```
# Set up CUDA
#First Change runtime to GPU and run this cell
!pip install git+https://github.com/afnan47/cuda.git
%load_ext nvcc_plugin
```

```
Collecting git+https://github.com/afnan47/cuda.git
  Cloning https://github.com/afnan47/cuda.git to /tmp/pip-req-build-k90lmsym
  Running command git clone --filter=blob:none --quiet https://github.com/afnan47/cuda.git /tmp/pip-req-build-k90lmsym
  Resolved https://github.com/afnan47/cuda.git to commit aac710a35f52bb78ab34d2e52517237941399eff
  Preparing metadata (setup.py) ... done
The nvcc_plugin extension is already loaded. To reload it, use:
  %reload_ext nvcc_plugin
```

```
%%cu
#include <iostream>
using namespace std;


// CUDA code to multiply matrices
__global__ void multiply(int* A, int* B, int* C, int size) {
    // Use thread indices and block indices to compute each element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < size && col < size) {
        int sum = 0;
        for (int i = 0; i < size; i++) {
            sum += A[row * size + i] * B[i * size + col];
            // A : ith col and rowth row
            // B : colth col and ith row
            // Recall ith row and jth col is accessed as matrix[i*size+j]
        }
        C[row * size + col] = sum;
    }
}


void initialize(int* matrix, int size) {
    for (int i = 0; i < size * size; i++) {
        matrix[i] = rand() % 10;
    }
}

void matrix_multiplication_cpu(int *a, int *b, int *c, int common, int c_rows,int c_cols){
    for(int i = 0; i < c_rows; i++){
        for(int j = 0; j < c_cols; j++){
            int sum = 0;
            for(int k = 0; k < common; k++){
                sum += a[i*common + k] * b[k*c_cols + j];
            }
            c[i*c_cols + j] = sum;
        }
    }
}


void print(int* matrix, int size) {
    for (int row = 0; row < size; row++) {
        for (int col = 0; col < size; col++) {
            cout << matrix[row * size + col] << " ";
        }
        cout << '\n';
    }
    cout << '\n';
}


int main() {

    int N = 2;

    int matrixSize = N * N;
    size_t matrixBytes = matrixSize * sizeof(int);

    int* A = new int[matrixSize];
    int* B = new int[matrixSize];
```

```
int* C = new int[matrixSize];
int* D = new int[matrixSize];
// We store A,B,C actually as 1 D arrays for easy access to GPU operations,
// However they actually represent 2 D arrays mathematically.
// In such representations, to access element in ith row and jth column, use matrix[i*n+j]

initialize(A, N);
initialize(B, N);
cout << "Matrix A: \n";
print(A, N);

cout << "Matrix B: \n";
print(B, N);


int* X, * Y, * Z;
// Allocate space
cudaMalloc(&X, matrixBytes);
cudaMalloc(&Y, matrixBytes);
cudaMalloc(&Z, matrixBytes);

// Copy values from A to X
cudaMemcpy(X, A, matrixBytes, cudaMemcpyHostToDevice);

// Copy values from A to X and B to Y
cudaMemcpy(Y, B, matrixBytes, cudaMemcpyHostToDevice);

// Threads per CTA dimension ie number of Threads per Block
int THREADS = 2;

// Blocks per grid dimension (assumes THREADS divides N evenly) ie Number of Blocks per Gride
int BLOCKS = N / THREADS;

// Use dim3 structs for block  and grid dimensions
dim3 threads(THREADS, THREADS);
dim3 blocks(BLOCKS, BLOCKS);

float gpu_elapsed_time;
cudaEvent_t gpu_start,gpu_stop;

cudaEventCreate(&gpu_start);
cudaEventCreate(&gpu_stop);
cudaEventRecord(gpu_start);

// Launch kernel
multiply<<<blocks, threads>>>(X, Y, Z, N);

cudaEventRecord(gpu_stop);
cudaEventSynchronize(gpu_stop);
cudaEventElapsedTime(&gpu_elapsed_time, gpu_start, gpu_stop);
cudaEventDestroy(gpu_start);
cudaEventDestroy(gpu_stop);

cudaMemcpy(C, Z, matrixBytes, cudaMemcpyDeviceToHost);
cout << "GPU result:\n";
print(C, N);
cout<<"GPU Elapsed time is: "<<gpu_elapsed_time<<" milliseconds\n"<<endl;

cudaEventCreate(&gpu_start);
cudaEventCreate(&gpu_stop);
cudaEventRecord(gpu_start);

matrix_multiplication_cpu(A,B,D,2,2,2);

cudaEventRecord(gpu_stop);
cudaEventSynchronize(gpu_stop);
cudaEventElapsedTime(&gpu_elapsed_time, gpu_start, gpu_stop);
cudaEventDestroy(gpu_start);
cudaEventDestroy(gpu_stop);

cout << "CPU result:\n";
print(D,N);
cout<<"CPU Elapsed time is: "<<gpu_elapsed_time<<" milliseconds"<<endl;

delete[] A;
delete[] B;
delete[] C;
delete[] D;
```

```
    delete[] D;

    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);

    return 0;
}
```

```
Matrix A:
3 6
7 5

Matrix B:
3 5
6 2

GPU result:
45 27
51 45

GPU Elapsed time is: 0.196608 milliseconds

CPU result:
45 27
51 45

CPU Elapsed time is: 0.002496 milliseconds
```