Computer Architecture - Final Project

0612020 吳峻陞

Topic:

Audio Analysis in Frequency Spectrum with Fast Fourier Transform (FFT)

Environment:

1. Virtual Box in Ubuntu-12.04

   VM Setting: 2G

   CPU: Dual-Core

2. Visual Studio 2019 in Window10

   CPU: i7-6700

   Graphic Card: GTX1060 3G

   RAM: 16G

會有兩個環境是因為 FFT 需要 complex<double>的函式才有辦法執行正常運算，而 hw1 時提供的 VB 不支援 complex 運算，所以我在第一個環境確認以 g++編譯執行 FFT_CPU function 是正確的之後，將程式搬到 Visual Studio 上面以自己的電腦執行 cuda_related function 得到正常結果，期間的參數和函式都沒有變更

File Description:

資料夾中包含了兩個 visual studio project，一個是 FFT_method1，另一個是 FFT_method2，FFT_method1 是 hw1 中第一種資料切割的專案實作，而 FFT_metho2 是第二種資料切割，下面是兩個資料夾之內的檔案介紹：

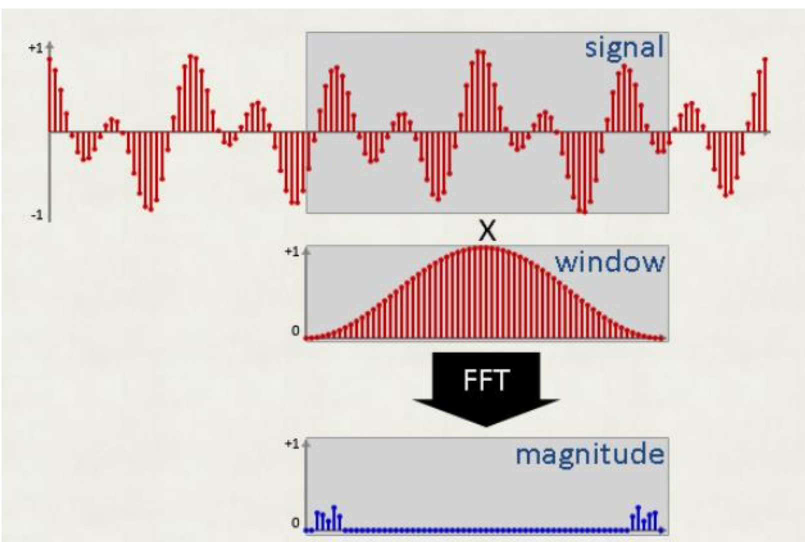
檔案	說明
kernel.cu	專案的主要執行檔(= main.cu)，裡面有對.wav 的音檔傳換(轉成 array)和 FFT_CPU function 以及對於 device 的資料傳遞
FFT_GPU_method1.cu	包含 FFT_GPU function 和第一種資料切割
FFT_GPU_method2.cu	包含 FFT_GPU function 和第二種資料切割
parameters.h	與 hw1 中相同設定 indexsave 和參數的標頭檔
re.wav	25 秒的 NCS 音樂，供 FFT 轉置運算
Result_methodx.dat	將經過 FFT 處理過的 complex 矩陣檔案存放
Demo.mkv	Demo 影片

Algorithm:

首先利用 File 將 audio 存入陣列，再將陣列轉成 complex<double> array 並除以 2^{bit-1} 來 normalize scale (bit = 16, channel = 2 => bit_total = 32)。

```cpp
const char* fileName = "re.wav";
const char* fileToSave = "result.dat";
FILE* fin = fopen(fileName, "rb");
short int* value = new short int[samples_count];
memset(value, 0, sizeof(short int) * samples_count);

//Reading data
for (int i = 0; i < samples_count; i++)
{
    fread(&value[i], sample_size, 1, fin);
}

//Time Counting
time_t start, end;
start = time(NULL);

//fft processing
printf("Processing FFT From CPU side\n");
complex<double> prefetch[samp];
complex<double> b[samp];
for (int i = 0; i < samp; i++) {
    prefetch[i] = complex<double>(value[i] / 32768., value[i] / 32768.);
}

```

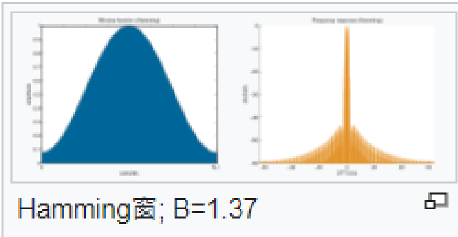
Preprocessing Data 之後先利用 hamming window 避免訊號直接送入 FFT 之後產生失真，接著用 Fast Fourier Transform 得到這段音頻的頻率分析。



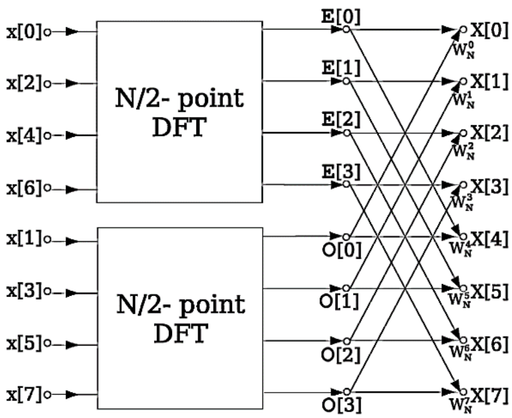
-> Abstract

Hamming Window Implementation

```cpp
//hamming_window
void Hamming_Window(double window[], int frame_size) {
    for (int i = 0; i < frame_size; i++) window[i] = 0.54f - 0.46f * cos((float)i * 2.0f * PI / float((frame_size - 1)));
}
```



Hamming Window Example



FFT Structure

```
//reserve bits for fft processing
unsigned int bitReverse(unsigned int x, int log2n) {
    int n = 0;
    for (int i = 0; i < log2n; i++) {
        if (x & (1 << i)) n |= 1 << (log2n - 1 - i);
    }
    return n;
}

//fft
template<class Iter_T>
void fft(Iter_T a, Iter_T b, int log2n)
{
    typedef typename iterator_traits<Iter_T>::value_type complex;
    const complex J(0, 1);
    int n = 1 << log2n;

    for (int i = 0; i < n; ++i) b[bitReverse(i, log2n)] = a[i];
    for (int s = 1; s <= log2n; ++s) {
        int m = 1 << s;
        int m2 = m >> 1;
        complex w(1, 0);
        complex wm = exp(-J * (PI / m2));
        for (int j = 0; j < m2; ++j) {
            for (int k = j; k < n; k += m) {
                complex t = w * b[k + m2];
                complex u = b[k];
                b[k] = u + t;
                b[k + m2] = u - t;
                w *= wm;
            }
        }
    }
}
```

FFT code Implementation

最後把經過 FFT 處理過的 complex 存入 result.dat 就完成處理了。

```
    FILE* fout = fopen(fileToSave, "w");
    for (int i = 0; i < samp; i++)
    {
        fprintf(fout, "%f + %fi\n", real(b[i]), imag(b[i]));
    }
    fclose(fin);
    fclose(fout);
```

CPU main function

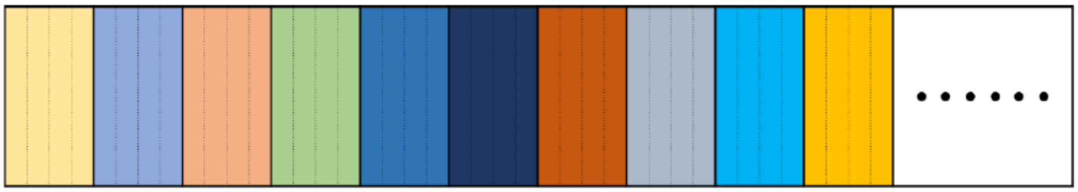
```
int main()
{
    //Create memory space
    IndexSave* indsave = new IndexSave[SIZE];
    short int* value = new short int[SIZE];
    memset(value, 0, sizeof(short int) * SIZE);

    /* CPU side*/
    value = process();

    /* GPU side*/
    GPU_kernel(value, indsave);
}
```

GPU Kernel Function

```

    // Allocate Memory Space on Device
    cudaMalloc((void**)&value, sizeof(int) * 2*SIZE);
    cudaMalloc((void**)&fft_, sizeof(thrust::complex<double>) * SIZE);

    // Allocate Memory Space on Device (for observation)
    cudaMalloc((void**)&dInd, sizeof(IndexSave) * SIZE);

    // Copy Data to be Calculated
    cudaMemcpy(value, v, sizeof(int) * 2*SIZE, cudaMemcpyHostToDevice);
    cudaMemcpy(fft_, f_, sizeof(thrust::complex<double>) * SIZE, cudaMemcpyHostToDevice);

    // Copy Data (indsave array) to device
    cudaMemcpy(dInd, indsave, sizeof(IndexSave) * SIZE, cudaMemcpyHostToDevice);

    // Start Timer
    cudaEventRecord(start, 0);

    // Launch Kernel
    dim3 dimGrid(8);
    dim3 dimBlock(128);
    cuda_kernel<<<dimGrid, dimBlock>>>(value, dInd, fft_);

    // Stop Timer


    // Copy Output back
    cudaMemcpy(v, value, sizeof(int) * 2*SIZE, cudaMemcpyDeviceToHost);
    cudaMemcpy(f_, fft_, sizeof(thrust::complex<double>) * SIZE, cudaMemcpyDeviceToHost);
    cudaMemcpy(indsave, dInd, sizeof(IndexSave) * SIZE, cudaMemcpyDeviceToHost);

    // Release Memory Space on Device
    cudaFree(value);
    cudaFree(fft_);
    cudaFree(dInd);

```

Data Decomposition:

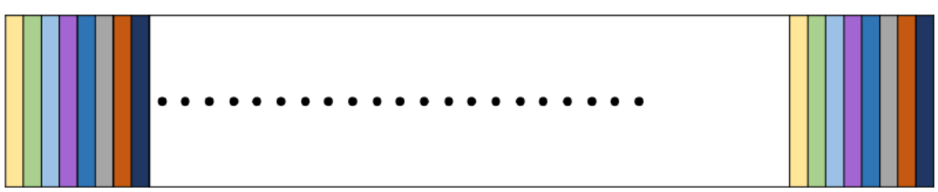
第一種方法是將 data 全部平均分配給 thread(如下圖)，讓每個 thread 處理相同數量的 task，這種方式因為一次性的 loading 較重使每個 thread 的執行速度會大大影響 CPU Time，我把<dimGrid, dimBlock> = <8, 128>，並用上述的方法執行 cuda file。

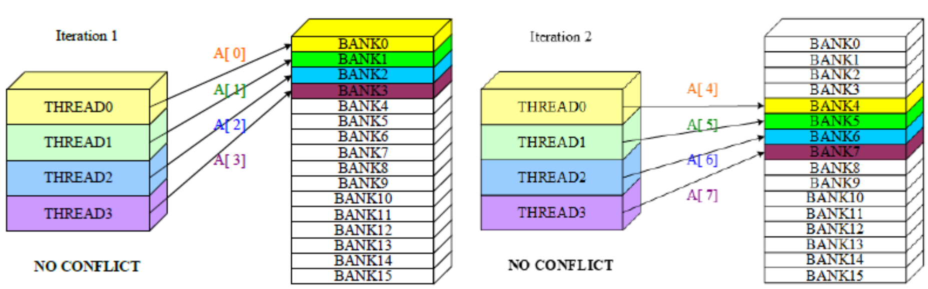


Cuda_method1 (process_gpu = FFT processing)

```
__global__ void cuda_kernel(int* value, IndexSave* dInd, thrust::complex<double>* fft_)
{
    // complete cuda kernel function
    int i = 0;
    int TotalThread = blockDim.x * gridDim.x;
    int stripe = SIZE / TotalThread;
    int head = (blockIdx.x * blockDim.x + threadIdx.x) * stripe;
    int LoopLim = head + stripe;

    for (i = head; i < LoopLim; i++) {
        process_gpu(value, fft_);
        dInd[i].blockInd_x = blockIdx.x;
        dInd[i].threadInd_x = threadIdx.x;
        dInd[i].head = head;
        dInd[i].stripe = stripe;
    }
};
```

第二種方法是將 data 依序給各個 thread，平均地把 loading 分給每個 thread，每個 thread 的總工作量相同但因為 sequentially assigned 的關係可以在 data 較多時讓擁有比第一種方法好的效率。





Cuda_method2 (process_gpu = FFT processing)

```
__global__ void cuda_kernel(int* value, IndexSave* dInd, thrust::complex<double>* fft_)
{
    // complete cuda kernel function
    int i = 0;
    int stripe = blockDim.x * gridDim.x;
    int head = blockIdx.x * blockDim.x + threadIdx.x;

    for (i = head; i < SIZE; i += stripe) {
        process_gpu(value, fft_);
        dInd[i].blockInd_x = blockIdx.x;
        dInd[i].threadInd_x = threadIdx.x;
        dInd[i].head = head;
        dInd[i].stripe = stripe;
    }
};
```

Result:

Method1 和 Method2 的執行時間和檔案讀取如下

```
File Type: RIFF?I
File Size: 4800178
WAV Marker: WAVEfmt
Format Name: fmt
Format Length: 16
Format Type: 1
Number of Channels: 2
Sample Rate: 48000
Sample Rate * Bits/Sample * Channels / 8: 192000
Bits per Sample * Channels / 8: 4
Bits per Sample: 16
id      size
LIST    134
data    4800000
Samples count = 2400000
.........................................................
Processing FFT From CPU side
Execution of CPU is Completed!
CPU Time: 51.51858ms
.........................................................
Processing FFT From GPU Side
Execution of Method1 is Completed!
GPU Time: 10094.1299 ms
.........................................................
```

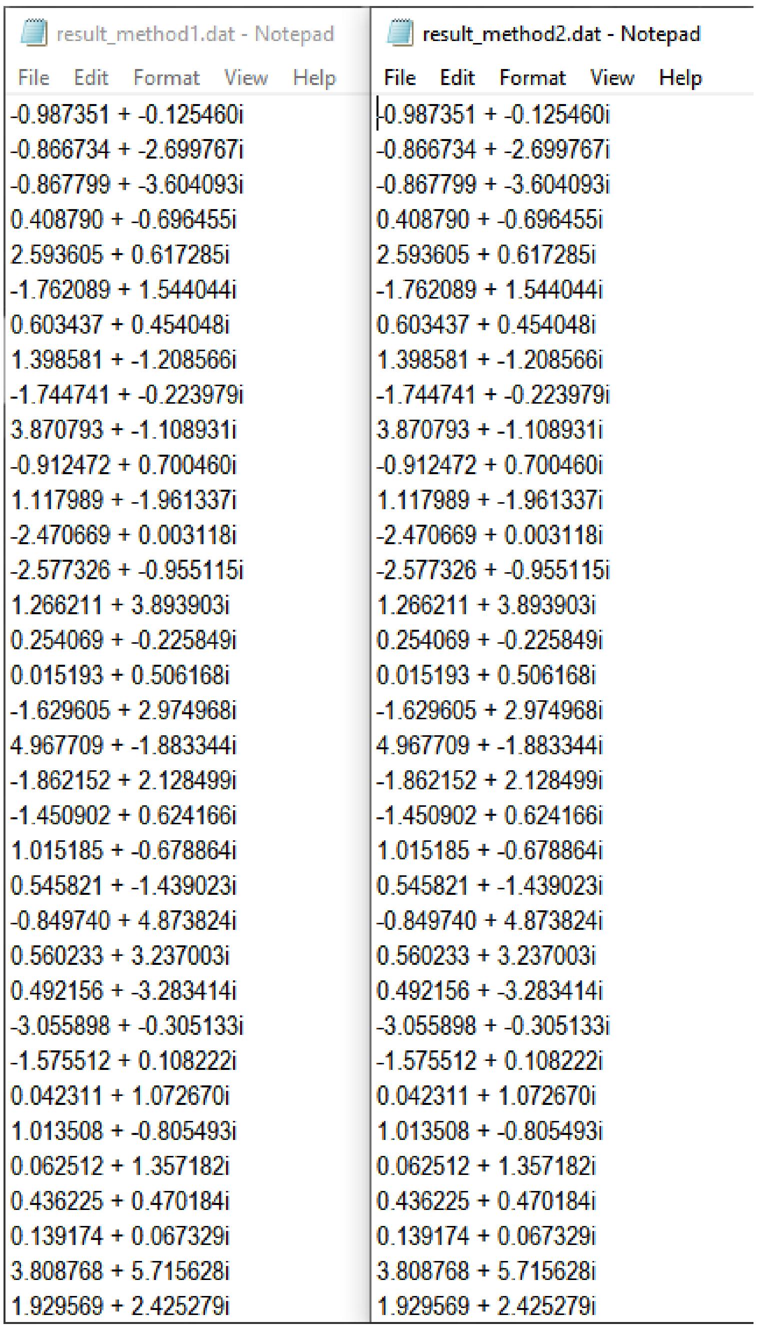
-> method1

```
File Type: RIFF?I
File Size: 4800178
WAV Marker: WAVEfmt
Format Name: fmt
Format Length: 16
Format Type: 1
Number of Channels: 2
Sample Rate: 48000
Sample Rate * Bits/Sample * Channels / 8: 192000
Bits per Sample * Channels / 8: 4
Bits per Sample: 16
id      size
LIST    134
data    4800000
Samples count = 2400000
.........................................................
Processing FFT From CPU side
Execution of CPU is Completed!
CPU Time: 54.54056ms
.........................................................
Processing FFT From GPU Side
Execution of Method2 is Completed!
GPU Time: 10049.9033 ms
.........................................................
```

-> method2

	Method1	Method2
CPU Time (ms)	51.51858	54.54056
GPU Time (ms)	10094.1299	10049.9033

為了算力及避免 array overlap，我取 2048 筆資料並以每 2^8 = 256 個點做 hamming window 來執行 n = 11 的 FFT，經過 FFT 的音訊則會被處理為 complex array 存入.dat 中，如下所示。



result_method1.dat - Notepad
File Edit Format View Help
-0.987351 + -0.125460i
-0.866734 + -2.699767i
-0.867799 + -3.604093i
0.408790 + -0.696455i
2.593605 + 0.617285i
-1.762089 + 1.544044i
0.603437 + 0.454048i
1.398581 + -1.208566i
-1.744741 + -0.223979i
3.870793 + -1.108931i
-0.912472 + 0.700460i
1.117989 + -1.961337i
-2.470669 + 0.003118i
-2.577326 + -0.955115i
1.266211 + 3.893903i
0.254069 + -0.225849i
0.015193 + 0.506168i
-1.629605 + 2.974968i
4.967709 + -1.883344i
-1.862152 + 2.128499i
-1.450902 + 0.624166i
1.015185 + -0.678864i
0.545821 + -1.439023i
-0.849740 + 4.873824i
0.560233 + 3.237003i
0.492156 + -3.283414i
-3.055898 + -0.305133i
-1.575512 + 0.108222i
0.042311 + 1.072670i
1.013508 + -0.805493i
0.062512 + 1.357182i
0.436225 + 0.470184i
0.139174 + 0.067329i
3.808768 + 5.715628i
1.929569 + 2.425279i

result_method2.dat - Notepad
File Edit Format View Help
-0.987351 + -0.125460i
-0.866734 + -2.699767i
-0.867799 + -3.604093i
0.408790 + -0.696455i
2.593605 + 0.617285i
-1.762089 + 1.544044i
0.603437 + 0.454048i
1.398581 + -1.208566i
-1.744741 + -0.223979i
3.870793 + -1.108931i
-0.912472 + 0.700460i
1.117989 + -1.961337i
-2.470669 + 0.003118i
-2.577326 + -0.955115i
1.266211 + 3.893903i
0.254069 + -0.225849i
0.015193 + 0.506168i
-1.629605 + 2.974968i
4.967709 + -1.883344i
-1.862152 + 2.128499i
-1.450902 + 0.624166i
1.015185 + -0.678864i
0.545821 + -1.439023i
-0.849740 + 4.873824i
0.560233 + 3.237003i
0.492156 + -3.283414i
-3.055898 + -0.305133i
-1.575512 + 0.108222i
0.042311 + 1.072670i
1.013508 + -0.805493i
0.062512 + 1.357182i
0.436225 + 0.470184i
0.139174 + 0.067329i
3.808768 + 5.715628i
1.929569 + 2.425279i