```
// Write a CUDA Program for :

// 1. Addition of two large vectors
// 2. Matrix Multiplication using CUDA C

#include <iostream>
#include <cuda_runtime.h>

#define N 1000000  // Vector size
#define MATRIX_SIZE 512  // Matrix size (N x N)

using namespace std;

// Vector addition kernel
__global__ void vectorAdd(float* A, float* B, float* C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        C[i] = A[i] + B[i];
}

// Matrix multiplication kernel
__global__ void matrixMulKernel(float* A, float* B, float* C, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < n && col < n) {
        float sum = 0;
        for (int i = 0; i < n; ++i)
            sum += A[row * n + i] * B[i * n + col];
        C[row * n + col] = sum;
    }
}

class ParallelComputations {
public:
    // Vector Addition
    void vectorAddition() {
        float *A, *B, *C, *d_A, *d_B, *d_C;
        size_t size = N * sizeof(float);

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

        // Initialize vectors
        for (int i = 0; i < N; i++) {
            A[i] = i;
            B[i] = 2 * i;
        }

        // Allocate device memory
        cudaMalloc(&d_A, size);
        cudaMalloc(&d_B, size);
```

```
        cudaMalloc(&d_C, size);

        // Copy data to device
        cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

        // Launch kernel
        int threadsPerBlock = 256;
        int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
        vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

        // Copy result back
        cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

        cout << "Sample Output for Vector Addition: C[0] = " << C[0] << ",
C[N-1] = " << C[N - 1] << endl;

        // Free memory
        delete[] A; delete[] B; delete[] C;
        cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    }

    // Matrix Multiplication
    void matrixMultiplication() {
        int size = MATRIX_SIZE * MATRIX_SIZE * sizeof(float);
        float *A, *B, *C, *d_A, *d_B, *d_C;

        // Allocate host memory
        A = new float[MATRIX_SIZE * MATRIX_SIZE];
        B = new float[MATRIX_SIZE * MATRIX_SIZE];
        C = new float[MATRIX_SIZE * MATRIX_SIZE];

        // Initialize matrices
        for (int i = 0; i < MATRIX_SIZE * MATRIX_SIZE; i++) {
            A[i] = 1.0f;
            B[i] = 2.0f;
        }

        // Allocate device memory
        cudaMalloc(&d_A, size);
        cudaMalloc(&d_B, size);
        cudaMalloc(&d_C, size);

        // Copy to device
        cudaMemcpy(d_A, A, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_B, B, size, cudaMemcpyHostToDevice);

        // Kernel launch config
        dim3 threadsPerBlock(16, 16);
        dim3 blocksPerGrid((MATRIX_SIZE + 15) / 16, (MATRIX_SIZE + 15) / 16);
        matrixMulKernel<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
MATRIX_SIZE);

        // Copy result back
```

```
        cudaMemcpy(C, d_C, size, cudaMemcpyDeviceToHost);

        cout << "Sample Output for Matrix Multiplication: C[0] = " << C[0] << ",
C[N*N-1] = " << C[MATRIX_SIZE*MATRIX_SIZE - 1] << endl;

        // Free memory
        delete[] A; delete[] B; delete[] C;
        cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
    }
};

int main() {
    ParallelComputations pc;

    // Perform Vector Addition
    cout << "\nVector Addition:" << endl;
    pc.vectorAddition();

    // Perform Matrix Multiplication
    cout << "\nMatrix Multiplication:" << endl;
    pc.matrixMultiplication();

    return 0;
}
/*g++ -fopenmp filename.cpp -o filename.exe
Filename.exe
gcc -v takaycha command prompt la
https://www.winlibs.com/*/
```