```
!nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

```
!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-j1vz2m13
  Running command git clone --filter=blob:none --quiet https://github.com/afnan47/cuda.git /tmp/pip-req-build-j1vz2m13
  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
```

```
%%writefile add.cu

#include <iostream>
#include <cstdlib> // Include <cstdlib> for rand()
using namespace std;

__global__ void add(int* A, int* B, int* C, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        C[tid] = A[tid] + B[tid];
        printf("Thread %d: %d + %d = %d\n", tid, A[tid], B[tid], C[tid]);
    }
}

void print(int* vector, int size) {
    for (int i = 0; i < size; i++) {
        cout << vector[i] << " ";
    }
    cout << endl;
}

int main() {
    int N;
    cout << "Enter the size of the vectors: ";
    cin >> N;

    // Allocate host memory
    int* A = new int[N];
    int* B = new int[N];
    int* C = new int[N];

    // Initialize host arrays
    cout << "Enter elements for vector A: ";
    for (int i = 0; i < N; i++) {
        cin >> A[i];
    }

    cout << "Enter elements for vector B: ";
    for (int i = 0; i < N; i++) {
        cin >> B[i];
    }

    cout << "Vector A: ";
    print(A, N);
    cout << "Vector B: ";
    print(B, N);

    int* X, * Y, * Z;
    size_t vectorBytes = N * sizeof(int);

    // Allocate device memory
    cudaMalloc(&X, vectorBytes);
    cudaMalloc(&Y, vectorBytes);
    cudaMalloc(&Z, vectorBytes);

    // Copy data from host to device
```

```
    cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(Y, B, vectorBytes, cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    // Launch kernel
    add<<<blocksPerGrid, threadsPerBlock>>>(X, Y, Z, N);

    // Copy result from device to host
    cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);

    cout << "Addition: ";
    print(C, N);

    // Free device memory
    cudaFree(X);
    cudaFree(Y);
    cudaFree(Z);

    // Free host memory
    delete[] A;
    delete[] B;
    delete[] C;

    return 0;
}
```

```
Overwriting add.cu
```

```
!nvcc add.cu -o add
!./add
```

```
Enter the size of the vectors: 6
Enter elements for vector A: 1 2 3 1 2 3
Enter elements for vector B: 1 1 1 1 1 1
Vector A: 1 2 3 1 2 3
Vector B: 1 1 1 1 1 1
Thread 0: 1 + 1 = 2
Thread 1: 2 + 1 = 3
Thread 2: 3 + 1 = 4
Thread 3: 1 + 1 = 2
Thread 4: 2 + 1 = 3
Thread 5: 3 + 1 = 4
Addition: 2 3 4 2 3 4
```

```
%%cu
#include <stdio.h>

__global__ void vectorAddKernel(float* deviceInput1, float* deviceInput2, float* deviceOutput, int size) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < size) {
        deviceOutput[tid] = deviceInput1[tid] + deviceInput2[tid];
        printf("Thread %d: %.2f + %.2f = %.2f\n", tid, deviceInput1[tid], deviceInput2[tid], deviceOutput[tid]);
    } else {
        printf("Thread %d out of bounds\n", tid);
    }
}

int main() {
    int size = 6; // Size of the input vectors
    float hostInput1[size] = {1, 2, 3,1,3,5};
    float hostInput2[size] = {5, 6, 7,4,5,1};
    float hostOutput[size];

    float *deviceInput1, *deviceInput2, *deviceOutput;
    cudaError_t cudaStatus;

    cudaStatus = cudaMalloc(&deviceInput1, size * sizeof(float));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed for deviceInput1: %s\n", cudaGetErrorString(cudaStatus));
        return 1;
    }

    cudaStatus = cudaMalloc(&deviceInput2, size * sizeof(float));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed for deviceInput2: %s\n", cudaGetErrorString(cudaStatus));
        cudaFree(deviceInput1);
        return 1;
    }

    cudaStatus = cudaMalloc(&deviceOutput, size * sizeof(float));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed for deviceOutput: %s\n", cudaGetErrorString(cudaStatus));
        cudaFree(deviceInput1);
        cudaFree(deviceInput2);
        return 1;
    }

    cudaMemcpy(deviceInput1, hostInput1, size * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(deviceInput2, hostInput2, size * sizeof(float), cudaMemcpyHostToDevice);

    int blockSize = 4; // 4 threads per block
    int numBlocks = (size + blockSize - 1) / blockSize; // Adjust grid size to cover all elements

    vectorAddKernel<<<numBlocks, blockSize>>>(deviceInput1, deviceInput2, deviceOutput, size);
    cudaDeviceSynchronize(); // Wait for all threads to finish

    cudaMemcpy(hostOutput, deviceOutput, size * sizeof(float), cudaMemcpyDeviceToHost);

    printf("Vector Addition Result:\n");
    for (int i = 0; i < size; i++) {
        printf("%.2f + %.2f = %.2f\n", hostInput1[i], hostInput2[i], hostOutput[i]);
    }

    cudaFree(deviceInput1);
    cudaFree(deviceInput2);
    cudaFree(deviceOutput);

    return 0;
}
```

```
Thread 6 out of bounds
Thread 7 out of bounds
Thread 0: 1.00 + 5.00 = 6.00
Thread 1: 2.00 + 6.00 = 8.00
Thread 2: 3.00 + 7.00 = 10.00
Thread 3: 1.00 + 4.00 = 5.00
Thread 4: 3.00 + 5.00 = 8.00
Thread 5: 5.00 + 1.00 = 6.00
Vector Addition Result:
1.00 + 5.00 = 6.00
2.00 + 6.00 = 8.00
```

```
3.00 + 7.00 = 10.00
1.00 + 4.00 = 5.00
3.00 + 5.00 = 8.00
5.00 + 1.00 = 6.00
```

```
%%cu
#include <iostream>
#include <cuda.h>

using namespace std;

#define BLOCK_SIZE 2

__global__ void gpuMM(float *A, float *B, float *C, int N)
{

    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;

    float sum = 0.f;
    for (int n = 0; n < N; ++n)
        sum += A[row*N+n]*B[n*N+col];

    C[row*N+col] = sum;
}

int main(int argc, char *argv[])
{int N;float K;

    cout<<"Enter a Value for Size/2 of matrix";
    cin>>K;

    K = 1;
    N = K*BLOCK_SIZE;

    cout << "\n Executing Matrix Multiplcation" << endl;
    cout << "\n Matrix size: " << N << "x" << N << endl;


    float *hA,*hB,*hC;
    hA = new float[N*N];
    hB = new float[N*N];
    hC = new float[N*N];


    for (int j=0; j<N; j++){
        for (int i=0; i<N; i++){
           hA[j*N+i] = 2;
           hB[j*N+i] = 4;

        }
    }


    int size = N*N*sizeof(float);
    float *dA,*dB,*dC;
    cudaMalloc(&dA,size);
    cudaMalloc(&dB,size);
    cudaMalloc(&dC,size);

    dim3 threadBlock(BLOCK_SIZE,BLOCK_SIZE);
    dim3 grid(K,K);
    cout<<"\n Input Matrix 1 \n";
    for (int row=0; row<N; row++){
            for (int col=0; col<N; col++){

                    cout<<hA[row*col]<<" ";

            }
            cout<<endl;
        }
    cout<<"\n Input Matrix 2 \n";
    for (int row=0; row<N; row++){
            for (int col=0; col<N; col++){

                    cout<<hB[row*col]<<" ";

            }
            cout<<endl;
        }

    cudaMemcpy(dA,hA,size,cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(dB,hB,size,cudaMemcpyHostToDevice);


    gpuMM<<<grid,threadBlock>>>(dA,dB,dC,N);

    // Now do the matrix multiplication on the CPU
   /*float sum;
    for (int row=0; row<N; row++){
        for (int col=0; col<N; col++){
            sum = 0.f;
            for (int n=0; n<N; n++){
                sum += hA[row*N+n]*hB[n*N+col];
            }
            hC[row*N+col] = sum;
            cout << sum <<" ";


        }
        cout<<endl;
    }*/


    float *C;
    C = new float[N*N];


    cudaMemcpy(C,dC,size,cudaMemcpyDeviceToHost);

    cout <<"\n\n\n\n\n Resultant matrix\n\n";
    for (int row=0; row<N; row++){
        for (int col=0; col<N; col++){

               cout<<C[row*col]<<"  ";

        }
        cout<<endl;
    }

    cout << "Finished." << endl;
}
```

```
 Enter a Value for Size/2 of matrix
  Executing Matrix Multiplcation

  Matrix size: 2x2

  Input Matrix 1
 2       2
 2       2

  Input Matrix 2
 4       4
 4       4




  Resultant matrix

 16      16
 16      16
 Finished.
```

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

// CUDA code to multiply matrices
__global__ void multiply(int* A, int* B, int* C, int size) {
    // Uses 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];
        }
        C[row * size + col] = sum;

        // Print thread index and operation
        printf("Thread (%d, %d) performed multiplication for C[%d][%d]\n", threadIdx.x, threadIdx.y, row, col);
    }
}

// Initialize matrix C with zeros
void initializeZero(int* matrix, int size) {
    for (int i = 0; i < size * size; i++) {
        matrix[i] = 0;
    }
}

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;
    cout << "Enter the size of the matrices: ";
    cin >> N;
```