Università degli Studi di Modena e Reggio Emilia

FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

Prova Finale

Studio e implementazione di alcuni algoritmi DSP su CPU e GPU

Candidato:
Gabriele Masini
Matricola 108456

Relatore:
Nicola Bicocchi

Anno Accademico 2018-2019

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

La disciplina denominata *DSP*, dall'inglese *Digital Signal Processing* ovvero "elaborazione digitale di segnali", è stata ed è tuttora parte fondamentale dello sviluppo tecnologico digitale che caratterizza la storia dell'umanità a partire dalla seconda metà del ventesimo secolo. Ciò è dovuto al fatto che gran parte delle grandezze di interesse scentifico e ingegneristico che debbono essere analizzate ed elaborate hanno natura di segnali, i quali necessitano di particolari accortezze e algoritmi per essere elaborati da un dispositivo a capacità di calcolo e memoria limitate, come i calcolatori elettronici.

L'elaborazione digitale dei segnali interessa particolarmente il campo delle telecomunicazioni, dove si lotta per ottenere bitrate sempre maggiori su lunghissime distanze. Un esempio lampante sono le linee telefoniche: dai commutatori analogici, costosi e poco pratici, si è passati ai canali digitali, che a parità di qualità audio offrono un maggior numero di connessioni contemporanee sullo stesso supporto (il doppino telefonico), non necessitano di interruttori analogici e soprattutto hanno un costo sia in termini di costruzione sia di messa in operazione e manutenzione nettamente minore.

Non bisogna però restringere il proprio campo visivo alle sole telecomu-

nicazioni, poiché le tecnologie DSP vengono largamente utilizzate anche in altri campi e risultano fondamentali per applicazioni come video e audio processing, applicazioni mediche, militari, finanziarie e di ricerca. Tutto ciò rende la conoscenza delle basi dell'elaborazione digitale fondamentale nei curricula ingegneristici, motivo per cui è presente nei corsi di laurea inerenti alla materia un esame di "telecomunicazioni", di cui la presente tesi propone come un approfondimento.

1.1 Scopo

La presente tesi si pone come obbiettivo quello di studiare e implementare alcuni dei tanti algoritmi che vengono utilizzati nell'ambito DSP sia dal punto di vista essenzialmente sequenziale del processore, sia una loro possibile parallelizzazione su scheda grafica. Ciò non significa che gli algoritmi riportati siano nella loro forma più efficiente o performante, bensì sono mostrati in modo da far risaltare le differenze implementative che espongono sulle due piattaforme. Inoltre gli algoritmi vengono realizzati con un minimo utilizzo di librerie esterne, in quanto è nell'interesse della tesi e dell'autore l'approfondimento degli algoritmi stessi e lo studio del loro funzionamento interno.

Per portare a termine tale scopo è stata necessaria la stesura di un programma in grado di eseguire gli algoritmi studiati e implementati sia su CPU sia su GPU. Si è deciso di limitarsi all'elaborazione di file audio poiché oltre ad ottenere un riscontro in termini di forma d'onda e spettri di frequenza e fase è possibile anche verificarne il funzionamento in base all'effetto che si ottiene nell'ascolto del risultato. Il programma, con le dovute modifiche, si può estendere anche all'elaborazione di segnali diversi dall'audio.

1.2 Inquadramento

Il presente elaborato espone nel capitolo **??** alcune nozioni matematiche necessarie per la corretta comprensione e spiegazione delle implementazioni degli algoritmi e inoltre viene spiegara brevemente come è strutturata l'architettura di una GPGPU e come vi si interfaccia a livello di programmazione attraverso le API di CUDA.

Nel capitolo **??** vengono presentate e spiegate le parti più interessanti degli algoritmi studiati, i quali sono disponibili in forma intera nel repository GitHub della tesi [4].

Capitolo 2

Nozioni prerequisite

Prima di cimentarsi in implementazioni di algoritmi è assolutamente necessario comprenderne la loro natura matematica. A tale scopo vengono presentate brevemente in questa sezione alcune nozioni fondamentali per l'elaborazione di segnali digitali.

2.1 Concetti fondamentali in DSP

2.1.1 Segnali, sistemi lineari

Nel corso di questa tesi verranno utilizzati spesso i termini *segnale* e *sistema* per cui è necessario definirli. Un segnale è "una descrizione di come un parametro varia rispetto ad un altro parametro" [6, pp. 87-88]. Esempi di segnali sono: pressione dell'aria nel tempo (audio), coppia motrice rispetto al numero di giri di un motore etc.

Per studiare i segnali e modificarne l'andamento si fa riferimento ai sistemi. Un sistema è "un qualsiasi processo che produce un segnale in uscita in risposta ad un segnale in ingresso" [6, pp. 87-88]. In particolare si studiano i sistemi lineari, ovvero sistemi che seguono le proprietà necessarie per la linearità

algebrica (omogeneità e additività). La proprietà di linearità è fondamentale nel DSP perché permette di scomporre il problema in sottoproblemi più piccoli e facilmente risolvibili per poi ricombinarli assieme e ottenere il risultato del problema di partenza.

Il comportamento di un sistema è descritto dalla risposta all'impulso o dalla sua funzione di trasferimento. La prima è il risultato del sistema quando all'ingresso viene applicata una funzione impulsiva (ovvero la funzione "delta di Dirac" nel caso continuo; nel caso discreto si utilizza una funzione "delta di Kronecker"). La funzione di trasferimento invece è il rapporto tra lo spettro di un generico segnale di uscita e il rispettivo spettro del segnale di ingresso. È dimostrabile che la funzione di trasferimento è la trasformata di Fourier della risposta all'impulso [1, pp. 3.29-3.31].

2.1.2 Convoluzione

Il primo strumento fondamentale per comprendere gli algoritmi DSP è l'operazione di convoluzione tra segnali. Essa è definita nel seguente modo [1, p. 2.10]:

$$x(t) * y(t) = \int_{-\infty}^{+\infty} x(\tau)y(t-\tau)d\tau \tag{2.1}$$

Dove x e y sono i due segnali da convolvere.

La convoluzione per segnali discreti è definita da una sommatoria [6, p. 120]:

$$y[k] = \sum_{j=0}^{M-1} h[j]x[i-j] \tag{2.2}$$

Dove x e h sono i segnali da convolvere, y il risultato della loro convoluzione. M è il numero di punti del segnale h. È fondamentale precisare che il segnale risultante dalla convoluzione discreta di x e h contiene $N + M - 1$ punti, dove N indica il numero di punti del segnale x.

La convoluzione è importante nella elaborazione dei segnali perché costituisce il primo strumento con cui si può ottenere l'uscita di un sistema a partire dal segnale temporale in ingresso e la risposta impulsiva del sistema stesso; tale segnale in uscita è infatti la convoluzione tra il segnale in ingresso e la risposta impulsiva del sistema.

2.1.3 La trasformata di fourier

Strumento essenziale per gli algoritmi DSP, la trasformata di Fourier scompone un segnale nel dominio del tempo nelle sue componenti nel dominio delle frequenze. Esistono trasformate diverse per diversi tipi di rappresentazione dei segnali nel tempo e una classificazione accettata a livello matematico e ingegneristico [6, p. 144] è la seguente:

Tipo segnale	Trasformata utilizzata
Aperiodico tempo-continuo	Trasformata di fourier continua (CFT)
Periodico tempo-continuo	Serie di fourier
Aperiodico tempo-discreto	Trasformata di fourier tempo discreta (DTFT)
Periodico tempo-discreto	Trasformata di fourier discreta (DFT)

Tabella 2.1: Famiglia di trasformate di Fourier

Le formule (trasformazione e antitrasformazione) della CFT sono definite nel modo seguente [1, p. 2.7]:

$$F(\omega) = \int_{-\infty}^{+\infty} f(t)e^{-j\omega t}dt \tag{2.3}$$

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{+\infty} F(\omega) e^{+j\omega t} d\omega \tag{2.4}$$

Dove $f(t)$ identifica il segnale nel dominio dei tempi, $F(\omega)$ la sua trasformata e j è l'unità immaginaria tale che $j^2 = -1$.

La serie di fourier invece è definita come [1, p. 2.4]:

$$f(t) = \sum_{n=-\infty}^{+\infty} c_n e^{jn\omega_0 t} \quad , \quad \omega_0 = \frac{2\pi}{T} \tag{2.5}$$

con

$$c_n = \frac{1}{T} \int_{-\frac{T}{2}}^{+\frac{T}{2}} f(t) e^{-jn\omega_0 t} dt \tag{2.6}$$

Dove T è il periodo della funzione e c_n è il coefficiente n-esimo della serie. Gli spettri di ampiezza e di fase si ricavano dai valori di A_n e φ_n definiti nel seguente modo [1, p. 2.5]:

$$c_0 = A_0 \tag{2.7}$$

$$2c_n = A_n e^{-j\varphi_n} \quad , \quad n \geq 1 \tag{2.8}$$

Essi generano degli spettri a righe in corrispondenza delle pulsazioni multiple di ω_0 [1, p. 2.5].

Per quanto la CFT e la serie di Fourier siano degli strumenti matematici indispensabili per comprendere l'analisi dei segnali, esse trovano relativamente poca applicazione pratica nella elaborazione dei segnali, poiché solitamente i segnali che devono essere processati da un calcolatore hanno natura tempo-discreta (ovvero sono stati campionati) e necessitano, quindi, dell'utilizzo della DTFT o della DFT. Come vedremo in seguito, però, si utilizza sempre la DFT, in quanto non è possibile modellare in un calcolatore il concetto di "infinito" fondamentale per la definizione e il calcolo della DTFT [6, pp. 144-145].

2.1.4 La trasformata di Fourier Discreta (DFT)

Come presentato nella sezione precedente, la DFT trasforma un segnale tempo-discreto periodico nelle sue componenti nel dominio delle frequenze;

essa per un segnale di N punti è definita nel seguente modo [6, p. 570]:

$$X[k] = \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N} \tag{2.9}$$

Si presti attenzione al fatto che la trasformata è definita su N punti da 0 a $N-1$ ed essi rappresentano le frequenze positive per $0 \leq n \leq N/2$ e frequenze negative per $N/2 \leq n \leq N-1$, poiché lo spettro di frequenze di un segnale discreto è periodico. Questo comporta che la trasformata di un segnale reale (con parte immaginaria nulla per ogni campione) abbia la parte reale dello spettro in simmetria pari rispetto a $N/2$ e parte immaginaria in simmetria dispari rispetto a $N/2$ [6, p. 570].

L'operazione di antitrasformazione è definita nel modo seguente [6, p. 572]:

$$x[n] = \sum_{k=0}^{N-1} X[k] e^{j2\pi kn/N} \tag{2.10}$$

Nonostante la DFT sia un ottimo punto di partenza per poter calcolare la trasformata di un segnale discreto con l'utilizzo del calcolatore elettronico, essa è limitata dal fatto che è onerosa in termini di tempi di calcolo; come vedremo in seguito nel capitolo **??** al suo posto si utilizza l'algoritmo di Cooley-Tukey [3], denominato "FFT" ovvero *fast Fourier transform*.

2.2 GPU e CUDA

Lo studio delle implementazioni su GP-GPU viene effettuato tramite l'utilizzo della tecnologia CUDA proprietaria di Nvidia. Essa espone una interfaccia di programmazione per l'utilizzo del calcolo parallelo delle schede grafiche di proprietà di Nvidia in linguaggio di programmazione C++.

È vantaggioso studiare implementazioni su schede video in quanto esse permettono di effettuare un elevato numero di operazioni parallele su un

certo insieme di dati; infatti il termine scheda "video" deriva dal fatto che in principio erano utilizzate soprattutto per l'elaborazione e rendering di video, dove sono necessarie elaborazioni simili per tanti dati quanti sono i pixel dell'immagine. In tempi relativamente recenti, però, si è smesso di parlare di GPU come schede video, ma si parla di GP-GPU acronimo di *General Purpose Graphical Processing Unit*, ovvero "Schede video per scopi generici", poiché tali dispositivi hanno trovato largo utilizzo per quanto riguarda elaborazioni ad alte prestazioni, soprattutto nell'ambito di ricerca di intelligenze artificiali come le reti neurali.

2.2.1 Programmazione in CUDA

Facendo riferimento alla guida per la programmazione in CUDA di Nvidia[5] il principale elemento di calcolo che viene somministrato alla GPU è il *kernel*, il quale è una funzione che viene eseguita un numero definito di volte in parallelo sulla GPU quando essa viene invocata. Un kernel viene dichiarato utilizzando il modificatore `__global__`. Inoltre è importante tenere a mente che al kernel possono essere passati come parametri solo tipi predefiniti, puntatori o struct di tipi predefiniti o puntatori, a patto che i puntatori facciano riferimento ad un'area di memoria sulla GPU, non nella RAM. Questo comporta che i dati da elaborare vadano prima copiati dalla memoria centrale alla memoria della GPU, elaborati e poi ricopiati nella RAM. Le API di CUDA mettono a disposizione le funzioni necessarie per operare questi spostamenti di memoria.

I kernel possono essere raggruppati in *stream* (flussi). I kernel facenti parte dello stesso stream vengono eseguiti in successione, ma stream diversi possono essere eseguiti in parallelo. Non c'è modo di sapere in quale istante un kernel entrerà in esecuzione e nemmeno quando esso finirà, per tale motivo risulta necessario porre attenzione alla sincronizzazione dei da-

ti su cui operano i diversi kernel. Anche in questo caso le API fornite da Nvidia mettono a disposizione varie funzioni di sincronizzazione sia all'interno dei kernel di uno stesso blocco (tramite `__syncthreads()`), sia per stream (con `cudaStreamSynchronize(<stream>)`) sia per la GPU intera (con `cudaDeviceSynchronize()`).

Le unità di elaborazione all'interno della GPU sono divise in blocchi ciascuno dei quali è in grado di eseguire un certo numero di thread. Compito del programmatore è distribuire queste risorse ai kernel necessari. I blocchi e i thread al loro interno sono organizzati secondo una matrice 3-dimensionale, e i loro indici sono accessibili all'interno del kernel usando le variabili `threadIdx` e `blockIdx`.

Un esempio mostrato nella guida alla programmazione di Nvidia [5] è il seguente, il quale esegue la somma di elementi di due vettori:

```
1   // Kernel definition
2   __global__ void VecAdd(float* A, float* B, float* C)
3   {
4       int i = threadIdx.x;
5       C[i] = A[i] + B[i];
6   }
7
8   int main()
9   {
10      ...
11      // Kernel invocation with N threads
12      VecAdd<<<1, N>>>(A, B, C);
13      ...
14  }
```

Come è possibile notare, al kernel vengono passati tre parametri, ovvero i vettori coi dati da elaborare `A` e `B` e il vettore risultante dalla somma elemento per elemento dei due precedenti, `C`. Il kernel viene invocato su un blocco di N thread. All'interno di ogni singolo thread è possibile ottenere l'indice del thread utilizzando `threadIdx.x`; in questo caso essendo la definizione del numero di thread monodimensionale (N), l'indice del thread corrente si ottiene

solo dalla prima componente della variabile tre-dimensionale `threadIdx`.

Capitolo 3

Implementazione

Per lo studio di quanto descritto fino ad ora si è costruito un programma che possa caricare un file wav in buffer di dimensione arbitraria in memoria per poterne elaborare il contenuto; dopodiché è possibile specificare una catena di operazioni da effettuare sul segnale e un metodo di salvataggio del risultato (.wav o .csv). La catena di operazioni e il salvataggio vengono specificati in un file di testo. Per la lettura e scrittura su file .wav si utilizza la libreria "libsndfile" scritta da Erik de Castro Lopo[2].

3.1 Strutture dati utilizzate

Per la rappresentazione dei dati si utilizza una struttura nominata `SignalBuffer_t` definita nel seguente modo:

```
1 struct SignalBuffer_t
2 {
3         cuComplex* samples;
4         size_t channels;
5         size_t* channel_size;
6         size_t max_size;
7 };
```

Essa continene un array di campioni, il numero di canali, la lunghezza dei buffer di ogni canale e la lunghezza massima disponibile dell'array. A prima vista può sembrare strana ma è necessaria questa configurazione per facilitare le operazioni di input/output dei file wav e soprattutto le operazioni di trasporto della memoria dalla e alla GPU.

`cuComplex` è un tipo importato dalla libreria di cuda il quale rappresenta un numero complesso. Esso può essere utilizzato, con le apposite operazioni, sia dalla CPU sia dalla GPU.

Vengono definite anche operazioni su questa struttura, di cui viene riportata l'implementazione delle due principali. Tutte le funzioni sono dichiarate con i modificatori `__host__ __device__ inline` per segnalare che sono funzioni che possono essere usate sia dalla cpu, sia dalla gpu; inoltre sono inline per snellire la loro chiamata in quanto sono usate spesso.

```
1     size_t get_channels(SignalBuffer_t buffer)
2     size_t get_channel_buffer_size(SignalBuffer_t buffer,
         size_t channel);
3     size_t get_max_buffer_size(SignalBuffer_t buffer);
4     size_t get_max_channel_buffer_size(SignalBuffer_t buffer)
         ;
5     size_t get_max_possible_channel_buffer_size(
         SignalBuffer_t buffer, size_t channel);
6     int set_channel_buffer_size(SignalBuffer_t buffer, size_t
          channel, size_t size);
7     size_t get_signal_buffer_channel_sample_index(
         SignalBuffer_t buffer, size_t channel, size_t index)
8     {
9         return index * buffer.channels + channel;
10    }
11
12    cuComplex get_signal_buffer_sample(SignalBuffer_t buffer,
          size_t channel, size_t index)
13    {
14        size_t buffer_size = get_channel_buffer_size(buffer,
             channel);
15        if (index >= buffer_size)
16            return make_cuComplex(0, 0);
17        return buffer.samples[
             get_signal_buffer_channel_sample_index(buffer,
             channel, index)];
```

```
18      }
19
20      int set_signal_buffer_sample(SignalBuffer_t buffer,
            size_t channel, size_t index, cuComplex value)
21      {
22          size_t current_buffer_size = get_channel_buffer_size(
                buffer, channel);
23          if (index >= current_buffer_size)
24          {
25              size_t new_current_buffer_size = index + 1;
26              if (!set_channel_buffer_size(buffer, channel,
                    new_current_buffer_size)) {
27                  // buffer overflow
28                  return 0;
29              }
30          }
31          buffer.samples[get_signal_buffer_channel_sample_index
                (buffer, channel, index)] = value;
32
33          return 1;
34      }
```

3.2 DFT

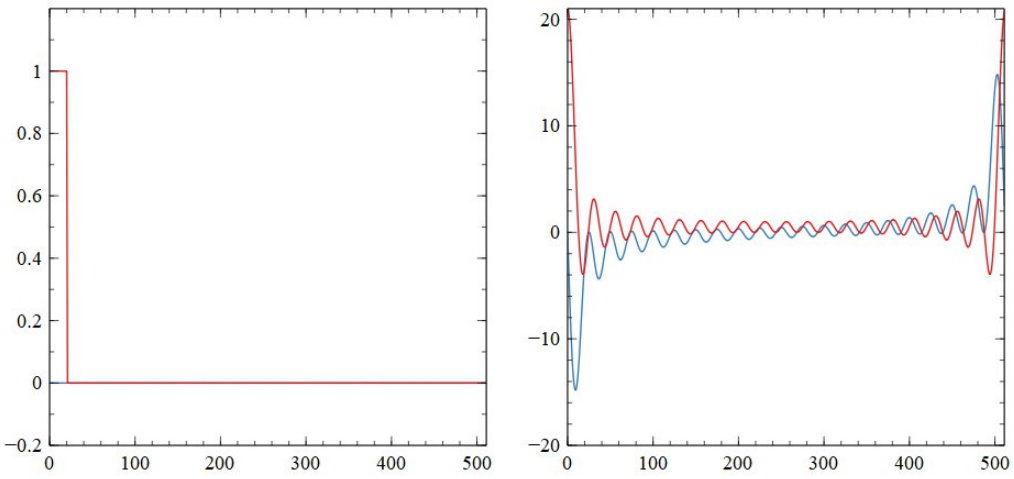
3.2.1 CPU

L'operazione di trasformata di Fourier discreta è implementata sulla CPU nel seguente modo:

```
1  void dft_wsio(SignalBuffer_t* bufferIn, SignalBuffer_t*
       bufferOut, size_t channel, size_t size)
2  {
3          cuComplex* tmp = new cuComplex[size];
4          cuComplex sample, s;
5          for (size_t k = 0; k < size; k++)
6          {
7                  tmp[k] = make_cuFloatComplex(0,0);
8                  for (size_t i = 0; i < size; i++)
9                  {
10                         s = cuComplex_exp(-2 * M_PI * k * i /
                               size);
11                         sample = get_signal_buffer_sample(*
                               bufferIn, channel, i);
```

```
12                          tmp[k] = cuCaddf(tmp[k], cuCmulf(
                                sample, s));
13                  }
14          }
15          for (size_t k = 0; k < size; k++)
16          {
17                  set_signal_buffer_sample(*bufferOut, channel,
                        k, tmp[k]);
18          }
19          delete[] tmp;
20  }
```

`cuComplex_exp(float x)` è una funzione che restituisce il numero complesso e^{jx}. L'algoritmo prende in input un buffer di cui effettuare la trasformata, un buffer in cui inserire il risultato della trasformazione, il canale dei buffer su cui operare e la dimensione in punti della trasformata. Come esposto in precedenza la funzione `get_signal_buffer_sample` restituisce il numero complesso 0 nel caso il valore dell'indice sia fuori range. Questo permette di implementare facilmente la dft di un buffer con un pad di zeri alla sua destra.



(a) Impulso (b) Dft

Figura 3.1: DFT di un impulso. In rosso è segnata la parte reale e in blu la parte immaginaria

Inserendo nel programma l'impulso di 512 campioni mostrato in figura 3.1a si ottiene la trasformata in figura 3.1b. Come è facile notare, il segnale di partenza ha solo componente reale, quindi la sua trasformata è simmetrica rispetto a $N/2$ in modo pari per la parte reale e in modo dispari per la parte immaginaria.

3.2.2 GPU

L'operazione di trasformata di Fourier discreta è implementata sulla GPU utilizzando i seguenti kernel:

```
1  __global__ void cudadft_kernel_dft(SignalBuffer_t
       device_buffer, SignalBuffer_t tmp, size_t channel)
2  {
3      int k = blockIdx.x * blockDim.x + threadIdx.x;
4      size_t size = get_channel_buffer_size(device_buffer,
           channel);
5      cuComplex temp = make_cuComplex(0, 0);
6      cuComplex sample, s;
7
8      for (int i = 0; i < size; i++)
9      {
10         sample = get_signal_buffer_sample(device_buffer,
               channel, i);
11
12         s = cuComplex_exp(-2.0f * PI * k * i / size);
13
14         temp = cuCaddf(temp, cuCmulf(sample, s));
15     }
16
17     set_signal_buffer_sample(tmp, channel, k, temp);
18 }
19
20 __global__ void cudadft_kernel_copy(SignalBuffer_t
       device_buffer, SignalBuffer_t tmp, size_t channel)
21 {
22     int k = blockIdx.x * blockDim.x + threadIdx.x;
23     cuComplex sample = get_signal_buffer_sample(tmp, channel,
           k);
24     set_signal_buffer_sample(device_buffer, channel, k,
           sample);
25 }
```

Il primo kernel si occupa di calcolare la DFT del segnale, mentre il secondo è un kernel che viene eseguito dopo che tutti i thread del primo sono completati e si occupa di ricopiare il risultato della dft da tmp al buffer originario. È necessaria questa divisione di compiti per evitare che vengano scritti dei valori nel buffer originario prima che tutti i thread abbiano finito di accedervi. Inserendo l'impulso di figura 3.1a nel programma con dft in CUDA si ottiene lo stesso risultato di 3.1b.

3.3 Fast Fourier Transform

L'operazione di DFT è onerosa in termini di calcolo in quanto ha complessita $O(N^2)$, per cui si utilizza spesso la Fast Fourier Transform al suo posto (FFT). Uno degli algoritmi più popolari per il calcolo della FFT è quello ideato da Cooley e Tukey nel 1965. Esso fa uso della decomposizione interlacciata e delle somme a "farfalla".

[da espandere con più info]

3.3.1 CPU

La FFT è implementata sulla CPU nel seguente modo:

```
1    void fft_wsio(SignalBuffer_t* bufferIn, SignalBuffer_t*
        bufferOut, size_t channel, size_t size_in)
2    {
3        cuComplex w, wm;
4        size_t levels;
5        size_t index_a, index_b;
6        size_t size = (size_t)pow(2, ceil(log2(size_in)));;
7        levels = (size_t)log2(size);
8        bit_reversal_sort_wsio(bufferIn, bufferOut, channel,
            size);
9        for (size_t level = 0; level < levels; level++)
10       {
11           size_t butterflies_per_dft = (size_t)pow(2, level
                );
```

```
12              size_t dfts = size / (butterflies_per_dft * 2);
13              wm = cuComplex_exp(-(M_PI / butterflies_per_dft))
                    ;
14              w = make_cuComplex(1,0);
15              for (size_t butterfly = 0; butterfly <
                    butterflies_per_dft; butterfly++)
16              {
17                  for (size_t dft = 0; dft < dfts; dft++)
18                  {
19                      index_a = butterfly + dft * (
                            butterflies_per_dft * 2);
20                      index_b = index_a + butterflies_per_dft;
21                      cuComplex a = get_signal_buffer_sample(*
                            bufferOut, channel, index_a);
22                      cuComplex b = get_signal_buffer_sample(*
                            bufferOut, channel, index_b);
23                      butterfly_calculation(&a, &b, w);
24                      set_signal_buffer_sample(*bufferOut,
                            channel, index_a, a);
25                      set_signal_buffer_sample(*bufferOut,
                            channel, index_b, b);
26                  }
27                  w = cuCmulf(w, wm);
28              }
29          }
30      }
```

Le funzioni `bit_reversal_sort_wsio` e `butterfly_calculation` sono consultabili in appendice.

[Più info su come funziona].

3.3.2 GPU

[da fare]

3.4 Convoluzione

[info]

3.4.1 CPU

La convoluzione è implementata sulla CPU nel seguente modo:

```
1     size_t buffer_size = get_channel_buffer_size(*buffer,
          channel);
2     size_t signal_size = get_channel_buffer_size(this->signal
          , SIGNAL_CHANNEL);
3     size_t remaining_samples = this->samples_remaining[
          channel];
4
5     size_t temp_index = this->temp_indexes[channel];
6     if ((buffer_size == 0 || signal_size == 0) &&
          remaining_samples > 0)
7     {
8         size_t count = 0;
9         cuComplex sample = get_signal_buffer_sample(this->
              temp, channel, temp_index);
10        while (remaining_samples > 0 &&
11            set_signal_buffer_sample(*buffer, channel, count,
                  sample)) {
12
13            count++;
14            temp_index = bounded_index(this->temp, channel,
                  temp_index + 1);
15            sample = get_signal_buffer_sample(this->temp,
                  channel, temp_index);
16            remaining_samples--;
17        }
18        temp_indexes[channel] = temp_index;
19        samples_remaining[channel] = remaining_samples;
20        continue;
21    }
22    size_t total = buffer_size + signal_size - 1;
23    for (size_t i = 0; i < buffer_size; i++)
24    {
25        cuComplex in_sample = get_signal_buffer_sample(*
              buffer, channel, i);
26        for (size_t j = 0; j < signal_size; j++)
27        {
28            cuComplex signal_sample =
                  get_signal_buffer_sample(this->signal,
                  SIGNAL_CHANNEL, j);
29            size_t index = bounded_index(this->temp, channel,
                  temp_index + i + j);
30            cuComplex out_sample = get_signal_buffer_sample(
                  this->temp, channel, index);
31            cuComplex result = cuCaddf(out_sample, cuCmulf(
                  in_sample, signal_sample));
```

```
32              set_signal_buffer_sample(this->temp, channel,
                    index, result);
33          }
34      }
35      for (size_t i = 0; i < buffer_size; i++)
36      {
37          size_t index = bounded_index(this->temp, channel,
                temp_index + i);
38          cuComplex out_sample = get_signal_buffer_sample(this
                ->temp, channel, index);
39          set_signal_buffer_sample(*buffer, channel, i,
                out_sample);
40          set_signal_buffer_sample(this->temp, channel, index,
                make_cuComplex(0, 0));
41      }
42      this->temp_indexes[channel] = bounded_index(this->temp,
            channel, temp_index + buffer_size);
43      this->samples_remaining[channel] = signal_size - 1;
```

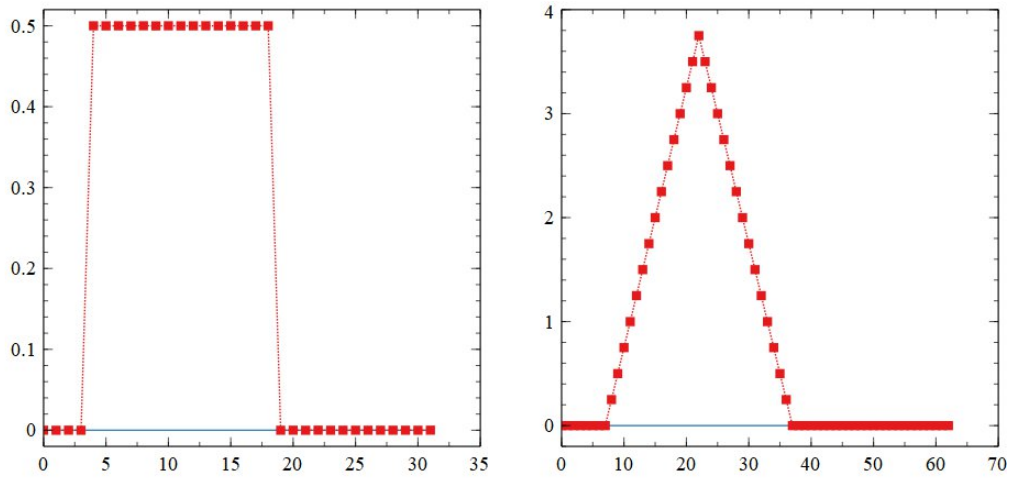
3.4.2 GPU

Sulla GPU si utilizzano i seguenti kernel.

```
1  __global__ void cudaconvolver_kernel_output(SignalBuffer_t
       device_buffer, SignalBuffer_t signal, SignalBuffer_t tmp,
       size_t channel, size_t temp_index)
2  {
3          int k = blockIdx.x * blockDim.x + threadIdx.x;
4          size_t out_size = get_channel_buffer_size(tmp,
               channel);
5          if (k >= out_size)
6                  return;
7          size_t signal_size = get_channel_buffer_size(signal,
               SIGNAL_CHANNEL);
8          cuComplex temp = make_cuComplex(0, 0);
9          cuComplex signal_sample, input_sample;
10         size_t index = cuda_bounded_index(tmp, channel,
               temp_index + k);
11         cuComplex temp_sample = get_signal_buffer_sample(tmp,
               channel, index);
12         for (int i = 0; i < signal_size; i++)
13         {
14                 signal_sample = get_signal_buffer_sample(
                       signal, SIGNAL_CHANNEL, i);
15                 if (i > k)
16                         input_sample = make_cuComplex(0,0);
17                 else
```

```
18                          input_sample =
                               get_signal_buffer_sample(
                               device_buffer, channel, k-i);
19                  temp_sample = cuCaddf(temp_sample, cuCmulf(
                       signal_sample, input_sample));
20          }
21          set_signal_buffer_sample(tmp, channel, index,
               temp_sample);
22  }
23  __global__ void cudaconvolver_kernel_copy(SignalBuffer_t
       device_buffer, SignalBuffer_t tmp, size_t channel, size_t
       temp_index)
24  {
25          int k = blockIdx.x * blockDim.x + threadIdx.x;
26
27          size_t tmp_size = get_channel_buffer_size(tmp,
               channel);
28          size_t out_size = get_channel_buffer_size(
               device_buffer, channel);
29          if (k >= tmp_size)
30                  return;
31
32          size_t index = cuda_bounded_index(tmp, channel,
               temp_index + k);
33          cuComplex sample = get_signal_buffer_sample(tmp,
               channel, index);
34          set_signal_buffer_sample(device_buffer, channel, k,
               sample);
35          if (k < out_size)
36                  set_signal_buffer_sample(tmp, channel, index,
                       make_cuComplex(0,0));
37  }
```

In figura 3.2 si può apprezzare una convoluzione di un impulso di 32 campioni con se stesso ottenuto dagli algoritmi presentati.



(a) Impulso (b) Convoluzione

Figura 3.2: Convoluzione di un impulso rettangolare con sé stesso. In rosso è segnata la parte reale e in blu la parte immaginaria

Capitolo 4

Conclusioni

Il mondo dei DSP è vastissimo ed entrare a contatto con una parte di esso è stata una esperienza alquanto entusiasmante. Inoltre la passione che l'autore nutre per la musica ha contribuito, in parte, a scegliere questo argomento per la propria tesi di laurea. Implementare gli algoritmi proposti è stata una sfida, soprattutto perché era desiderio dell'autore implementare gli algoritmi di propria iniziativa per poterne capire l'intricato funzionamento interno e quindi riproporne una possibile implementazione in una ottica diversa: il parallelismo sulla GPU.

[altre conclusioni, benchmark di tempo etc...]

Appendice A

Funzioni aggiuntive

```
1     void bit_reversal_sort_wsio(SignalBuffer_t* bufferIn,
          SignalBuffer_t* bufferOut, size_t channel, size_t size
          )
2     {
3         cuComplex sample, temporary;
4         size_t buffer_size = get_channel_buffer_size(*
              bufferIn, channel);
5         size_t j, k, halfSize;
6
7         halfSize = size / 2;
8         j = halfSize;
9
10        sample = get_signal_buffer_sample(*bufferIn, channel,
              0);
11        set_signal_buffer_sample(*bufferOut, channel, 0,
              sample);
12
13        sample = get_signal_buffer_sample(*bufferIn, channel,
              size-1);
14        set_signal_buffer_sample(*bufferOut, channel, size -
              1, sample);
15
16        for (size_t i = 1; i < size - 2; i++)
17        {
18            if (i < j)
19            {
20                temporary = get_signal_buffer_sample(*
                      bufferIn, channel, j);
21                sample = get_signal_buffer_sample(*bufferIn,
                      channel, i);
22
```

```
23                  if (i >= buffer_size)
24                      sample = make_cuComplex(0,0);
25                  if (j >= buffer_size)
26                      temporary = make_cuComplex(0, 0);
27
28                  set_signal_buffer_sample(*bufferOut, channel,
                        j, sample);
29                  set_signal_buffer_sample(*bufferOut, channel,
                        i, temporary);
30              }
31              else if (i == j) {
32                  sample = get_signal_buffer_sample(*bufferIn,
                       channel, i);
33                  if (i >= buffer_size)
34                      sample = make_cuComplex(0, 0);
35                  set_signal_buffer_sample(*bufferOut, channel,
                        i, sample);
36              }
37              k = halfSize;
38              while (k <= j)
39              {
40                  j = j - k;
41                  k = k / 2;
42              }
43              j = j + k;
44          }
45      }
```

```
1       void butterfly_calculation(cuComplex* a, cuComplex* b,
           cuComplex w)
2       {
3           cuComplex aa = *a;
4           cuComplex bw = cuCmulf(*b, w);
5
6           *a = cuCaddf(aa, bw);
7           *b = cuCsubf(aa, bw);
8       }
```

Bibliografia

[1] Leonardo Calandrino e Gianni Immovilli. *Schemi delle lezioni di comunicazioni elettriche*. Pitagora Editrice Bologna, 1991.

[2] Erik de Castro Lopo. *libsndfile*. URL: `http://www.mega-nerd.com/libsndfile/` (visitato il 12/03/2020).

[3] James W. Cooley e John W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of Computation* (1965).

[4] Gabriele Masini. *Sorgenti tesi*. URL: `https://github.com/gmasini97/thesis-src` (visitato il 12/03/2020).

[5] Nvidia. *CUDA C++ Programming Guide*. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (visitato il 12/03/2020).

[6] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997. ISBN: 978-0966017632.