Module 5

29 November 2024 01:32 AM

Vector addition

```
#include <stdio.h>
#include <cuda_runtime.h>

// Kernel function to add two vectors
__global__ void vectorAdd(const float* A, const float* B, float* C, int n) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}

int main() {
    int n = 1024; // Size of the vectors
    size_t size = n * sizeof(float);

    // Allocate memory on the host
    float *h_A = (float*)malloc(size);
    float *h_B = (float*)malloc(size);
    float *h_C = (float*)malloc(size);

    // Initialize input vectors
    for (int i = 0; i < n; i++) {
        h_A[i] = i * 1.0f;
        h_B[i] = i * 2.0f;
    }

    // Allocate memory on the device
    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 the kernel with 256 threads per block
    int threadsPerBlock = 256;
    int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, n);

    // Copy the result from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Print a few results
    for (int i = 0; i < 10; i++) {
        printf("C[%d] = %f\n", i, h_C[i]);
    }

    // Free device memory
```

```
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

Matrix multiplication

```
#include <stdio.h>
#include <cuda_runtime.h>

// Kernel function for matrix-vector multiplication
__global__ void matrixVectorMul(const float* matrix, const float* vector, float* result, int rows, int
cols) {
    int row = blockIdx.x * blockDim.x + threadIdx.x; // Calculate the row index
    if (row < rows) {
        float sum = 0.0f;
        for (int col = 0; col < cols; col++) {
            sum += matrix[row * cols + col] * vector[col];
        }
        result[row] = sum;
    }
}

int main() {
    // Matrix dimensions
    int rows = 4;  // Number of rows in the matrix
    int cols = 3;  // Number of columns in the matrix
    size_t matrixSize = rows * cols * sizeof(float);
    size_t vectorSize = cols * sizeof(float);
    size_t resultSize = rows * sizeof(float);

    // Allocate and initialize host memory
    float h_matrix[] = {
        1.0f, 2.0f, 3.0f,
        4.0f, 5.0f, 6.0f,
        7.0f, 8.0f, 9.0f,
        10.0f, 11.0f, 12.0f
    };
    float h_vector[] = {1.0f, 2.0f, 3.0f};
    float h_result[rows];

    // Allocate device memory
    float *d_matrix, *d_vector, *d_result;
    cudaMalloc((void**)&d_matrix, matrixSize);
    cudaMalloc((void**)&d_vector, vectorSize);
    cudaMalloc((void**)&d_result, resultSize);

    // Copy data from host to device
    cudaMemcpy(d_matrix, h_matrix, matrixSize, cudaMemcpyHostToDevice);
```

```
    cudaMemcpy(d_vector, h_vector, vectorSize, cudaMemcpyHostToDevice);

    // Launch the kernel with one thread per row
    int threadsPerBlock = 256;
    int blocksPerGrid = (rows + threadsPerBlock - 1) / threadsPerBlock;
    matrixVectorMul<<<blocksPerGrid, threadsPerBlock>>>(d_matrix, d_vector, d_result, rows, cols);

    // Copy the result from device to host
    cudaMemcpy(h_result, d_result, resultSize, cudaMemcpyDeviceToHost);

    // Print the result
    printf("Result vector:\n");
    for (int i = 0; i < rows; i++) {
        printf("%f\n", h_result[i]);
    }

    // Free device memory
    cudaFree(d_matrix);
    cudaFree(d_vector);
    cudaFree(d_result);

    return 0;
}
```

Matrix Matrix Multiplication:

```
#include <stdio.h>
#include <cuda_runtime.h>

// Kernel function for matrix multiplication
__global__ void matrixMultiply(const float* A, const float* B, float* C, int N) {
    // Compute row and column index of the element
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Ensure the indices are within bounds
    if (row < N && col < N) {
        float sum = 0.0f;
        for (int k = 0; k < N; k++) {
            sum += A[row * N + k] * B[k * N + col];
        }
        C[row * N + col] = sum;
    }
}

int main() {
    // Define matrix dimensions (square matrix of size NxN)
    int N = 4;
    size_t size = N * N * sizeof(float);

    // Allocate and initialize host matrices
    float h_A[N * N], h_B[N * N], h_C[N * N];
    for (int i = 0; i < N * N; i++) {
        h_A[i] = i + 1;  // Example initialization
        h_B[i] = i + 1;  // Example initialization
    }

    // 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 matrices from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Configure block and grid dimensions
    int blockSize = 16; // Threads per block dimension (blockSize x blockSize)
    dim3 threadsPerBlock(blockSize, blockSize);
    dim3 blocksPerGrid((N + blockSize - 1) / blockSize, (N + blockSize - 1) / blockSize);

    // Launch the kernel
    matrixMultiply<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy the result from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Print the result
    printf("Resultant matrix C:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%f ", h_C[i * N + j]);
        }
        printf("\n");
    }

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    return 0;
}
```