HIGH PERFORMANCE COMPUTING

Project - CUDA

Topic - Using Discrete Fourier Transform for finding the normalized frequency of a random wave and parallelizing the code using CUDA.

CONTENTS

S.no.	Topic	Page no.
1.	Introduction	2
2.	Formulae used	4
3.	Functions used	4
4.	Procedure	5
5.	Sample inputs and outputs	6
6.	Execution time, speed-up and parallel fraction	9
7.	Code	11
8.	Profiling	13

Bhaavanaa Thumu - CED17I021

1. Introduction

About discrete fourier transform :

The discrete fourier transform converts a finite sequence of equally-spaced samples of a function into a same-length sequence of equally-spaced samples of the discrete-time Fourier transform (DTFT), which is a complex valued function of frequency. The DFT is therefore said to be a frequency domain representation of the original input sequence.

The DFT is the most important discrete transform, used to perform Fourier analysis in many practical applications. In digital signal processing, the function is any quantity or signal that varies over time, such as the pressure of a sound wave, a radio signal, or daily temperature readings, sampled over a finite time interval. In image processing, the samples can be the values of pixels along a row or column of a raster image. The DFT is also used to efficiently solve partial differential equations, and to perform other operations such as convolutions or multiplying large integers.

Since it deals with a finite amount of data, it can be implemented in computers by numerical algorithms or even dedicated hardware. The advantage of DFT over FFT is the restriction of 2^n samples and fixed/discrete sample rates for the FFT.

About power spectrum :

For a given signal, the power spectrum gives a plot of the portion of a signal's power (energy per unit time) falling within given frequency bins. The most common way of generating a power spectrum is by using the discrete fourier transform, and other techniques such as the maximum entropy method can also be used.

About normalized frequency :

Normalised frequency is frequency in Hz (or more generically cycles/second or some other unit) divided by the sample frequency of your signal in Hz (or the same units as your original frequency).

2. Formulae used

Speed-up = T(1) / T(P)

Parallel fraction, f = (1 - T(P)/T(1)) / (1 - (1/P))

where,

T(1) - time taken for serial execution

T(P) - time taken for parallel execution

P - number of processes/threads/processors

Formula for finding DFT of a sample -

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}$$

3. Functions used

cudaMalloc - allocates a block of memory

cudaMemcpy - copies ‘n’ characters from source to destination memory area.

4. Procedure

The steps for performing the task of finding the normalized frequency of a random wave using discrete fourier transform are -

a. Produce an array of N random numbers within a range, e.g. -1 to 1 using the rand() function. This would be our array of samples.
b. Define a struct which contains real and imaginary double precision numbers, so that it can store the real and imaginary coefficients of the output of DFT. Initialize the real and imaginary values to 0.
c. Find the DFT of all the samples present in the array.

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}$$

d. For every element in the array of structures containing the coefficients of the DFT obtained above, find the power spectrum of it. It is the sum of the square of the real coefficient and the square of the imaginary coefficient of the DFT.
e. Find the sum of the product of the frequency and the power spectrum value of all the elements. Here,

   frequency of an element = 2*PI*k/N

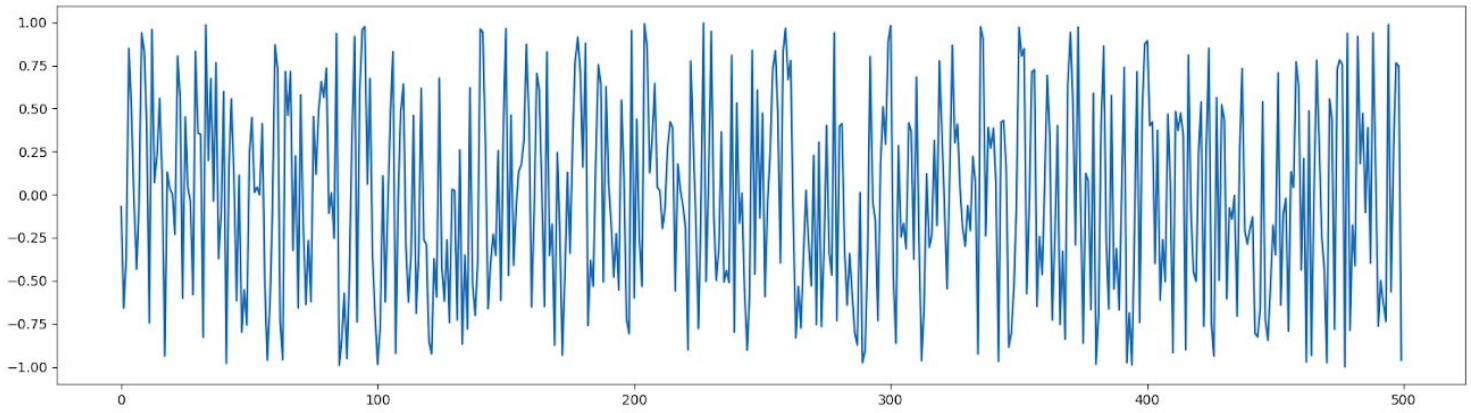
f. Find the sum of the power spectrum values.
g. The normalized frequency of the random wave generated above is the sum of the product of the frequency and the power spectrum value of all the elements (obtained in step e) divided by the sum of the power spectrum values (obtained in step f).

5. Sample inputs and outputs

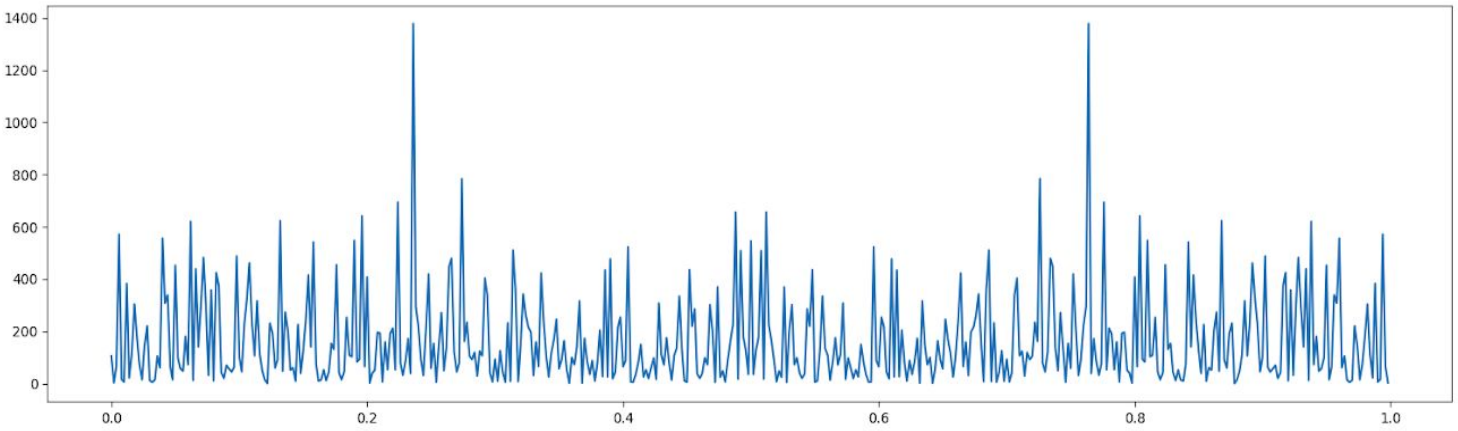
Example 1:

Input :- No. of random samples : 500

Range of the random samples : -1 to 1



Random input signal



Power spectrum vs k/N of the random input signal

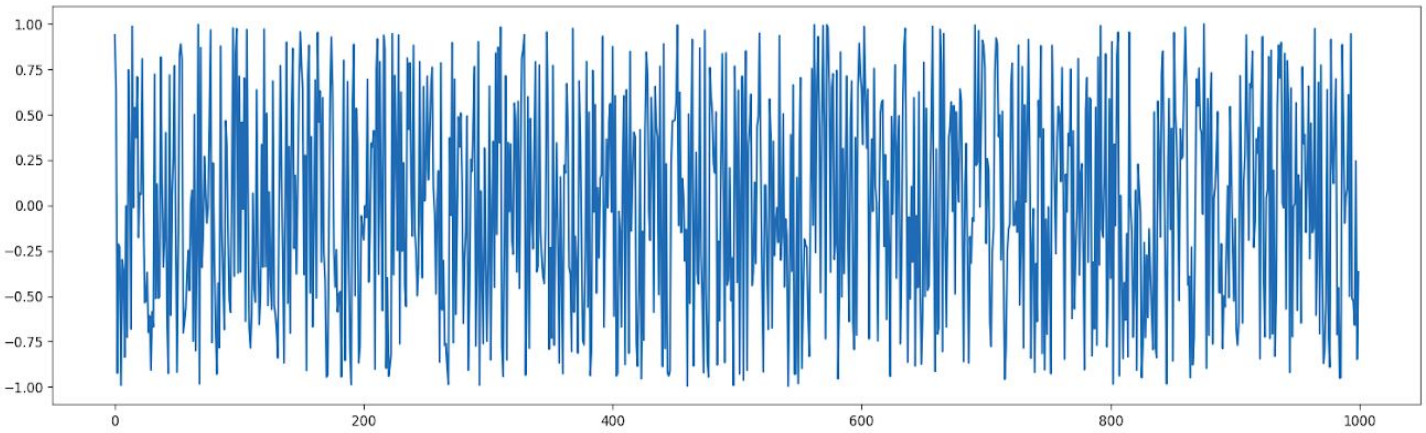
Output :- Normalized frequency of the wave : 3.137617 Hz

Time taken for the program for the given number of processes.

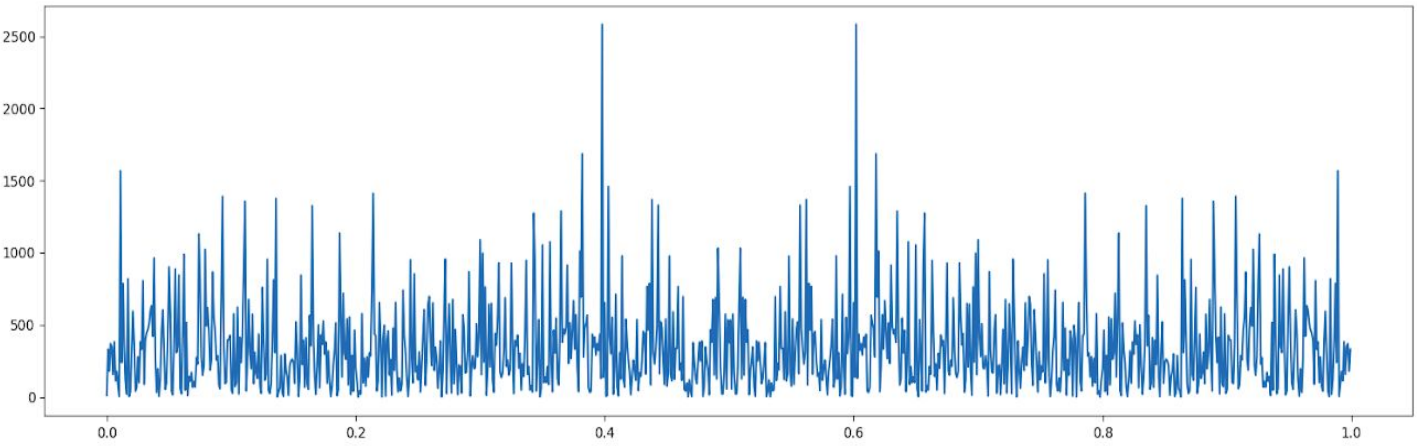
Example 2:

Input :- No. of random samples : 1,000

Range of the random samples : -1 to 1



Random input signal



Power spectrum vs k/N of the random input signal

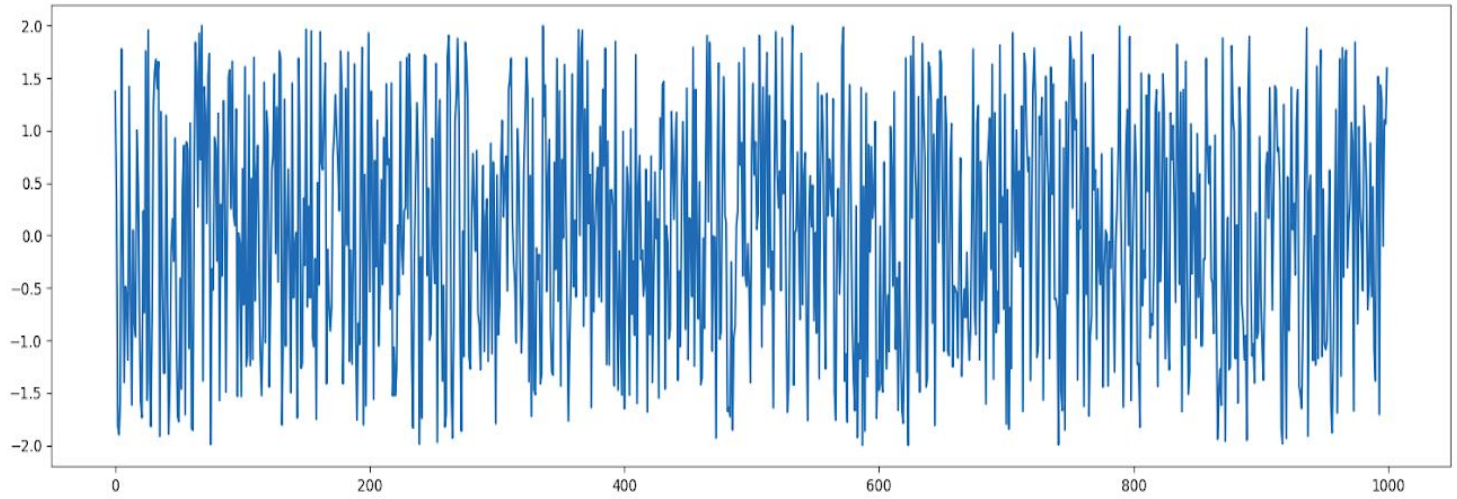
Output :- Normalized frequency of the wave : 3.141485 Hz

Time taken for the program for the given number of processes.

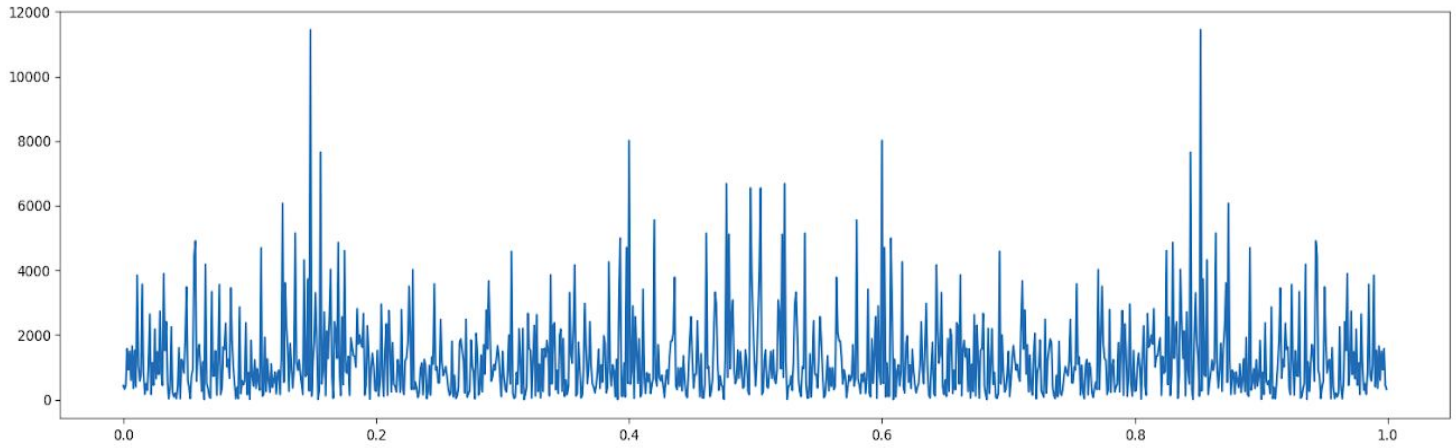
Example 3:

Input :- No. of random samples : 1000

Range of the random samples : -2 to 2



Random input signal



Power spectrum vs k/N of the random input signal

Output :- Normalized frequency of the wave : 3.140520 Hz

Time taken for the program for the given number of processes.

6. Execution time, speed-up and parallel fraction

For the below analysis, the program contains 10,00,000 sample points in the random wave. The range for these random points, considered here, is -1 to 1.

The number of processors initialized and the corresponding execution time, speed-up and parallel fraction is as shown in the table below.

No. of processes	Execution time	Speed-up	Parallel fraction
1	131.711246	1	#DIV/0!
2	132.350666	0.9951687436	-0.009709421472
4	65.922643	1.997966708	0.6659882128
8	33.052886	3.984863712	0.856057587
16	16.692128	7.890620417	0.93148507
32	8.578441	15.35375087	0.9650264106
64	4.625175	28.47702973	0.9801995968
128	2.949827	44.65049849	0.9853015014
256	2.435393	54.08213212	0.985358662
500	**2.015343**	**65.35425781**	**0.9866721224**

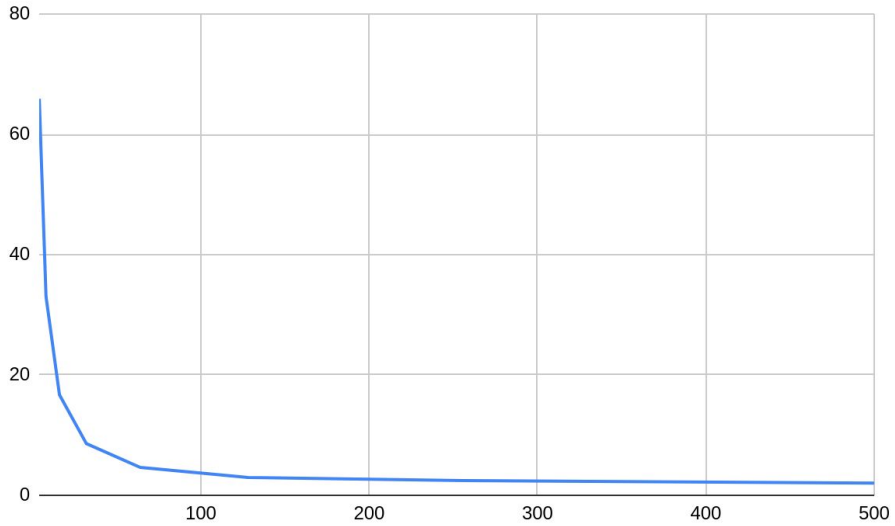
Hence, in this case, the optimal number of processes is 500. In this case-

Execution time = 2.015343

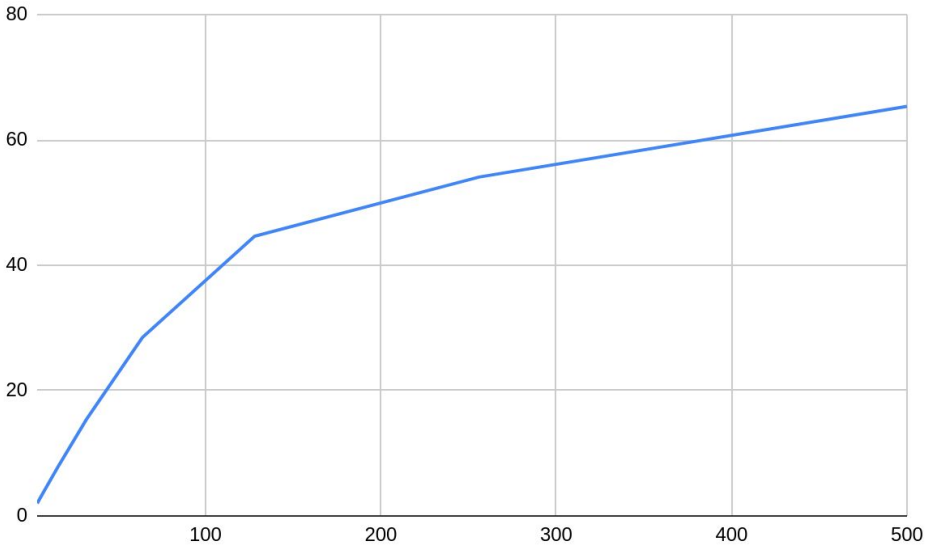
Speed-up = 65.35425781

Parallel fraction = 0.9866721224

The plot below is the graph between the number of processes and execution time.



The plot below is the graph between the number of processes and speed-up.



In this case, finding the DFT of the elements is parallelized. The load is equally distributed among all the threads and the leftover load is also done by the master process.

7. Code

```
%%cu

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include <sys/time.h>

#define PI 3.14159265

#define N 10000                                 // the number of random samples
#define thread_count 500                        // the number of threads initialized


__global__ void dft(double *d_samples, double *d_dft_real, double *d_dft_img, double *d_k_by_N, double *d_power_spectrum, double *d_sum_power_spectrum, double *d_sum_freq_power)
{
    int load = N/thread_count;                                          // calculating the load per thread

    __shared__ double partial_sum_power_spectrum[thread_count];         // initializing a partial sum variable which will find the partial sums of the power_spectrum for the load assigned to it
    __shared__ double partial_sum_freq_power[thread_count];             // initializing a partial sum variable which will find the partial sums of the freq_power for the load assigned to it

    partial_sum_power_spectrum[threadIdx.x] = 0;                        // initializing the partial sum variables to 0
    partial_sum_freq_power[threadIdx.x] = 0;

    if(threadIdx.x != thread_count-1)                                   // if it is not the last thread, it will do load amt of work
    {
        for(int k=threadIdx.x*load; k<(threadIdx.x+1)*load; k++)        // for every element in the load
        {
            for(int n=0; n<N; n++)
            {
                d_dft_real[k] += d_samples[n] * cos((2*PI*k*n)/N);      // find the real and img part of the dft
                d_dft_img[k] += d_samples[n] * sin((2*PI*k*n)/N);
            }

            d_k_by_N[k] = (double)k/N;                                  // find the k/N value

            d_power_spectrum[k] = d_dft_real[k]*d_dft_real[k] + d_dft_img[k]*d_dft_img[k];          // find the power spectrum value

            partial_sum_power_spectrum[threadIdx.x] += d_power_spectrum[k];                         // obtain the partial power_spectrum sum of the load assigned to the thread
            partial_sum_freq_power[threadIdx.x] += 2*PI*d_k_by_N[k]*d_power_spectrum[k];            // obtain the partial freq_power sum of the load assigned to the thread
        }
    }
    else                                                                // if it is last thread, then it will do load+leftover work
    {
        for(int k=threadIdx.x*load; k<N; k++)
        {
            for(int n=0; n<N; n++)
            {
                d_dft_real[k] += d_samples[n] * cos((2*PI*k*n)/N);
                d_dft_img[k] += d_samples[n] * sin((2*PI*k*n)/N);
            }

            d_k_by_N[k] = (double)k/N;

            d_power_spectrum[k] = d_dft_real[k]*d_dft_real[k] + d_dft_img[k]*d_dft_img[k];

            partial_sum_power_spectrum[threadIdx.x] += d_power_spectrum[k];
            partial_sum_freq_power[threadIdx.x] += 2*PI*d_k_by_N[k]*d_power_spectrum[k];
        }
    }

    __syncthreads();

    if(threadIdx.x == 0)                                                // if it is the master process
    {
        *d_sum_power_spectrum = 0;                                      // initialize the total sums to 0
        *d_sum_freq_power = 0;

        for(int i=0; i<thread_count; i++)                               // find total sum = sum of partial sums
        {
            *d_sum_power_spectrum += partial_sum_power_spectrum[i];
            *d_sum_freq_power += partial_sum_freq_power[i];
        }
    }
}
```

```
int main()
{
    struct timeval t1, t2;                                                          // for keeping track of the startand end time

    double *samples, *d_samples;                                                    // array which contains the sample values
    double *dft_real, *d_dft_real, *dft_img, *d_dft_img;                            // array which stores the real and img part of the DFT of the sample
    double *k_by_N, *d_k_by_N, *power_spectrum, *d_power_spectrum;                  // arrays which store the k/N and power spectrum
    double sum_power_spectrum, *d_sum_power_spectrum, sum_freq_power, *d_sum_freq_power;      // for storing the final sum of the power spectrum and the total frequency

    samples = (double*)malloc(sizeof(double) * N);                                  // allocating space for samples in the host
    dft_real = (double*)malloc(sizeof(double) * N);                                 // allocating space for dft_real in the host
    dft_img = (double*)malloc(sizeof(double) * N);                                  // allocating space for dft_img in the host
    k_by_N = (double*)malloc(sizeof(double) * N);                                   // allocating space for k_by_N in the host
    power_spectrum = (double*)malloc(sizeof(double) * N);                           // allocating space for power_spectrum in the host

    cudaMalloc((void**)&d_samples, sizeof(double) * N);                             // allocating space for d_samples in the device
    cudaMalloc((void**)&d_dft_real, sizeof(double) * N);                            // allocating space for d_dft_real in the device
    cudaMalloc((void**)&d_dft_img, sizeof(double) * N);                             // allocating space for d_dft_img in the device
    cudaMalloc((void**)&d_k_by_N, sizeof(double) * N);                              // allocating space for d_k_by_N in the device
    cudaMalloc((void**)&d_power_spectrum, sizeof(double) * N);                      // allocating space for d_power_spectrum in the device

    cudaMalloc((void**)&d_sum_power_spectrum, sizeof(double));                      // allocating space for sum_power_spectrum in the device
    cudaMalloc((void**)&d_sum_freq_power, sizeof(double));                          // allocating space for sum_freq_power in the device

    for(int i=0; i<N; i++)
    {
        samples[i] = ((float)rand()/RAND_MAX)*(float)(2.0) - 1;                     // assigning random values to samples
        // samples[i] = i;
        dft_real[i] = 0;                                                            // initializing the elements of the other arrays to 0
        dft_img[i] = 0;
        k_by_N[i] = 0;
        power_spectrum[i] = 0;
    }

    gettimeofday(&t1, 0);                                                           // obtaining the start time

    cudaMemcpy(d_samples, samples, sizeof(double) * N, cudaMemcpyHostToDevice);     // copying the host variables to the device variables using cudaMemcpy
    cudaMemcpy(d_dft_real, dft_real, sizeof(double) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_dft_img, dft_img, sizeof(double) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_k_by_N, k_by_N, sizeof(double) * N, cudaMemcpyHostToDevice);
    cudaMemcpy(d_power_spectrum, power_spectrum, sizeof(double) * N, cudaMemcpyHostToDevice);

    dft<<<1,thread_count>>>(d_samples, d_dft_real, d_dft_img, d_k_by_N, d_power_spectrum, d_sum_power_spectrum, d_sum_freq_power);        // function call

    cudaMemcpy(&sum_power_spectrum, d_sum_power_spectrum, sizeof(double), cudaMemcpyDeviceToHost);              // copying the results from the device variables to the host variables
    cudaMemcpy(&sum_freq_power, d_sum_freq_power, sizeof(double), cudaMemcpyDeviceToHost);

    gettimeofday(&t2, 0);                                                           // obtaining the end time

    float normalized_freq = (float)sum_freq_power/sum_power_spectrum;               // obtaining the normalized frequency

    printf("The normalized frequency of the random wave is %f\n\n", normalized_freq);

    double time = (1000000.0*(t2.tv_sec-t1.tv_sec) + t2.tv_usec-t1.tv_usec)/1000000.0;        // calculating the time taken
    printf("time taken : %lf sec\n", time);

    return 0;
}
```

8. Profiling

Gprof

Profiling tool is gprof from the GNU binutils package. It uses both sampling and instrumentation and gives us a flat profile and call graph which gives various insights in the users code.

Commands used:

- gcc -pg sample.c
- ./a.out
- gprof -b a.out gmon.out

The above commands give us a flat profile and call graph.

Flat profile -

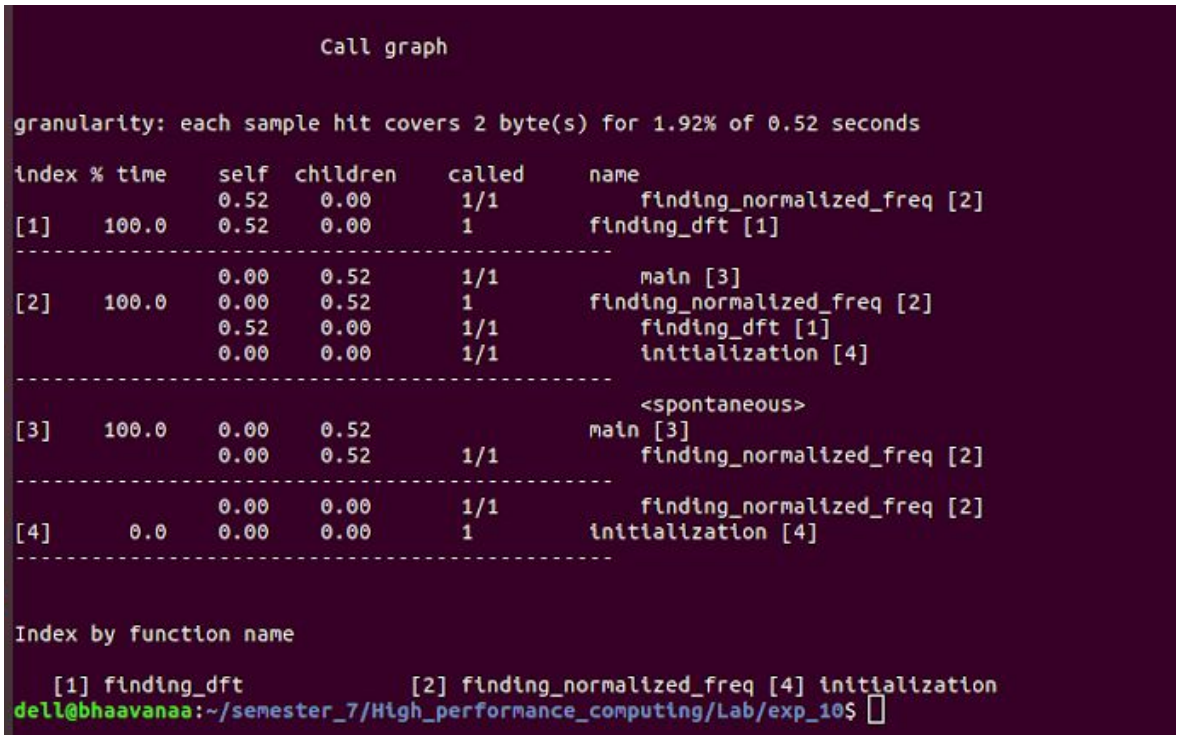
```
dell@bhaavanaa:~/semester_7/High_performance_computing/Lab/exp_10$ gprof -b ./a.out gmon.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 98.38      0.52     0.52        1   521.40   521.40  finding_dft
  0.00      0.52     0.00        1     0.00   521.40  finding_normalized_freq
  0.00      0.52     0.00        1     0.00     0.00  initialization
```

The flat profile contains information about execution times of all the program's functions and how often they were called.

Inference : By correlating flat profile and call graph, we see that 98% of the time is spent in find_dft() function, (called by the finding_normalized_freq() function, which is called by the main() function), out of the three functions (initialization(), finding_dft() and finding_normalized_freq()), even though it

is called once (which rules out the possibility that more calls take more time). So there can be a potential hotspot inside this function.

Call graph -

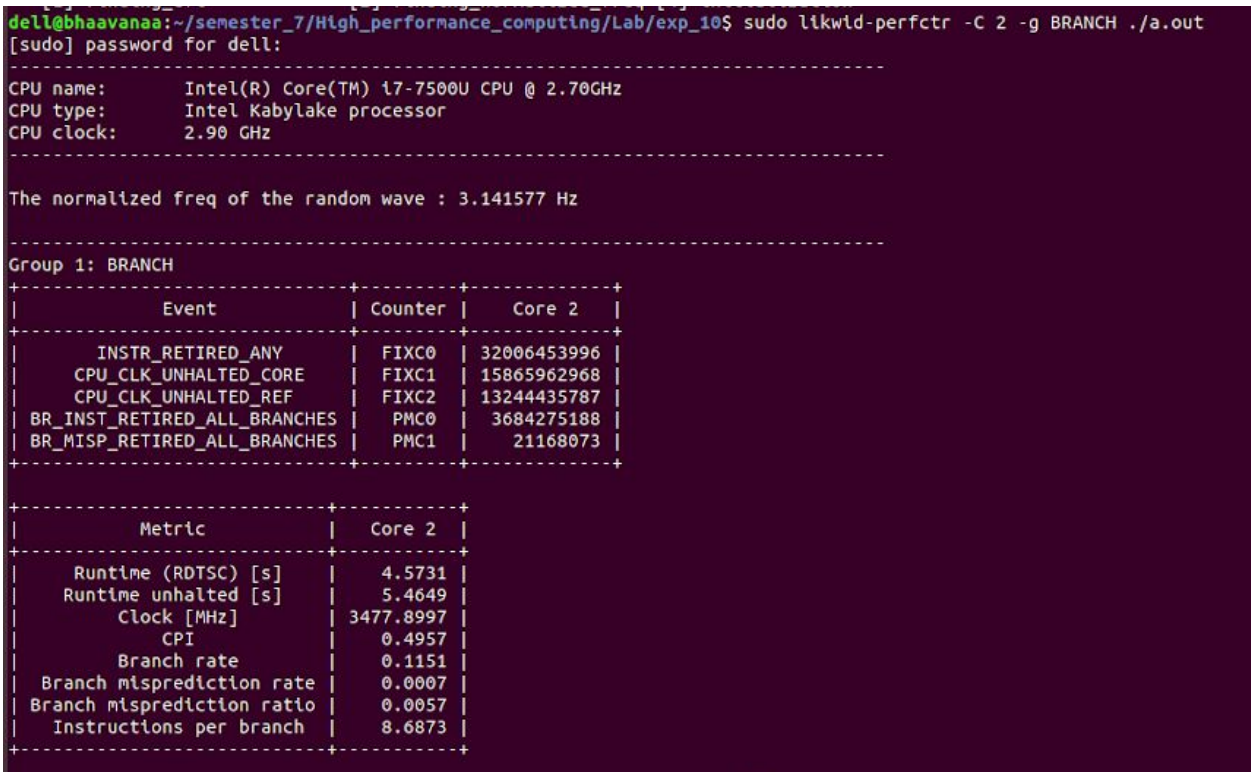


Call graph is in depth analysis of flat profile. It basically says what are all the functions which called it and what all functions it calls and other insights.

Inference : We can say that the finding_dft() function seems to be the hotspot in the code and can be a potential bottleneck if the input is large in size.

Likwid

A mini tool suite with various performance an\lysis measures. LIKWID stands for "***Like I Knew What I'm Doing***". It is an easy to use yet powerful command line performance tool suite for the GNU/Linux operating system. While the focus of LIKWID is on x86 processors, some of the tools are portable and not limited to any specific architecture.



Likwid-perfctr:

Count hardware performance events. It can be used as a wrapper application, which does not require modification of the code to be analyzed, or with a marker API, which restricts the event counting to parts of the code.

So this gives us the overall performance of the program like clocks per instruction, branch rate, branch misprediction rate, branch misprediction ratio.

Inference : We can infer that since branch misprediction rate is not even 1%, there is very less of miss penalty added to the program. Given the case that there is 11% branching happening it is safe to say branch misprediction does not have an effect on the program.