CUDA: 1) Hello World Program 2) Vector Addition and 3) Vector Multiplication. Use input as a larger double number (64-bit). Run experiment for Threads = {1, 2, 4, 8, 16, 32, 64, 128, 256, 500} Estimate the parallelization fraction. Document it and report.

```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```

```
Collecting git+git://github.com/andreinechaev/nvcc4jupyter.git
  Cloning git://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-_74mwoc4
  Running command git clone -q git://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-
Building wheels for collected packages: NVCCPlugin
  Building wheel for NVCCPlugin (setup.py) ... done
  Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-cp36-none-any.whl size=4307 sh
  Stored in directory: /tmp/pip-ephem-wheel-cache-r5fo2si3/wheels/10/c2/05/ca241da37bff7
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

```
%reload_ext nvcc_plugin
```

```
directory /content/src already exists
Out bin /content/result.out
```

```
%%cu
#include<bits/stdc++.h>
using namespace std;
int main() {
    cout<<"Hello This is Amar Kumar - CED17I029"<<endl;
    return 0;
}
```

```
Hello This is Amar Kumar - CED17I029
```

```
%%cu

__global__ void add(int *a, int *b, int *c) {
  *c = *a + *b;
 }

#include <bits/stdc++.h>
using namespace std;
int main(void) {
  int a, b, c;   // host copies of a, b, c
  int *d_a, *d_b, *d_c;  // device copies of a, b, c
  int size = sizeof(int);

  // Allocate space for device copies of a, b, c
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_a, size);
  cudaMalloc((void **)&d_b, size);
  cudaMalloc((void **)&d_c, size);

  // Setup input values
  a = 2;
  b = 7;
// Copy inputs to device
  cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

  // Launch add() kernel on GPU
  add<<<1,1>>>(d_a, d_b, d_c);

  // Copy result back to host
  cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
    printf("Added Value = %d", c);
  // Cleanup
  cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
  return 0;
 }
```

```
Added Value = 9
```

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include<bits/stdc++.h>
#include<chrono>
using namespace std::chrono;
using namespace std;
#define N 100000
#define M 1
__global__ void vecAdd(double *a, double *b, double *c,int th){
    int id = threadIdx.x;
    for(int i=id ; i<N ; i+=th){
        c[i] = a[i] + b[i];
    }
}

int main( int argc, char* argv[] ){
    double *a,*b,*c;
    double *d_a,*d_b,*d_c;

    size_t size = N*sizeof(double);

    a = (double*)malloc(size);
    b = (double*)malloc(size);
    c = (double*)malloc(size);

    cudaMalloc(&d_a, size);
    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);

    int i;
    for( i = 0; i < N; i++ ) {
        a[i] = rand()%100000 + (1.0/(rand()%1000));
        b[i] = rand()%100000 + (1.0/(rand()%1000));
    }

    // Copy host vectors to device
    cudaMemcpy( d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, b, size, cudaMemcpyHostToDevice);

    int tt[10] ={1,2,4,8,16,32,64,128,256,500};

    for(int t=0 ; t<10 ; ++t){
        auto start = high_resolution_clock::now();
        vecAdd<<<1, tt[t]>>>(d_a, d_b, d_c,tt[t]);
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(stop - start);
    // cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;
        cout <<duration.count()<<endl;
    }

    //printf("execution time : %lf\n",(end-start));
    cudaMemcpy( c, d_c, size, cudaMemcpyDeviceToHost );

    //for(i=0; i<N; i++)
      //printf("%lf + %lf = %lf \n",a[i],b[i],c[i]);

    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

```
21
12
7
6
4
5
6
5
```

```
6
5
```

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include<bits/stdc++.h>
#include<chrono>
using namespace std::chrono;
using namespace std;
#define N 100000
#define M 1
__global__ void vecAdd(double *a, double *b, double *c,int th){
    int id = threadIdx.x;
    for(int i=id ; i<N ; i+=th){
        c[i] = a[i] * b[i];
    }
}

int main( int argc, char* argv[] ){
    double *a,*b,*c;
    double *d_a,*d_b,*d_c;

    size_t size = N*sizeof(double);

    a = (double*)malloc(size);
    b = (double*)malloc(size);
    c = (double*)malloc(size);

    cudaMalloc(&d_a, size);
    cudaMalloc(&d_b, size);
    cudaMalloc(&d_c, size);

    int i;
    for( i = 0; i < N; i++ ) {
        a[i] = rand()%100000 + (1.0/(rand()%1000));
        b[i] = rand()%100000 + (1.0/(rand()%1000));
    }

    // Copy host vectors to device
    cudaMemcpy( d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( d_b, b, size, cudaMemcpyHostToDevice);

    int tt[10] ={1,2,4,8,16,32,64,128,256,500};

    for(int t=0 ; t<10 ; ++t){
        auto start = high_resolution_clock::now();
        auto start = high_resolution_clock::now();
        vecAdd<<<1, tt[t]>>>(d_a, d_b, d_c,tt[t]);
        auto stop = high_resolution_clock::now();
        auto duration = duration_cast<microseconds>(stop - start);
    // cout << "Time taken by function: "<< duration.count() << " microseconds" << endl;
        cout <<duration.count()<<endl;
    }

    //printf("execution time : %lf\n",(end-start));
    cudaMemcpy( c, d_c, size, cudaMemcpyDeviceToHost );

    for(i=0; i<N; i++)
      printf("%lf * %lf = %lf \n",a[i],b[i],c[i]);

    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(a);
    free(b);
    free(c);

    return 0;
}
```

```
22
10
6
6
5
5
6
5
5
6
```