[i].imag = imag;
    }
}
```

Table 7: speedup v5.0

Funzione	**v4.9**	**v5.0**	**Speedup**
DFT	$0.030793s$	$0.042509s$	-40%
Filtro	$0.000024s$	$0.000047s$	-95.83%
IDFT	$0.083755s$	$0.041276s$	51.01%
Totale	$0.114572s$	$0.083832s$	27.19%

L'ottimizzazione apportata utilizzando la `shared memory` ha portato a un ulteriore miglioramento passando da $0.114s$ della versione `4.0` a $0.08s$. Dunque con questa versione abbiamo raggiunto il miglior tempo di esecuzione.

Segue dunque il confronto fra i tempi di esecuzione del programma sequenziale e la migliore ottimizzazione realizzata.

Table 8: speedup totale

Funzione	**sequenziale**	**v5.0**	**Speedup**
DFT	$215.199623s$	$0.042509s$	99.98%
Filtro	$0.000159s$	$0.000047s$	70.44%
IDFT	$215.232599s$	$0.041276s$	99.98%
Totale	$430.432381s$	$0.083832s$	99.98%

5 Conclusioni

Al termine di questa fase di test osserviamo che il migliore upgrade ottenuto è stato a seguito dell'impiego delle istruzioni intrinseche, che ci ha portato la migliore performance in tempi di esecuzione. Utilizzando un file di dimensioni 6 volte maggiori rispetto a quello utilizzato nella fase di upgrade osserviamo.