

FFT GPGPU

Edoardo Alessandro Scarpaci

Maggio 2022

1 Introduzione

La DFT(Discrete Fourier Transform) é una delle piu importanti trasformazioni discrete, usata nelle analisi di Fourier. La DFT permette di convertire una sequenza finita di samples, definiti nel dominio del tempo, in una sequenza della stessa lunghezza della prima, nella quale i samples sono mappati nel dominio delle frequenze. La trasformazione é usata in moltissimi ambiti: analisi spettrali, ottica, diffrazione, compressioni di dati, equazioni differenziali parziali.

Per via dell'ampio utilizzo della trasformata in moltissimi applicazioni, sono stati ricercati degli algoritmi, molto ottimizzati chiamati FFT, che é l'acronimo di Fast Fourier Transform.

Questa classe di algoritmi(FFT) implementano una DFT, alcune volte approssimata altre volte esatta, cercando di ottimizzare o i passi computazionali oppure gli accessi in memoria, in modo da essere cache friendly. Una delle piu famose implementazioni delle FFT é chiamata Cooley-Tukey, in onore dei ricercatori che hanno implementato l'algoritmo. Cooley-Tukey FFT utilizza il meccanismo di divide and conquer per ottimizzare il calcolo della DFT, infatti decomponete la DFT in DFT piu piccole per poi in seguito ricomporle in quella completa.

2 Algoritmi

2.1 DFT

La DFT permette di convertire una sequenza finita di samples di una determinata funzione in una sequenza della stessa lunghezza della prima, nella quale i samples sono mappati in uno spazio in funzione delle frequenze. La DFT transforma una sequenza di lunghezza N di numeri complessi $\{x_n\} := x_0, x_1 \dots x_{N-1}$ in un'altra sequenza, di lunghezza N, di numeri complessi $\{X_k\} := X_0, X_1 \dots X_{N-1}$ definita in funzione della prima dalla relazione: $X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N} kn}$.

L'algoritmo ha una complessità computazionale pari a $O(N^2)$ poiché ogni sample richiede $O(N)$ operazione da effettuare.

2.2 Cooley-Tukey FFT

FFT Cooley-Tukey esprime la DFT come una arbitraria composizioni in due DFT di grandezza N_1 ed N_2 , in cui N_1 ed N_2 sono definite dalla relazione $N = N_1 N_2$, in cui N è la dimensione della DFT prima della decomposizione. Effettuando questa decomposizioni ricorsivamente l'algoritmo è in grado di diminuire la complessità computazionale da $O(N^2)$ ad $O(N \log N)$.

Una delle più famose implementazione dell'algoritmo Cooley-Tukey, è chiamata radix-2 DIT, la differenze dalle altre implementazioni è che questo divide le DFT sempre in due DFT di dimensione $N/2$. Data la DFT definita come: $X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N} kn}$ la radix-2 DIT divide la sequenza di samples in due sequenze, quella degli indici pari $\{x_{2m}\} := x_0, x_2, \dots, x_{N-2}$ ed in quella degli indici dispari $\{x_{2m+1}\} := x_1, x_3, \dots, x_{N-1}$, ed su ognuna di queste sequenza effettua una DFT. Questa decomposizione di DFT viene eseguita ricorsivamente, per poi ricomporre il tutto nella DFT originale. Le DFT scomposte sono definite come: $X_k = \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-\frac{i2\pi}{N} km} + \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-\frac{i2\pi}{N} km}$. Definendo come $E_k = \sum_{m=0}^{N/2-1} x_{2m} \cdot e^{-\frac{i2\pi}{N} km}$ la DFT degli indici pari e $O_k = \sum_{m=0}^{N/2-1} x_{2m+1} \cdot e^{-\frac{i2\pi}{N} km}$ quella degli indici dispari, è possibile definire $X_k = E_k + e^{-\frac{i2\pi}{N} k} O_k$, grazie alla periodicità degli esponenti complessi è possibile riscrivere la precedente equazioni in due equazioni, $X_k = E_k + e^{-\frac{i2\pi}{N} k} O_k$ ed $X_{k+\frac{N}{2}} = E_k - e^{-\frac{i2\pi}{N} k} O_k$.

3 Implementazioni

Tutti gli algoritmi si basano, sulla decomposizione e successiva ricomposizioni di diverse DFT sempre di dimensione più piccola. L'algoritmo quindi si basa su delle operazioni di mapping e delle operazioni di ricomposizione per ogni DFT.

3.0.1 Mapping

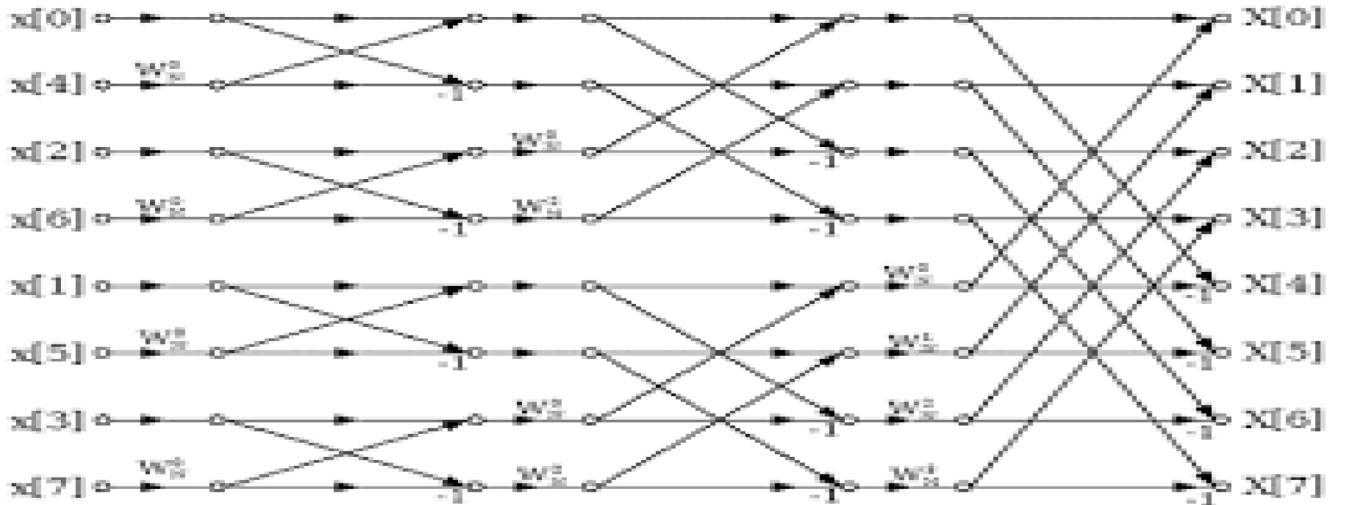
Operazione di mapping consiste nel mappare gli indici delle sequenze decomposte in un ordine definito dall'operazione di bit-reversal permutation.

3.0.2 Ricomposizione

L'operazione di ricomposizione delle DFT si basa su delle operazioni di sin e cos, per calcolare i fattori di ricomposizione, ed in delle somme, per ricomporre le DFT.

3.0.3 Butterfly flow diagram

È possibile visualizzare tutte le operazioni fatte da una fft Cooley-Tukey usando un diagramma di flusso a farfalla.



3.1 Naive CPU

L'algoritmo implementato sulla cpu, è implementato usando un singolo thread, ed usando la ricorsione per andare ad effettuare l'operazione di mapping ed ricomposizione. Ogni iterazione dell'algoritmo andrà ricorsivamente a dividere le sequenze di samples nelle due sequenze, di dimensione $N/2$, di indici pari ed indici dispari. Arrivati al caso base si ricostruiscono le DFT usando le precedenti formule.

```
void FFT::fft(std::complex<float>* A, size_t N,int iter){

    if (N<=1) return;

    std::complex<float>* even = new std::complex<float>[N/2];
    std::complex<float>* odd = new std::complex<float>[N/2];

    for(size_t i=0;i<N/2;i++){
        even[i] = A[i*2];
        odd[i] = A[i*2+1];
    }

    fft(even,N/2,iter-1);
    fft(odd,N/2,iter-1);
    for(int i=0;i<N/2;i++){
        std::complex<float> t = std::exp(std::complex<float>(0, -2 * M_PI * i / N));
        std::complex<float> t_odd = t * odd[i];

        A[i] = even[i] + t_odd;
        A[i + N/2] = even[i] - t_odd;
    }

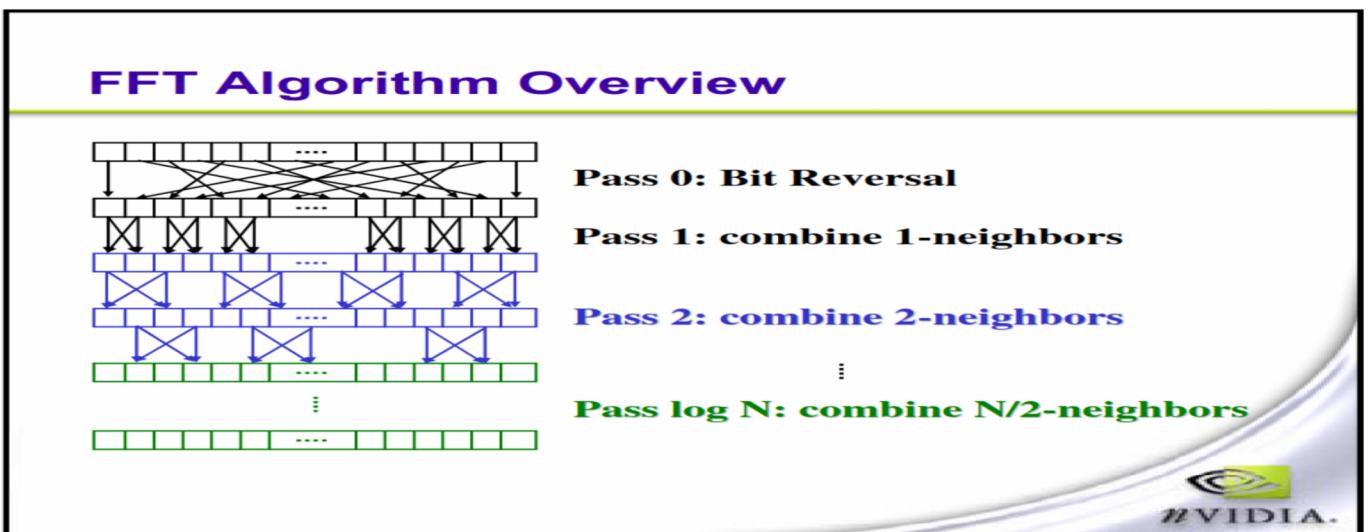
    delete[] even;
    delete[] odd;
}
```

3.2 FFTW

FFTW è una libreria multipiattaforma che implementa vari algoritmi FFT, tra cui Cooley-Tukey, la userò come baseline per le performance reali di un algoritmo fft ottimizzato per essere eseguito sulla CPU.

3.3 GPU Implementation

Le implementazioni su GPU useranno le due operazioni, Mapping, Ricomposizione, parallelamente. Ogni thread della gpu si occuperà di ricomporre due elementi della sequenza in un elemento della DFT. Per questo motivo saranno eseguiti contemporaneamente $N/2$ work-item. L'algoritmo sarà eseguito $\log_2(N)$ volte per ricomporre totalmente le DFT, ogni iterazione ricomporre le DFT di dimensione 2^{iter} .



3.3.1 Bit Reversal Permutation

L'algoritmo di bit reversal permutation per radix-2 effettua semplicemente uno scambio degli $N/2$ bit più significativi e degli $N/2$ bit meno significativi.

Index	Binary	Bit-Reversed	Bit-Reversed
		Binary	Index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

```
int generatePermutation(int index,int logLength){  
    int a = index;  
    int b = 0;  
    int j = 0;  
  
    while(j++ < logLength){  
        b = (b << 1) | (a & 1);  
        a >>= 1;  
    }  
    return b;  
}
```

3.3.2 GPU FFT 1

La prima implementazione su GPU non contiene nessun tipo di ottimizzazione.

L'operazione di mapping é effettuata durante le operazione di ricomposizione.

Ogni work-item effettua il mapping sui due indice che deve ricomporre. Nell'ultima operazione non bisogna di seguire l'operazione di mapping per l'output poiché é l'ultima ricomposizione quindi l'ordine é quello definito solo dal calcolo degli indici.

```
kernel void fft_1(global float2* input, global float2 *output,int length,int iter){
    const int gid = get_global_id(0);
    if (gid >= length/2) return;
    int logLength = round(log2((float)length));

    //butterfly group_size
    const int b_size = round(pow(2.,iter));
    //stride to pick the odd term (1,2,4,8....)
    const int stride = round(pow(2.,iter-1));

    //Number of group
    const int n_group = length / pow(2.,iter);

    const int id = (gid / stride) * b_size + (gid%stride);
    const int j = (gid%stride) * n_group;
    const int id_strided = id + stride;

    const int index_permutated = generatePermutation(id,logLength);
    const int index_permutated_strided= generatePermutation(id_strided,logLength);

    const float2 even = input[index_permutated];
    const float2 odd = input[index_permutated_strided];

    const float angle = -2 * M_PI * j / length;
    float2 wj;
    wj.x = cos(angle);
    wj.y = sin(angle);

    const float2 wj_odd = cmult(wj,odd);

    const float2 even_output = even + wj_odd;
    const float2 odd_output = even - wj_odd;

    if(iter == logLength){
        output[id] = even_output;
        output[id_strided] = even - wj_odd;
    }
    else{
        output[index_permutated] = even_output;
        output[index_permutated_strided] = even - wj_odd;
    }
}
```

3.3.3 GPU FFT 2

La seconda implementazione contiene ottimizzazione a solo livello di computazione.

Poiché usiamo un Cooley-Tukey Radix-2 tutte le dimensioni delle DFT decomposte sono potenze di 2, quindi è possibile sostituire tutte le operazioni: potenza, modulo, moltiplicazioni e divisione, con operazioni bitwise.

L'operazione di Mapping è sempre ed eseguita durante la ricomposizione ad ogni iterazione;

```
kernel void fft_2(global float2* input, global float2 *output,int length,int iter){
    const int gid = get_global_id(0);
    if (gid >= length >> 1) return;
    int logLength = round(log2((float)length));

    //butterfly group_size
    const int b_size = 1 << iter;
    //stride to pick the odd term (1,2,4,8...)
    const int stride = 1 << (iter-1);

    //Number of group
    const int n_group = length >> iter;

    const int base_offset = gid & (n_group -1);

    // 0 1 2 3 = [0,1] = 0   e [2,3] = 1
    const int group_id = gid >> n_group;
    //0,1,2,3 = [0,1,2,3] = 0;

    const int id = (gid >> ( iter -1 )) * b_size + (gid & (stride -1));
    const int id_strided = id + stride;

    const int index_permutated = generatePermutation(id,logLength);
    const int index_permutated_strided= generatePermutation(id_strided,logLength);

    const int j = (gid & (stride -1) ) * n_group;

    const float2 even = input[index_permutated];
    const float2 odd = input[index_permutated_strided];

    const float angle = -2 * M_PI * j / length;
    float2 wj;
    wj.x = cos(angle);
    wj.y = sin(angle);

    const float2 wj_odd = cmult(wj,odd);

    const float2 even_output = even + wj_odd;
    const float2 odd_output = even - wj_odd;

    if(iter == logLength){
        output[id] = even_output;
        output[id_strided] = odd_output;
    }
    else{
        output[index_permutated] = even_output;
        output[index_permutated_strided] = odd_output;
    }
}
```

3.3.4 GPU FFT 3

La terza implementazione contiene le ottimizzazioni precedenti ed in più ottimizzazioni per l'accesso alla memoria.

In questo caso utilizziamo N work-item per eseguire l'operazione di Mapping in tutto l'array dei sample,in modo da rendere gli accessi alla memoria il più coalescente possibile, evitando salti di grandi dimensioni nella memoria .

L'algoritmo prima di effettuare le operazioni di ricomposizione, esegue l'operazione di mapping su tutto l'array di input, ed infine sull'array già mappato si eseguono le operazioni di ricomposizione.

```

kernel void bitReverse(global float2* input,global float2* output,int logLength){
    const int gid = get_global_id(0);
    const int gid_permuted = generatePermutation(gid,logLength);
    if(gid >= 1<<logLength) return;

    output[gid] = input[gid_permuted];
}

kernel void fft_3(global float2* input, global float2 *output,int length,int iter){
    const int gid = get_global_id(0);
    if (gid >= (length >> 1)) return;

    //butterfly group_size
    const int b_size = 1 << iter;
    //stride to pick the odd term (1,2,4,8...)
    const int stride = 1 << (iter-1);

    //Number of group
    const int n_group = length >> iter;

    const int base_offset = gid & (n_group -1);
    // 0 1 2 3 = [0,1]= 0   e [2,3] = 1
    const int group_id = gid >> n_group;
    //0,1,2,3 = [0,1,2,3] = 0;
    const int id = (gid >> ( iter -1 )) * b_size + (gid & (stride -1));
    const int id_strided = id + stride;

    /*if(gid == 1)
        printf("Ids:(%d, %d, %d)\n",gid,id,id_strided);
    */

    const int j = (gid & (stride -1) ) * n_group;

    const float2 even = input[id];
    const float2 odd = input[id_strided];

    const float angle = -2 * M_PI * j / length;
    float2 wj;

    wj.x = cos(angle);
    wj.y = sin(angle);

    const float2 wj_odd = cmult(wj,odd);

    const float2 even_output = even + wj_odd;
    const float2 odd_output = even - wj_odd;

    output[id] = even_output;
    output[id_strided] = even - wj_odd;
}

```

3.3.5 GPU FFT 4

La quarta implementazione cerca di ottimizzare ancora una volta gli accessi in lettura alla memoria, permettendo di effettuare, gli accessi in lettura coalescenti.

L'operazione di mapping è sempre eseguita in tutto l'array prima di eseguire l'algoritmo fft.

Dopo ogni esecuzione della ricomposizione si effettua un'operazione di riordinamento degli elementi, permettendo così all'operazione di ordinamento successiva di eseguire accessi in lettura coalescenti.

```

kernel void permuteArray(global float2* input,global float2* output,int lenght,int iter){}
    const int gid = get_global_id(0);
    if(gid >= lenght/2) return;

    const int id = calculateIndex(gid,lenght,iter);
    const int stride = 1 << (iter-1);
    const int id_strided = id +stride;

    output[gid*2] = input[id];
    output[gid*2+1] = input[id_strided];
}

```

```

kernel void fft_4(global float2* input, global float2 *output,int length,int iter,int logLength){}
    const int gid = get_global_id(0);
    if (gid >= (length >> 1)) return;

    const int id = gid*2;
    const int id_strided = id+1;

    const float2 even = input[id];
    const float2 odd = input[id_strided];
    const int j = calculateJ(gid,length,iter);
    const float angle = -2 * M_PI * j / length;

    float2 wj;
    wj.x = cos(angle);
    wj.y = sin(angle);

    const float2 wj_odd = cmult(wj,odd);

    const float2 even_output = even + wj_odd;
    const float2 odd_output = even - wj_odd;

    const int output_id = calculateIndex(gid,length,iter);
    const int output_id_strided = output_id + (1<<(iter-1));

    output[output_id] = even_output;
    output[output_id_strided] = even - wj_odd;
}

```

3.4 Performance

Gli algoritmi implementati nella gpu saranno confrontati con: cpu Naive e fftw, con dimensione dei samples che andranno da 2^3 ad 2^{20} .

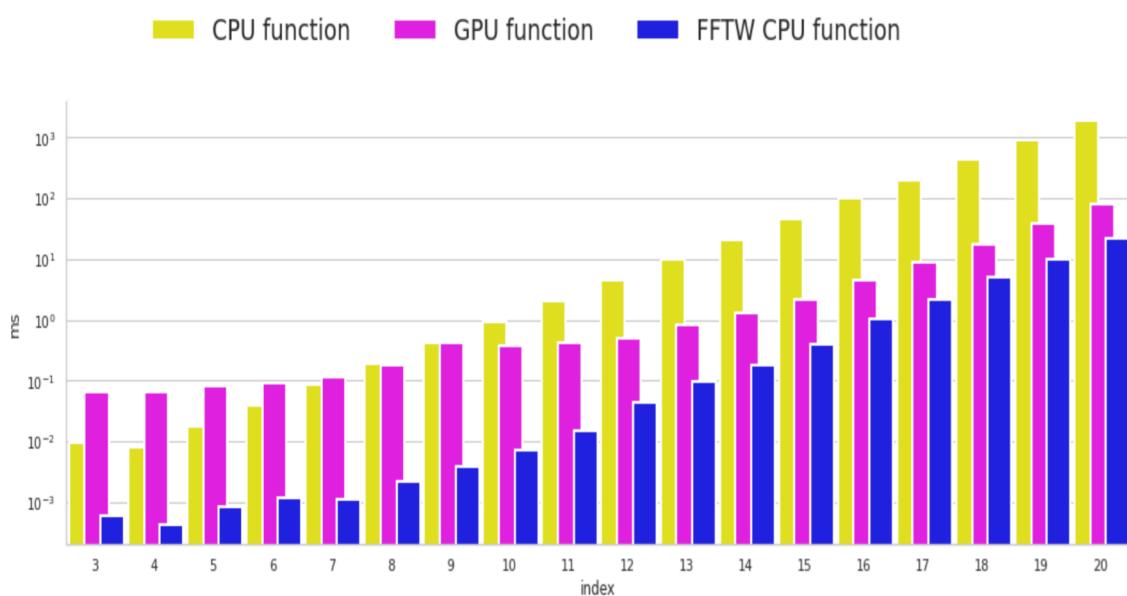
Tutti i grafici saranno in scala logaritmica nell'asse delle y. Tutte le misure di performance calcolate nei grafici seguenti, sono state calcolate usando una media aritmetica di 5 run di ogni algoritmo per le varie dimensioni dei samples.

Tutti i benchmark sono state effettuate sul seguente sistema:

CPU	GPU	OP
Ryzen 5 5600h	GTX 1650	Linux Mint 20.3

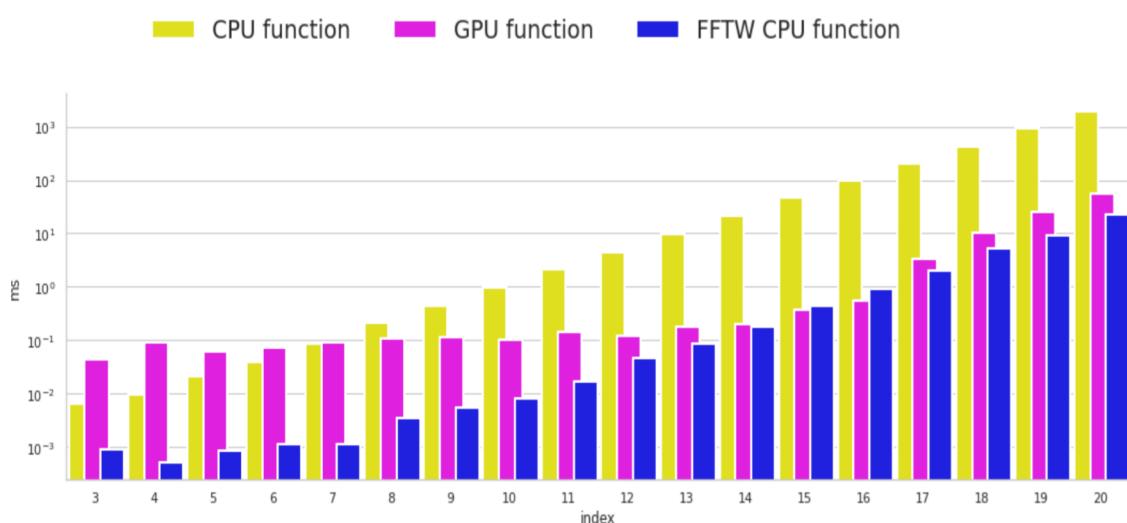
3.4.1 GPU FFT 1

Per via della mancanza di ottimizzazioni l'algoritmo la prima iterazione del FFT implementato su GPU risulta essere sempre più lento, di fftw, invece per dimensioni superiori a 2^{10} risulta essere più veloce di CPU naive.



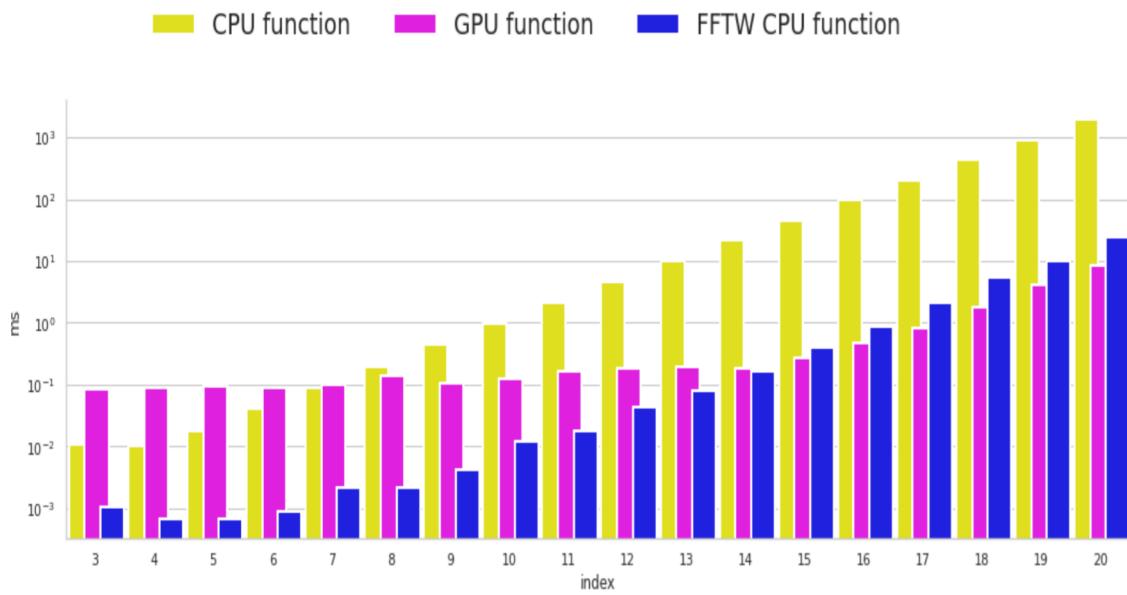
3.4.2 GPU FFT 2

Grazie alle ottimizzazioni a livello di computazione per alcune dimensioni l'algoritmo gpu fft2 risulta essere più veloce delle controparti su cpu. In più è possibile notare che poiché lo speedup si ha solo in un range limitato, e non scala linearmente, l'algoritmo è probabilmente IO bound per elevate quantità di dati.



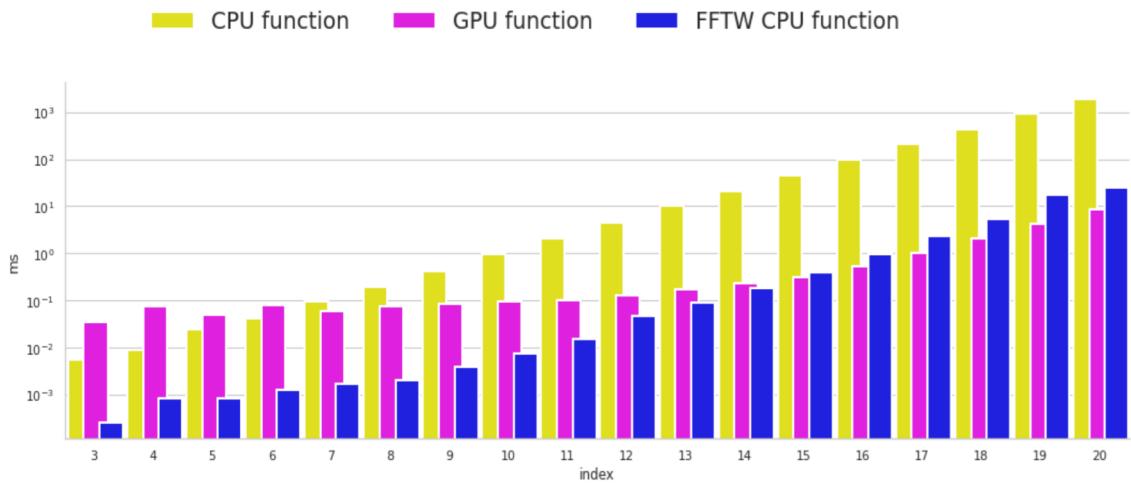
3.4.3 GPU FFT 3

Poiche per come si é potuto vedere dalle misure di performance precedenti l'algoritmo sembra essere IO bound, l'algoritmo cerca di ottimizzare gli accessi della memoria, riordinando l'array di samples prima delle computazioni. In questo modo, soprattutto nelle prime run, l'algoritmo tende ad effettuare accessi alla memoria molto vicini tra di loro, quindi dimuendo il problema IO. In questo caso é possibile notare che per piccole dimensioni l'algoritmo risulta essere sempre IO bound,ma per dimensioni maggiori di 2^{13} l'algoritmo sembra scalare linearmente con il numero di samples.



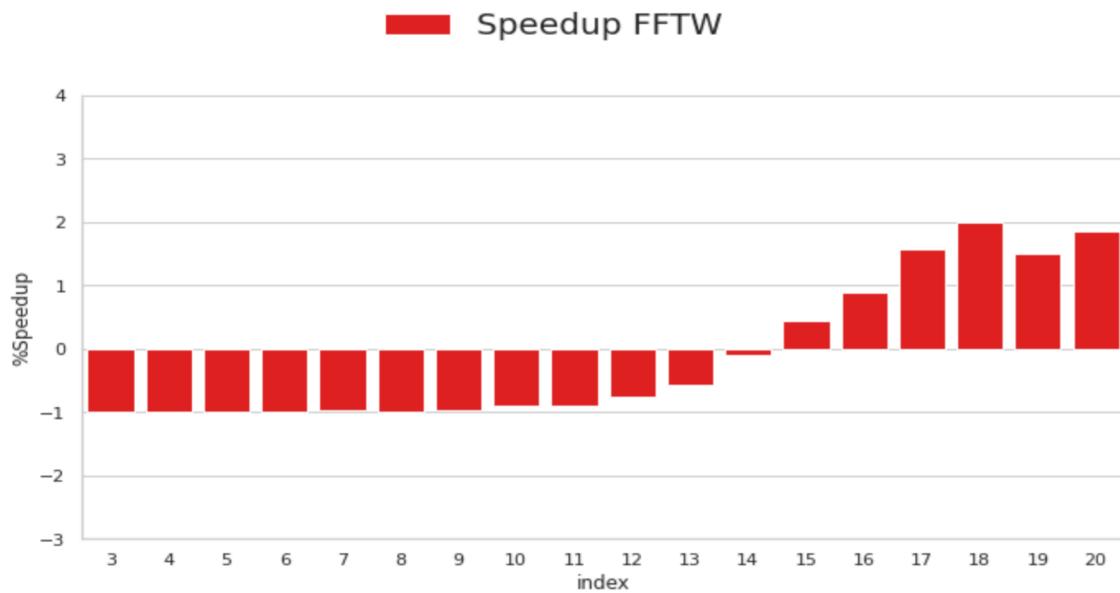
3.4.4 GPU FFT 4

L'algoritmo precedente permette di limitare l'essere IO bound,pero soprattutto per piccoli samples di dati risulta essere sempre IO bound. Questa iterazione dell'algoritmo rende effettivamente gli accessi in memoria ,durante la lettura, coalescenti. Pero per renderli coalescenti effettua un operazioni di reordering dei samples dopo ogni operazione di ricomposizione. L'aggiunta di questo riordinamento rende la computazione della ricomposizione molto più veloce ma purtroppo aggiunge un passo che rallenta la computazione nel complesso. Infatti quest'algoritmo risulta essere equiparabile, o poco più lento, dell'algoritmo precedente per sample abbastanza grandi,invece per sample piu piccoli risulta essere un minimo più veloce dei precedenti, anche se sempre più lento delle controparti CPU.



3.4.5 Speedup

Confrontato la migliore iterazione ,la 3,dell'algoritmo su GPU con l'algoritmo fftw è possibile notare come per dimensioni dei samples maggiori di 2^{14} si ottiene un speedup considerevole,che tende ad aumentare con la dimensione dei dati, seppur non linearmente.



4 Conclusioni

Vedendo le varie performance degli algoritmi,è possibile affermare che per dimensioni elevate di sample,per l'esattezza maggiori di 2^{14} , è conveniente offloadare la computazione su gpu, ottenendo uno speedup fino al 3x rispetto anche alla migliore implementazioni su CPU,però per piccoli samples di dati piccoli è sempre conveniente usare gli algoritmi implementati su CPU.

4.0.1 Possibili Miglioramenti

Per come si é potuto vedere anche la migliore iterazione dell'algoritmo su gpu risulta essere sempre IO bound. I possibili miglioramenti che in futuro sarebbe possibile applicare:

- Trovare un altro layout in memoria delle DFT decomposte in modo da rendere gli accessi sia in lettura che scrittura coalescenti
- Utilizzare delle lookuptable per le operazioni di Sin e Cos

Il principale problema nell'implementazione del primo miglioramento, che é quello che porterebbe il maggior miglioramento, é nell'implementazione stessa di cooley-tukey, per come é definito l'algoritmo, risulta non essere cache coherent, per via di tutti i salti che effettua nei diversi spazi di memoria.

5 Crediti

[John Spitzer NVIDIA Corporation] GPU Implementation

[Benjamin Blunt] Cooley Tukey FFT

[lukicdarkoo] Cpu Naive FFT

FFTW