Università degli Studi di Modena e Reggio Emilia

FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

Prova Finale

Studio e implementazione, per CPU e GPU, di alcuni algoritmi utilizzati nell'elaborazione digitale di segnali

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 il calcolatore elettronico.

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.

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 a canale singolo 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 2 alcune nozioni matematiche necessarie per la corretta comprensione e spiegazione delle implementazioni degli algoritmi presentati in seguito. Viene spiegato, inoltre, come si utilizza la GPU dal punto di vista implementativo, quali sono i "componenti di calcolo" principali e come vi si interfaccia con le API di CUDA.

Nel capitolo 3 vengono presentate e spiegate le parti più interessanti degli algoritmi studiati, i quali sono interamente disponibili 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" [7, 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" [7, 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 [7, 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 [7, 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 [7, 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. Nella pratica dei DSP essa viene utilizzata anche per segnali aperiodici con particolari accortezze (per evitare fenomeni di aliasing o convoluzione circolare).

Per un segnale di N punti è definita nel seguente modo [1, p. 2.50]:

$$X_n = \sum_{k=0}^{N-1} x_k e^{-j2\pi kn/N} \tag{2.9}$$

Si presti attenzione al fatto che la trasformata è periodica di periodo N, conseguenza del fatto che il segnale in ingresso è discreto [1, p. 2.30]. Inoltre i punti da 0 a $N/2$ rappresentano le frequenze positive e da $N/2$ a $N-1$ le frequenze negative. 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 $N/2$ [7, p. 570].

L'operazione di antitrasformazione è definita nel modo seguente [1, p. 2.50]:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{j2\pi kn/N} \tag{2.10}$$

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 3 al suo posto si utilizza l'algoritmo di Cooley-Tukey [3], denominato "FFT" ovvero *Fast Fourier Transform*, il quale permette di ridurre la complessità computazionale della trasformazione.

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 intendere le GPU solamente 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 memoria della GPU, non nella memoria centrale

della CPU. 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 dati 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 elemento per elemento 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 [6].

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 size;
5     size_t max_size;
6 };
```

Essa continene un array di campioni, la lunghezza corrente e la lunghezza massima dello stesso. Questa struttura dati viene utilizzata sia dalla CPU sia dalla GPU e l'utilizzo di un array è necessario perché CUDA, senza librerie aggiuntive esterne, non riesce a gestire nei kernel gli oggetti della libreria standard di template di C++, di cui fa parte il template `vector`.

Vengono definite anche operazioni su questa struttura, le quali ne garantiscono la corretta creazione, eliminazione e la corrispettiva gestione dei campioni all'interno della stessa. Le funzioni utilizzate sono dichiarate con i modificatori `__host__ __device__` i quali sono necessari per segnalare al compilatore `nvcc` che tali funzioni possono essere sia eseguite sulla CPU (host), sia sulla scheda grafica (device).

Infine `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.

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, a patto che si conosca la risposta impulsiva del sistema stesso.

La definizione di convoluzione tra segnali discreti è descritta dall'equazione 2.2. 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; ciò comporta che sia necessario 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 un'operazione che necessita di due segnali in entrata, si è supposto che il secondo segnale 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" [7, 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

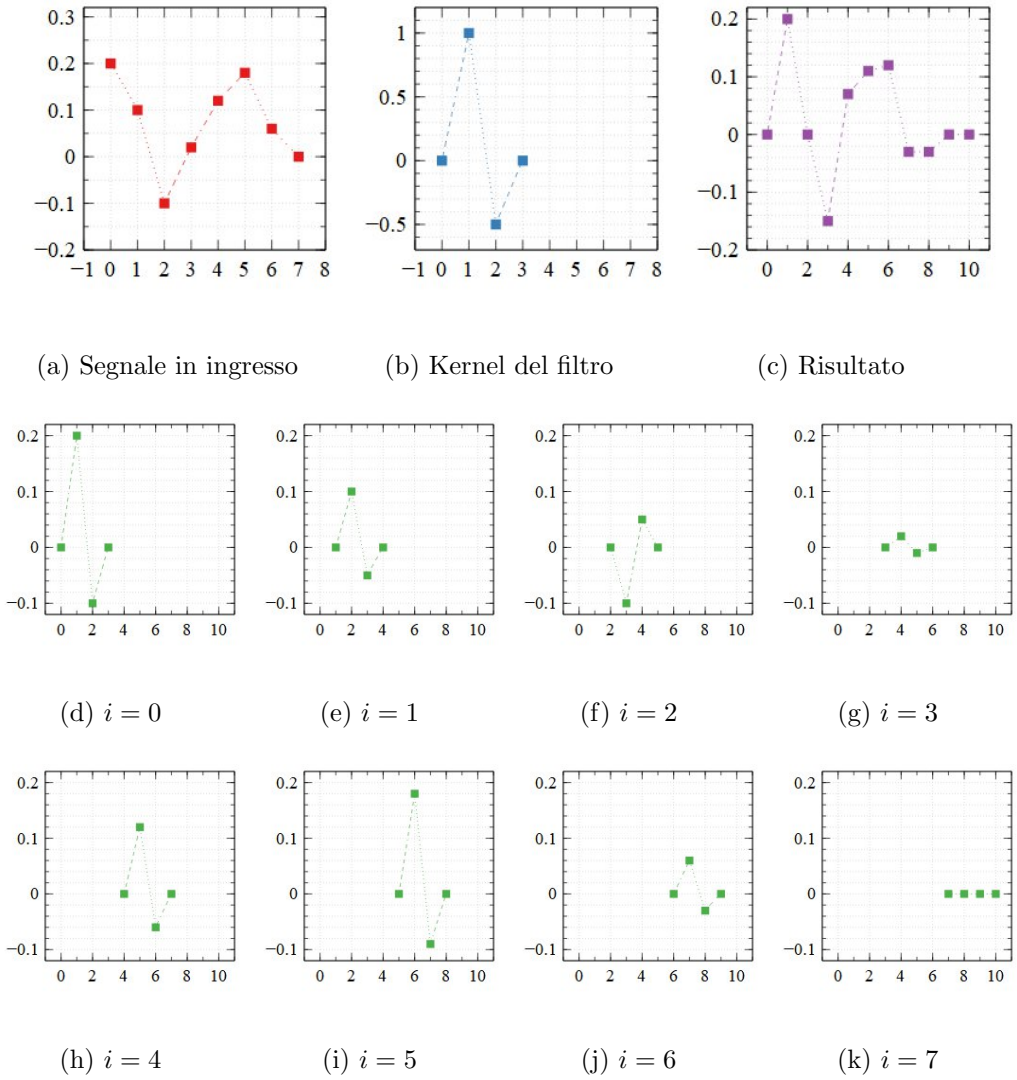
la corretta gestione del fenomeno dell'"overlap"; il buffer temporaneo alla fine della convoluzione conterrà i campioni in uscita dalla convoluzione e la "coda" da sommare alla convoluzione successiva.

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

```
1  ...
2  for (size_t i = 0; i < buffer_size; i++)
3  {
4      cuComplex in_sample = get_sample(buffer, i);
5      for (size_t j = 0; j < kr_size; j++)
6      {
7          size_t index = i + j;
8          cuComplex kr_sample = get_sample(signal, j);
9          cuComplex out_sample = get_sample(temp, index);
10         cuComplex result = cuCaddf(out_sample,
11                                    cuCmulf(in_sample, kr_sample));
12         set_sample(temp, index, result);
13     }
14 }
15 ...
```

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.

Questo tipo di implementazione viene chiamata da Smith "algoritmo dal lato dell'ingresso" [7, pp. 112-115], poiché per ogni singolo elemento dell'ingresso ne elabora il contributo a più posizioni nell'uscita. Esso equivale ad eseguire la somma dei "sottosegnali" ottenuti dal prodotto tra il kernel del filtro e il valore del campione in ingresso e dalla traslazione rispetto al suo indice. Una visualizzazione grafica dell'algoritmo è riportata in figura 3.1.



(a) Segnale in ingresso (b) Kernel del filtro (c) Risultato

(d) $i = 0$ (e) $i = 1$ (f) $i = 2$ (g) $i = 3$

(h) $i = 4$ (i) $i = 5$ (j) $i = 6$ (k) $i = 7$

Figura 3.1: Scomposizione dei singoli contributi del segnale in ingresso in una convoluzione.

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 "algo-

ritmo dal lato dell'uscita" [7, 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 in_buf,
       SignalBuffer_t kr_buf, SignalBuffer_t tmp)
2  {
3      int k = blockIdx.x * blockDim.x + threadIdx.x;
4      ...
5      cuComplex kr_s, in_s;
6      cuComplex tmp_s = make_cuComplex(0, 0);
7      for (int i = 0; i < kr_buf_size; i++)
8      {
9          kr_s = get_sample(kr_buf, i);
10         in_s = get_sample(in_buf, k-i);
11         tmp_s = cuCaddf(tmp_s, cuCmulf(kr_s, in_s));
12     }
13     set_sample(tmp, index, tmp_s);
14     ...
15 }
```

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; per questo motivo 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 relativa alla gestione dell'effetto di "overlap" in modo da rendere evidenti le parti fondamentali dell' implementazione per la GPU.

È inoltre opportuno specificare che la funzione `get_sample` restituisce un numero complesso con parte reale e immaginaria nulla nel caso l'indice richiesto sia all'esterno della dimensione del buffer.

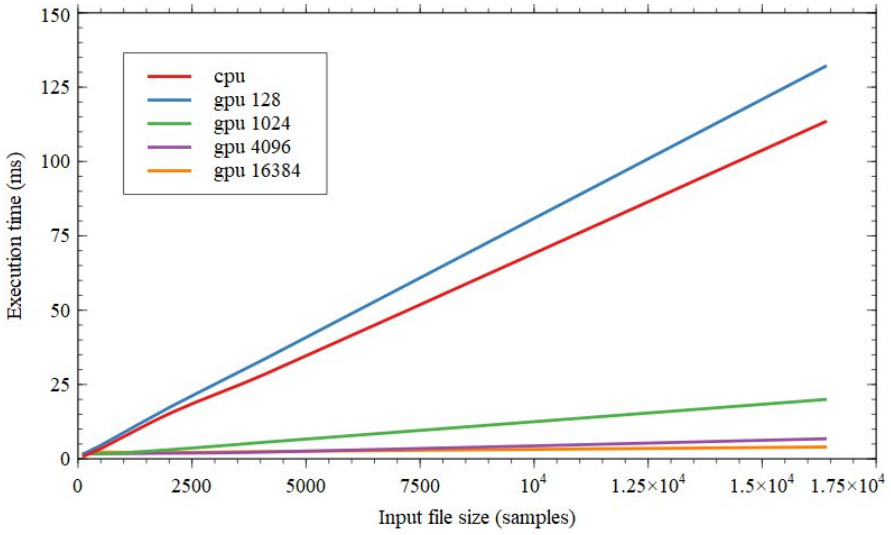


Figura 3.2: Tempo di convoluzione, CPU e GPU a confronto

Il grafico in figura 3.2 rappresenta il tempo necessario per compiere una convoluzione utilizzando un kernel di un filtro di dimensioni fisse (16 punti) al variare della 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. Si può anche notare che la GPU ottiene dei tempi di elaborazione peggiori della CPU quando il buffer non è sufficientemente grande: il risultato è giustificato dal fatto che in tale circostanza sono necessari più spostamenti

di dati tra memoria RAM e memoria della GPU i quali agiscono da "collo di bottiglia" nell'elaborazione. Per come è stato progettato il programma è necessario che un buffer venga elaborato prima che si possa elaborare il seguente (non è possibile elaborare più buffer in parallelo).

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 2.9, la DFT è sostanzialmente una sommatoria di un prodotto di numeri complessi, notiamo però che a differenza della equazione della convoluzione (2.2) 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.

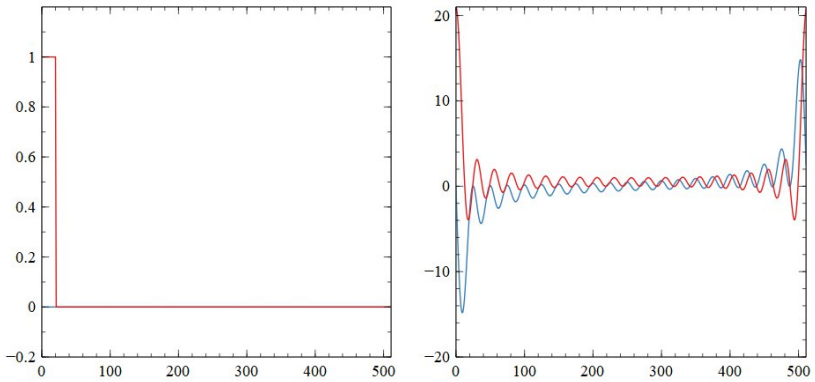
3.4.1 CPU

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

```
1 void dft(SignalBuffer_t &in_buf, SignalBuffer_t &out_buf,
    size_t size)
2 {
3     cuComplex s, in_s, out_s;
4     ...
5     for (size_t k = 0; k < size; k++)
6     {
```

```
7           out_s = get_sample(out_buf, k);
8           ...
9           for (size_t i = 0; i < size; i++)
10          {
11              s = cuComplex_exp(-2 * M_PI * k * i / size);
12              in_s = get_sample(in_buf, i);
13              out_s = cuCaddf(out_s, cuCmulf(s, in_s));
14          }
15          set_sample(out_buf, k, out_s);
16      }
17      ...
18  }
```

`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 3.3: 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

3.3a si ottiene la trasformata in figura 3.3b. 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(SignalBuffer_t in_buf,
       SignalBuffer_t out_buf)
2  {
3      int k = blockIdx.x * blockDim.x + threadIdx.x;
4      ...
5      size_t size = get_max_buffer_size(out_buf);
6      cuComplex out_s = make_cuComplex(0, 0);
7      cuComplex in_s, s;
8      for (int i = 0; i < size; i++)
9      {
10         in_s = get_sample(device_buffer, i);
11         s = cuComplex_exp(-2.0f * PI * k * i / size);
12         out_s = cuCaddf(out_s, cuCmulf(in_s, s));
13     }
14     set_sample(out_buf, k, out_s);
15     ...
16 }
```

Anche in questo caso, il kernel viene eseguito in parallelo su tutti i campioni del segnale risultando più performante della rispettiva implementazione sulla CPU. In particolare si può già notare che la complessità di ogni singolo thread si è ridotta a $O(N)$ e, essendo i thread eseguiti contemporaneamente, la complessità reale non dovrebbe discostarsi troppo da quest'ultima; come dimostrato dai dati relativi al tempo di elaborazione ricavati dal programma e mostrati in figura 3.4.

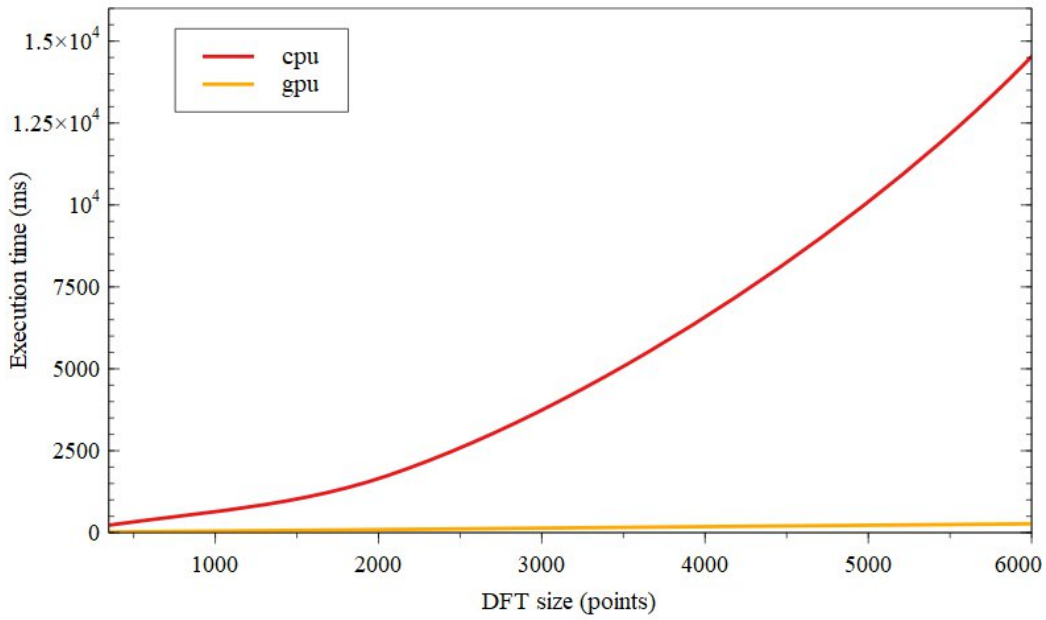


Figura 3.4: Tempo di calcolo per DFT tra GPU e CPU, al variare del numero di punti della DFT.

3.5 Fast Fourier Transform

L'operazione di DFT è onerosa in termini di calcolo in quanto ha complessità $O(N^2)$, 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" le quali permettono all'algoritmo di ottenere una complessità di tipo $O(Nlog_2N)$ [3].

3.5.1 Spiegazione matematica

La FFT utilizza le proprietà di periodicità e di "simmetria complessa coniugata" del numero complesso $e^{-2\pi j\frac{kn}{N}}$ che si può trovare a secondo membro

dell'equazione 2.9. Infatti, definendo $W_N = e^{-\frac{2\pi j}{N}}$ è immediato verificare che:

$$W_N^{k(N-n)} = e^{-2\pi jk} e^{+2\pi j \frac{kn}{N}} = W_N^{-kn} = (W_N^{kn})^* \tag{3.1}$$

$$W_N^{kn} = W_N^{(k+N)n} = W_N^{k(n+N)} \tag{3.2}$$

Separando la sommatoria della DFT in equazione 2.9 in termini pari e in termini dispari si ottiene:

$$X_n = \sum_{k=0}^{N-1} x_k W_N^{kn} = \sum_{k\ pari} x_k W_N^{kn} + \sum_{k\ dispari} x_k W_N^{kn} \tag{3.3}$$

Supponendo $N = 2^m$ con $m = \mathbb{N}$, gli indici delle sommatorie possono essere scritti come $2p$ e $2p+1$ con $p = 0, 1, \ldots, \frac{N}{2} - 1$:

$$X_n = \sum_{p=0}^{\frac{N}{2}-1} x_{2p} W_N^{2pn} + \sum_{p=0}^{\frac{N}{2}-1} x_{2p+1} W_N^{(2p+1)n} \tag{3.4}$$

Raccogliendo nella seconda sommatoria si ottiene:

$$X_n = \sum_{p=0}^{\frac{N}{2}-1} x_{2p} (W_N^2)^{pn} + W_N^n \sum_{p=0}^{\frac{N}{2}-1} x_{2p+1} (W_N^2)^{pn} \tag{3.5}$$

Notando che $W_N^2 = e^{-2\pi j \frac{2}{N}} = e^{-\frac{2\pi j}{N/2}} = W_{N/2}$ si ottiene:

$$X_n = \sum_{p=0}^{\frac{N}{2}-1} x_{2p} W_{N/2}^{pn} + W_N^n \sum_{p=0}^{\frac{N}{2}-1} x_{2p+1} W_{N/2}^{pn} \tag{3.6}$$

Le due sommatorie nella 3.6 sono due DFT di $N/2$ elementi e la loro somma equivale alla DFT originaria, X_n.

Il ragionamento utilizzato fino ad ora si può quindi riapplicare alle DFT di $N/2$ elementi ottenute nella 3.6, fino ad ottenere, dopo $log_2 N$ volte, delle DFT di un singolo elemento.

Se per una DFT bisogna compiere N^2 operazioni, dopo ogni suddivisione in pari/dispari la DFT viene riscritta come due DFT di $N/2$ elementi e N operazioni. Questo significa che il numero di operazioni sono diventate $2(\frac{N}{2})^2 + N = \frac{N^2}{2} + N$ e procedendo in maniera analoga si arriva al numero di operazioni dopo $log_2 N$ suddivisioni: $N + Nlog_2 N$ che equivale, quindi, ad una complessità di $O(Nlog_2 N)$ per $N >> 1$.

3.5.2 Spiegazione dell'algoritmo

Data la spiegazione matematica della sezione precedente, è immediato notare che il primo passo per calcolare la FFT di un array di $N = 2^m$, $m \in \mathbb{N}$ è quello di riordinare gli elementi in modo che nella prima metà dell'array ci siano gli elementi in posizione pari, mentre nella seconda metà gli elementi di posizione dispari. Dopodiché si ripete questa separazione per i due "sottoarray" di $N/2$ elementi. Su un calcolatore ad aritmetica binaria questo è facilmente riconducibile ad un ordinamento a "bit rovesciati" degli indici dell'array, come mostrato in figura 3.5 per un array di dimensione $N = 8$.

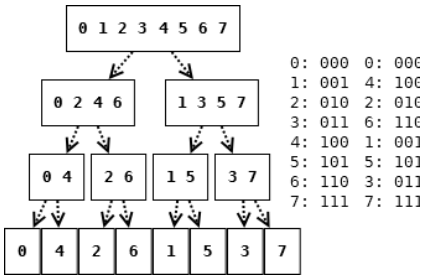


Figura 3.5: Scomposizione interlacciata per il calcolo della FFT di un array di 8 elementi.

In seguito bisogna calcolare la DFT di ogni singolo elemento, un calcolo banale in quanto il risultato della DFT di un singolo elemento è l'elemento stesso.

A questo punto bisogna effettuare le operazioni di somma all'indietro rispettando, quindi, l'ordine con cui si è effettuata la suddivisione dell'array. Notiamo, innanzitutto, che ad ogni "livello" di suddivisione si effettua una moltiplicazione e una somma; inoltre è importante notare che in una DFT di N punti si ha $W_N^{k+N/2} = -W_N^k$ e quindi che, grazie alla 3.2:

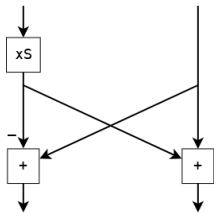


Figura 3.6: La "farfalla", operazione fondamentale per la FFT.

$$X_{n+N/2} = \sum_{p=0}^{\frac{N}{2}-1} x_{2p} W_{N/2}^{pn} - W_N^n \sum_{p=0}^{\frac{N}{2}-1} x_{2p+1} W_{N/2}^{pn} \tag{3.7}$$

Disegnando, quindi, un diagramma per il calcolo di X_n e $X_{n+N/2}$ si ottiene un diagramma in figura 3.6 che viene denominato "farfalla" per la sua forma. Le operazioni a "farfalla" si combinano dall'array iniziale ordinato a "bit invertiti" nel modo raffigurato in figura 3.7 per un array iniziale di dimensione $N = 4$.

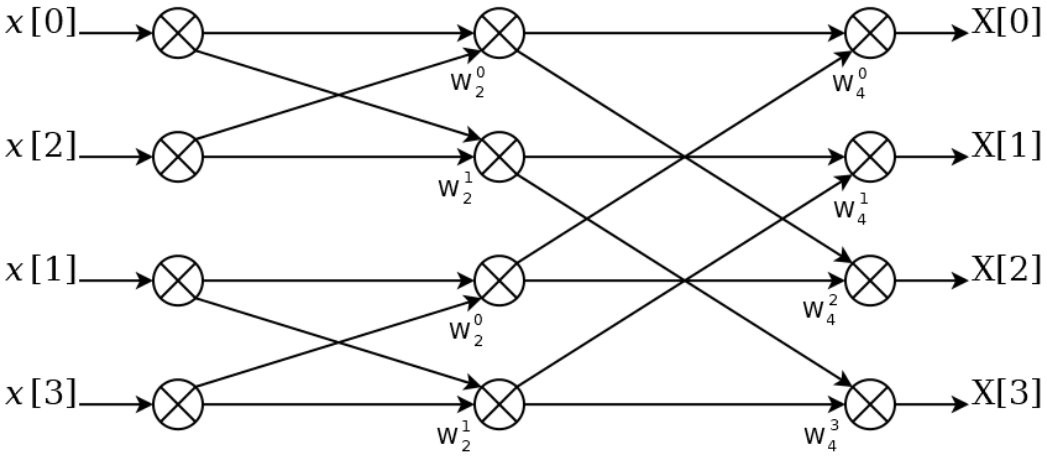


Figura 3.7: Operazioni a farfalla consecutive per il calcolo della FFT di un array di 4 elementi.

3.5.3 CPU

Per implementare sulla CPU la FFT è necessario, quindi, la creazione di una procedura di ordinamento a bit invertiti e una per effettuare le operazioni a "farfalla".

```
1  void fft_wsio(SignalBuffer_t &b_in, SignalBuffer_t &b_out,
       size_t points)
2  {
3      ... // declare vars, assign sizes
4      bit_reversal_sort_wsio(b_in, b_out, points);
5      for (size_t level = 0; level < levels; level++){
6          ...
7          wm = cuComplex_exp(-(M_PI / butterflies_per_dft));
8          w = make_cuComplex(1,0);
9          for (size_t butterfly = 0; butterfly <
               butterflies_per_dft; butterfly++){
10             for (size_t dft = 0; dft < dfts; dft++){
11                 ... // calculate indexes
12                 cuComplex a = get_sample(b_out, index_a);
13                 cuComplex b = get_sample(b_out, index_b);
14                 butterfly_calculation(&a, &b, w);
15                 ... // set out samples
16             }
17             w = cuCmulf(w, wm);
18         }
19         ...
20     }
21 }
```

Nell'implementazione si può notare che per ogni livello si scorrono prima le operazioni a farfalla e poi le "sotto DFT" corrispondenti; in questo modo si può utilizzare lo stesso valore di W_N^k (si faccia riferimento ad esempio alla figura 3.7, dove W_2^0 e W_2^1 appaiono due volte al primo livello). Dopodiché il valore successivo W_N^{k+1} si ricava dalla moltiplicazione di W_N^k con il valore di `wm`, il quale contiene il numero complesso con argomento necessario per calcolare W_N^{k+1}; questo evita il calcolo tramite funzioni trigonometriche.

3.5.4 GPU

L'implementazione per GPU dell'algoritmo della FFT segue lo stesso schema della CPU, ovvero: si riordina l'array e si effettuano le operazioni a "farfalla". La scomposizione in kernel utilizzata consiste nell'operare il riordino tramite una funzione kernel eseguita su ogni elemento dell'array, dopodiché per ogni livello si utilizza un kernel che esegue le somme a farfalla; è chiaro, infatti, che il numero di operazioni a farfalla per ogni livello è uguale a quello degli altri livelli ed è pari ad $N/2$. Quello che cambia da un livello all'altro sono quali coppie di elementi le farfalle elaborano e per questo motivo non è possibile eseguire l'intera FFT con una sola invocazione di kernel, bensì è necessario invocare il kernel per ogni livello (ovvero un numero pari a $log_2 N$ volte), attendendo che i dati siano sincronizzati dopo ogni calcolo.

Il kernel utilizzato per le operazioni a farfalla è qui riportato:

```
1  __global__ void cudafft_kernel_butterflies(SignalBuffer_t
       in_buf, SignalBuffer_t out_buf, size_t level)
2  {
3      unsigned int k = blockIdx.x * blockDim.x + threadIdx.x;
4      size_t bpd = (size_t)powf(2, level);
5      size_t index_a = k + (size_t)(k / bpd) * bpd;
6      size_t index_b = index_a + bpd;
7      // get a,b samples
8      cuComplex w = cuComplex_exp(-(2*PI*index_a)/(bpd*2));
9      cuda_butterfly_calculation(&a, &b, w);
10     // set a,b samples
11 }
```

La funzione `cuda_bufferfly_calculation` esegue l'operazione a farfalla, come nella implementazione per la CPU. La parte interessante del kernel riguarda come ricavare gli indici di `a` e `b` poiché non è possibile sapere direttamente in quale "sotto DFT" è la farfalla, ma è necessario ricavare tale indice.

In figura 3.8 sono raffigurati i tempi di calcolo rispetto alla lunghezza della fft. Le linee sono tratteggiate in quanto la FFT non può essere calcolata, con gli algoritmi presentati, per qualsiasi numero di punti, bensì solo per

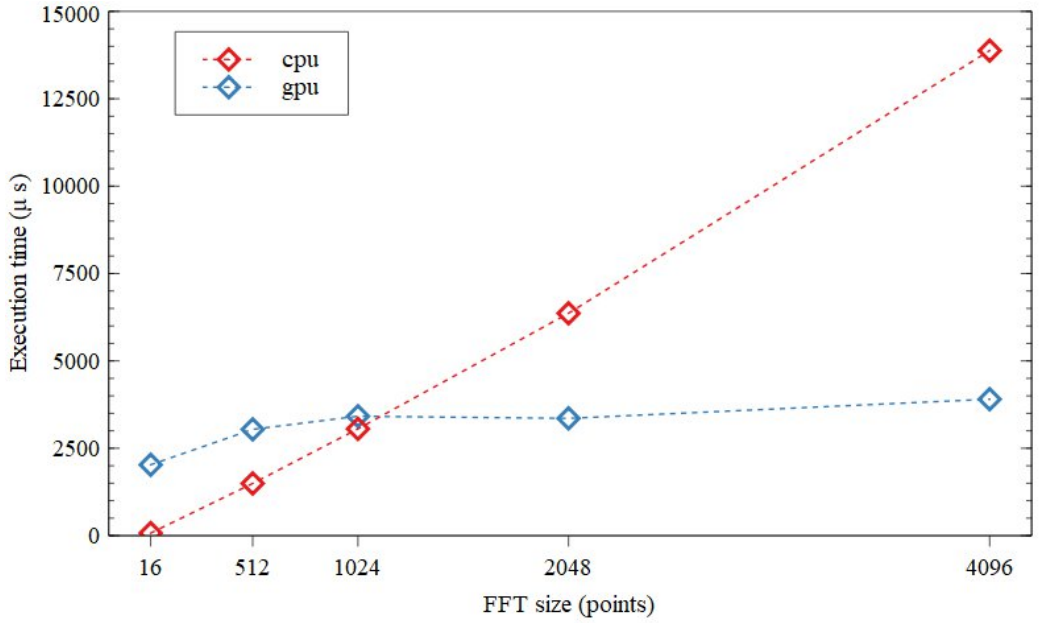


Figura 3.8: Tempo di calcolo per FFT tra GPU e CPU, al variare del numero di punti della FFT.

le potenze di 2. I tempi di elaborazione rispetto alla DFT sono bassissimi (si noti la scala dell'asse y in microsecondi) e la GPU riesce comunque ad ottenere prestazioni migliori al crescere del numero di punti rispetto alla CPU, seppur quest'ultima abbia tempi di calcolo minori per valori di N bassi, dove la GPU non riesce a competere perché è limitata dagli spostamenti di memoria.

Capitolo 4

Conclusioni

L'elaborazione digitale di segnali è un argomento vastissimo, con varie sfaccettature e varianti: una cosa che funziona molto bene in un caso, potrebbe funzionare molto male in un altro caso e tutto dipende a cosa bisogna applicare tale operazione. Un esempio lampante è il tempo di elaborazione della FFT presentate nell'ultimo capitolo: la CPU ha prestazioni eccezionali per pochi punti, mentre la GPU ha prestazioni incredibili nel caso opposto. Questo non significa che l'utilizzo di una o l'altra piattaforma sia esclusiva, anzi, per come è architetturata ora la tecnologia il caso ottimo è quello di utilizzare entrambi i "centri di calcolo".

La sfida rimane quella di modificare gli algoritmi preesistenti in una loro versione che possa essere calcolata in modo parallelo su una GPU. Come si è visto, non sempre è possibile parallelizzare l'intero processo in una volta sola e a volte bisogna scendere a compromessi e utilizzare funzioni di sincronizzazione.

Nonostante ciò, le prestazioni ottenute dalla GPU sono a dir poco incredibili e soprattutto i grafici presentati mostrano uno *scaling* molto promettente. Non a caso, infatti, questi dispositivi vengono utilizzati per la ricerca nel

campo delle intelligenze artificiali e nella costruzione di supercomputer.

Appendice A

Aggiunte

A.1 Attrezzatura utilizzata

Il programma che è stato utilizzato nel corso di questa tesi è stato scritto ed eseguito sulla macchina seguente:

- CPU: Intel Core i5-6500 @3.20GHz, 4 Cores.
- RAM: 16GB
- GPU: Nvidia GeForce GTX 970
  - CUDA Driver version: 10.2
  - CUDA Capability: 5.2
  - Global Memory: 4096MB
  - CUDA Cores: 1664
  - Max clock rate: 1.25GHz
  - Memory clock rate: 3505Mhz
  - Memory bus width: 256 bit
  - Max thread per block: 1024
  - nvcc version: V10.2.89

A.2 Codice operazione a “farfalla”

```
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-prj` (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] Jeremy Sanders. *Veusz*. URL: `https://veusz.github.io/` (visitato il 16/03/2020).

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