FFT GPGPU

Edoardo Alessandro Scarpaci

Maggio 2022

1 Introduzione

La DFT(Discrete Fourier Transform é una delle più 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 ,nella quale i samples sono mappati nel dominio delle frequenze.La trasformazioni é usata in moltissimi ambiti:analisi spettrali, ottica,diffrazione, compressioni di dati, equazioni differenziali parziali.

Per via dell'ampio utilizzo della trasformata,sono stati creati degli algoritmi chiamati FFT(Fast Fourier Transform).

Le FFT implementano una trasformata di Fourier, 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 più 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,cioè decomponendo le DFT in DFT sempre più piccole, ed infine ricomponendole.

2 Algoritmi

2.1 DFT

La DFT trasforma una sequenza di lunghezza N di numeri complessi $\{x_n\} := x_0, x_1 ... x_{N-1}$ in un altra sequenza, di lunghezza N, di numeri complessi $\{X_k\} := X_0, X_1 ... X_{N-1}$ 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 di 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 é la dimensione della DFT prima della decomposizione. Effettuando questa decomposizioni ricorsivamente l'algoritmo é 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,é chiamata radix-2 DIT.Implementazione Radix-2 DIT sceglie entrambi N_1 ed N_2 come $N/2$. Data una 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 $\{x_{2m}\} := x_0, x_2 ... x_{N-2}$ ed in quella degli indici dispari $\{x_{2m+1}\} := x_1, x_3 ... x_{N-1}$, ed su ognuna di queste sequenza effettua una trasformata di Fourier. 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 é possibile definire $X_k = E_k + e^{-\frac{i2\pi}{N}k} O_k$, infine grazie alla periodicità degli esponenti complessi é 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. L'algoritmo è possibile basarlo su operazioni di mapping e ricomposizione.

3.0.1 Mapping

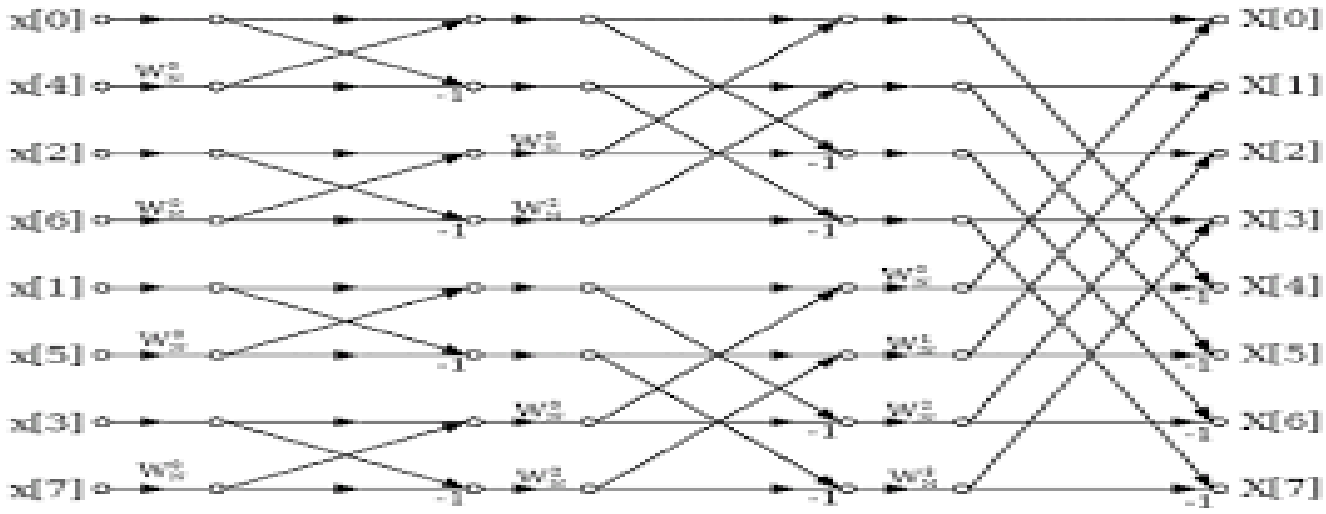
Operazione di mapping consiste nel mappare gli indici delle sequenze decomposte in un ordine definito dall'operazione di bit-reversal permutation.

3.0.2 Ricomposizione

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

3.0.3 Butterfly flow diagram

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



Tutti i benchmark sono state effetuate sul seguente sistema:

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

3.1 Naive CPU

L'algoritmo implementato sulla cpu,usa un singolo thread, ed usa la ricorsione per effettuare l'operazione di mapping ed ricomposizione. Ogni iterazione l'algoritmo andrà ricorsivamente a dividere le sequenze di samples in altre due sequenze di dimensione N/2,una conterrà gli indici pari ed l'altra quelli dispari. Arrivati al caso base si ricostruiscono le DFT usando le precedenti formule.

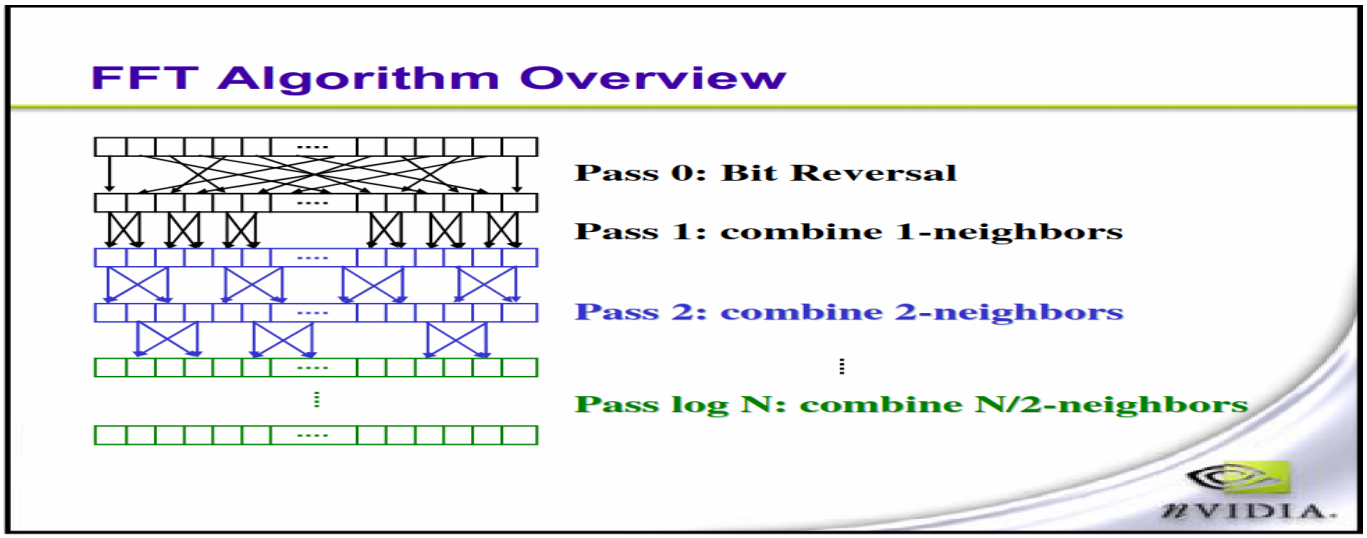
```
void FFT::fft(std::complex<float>* A, size_t N,int iter){

    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 é una libreria multi piattaforma che implementa vari algoritmi FFT, tra cui Cooley-Tukey.La userò come baseline per le performance di un algoritmo FFT ottimizzato per essere eseguito sulla CPU.

3.3 GPU Implementation

Le implementazioni su GPU useranno le due operazioni,Mapping e 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 Binary	Bit-Reversed 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 eseguire l'operazione di mapping sull'output poiché é l'ultima ricomposizione.

```
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(pown(2.,iter));
    //stride to pick the odd term (1,2,4,8....)
    const int stride = round(pown(2.,iter-1));

    //Number of group
    const int n_group = length / pown(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;

    float cosvalue;
    wj.y = sincos(angle,&cosvalue);
    wj.x = cosvalue;


    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 Performance FFT1

Per via della mancanza di ottimizzazioni la prima iterazione del FFT risulta essere piu lento di fftw per qualsiasi dimensione data in input, pero è possibile notare come per un numero di elementi superiore a 2^{10} risulta essere più veloce dell'implementazione CPU naive.

3.3.4 GPU FFT 2

La seconda implementazione contiene ottimizzazione solamente al livello di computazione.

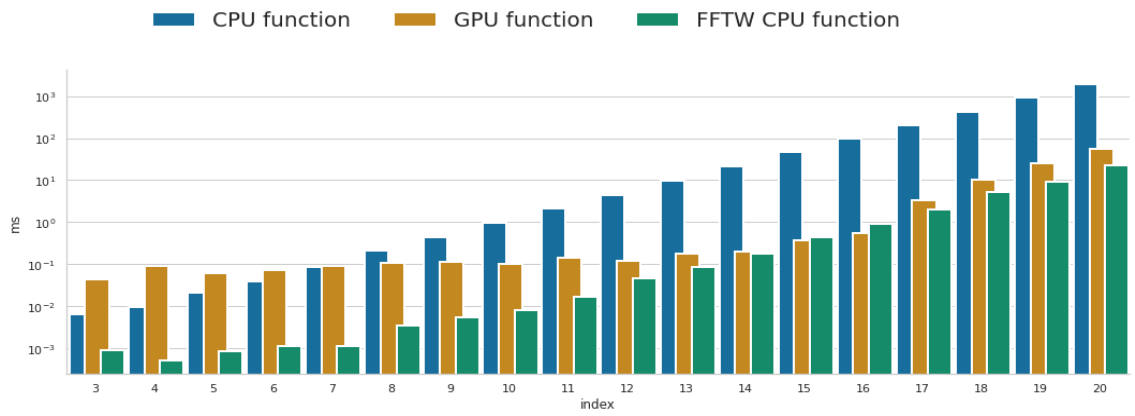
Poiché usiamo un Cooley-Tukey Radix-2 tutte le dimensioni delle DFT decomposte sono potenze di 2,quindi é possibile sostituire tutte le operazioni: potenza, modulo, moltiplicazioni e divisione, con operazioni bitwise.

L'operazione di Mapping é sempre ed eseguita durante la ricomposizione ad ogni iterazione.

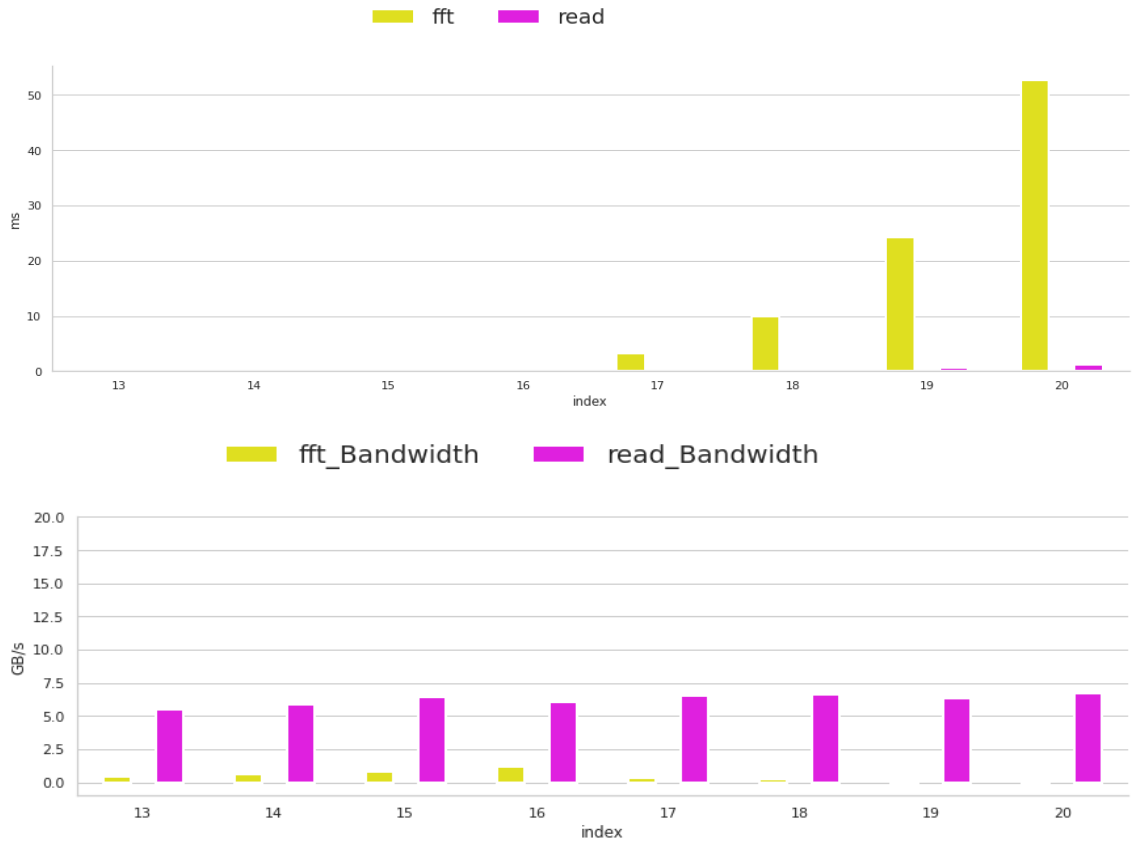
```
//optimization mod and pow using bit manipulation
kernel void fft_2(global float2* input, global float2 *output,int length,int iter){
    const int gid = get_global_id(0);
    if (gid >= length >> 1) return;
    int logLength = round(log2((float)length));

    //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;
    float cosvalue;
    wj.y = sincos(angle,&cosvalue);
    wj.x = cosvalue;

    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] = odd_output;
    }
}
```

3.3.5 Performance FFT2

Grazie alle ottimizzazione sulla computazione per alcune dimensioni l'algoritmo FFT2 risulta essere più veloce delle controparti su cpu. In più é possibile notare che poiché lo speedup si ha solo in un range limitato,e non scala linearmente con la quantitá di dati, l'algoritmo é probabilmente limitato dagli accessi in memoria non coalescenti.



infatti come si envice dai due grafi successivi non si hanno grossi miglioramenti dall'iterazione precedente dell'algoritmo



3.3.6 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 tutti i samples,in modo da rendere gli accessi alla memoria il piu coalescenti possibile,evitando salti di grandi dimensioni nella memoria .

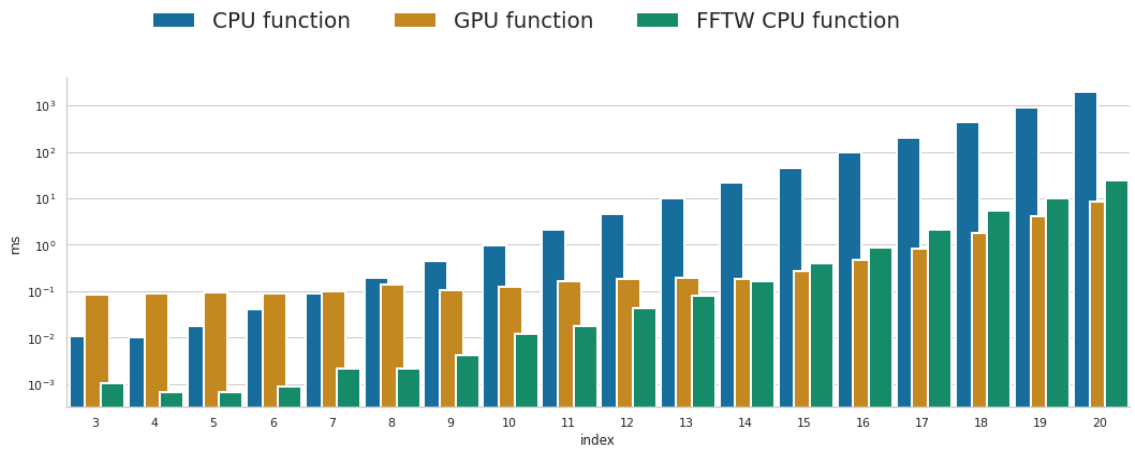
L'algoritmo prima di effettuare le operazioni di ricomposizione,esegue l'operazione di mapping su tutto l'array di input, ed infine sull'array gia mappato esegue tutte le operazione di ricomposizione.

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


    output[gid] = input[gid_permutated];

}
```

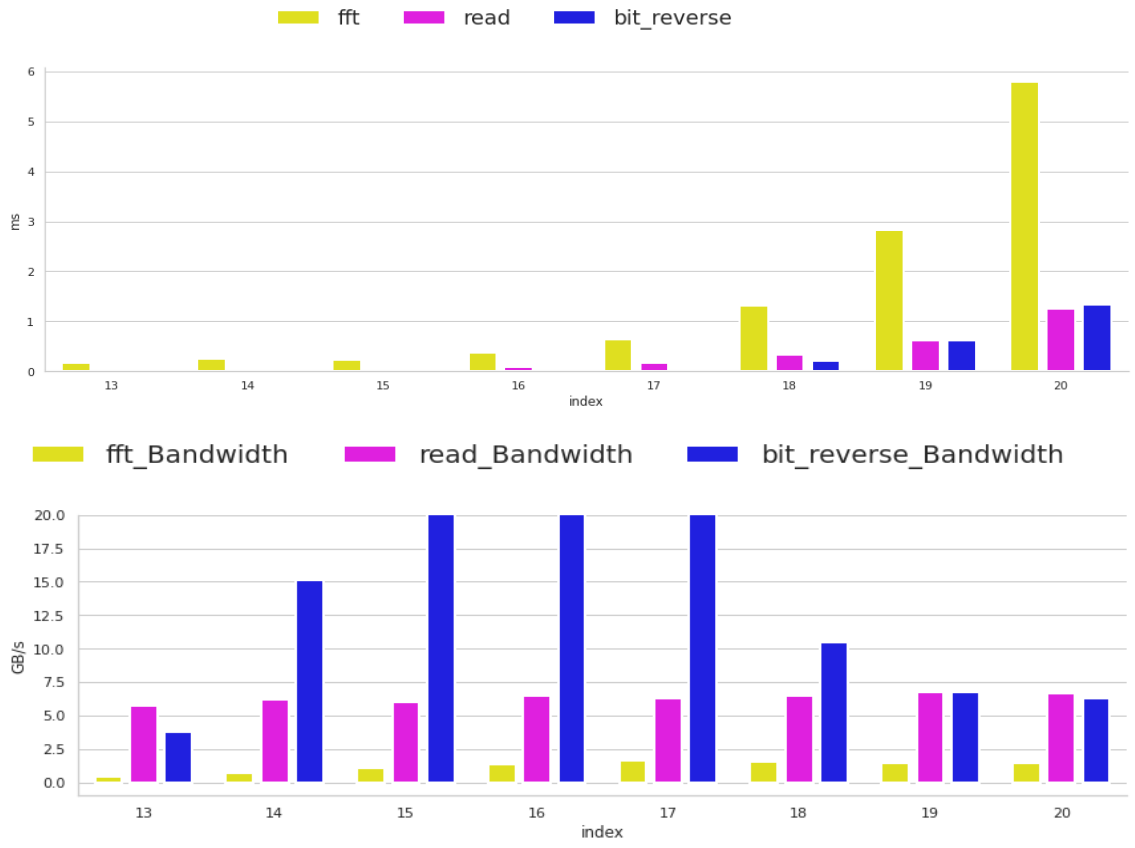
```
kernel void fft_3(global float2* input, global float2 *output,int length,int iter){
    const int gid = get_global_id(0);
    if (gid >= (length >> 1)) return;

    //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;

    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.y = sincos(angle,(float*)&wj);


    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] = odd_output;
}
```

3.3.7 Performance FFT 3

Come ipotizzato precedentemente l'algoritmo sembra essere limitato dagli accessi alla memoria non coalescenti,quindi questa implementazione si è concentrata sull'ottimizzare gli accessi alla memoria, riordinando l'array di samples prima delle computazioni. In questo modo, sopratutto nelle prime run, l'algoritmo tende ad effettuare accessi alla memoria molto vicini tra di loro, quindi diminuendo il problema degli accessi alla memoria non coalescenti.



Come si può notare sia dal grafo di runtime sia da quello del bandwitdth l'operazione di bitReverse ha velocizzato di molto l'operazione FFT,avvalorando l'ipotesi che l'algoritmo fosse limitato dagli accessi alla memoria poco ottimizati.

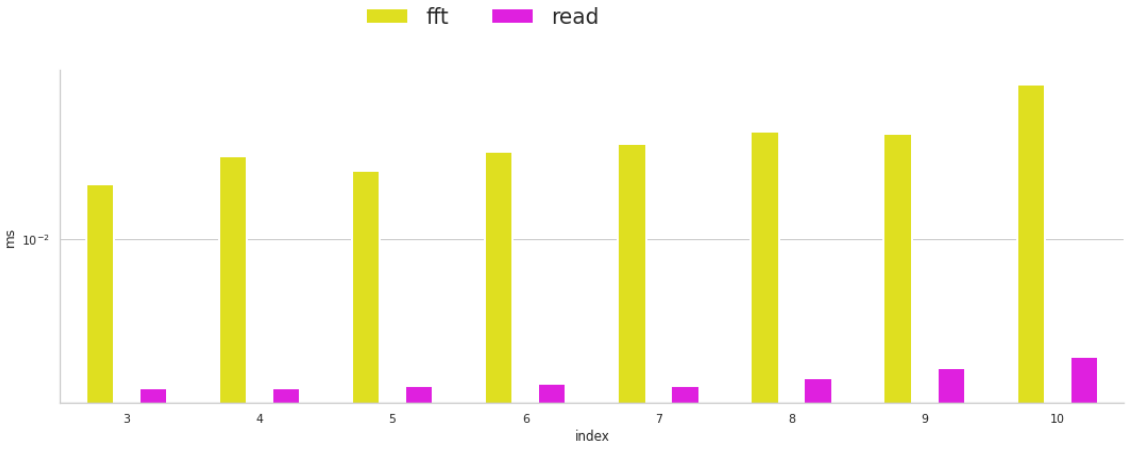


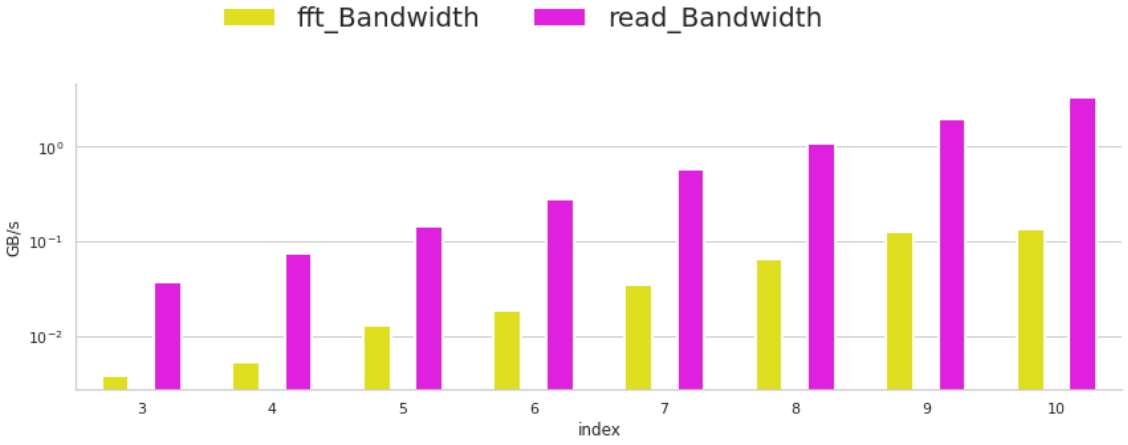
3.3.8 GPU Compact FFT 3

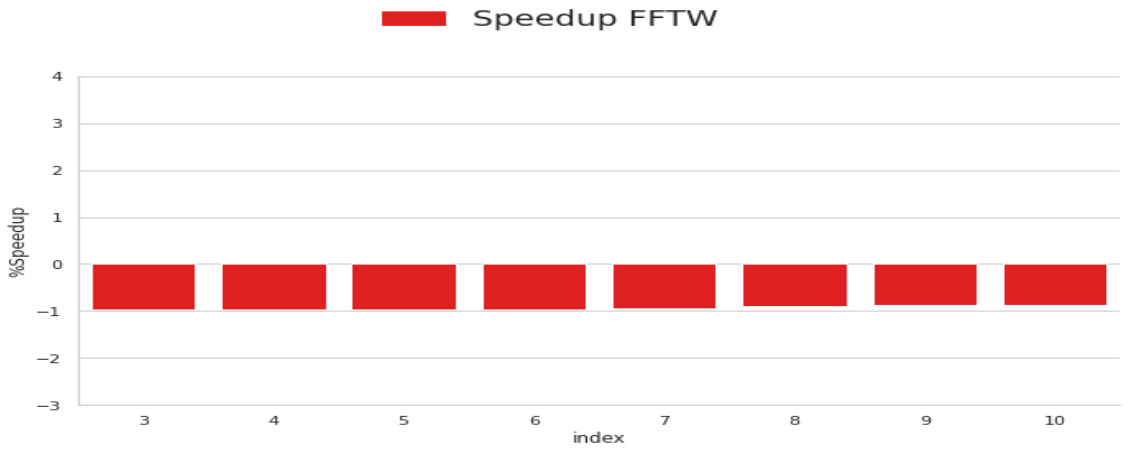
Come si è visto dai grafi precedenti per dimensione dei dati in input inferiori a 2^6 l'algoritmo sembra sprecare più tempo per il lancio dei singoli kernel che per la computazione stessa.Per ovviare al problema su input che rientrano in un singolo work-group è possibile eseguire le operazione usando un singolo kernel.

```
kernel void compact_fft_3(global float2* input, global float2 *output,
                          int length,int maxIter,local float2 * lmem){

    const int gid = get_global_id(0);
    if (gid >= (length >> 1)) return;

    const int gid_permutated = generatePermutation(gid,maxIter);
    const int gid_permutated_2 = generatePermutation(gid + (length >> 1),maxIter);

    lmem[gid] = input[gid_permutated];
    lmem[gid + (length >> 1)] = input[gid_permutated_2];

    float2* temp;

    for(int iter=1;iter<= maxIter;iter++){
        //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 j = (gid & (stride -1) ) * n_group;

        barrier(CLK_LOCAL_MEM_FENCE);
        const float2 even = lmem[id];
        const float2 odd = lmem[id_strided];

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

        wj.y = sincos(angle,(float*)&wj);

        const float2 wj_odd = cmult(wj,odd);

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

        barrier(CLK_LOCAL_MEM_FENCE);

        lmem[id] = even_output;
        lmem[id_strided] = odd_output;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    output[gid] = lmem[gid];
    output[gid + (length>>1)] = lmem[gid + (length>>1)];
}
```

Questa implementazione porta l'algoritmo ad essere molto più veloce dei precedenti su piccole dimensione di input, ma sempre più lento dell'algoritmo fftw. I grafi in questione sono in scala logaritmica per via delle grandi differenze dei valori.







3.3.9 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 é sempre eseguita in tutto l'array prima di eseguire l'algoritmo fft.

Alla fine di ogni esecuzione di ricomposizione si riordinano gli elementi,in modo tale nell'operazione di ricomposizione successiva l' i-esimo work-item acceda solamente alle locazione di memoria i ed $i + 1$,rendendo effettivamente gli accessi in memoria coalescenti.

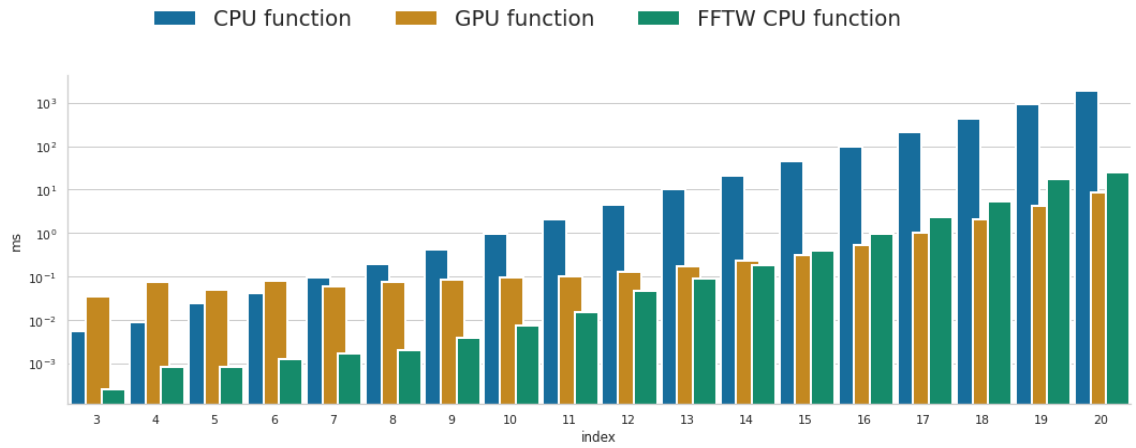
```
kernel void permutateArray(global float2* input,global float2* output,int lenght,int iter){
    const int gid = get_global_id(0);
    if(gid >= lenght/2) return;

    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;
}
```

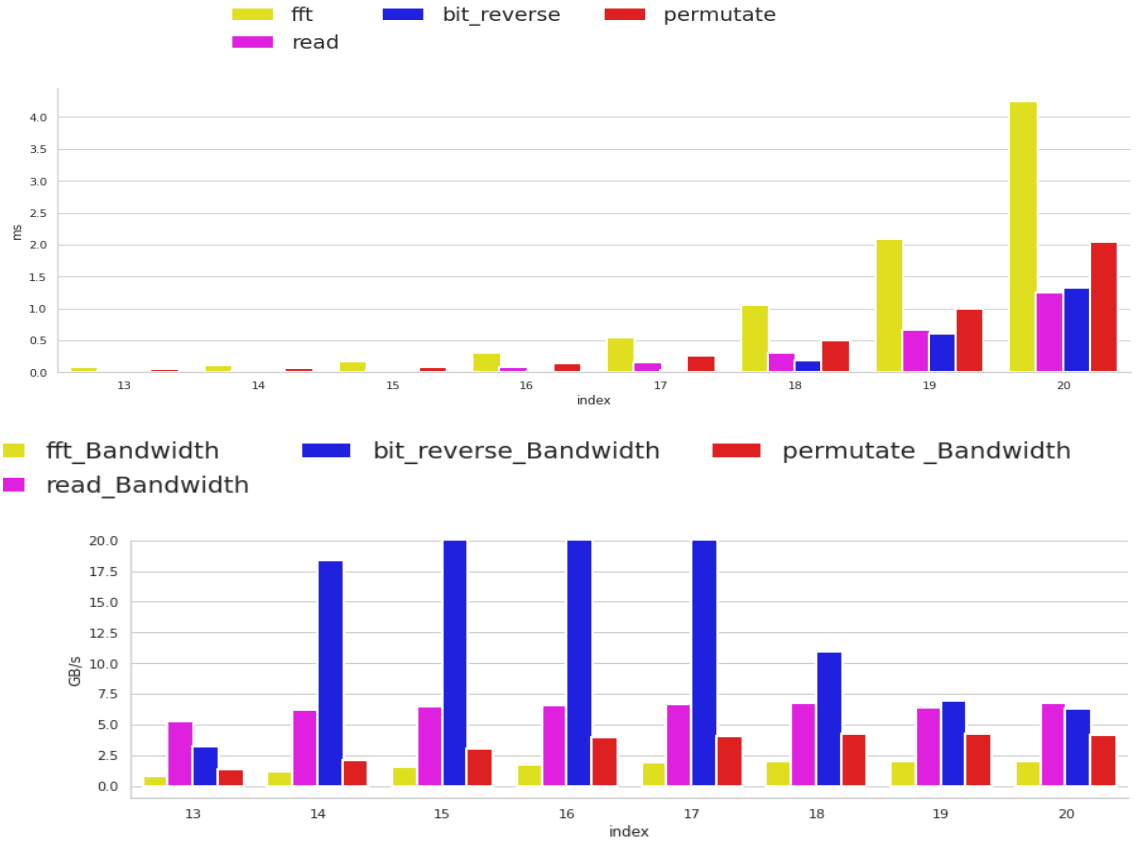
3.3.10 Performance FFT 4

L'algoritmo precedente permette di limitare il problema degli accessi alla memoria non coalescenti. 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 l'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.

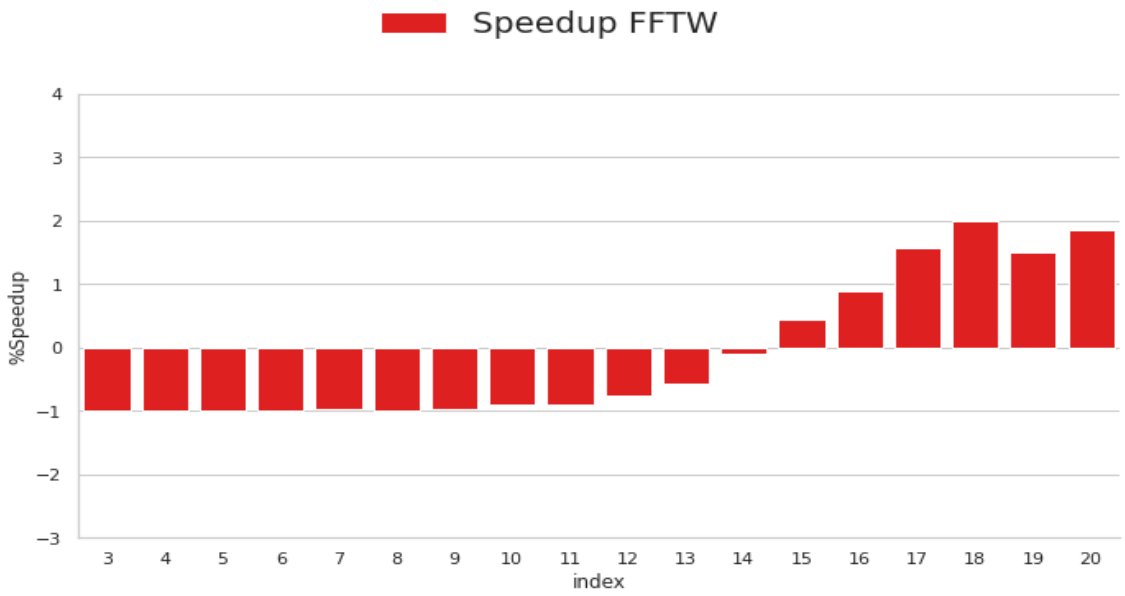


Come e‘ possibile notare l’operazione di permutazione velocizza l’operazione FFT rispetto alla precedente iterazione,avvalorando la mia ipotesi sul problema degli accessi alla memoria,pero questa operazione a sua volta aggiunge passi computazionali che portano l’algoritmo a non essere più’ veloce della precedente iterazione.



3.3.11 Speedup

Confrontato l’implementazione migliore,cioe FFT3,con l’algoritmo fftw é 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,é possibile affermare che per dimensioni elevate di sample,per l'esattezza maggiori di 2^{14}, é conveniente usare una gpu, ottenendo uno speedup fino ad un massimo di 3x rispetto anche a fftw, pero per piccoli samples di dati piu piccoli é 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 ORG] FFTW