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

1 Introduzione **6**

1.1 Scopo 7

1.2 Inquadramento 8

2 Nozioni prerequisite **9**

2.1 Concetti fondamentali in DSP 9

2.1.1 Segnali, sistemi lineari 9

2.1.2 Convoluzione 10

2.1.3 La trasformata di fourier 11

2.1.4 La trasformata di Fourier Discreta (DFT) 12

2.2 GPU e CUDA 13

2.2.1 Programmazione in CUDA 14

3 Implementazione **17**

3.1 Breve descrizione del programma 17

3.2 Strutture dati utilizzate 18

3.3 Convoluzione 19

3.3.1 CPU 20

3.3.2 GPU 21

3.4 DFT 24

3.4.1 CPU 24
3.4.2 GPU 26
3.5 Fast Fourier Transform 26
3.5.1 CPU 27
3.5.2 GPU 27

4 Conclusioni **28**

A Funzioni aggiuntive **29**

Elenco delle figure

3.1 Disposizione dei campioni nella struttura `SignalBuffer_t` supponendo m canali. 18
3.2 Convoluzione di un impulso rettangolare con sé stesso. In rosso è segnata la parte reale e in blu la parte immaginaria 21
3.3 Tempo di convoluzione, CPU e GPU a confronto 23
3.4 DFT di un impulso di 512 campioni. In rosso è segnata la parte reale e in blu la parte immaginaria 25

Elenco delle tabelle

2.1 Famiglia di trasformate di Fourier 11

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

3.1 Breve descrizione del programma

Per lo studio di quanto descritto fino ad ora si è costruito un programma in grado di 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].

L'invocazione del programma avviene tramite riga di comando ed è necessario utilizzare un programma esterno per visualizzare il contenuto dei file di output. I grafici dei risultati che verranno mostrati in questa tesi sono stati generati utilizzando un file `.csv` in uscita dal programma il quale viene poi visualizzato tramite il programma Veusz [**veusz**].

3.2 Strutture dati utilizzate

Per la rappresentazione dei dati dei campioni in ingressi 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. I campioni sono organizzati nell'array di numeri complessi secondo lo schema descritto in figura 3.1, ovvero lo stesso modo in cui vengono restituiti i dati dalle funzioni di lettura di `libsdnfile`.

indice array	0	1	...	m-1	m	m+1	...
numero canale	0	1	...	m-1	0	1	...
numero campione canale	0	0	...	0	1	1	...

Figura 3.1: Disposizione dei campioni nella struttura `SignalBuffer_t` supponendo m canali.

`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, le quali garantiscono la corretta gestione dei campioni all'interno della stessa. Le funzioni utilizzate sono dichiarate con i modificatori `__host__ __device__` i quali sono necessari al compilatore di CUDA per segnalare che tali funzioni possono essere sia eseguite sulla CPU (host), sia sulla scheda grafica (device).

3.3 Convoluzione

Il primo algoritmo che è stato studiato e implementato è la convoluzione tra segnali la quale è indispensabile perché è il primo strumento con cui si può calcolare la risposta di un sistema ad un determinato segnale se si conosce la risposta impulsiva del sistema stesso.

La definizione di convoluzione tra segnali discreti è descritta dall'equazione **??**. Bisogna porre particolare attenzione al fatto che la lunghezza del risultato della convoluzione è uguale alla somma delle lunghezze dei segnali in ingresso meno uno; questo comporta che bisogna far attenzione a gestire il fenomeno detto "overlap" tra buffer risultanti consecutivi; questo fenomeno, grazie alla linearità del sistema, si riduce solamente alla somma della "coda" del buffer in uscita precedente con i primi campioni del buffer in uscita seguente.

Visto che la convoluzione è una operazione che necessita di due segnali in entrata, si è supposto che il secondo segnale sia di un solo canale e non sia diviso in vari buffer, ma sia tutto contenuto in uno solo, mentre il primo segnale può essere diviso in più buffer. Questa decisione è giustificata dal fatto che spesso la convoluzione viene utilizzata tra un segnale molto lungo e uno di gran lunga più corto, poiché, come già detto, è una operazione onerosa in termini di calcolo. In particolare è spesso utilizzata per operazioni di filtraggio e in questo caso il secondo segnale prende il nome di "kernel del filtro" [6, p. 108]. Nelle spiegazioni seguenti si utilizzerà la parola "kernel" per indicare il secondo segnale della convoluzione; bisogna prestare attenzione a non confondere il "kernel" del filtro con le funzioni "kernel" di CUDA.

3.3.1 CPU

La convoluzione sulla CPU è implementata utilizzando due cicli `for`: uno che scorre i campioni del buffer in entrata di un determinato canale e l'altro invece che scorre i campioni del buffer contenente il secondo segnale. I due campioni ottenuti vengono quindi moltiplicati assieme e vengono poi aggiunti al valore di un buffer temporaneo. La presenza del buffer temporaneo è necessaria per la corretta gestione del fenomeno dell'"overlap"; il buffer temporaneo alla fine della convoluzione conterrà i campioni in uscita dalla convoluzione e la "coda" da sommare alla convoluzione successiva.

Una versione semplificata alle componenti fondamentali del codice che è stato utilizzato è il seguente:

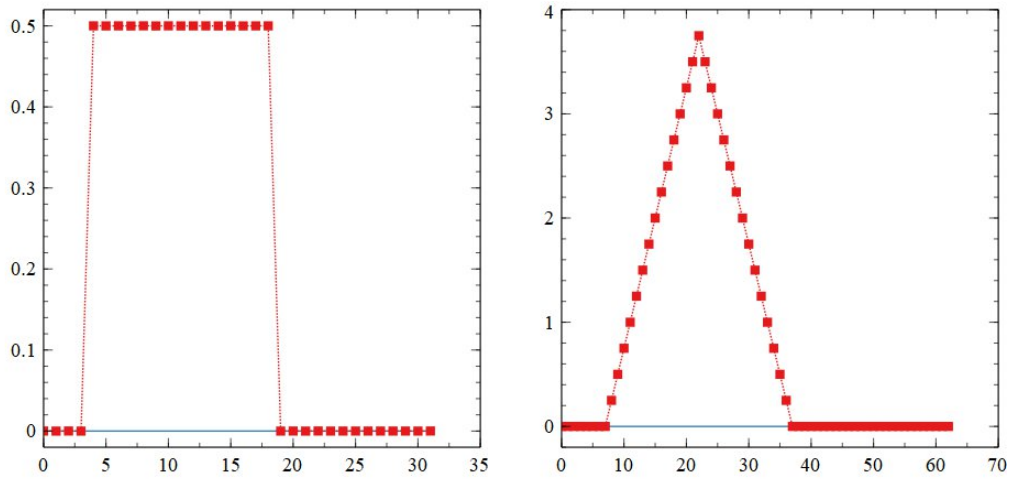
```
1  ...
2  for (size_t i = 0; i < input_size; i++)
3  {
4      cuComplex in_sample = get_sample(input, channel, i);
5      for (size_t j = 0; j < kernel_size; j++)
6      {
7          size_t index = i+j;
8          cuComplex kernel_sample = get_sample(kernel,
               SIGNAL_CHANNEL, j);
9          cuComplex out_sample = get_sample(temp, channel,
               index);
10         cuComplex result = cuCaddf(out_sample, cuCmulf(
               in_sample, kernel_sample));
11         set_sample(temp, channel, index, result);
12     }
13 }
14 ...
```

Come si può osservare sono presenti i due cicli `for` e le operazioni per effettuare l'accumulazione nel buffer temporaneo. Le funzioni `cuCaddf` e `cuCmulf` sono specificate da CUDA per effettuare rispettivamente la somma e il prodotto tra numeri complessi. È importante sapere che l'implementazione di `get_sample` è scritta in modo che restituisca un numero complesso con parte reale e immaginaria uguali a 0 (zero) nel caso l'indice richiesto sia fuori dalla

dimensione dell'array del canale.

Questo tipo di implementazione viene chiamata da Smith "algoritmo dal lato dell'ingresso" [6, pp. 112-115], poiché elabora il contributo di ogni singolo elemento dell'ingresso rispetto a più posizioni nell'uscita.

In figura 3.2 si può osservare il risultato della convoluzione di un impulso di 32 campioni con sé stesso ottenuto con l' algoritmo presentato.



(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

3.3.2 GPU

Per implementare la convoluzione sulla GPU è necessario individuare i thread che è possibile parallelizzare e racchiuderli in uno o più kernel. Nel caso in esame si è deciso di utilizzare l'implementazione che Smith chiama "algoritmo dal lato dell'uscita" [6, pp. 116-121], il quale differisce dall'algoritmo precedentemente illustrato nell'implementazione sulla CPU per il fatto che

si calcolano i contributi di vari campioni dell'ingresso rispetto ad un solo campione in uscita. I due algoritmi, seppur presentino due punti di vista diversi, sono equivalenti e restituiscono lo stesso risultato.

Bisogna prestare attenzione al fatto che dato un kernel di un filtro di M elementi, il valore dell'elemento i-esimo dell'uscita è uguale al prodotto dell'elemento $i - j$ esimo dell'ingresso con l'elemento j-esimo del kernel del filtro, con $j = 0...M - 1$.

Il kernel CUDA utilizzato per compiere questa operazione ridotto alle sue operazioni essenziali è il seguente:

```
1  __global__ void cudaconvolver_kernel(SignalBuffer_t
       device_buffer, SignalBuffer_t kernel_buffer,
       SignalBuffer_t out_buffer, size_t channel)
2  {
3      int k = blockIdx.x * blockDim.x + threadIdx.x;
4      ...
5      cuComplex temp_sample = make_cuComplex(0,0);
6      for (int i = 0; i < kernel_size; i++)
7      {
8          kernel_sample =
9              get_sample(kernel_buffer, SIGNAL_CHANNEL, i);
10         input_sample =
11             get_sample(device_buffer, channel, k-i);
12         temp_sample = cuCaddf(temp_sample,
13             cuCmulf(kernel_sample, input_sample));
14     }
15     set_sample(out_buffer, channel, index, temp_sample);
16     ...
17 }
```

Si noti la presenza dell'indice k, il quale viene calcolato in base agli indici del thread corrente e del blocco corrente; esso identifica la posizione k dell'elemento dell'uscita da calcolare.

Se il segnale in ingresso è composto di N punti e il kernel del filtro è composto da M punti, il segnale in uscita è di $N + M - 1$ punti; infatti la funzione kernel viene eseguita $N + M - 1$, ovvero è presente una esecuzione della funzione per ogni elemento in uscita.

Nel codice mostrato è stato tolta la parte per la gestione dell'effetto di "overlap" in modo da rendere evidenti le parti fondamentali della sua implementazione per la GPU.

Il grafico in figura 3.3 rappresenta il tempo necessario per compiere una convoluzione utilizzando un kernel di un filtro di dimensioni fisse (128 punti) e facendo variare la lunghezza del file in ingresso da elaborare. Inoltre per quanto riguarda la GPU sono visualizzati anche valori diversi di lunghezza del buffer di elaborazione, in quanto, a differenza della CPU che non risente della grandezza dello stesso, la GPU presenta prestazioni via via migliori all'aumentare della dimensione del buffer, poiché un buffer più grande significa elaborare più dati in parallelo.

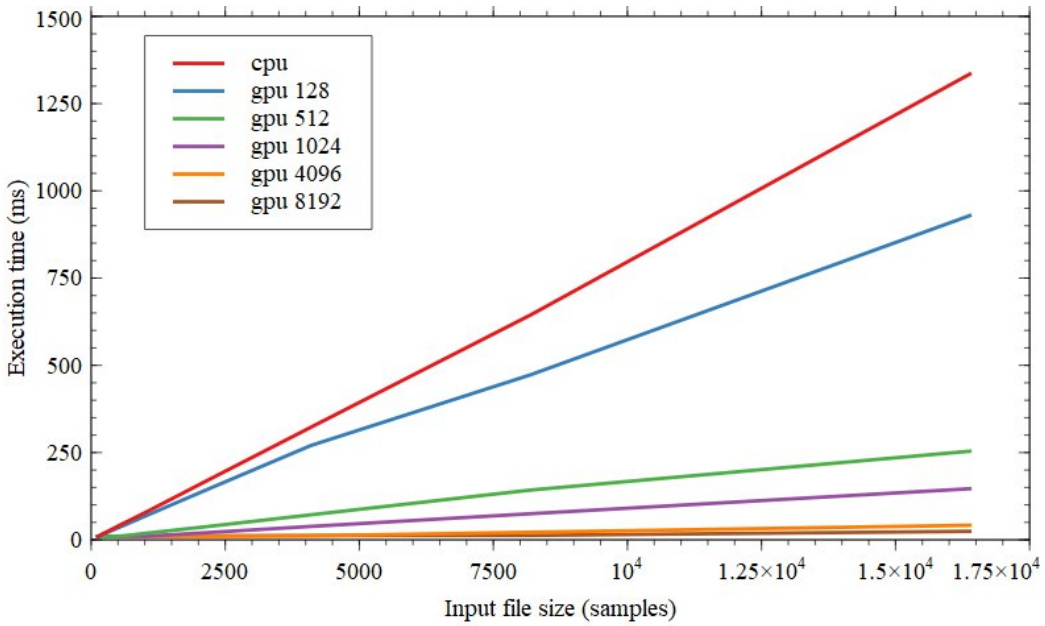


Figura 3.3: Tempo di convoluzione, CPU e GPU a confronto

3.4 DFT

Un secondo algoritmo interessante da implementare è la trasformata di Fourier discreta. Essa trova larga applicazione come strumento di analisi spettrale dei segnali, oltre ad essere un punto di partenza per poi introdurre la *Fast Fourier Transform* (FFT).

Come indicato nell'equazione **??**, la DFT è sostanzialmente una sommatoria di un prodotto di numeri complessi, notiamo però che a differenza della equazione della convoluzione (**??**) il valore massimo della sommatoria dipende dalla lunghezza del segnale. Questo significa che nella implementazione saranno necessari due cicli `for` innestati che dipendono entrambi dalla lunghezza del segnale. Tale configurazione di cicli `for` restituisce una complessità computazionale di tipo $O(N^2)$. Motivo ulteriore per cui si è sviluppato l'algoritmo della FFT.

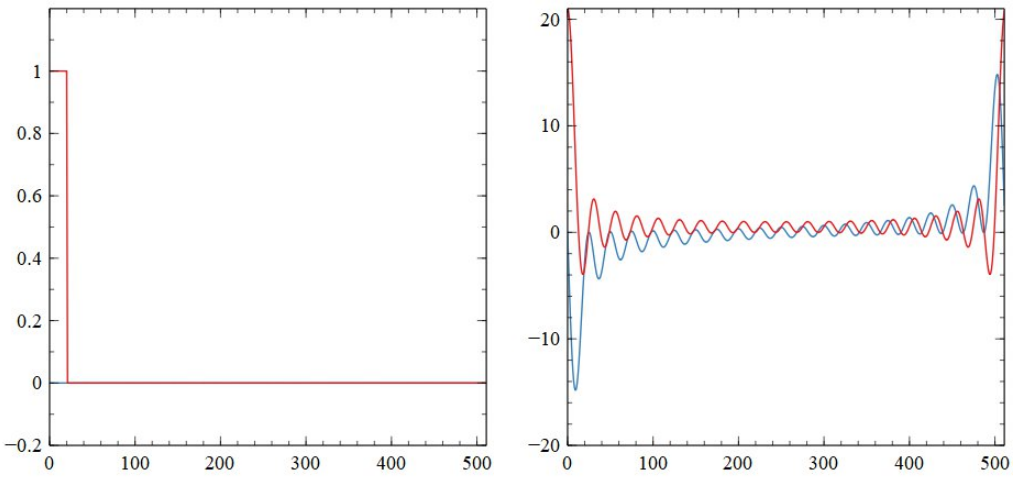
3.4.1 CPU

L'implementazione della trasformata di Fourier discreta è ricavata direttamente dalla sua equazione.

```
1  void dft_wsio(SignalBuffer_t* bufferIn, SignalBuffer_t*
       bufferOut, size_t channel, size_t size)
2  {
3      ...
4      for (size_t k = 0; k < size; k++)
5      {
6          ...
7          for (size_t i = 0; i < size; i++)
8          {
9              cuComplex s =
10                 cuComplex_exp(-2 * M_PI * k * i / size);
11             cuComplex sample =
12                 get_sample(*bufferIn, channel, i);
13             cuComplex outs =
14                 get_sample(*bufferOut, channel, k);
15             outs = cuCaddf(outs, cuCmulf(sample, s));
16             set_sample(*bufferOut, channel, k, outs);
```

```
17              }
18          }
19          ...
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_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 3.4: DFT di un impulso di 512 campioni. In rosso è segnata la parte reale e in blu la parte immaginaria

Inserendo nel programma l'impulso di 512 campioni mostrato in figura 3.4a si ottiene la trasformata in figura 3.4b. 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.

3.4.2 GPU

L'operazione di trasformata di Fourier discreta è implementata sulla GPU utilizzando il seguente kernel:

```
1  __global__ void cudadft_kernel_dft(SignalBuffer_t
       device_buffer, SignalBuffer_t out_buffer, size_t channel,
       size_t size)
2  {
3      int k = blockIdx.x * blockDim.x + threadIdx.x;
4      ...
5      cuComplex temp = make_cuComplex(0, 0);
6      for (int i = 0; i < size; i++)
7      {
8          cuComplex sample =
9              get_sample(device_buffer, channel, i);
10         cuComplex s
11             = cuComplex_exp(-2.0f * PI * k * i / size);
12         temp = cuCaddf(temp, cuCmulf(sample, s));
13     }
14     set_sample(out_buffer, channel, k, temp);
15 }
```

Anche in questo caso, il kernel viene eseguito in parallelo su tutti i campioni del segnale risultando più prestante della rispettiva implementazione sulla CPU. In particolare si può già notare che la complessità di ogni singolo thread si è ridotta a $O(N)$.

[grafici prestazioni]

3.5 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]

3.5.1 CPU

La FFT è implementata sulla CPU nel seguente modo:

```
1     [codice da ridimensionare]
```

[da fare].

3.5.2 GPU

[da fare]

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     codice da ridimensionare
```

```
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.