CUDA: 1) Matrix Addition 2) Matrix multiplication column major order 3) Matrix Multiplication Block based approach. 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

```
!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-
  Running command git clone -q git://github.com/andreinechaev/nvcc4jupyter.git /
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
  Stored in directory: /tmp/pip-ephem-wheel-cache-qsuqf__f/wheels/10/c2/05/ca241
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

```
%load_ext nvcc_plugin
```

```
created output directory at /content/src
Out bin /content/result.out
```

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include<bits/stdc++.h>
#include<chrono>
using namespace std::chrono;
using namespace std;
__global__ void Mat_add(double A[], double B[], double C[],int N,int th) {
   int id = threadIdx.x;
   for(int i=id ; i<N*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;
   int N=10000;
   size_t size = N*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,j;
     for (i=0; i<N; ++i){
        for(j=0 ; j<N ; ++j){
             A[i*N+j] = rand()%100000 + (1.0/(rand()%1000));
             B[i*N+j] = rand()%100000 + (1.0/(rand()%1000));
        }
    }

    /* Copy matrices from host memory to device memory */
    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();
          Mat_add<<<1, tt[t]>>>(d_A, d_B, d_C,N, tt[t]);
           auto stop = high_resolution_clock::now();
         auto duration = duration_cast<microseconds>(stop - start);
     // cout << "Time taken by function: "<< duration.count() << " microseconds" << en
         cout <<duration.count()<<endl;
     }

    /* Wait for the kernel to complete */
    cudaThreadSynchronize();
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

     /*for (int i=0; i<N; ++i){
        for(int j=0 ; j<N ; ++j){
            cout<<A[i*N + j]<<" + "<<B[i*N + j]<<" = "<<C[i*N + j]<<endl;
        }
    }*/
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(A); free(B); free(C);

    return 0;
}
```

```
36
9
5
5
5
4
4
5
5
5
```

```
%%cu
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include<bits/stdc++.h>
#include<chrono>
#define N 1000
using namespace std::chrono;
using namespace std;
__global__ void Mat_mul(double A[], double B[], double C[],int th) {
   int idx = threadIdx.x;

   for(int i = idx ; i<N*N ; i+=th){
       int temp=0;
       int y = i%N;
       int x = i/N;
       for(int k=0 ; k<N ; k++){
          temp += A[x*N+k] * B[k*N+y];
       }
       C[x*N+y] = temp;
   }
}

int main(int argc, char* argv[]) {
   double *A, *B, *C;
   double *d_A, *d_B, *d_C;
   //int N=4;
   size_t size = N*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);

    for (int i=0; i<N; ++i){
       for(int j=0 ; j<N ; ++j){
            A[i*N+j] = rand()%100000 + (1.0/(rand()%1000));
            B[i*N+j] = rand()%100000 + (1.0/(rand()%1000));
            /*A[i*N+j] = rand()%10;
            B[i*N+j] = rand()%10;*/
       }
   }
   /*cout<<"matrix A\n";
   for(int i=0 ; i<N ; ++i){
      for(int j=0 ; j<N ; ++j){
          cout<<A[i*N+j]<<" ";
      }
      cout<<endl;
  }
  cout<<endl;
```

```
    cout<<endl;
 cout<<"matrix B\n";
   for(int i=0 ; i<N ; ++i){
       for(int j=0 ; j<N ; ++j){
           cout<<B[i*N+j]<<" ";
       }
       cout<<endl;
   }
   cout<<endl;*/

    /* Copy matrices from host memory to device memory */
    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();
          Mat_mul<<<1, 2>>>(d_A, d_B, d_C, 2);
           auto stop = high_resolution_clock::now();
         auto duration = duration_cast<microseconds>(stop - start);
         cout << "Time taken by function: "<< duration.count() << " microseconds" << e
         //cout <<duration.count()<<endl;
     }

    /* Wait for the kernel to complete */
    //cudaThreadSynchronize();
    cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

 /*cout<<"matrix C\n";
   for(int i=0 ; i<N ; ++i){
       for(int j=0 ; j<N ; ++j){
           cout<<C[i*N+j]<<" ";
       }
       cout<<endl;
   }*/
    cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    free(A); free(B); free(C);

    return 0;
 }
```

```
Time taken by function: 27 microseconds
Time taken by function: 9 microseconds
Time taken by function: 4 microseconds
Time taken by function: 5 microseconds
Time taken by function: 5 microseconds
Time taken by function: 5 microseconds
Time taken by function: 4 microseconds
Time taken by function: 4 microseconds
Time taken by function: 4 microseconds
Time taken by function: 5 microseconds
```