**CUDA Programming: Vector Addition and Matrix Multiplication**

**1. Vector Addition**

**Explanation**

**Kernel Function (vectorAdd):**

* **Parallel Computation:** Each thread is responsible for computing one element of the result vector C.
* **Index Calculation:** The thread calculates its unique global index using blockIdx, threadIdx, and blockDim to determine which elements of vectors A and B should add.
* **Boundary Check:** Ensures that threads do not access memory beyond the vector size, preventing invalid memory operations.
* **Element-wise Addition:** Performs the addition of corresponding elements from vectors A and B and stores the result in vector C.

**Host Code:**

* **Initialization:** Initializes two input vectors, A and B with sample data.
* **Memory Allocation:** Allocates memory on the GPU (device) for vectors A, B, and C.
* **Data Transfer:** Copies the input vectors A and B from the host (CPU) memory to the device (GPU) memory.
* **Kernel Configuration:** Defines the number of threads per block and the number of blocks per grid to ensure all elements are processed.
* **Kernel Launch:** Executes the vectorAdd kernel on the GPU to perform parallel vector addition.
* **Result Retrieval:** Copies the resulting vector C from the device back to the host memory.
* **Verification:** Prints a subset of the results to verify the correctness of the computation.
* **Cleanup:** Frees the allocated memory on both the device and host to prevent memory leaks.

**Code, Compilation, Execution, and Output:**

#include <iostream>

#include <cuda\_runtime.h>

// CUDA kernel as defined above

int main() {

    int N = 1 << 20; // Number of elements

    size\_t size = N \* sizeof(float);

    // Allocate host memory

    float \*h\_A = (float \*)malloc(size);

    float \*h\_B = (float \*)malloc(size);

    float \*h\_C = (float \*)malloc(size);

    // Initialize vectors

    for (int i = 0; i < N; ++i) {

        h\_A[i] = static\_cast<float>(i);

        h\_B[i] = static\_cast<float>(i \* 2);

    }

    // Allocate device memory

    float \*d\_A, \*d\_B, \*d\_C;

    cudaMalloc((void \*\*)&d\_A, size);

    cudaMalloc((void \*\*)&d\_B, size);

    cudaMalloc((void \*\*)&d\_C, size);

    // Copy data from host to device

    cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

    cudaMemcpy(d\_B, h\_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 to host

    cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost);

    // Verify result

    for (int i = 0; i < 10; ++i) {

        std::cout << h\_C[i] << " ";

    }

    std::cout << std::endl;

    // Free memory

    cudaFree(d\_A);

    cudaFree(d\_B);

    cudaFree(d\_C);

    free(h\_A);

    free(h\_B);

    free(h\_C);

    return 0;

}

nvcc vector.cu -o vector

./vector

Output:

0 3 6 9 12 15 18 21 24 27

**2. Matrix Multiplication**

**Explanation**

**Kernel Function (matrixMulCUDA):**

* **Element-wise Computation:** Each thread computes one element of the result matrix C by calculating the dot product of a specific row from matrix A and a specific column from matrix B.
* **Index Calculation:** Determines the row and column indices using blockIdx, threadIdx, and blockDim to map each thread to a unique element in the result matrix.
* **Boundary Check:** Ensures that threads do not attempt to access elements outside the bounds of the matrices, maintaining memory safety.
* **Dot Product Calculation:** Iterates over the corresponding elements of the row from A and the column from B, multiplying them and accumulating the sum to compute C[i][j].

**Host Code:**

* **Initialization:** Initializes two input matrices A and B with sample data.
* **Memory Allocation:** Allocates memory on the GPU (device) for matrices A, B, and C.
* **Data Transfer:** Copies the input matrices A and B from the host (CPU) memory to the device (GPU) memory.
* **Kernel Configuration:** Defines a two-dimensional grid and block configuration (e.g., 16x16 threads per block) to map threads to matrix elements efficiently.
* **Kernel Launch:** Executes the matrixMulCUDA kernel on the GPU to perform parallel matrix multiplication.
* **Result Retrieval:** Copies the resulting matrix C from the device back to the host memory.
* **Verification:** Prints the result matrix C to verify the correctness of the computation.
* **Cleanup:** Frees the allocated memory on both the device and host to prevent memory leaks.

**Code, Compilation, Execution, and Output:**

#include <iostream>

#include <cuda\_runtime.h>

#define N 16  // Size of the matrices

// CUDA kernel as defined above

int main() {

    int size = N \* N \* sizeof(float);

    float h\_A[N\*N], h\_B[N\*N], h\_C[N\*N];

    // Initialize matrices

    for(int i = 0; i < N\*N; ++i) {

        h\_A[i] = static\_cast<float>(i);

        h\_B[i] = static\_cast<float>(i % N);

    }

    float \*d\_A, \*d\_B, \*d\_C;

    cudaMalloc((void\*\*)&d\_A, size);

    cudaMalloc((void\*\*)&d\_B, size);

    cudaMalloc((void\*\*)&d\_C, size);

    cudaMemcpy(d\_A, h\_A, size, cudaMemcpyHostToDevice);

    cudaMemcpy(d\_B, h\_B, size, cudaMemcpyHostToDevice);

    // Define block and grid dimensions

    dim3 threadsPerBlock(16, 16);

    dim3 blocksPerGrid((N + threadsPerBlock.x - 1) / threadsPerBlock.x,

                       (N + threadsPerBlock.y - 1) / threadsPerBlock.y);

    // Launch the kernel

    matrixMulCUDA<<<blocksPerGrid, threadsPerBlock>>>(d\_C, d\_A, d\_B, N);

    // Copy the result back to host

    cudaMemcpy(h\_C, d\_C, size, cudaMemcpyDeviceToHost);

    // Display the result matrix

    std::cout << "Result matrix C:" << std::endl;

    for(int i = 0; i < N; ++i) {

        for(int j = 0; j < N; ++j) {

            std::cout << h\_C[i \* N + j] << " ";

        }

        std::cout << std::endl;

    }

    // Free device memory

    cudaFree(d\_A);

    cudaFree(d\_B);

    cudaFree(d\_C);

    return 0;

}

nvcc matrixMul.cu -o matrixMul

./matrixMul

Output:

Result matrix C:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47

...

240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255