```
%%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];
  }
}
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;
  int* A, * B, * C;
  int vectorSize = N;
  size_t vectorBytes = vectorSize * sizeof(int);

  A = new int[vectorSize];
  B = new int[vectorSize];
  C = new int[vectorSize];

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

  cudaMalloc(&X, vectorBytes);
  cudaMalloc(&Y, vectorBytes);
  cudaMalloc(&Z, vectorBytes);

  if (X == nullptr || Y == nullptr || Z == nullptr) {
```

```
    cerr << "CUDA memory allocation failed" << endl;
    return 1;
  }

  cudaMemcpy(X, A, vectorBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(Y, B, vectorBytes, cudaMemcpyHostToDevice);
  int threadsPerBlock = 256;
  int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

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

  cudaError_t kernelLaunchError = cudaGetLastError();
  if (kernelLaunchError != cudaSuccess) {
  cerr << "CUDA kernel launch failed: " <<
  cudaGetErrorString(kernelLaunchError) << endl;
  return 1;
  }

  cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);

  cudaError_t memcpyError = cudaGetLastError();
  if (memcpyError != cudaSuccess) {
  cerr << "CUDA memcpy failed: " << cudaGetErrorString(memcpyError) <<
  endl;
  return 1;
  }
  cout << "Addition: ";
  print(C, N);

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

  delete[] A;
  delete[] B;
  delete[] C;
  return 0;
}
```

```
%%writefile matrix_mult.cu
#include <iostream>
#include <cuda.h>
using namespace std;
#define BLOCK_SIZE 1
__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;
// Get matrix size from user
cout << "Enter size of matrix (N): ";
cin >> N;
if (N % BLOCK_SIZE != 0) {
cerr << "Matrix size must be a multiple of BLOCK_SIZE." << endl;
return 1;
}
cout << "\nExecuting Matrix Multiplication" << endl;
cout << "Matrix size: " << N << "x" << N << endl;
// Allocate memory for matrices on the host
float *hA, *hB, *hC;
hA = new float[N * N];
hB = new float[N * N];
hC = new float[N * N];
// Read matrices from user
cout << "Enter elements of matrix A (" << N << "x" << N << "):" <<
endl;
for (int i = 0; i < N * N; ++i)
cin >> hA[i];
cout << "Enter elements of matrix B (" << N << "x" << N << "):" <<
endl;
for (int i = 0; i < N * N; ++i)
cin >> hB[i];
// Allocate memory for matrices on the device
int size = N * N * sizeof(float);
float *dA, *dB, *dC;
cudaMalloc(&dA, size);
cudaMalloc(&dB, size);
cudaMalloc(&dC, size);
// Copy matrices from the host to the device
cudaMemcpy(dA, hA, size, cudaMemcpyHostToDevice);
cudaMemcpy(dB, hB, size, cudaMemcpyHostToDevice);
dim3 threadBlock(BLOCK_SIZE, BLOCK_SIZE);
```

```
dim3 grid(N / BLOCK_SIZE, N / BLOCK_SIZE);
// Execute the matrix multiplication kernel
gpuMM<<<grid, threadBlock>>>(dA, dB, dC, N);
// Copy the result matrix from the device to the host
cudaMemcpy(hC, dC, size, cudaMemcpyDeviceToHost);
// Display the result matrix
cout << "\nResultant matrix:\n";
for (int row = 0; row < N; row++) {
for (int col = 0; col < N; col++) {
cout << hC[row * N + col] << " ";
}
cout << endl;
}
// Free device memory
cudaFree(dA);
cudaFree(dB);
cudaFree(dC);
// Free host memory
delete[] hA;
delete[] hB;
delete[] hC;
cout << "Finished." << endl;
return 0;
}
```