CUDA

cat>> ass1.cu

Nvcc ass1.cu

./a.out

1

MATRIX MULTIPLICATION

```
#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;
   }
```

```
}


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


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* A, * B, * C;
```

```
int N = 2;
int blockSize =  16;

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

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

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
int THREADS = 2;

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

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

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

cudaMemcpy(C, Z, matrixBytes, cudaMemcpyDeviceToHost);
cout << "Multiplication of matrix A and B: \n";
print(C, N);
```

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

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

    return 0;
}
```

VECTOR ADDITION

```
#include <iostream>
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 initialize(int* vector, int size) {
    for (int i = 0; i < size; i++) {
        vector[i] = rand() % 10;
    }
}

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

int main() {
    int N = 4;
    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];

initialize(A, vectorSize);
initialize(B, vectorSize);

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);

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);

cudaMemcpy(C, Z, vectorBytes, cudaMemcpyDeviceToHost);
```

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

    delete[] A;
    delete[] B;
    delete[] C;

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

    return 0;
}
```