**Lab Exercise 6.4 – Extend the Kernel to Perform Simple Arithmetic Operations**

**Objective:**  
To extend the basic CUDA kernel by performing simple arithmetic operations like addition, subtraction, multiplication, and division on elements of two arrays. This will help understand how to use thread indexing to access and process array elements in parallel on the GPU.



**Sample CUDA Program**

**Filename:** arithmetic\_operations.cu

#include <iostream>

#define N 16 // Number of elements in arrays

// CUDA kernel for performing arithmetic operations

\_\_global\_\_ void perform\_arithmetic\_operations(int \*a, int \*b, int \*sum, int \*diff, int \*prod, int \*quot) {

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

if (idx < N) {

// Perform simple arithmetic operations

sum[idx]= a[idx] + b[idx];

diff[idx] = a[idx] - b[idx];

prod[idx] = a[idx] \* b[idx];

if (b[idx] != 0) {

quot[idx] = a[idx] / b[idx];

} else {

quot[idx] = 0; // Prevent division by zero

}

}

}

int main() {

// Host arrays

int h\_a[N], h\_b[N], h\_sum[N], h\_diff[N], h\_prod[N], h\_quot[N];

// Initialize input arrays

for (int i = 0; i < N; i++) {

h\_a[i] = (i + 1) \* 2; // Elements 2, 4, 6, ... 32

h\_b[i] = (i + 1) ; // Elements 1 to 16

}

// Device arrays

int \*d\_a, \*d\_b, \*d\_sum, \*d\_diff, \*d\_prod, \*d\_quot;

// Allocate memory on device

cudaMalloc(&d\_a, N \* sizeof(int));

cudaMalloc(&d\_b, N \* sizeof(int));

cudaMalloc(&d\_sum, N \* sizeof(int));

cudaMalloc(&d\_diff, N \* sizeof(int));

cudaMalloc(&d\_prod, N \* sizeof(int));

cudaMalloc(&d\_quot, N \* sizeof(int));

// Copy data from host to device

cudaMemcpy(d\_a, h\_a, N \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, N \* sizeof(int), cudaMemcpyHostToDevice);

// Launch kernel with enough blocks and threads

int threadsPerBlock = 8;

int numBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;

perform\_arithmetic\_operations<<<numBlocks, threadsPerBlock>>>(d\_a, d\_b, d\_sum, d\_diff, d\_prod, d\_quot);

// Synchronize device

cudaDeviceSynchronize();

// Copy result back to host

cudaMemcpy(h\_sum, d\_sum, N \* sizeof(int), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_diff, d\_diff, N \* sizeof(int), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_prod, d\_prod, N \* sizeof(int), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_quot, d\_quot, N \* sizeof(int), cudaMemcpyDeviceToHost);

// Display results

std::cout << "Index\tA\tB\tSum\tDiff\tProd\tQuot" << std::endl;

for (int i = 0; i < N; i++) {

std::cout << i << "\t" << h\_a[i] << "\t" << h\_b[i] << "\t" << h\_sum[i] << "\t"

<< h\_diff[i] << "\t" << h\_prod[i] << "\t" << h\_quot[i] << std::endl;

}

// Free device memory

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_sum);

cudaFree(d\_diff);

cudaFree(d\_prod);

cudaFree(d\_quot);

return 0;

}

**Explanation:**

* **Host Arrays:** We define arrays h\_a and h\_b on the host (CPU) side, which will hold the input data for the arithmetic operations.
* **CUDA Kernel (perform\_arithmetic\_operations):**
  + **sum[idx] = a[idx] + b[idx]**: Adds corresponding elements from arrays a and b.
  + **diff[idx] = a[idx] - b[idx]**: Subtracts b[idx] from a[idx].
  + **prod[idx] = a[idx] \* b[idx]**: Multiplies corresponding elements.
  + **quot[idx] = a[idx] / b[idx]**: Divides a[idx] by b[idx], but we handle division by zero.
* **Grid and Block Configuration:**
  + We use 8 threads per block and compute the number of blocks required using the formula:  
    numBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;
* **CUDA Memory Management:**
  + Allocate memory on the GPU for the arrays.
  + Copy input data from host to device.
  + Copy results back to the host for printing.
* **Output:** After kernel execution, the results of the arithmetic operations are printed for each index.

**How to Compile and Run:**

nvcc arithmetic\_operations.cu -o arithmetic\_operations

./arithmetic\_operations

**Expected Output:**

Id A B Sum Diff Prod Quot

0 1 2 3 -1 2 0



1 2 4 6 -2 8 0

2 3 6 9 -3 18 0

3 4 8 12 -4 32 0

4 5 10 15 -5 50 0

5 6 12 18 -6 72 0

6 7 14 21 -7 98 0

7 8 16 24 -8 128 0

8 9 18 27 -9 162 0

9 10 20 30 -10 200 0

10 11 22 33 -11 242 0

11 12 24 36 -12 288 0

12 13 26 39 -13 338 0

13 14 28 42 -14 392 0

14 15 30 45 -15 450 0

15 16 32 48 -16 512 0

**Code Overview:**

This CUDA program performs simple arithmetic operations (addition, subtraction, multiplication, and division) on two arrays (a and b) in parallel using CUDA. The program defines a kernel that is launched with multiple threads across multiple blocks to compute these operations element-wise. Let's go through the code step by step.

**1. Include Header Files**

#include <iostream>

* We include the <iostream> library for standard input and output operations. It is used here to display the results of the arithmetic operations.

**2. Define Constants**

#define N 16 // Number of elements in arrays

* We define N as the number of elements in the input arrays a and b. Here, N is set to 16, which means both arrays will contain 16 elements.

**3. CUDA Kernel Function**

\_\_global\_\_ void perform\_arithmetic\_operations(int \*a, int \*b, int \*sum, int \*diff, int \*prod, int \*quot) {

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

if (idx < N) {

// Perform simple arithmetic operations

sum[idx] = a[idx] + b[idx];

diff[idx] = a[idx] - b[idx];

prod[idx] = a[idx] \* b[idx];

if (b[idx] != 0) {

quot[idx] = a[idx] / b[idx];

} else {

quot[idx] = 0; // Prevent division by zero

}

}

}

* **\_\_global\_\_ Keyword**: This specifies that the function perform\_arithmetic\_operations is a CUDA kernel. Kernels are executed on the GPU and can be launched from the host (CPU).
* **Parameters**:
  + int \*a, \*b: Pointers to input arrays a and b stored on the device (GPU).
  + int \*sum, \*diff, \*prod, \*quot: Pointers to output arrays on the device that will store the results of the arithmetic operations.
* **Thread Indexing**:
  + int idx = threadIdx.x + blockIdx.x \* blockDim.x;: This calculates the global thread index idx within the entire grid. It combines the block index (blockIdx.x), block dimension (blockDim.x), and thread index (threadIdx.x).
* **Condition if (idx < N)**:
  + This ensures that the thread only processes elements of the arrays if the global index is within the bounds of the arrays (N elements). This prevents threads from accessing memory out of bounds.
* **Arithmetic Operations**:
  + sum[idx] = a[idx] + b[idx];: Adds corresponding elements of a and b and stores the result in sum.
  + diff[idx] = a[idx] - b[idx];: Subtracts corresponding elements of b from a and stores the result in diff.
  + prod[idx] = a[idx] \* b[idx];: Multiplies corresponding elements of a and b and stores the result in prod.
  + quot[idx] = a[idx] / b[idx];: Divides corresponding elements of a by b, and stores the result in quot. We check if b[idx] != 0 to avoid division by zero, in which case we set quot[idx] = 0.

**4. Host Code**

**Define Host Arrays**

int h\_a[N], h\_b[N], h\_sum[N], h\_diff[N], h\_prod[N], h\_quot[N];

* We define six arrays on the host (CPU) to store the input and output data. Arrays h\_a and h\_b will hold the input data, while h\_sum, h\_diff, h\_prod, and h\_quot will hold the results of the arithmetic operations.

**Initialize Input Arrays**

for (int i = 0; i < N; i++) {

h\_a[i] = i + 1; // Elements 1 to 16

h\_b[i] = (i + 1) \* 2; // Elements 2, 4, 6, ... 32

}

* We initialize the input arrays h\_a and h\_b with some values:
  + h\_a[i] is initialized to i + 1, which results in values from 1 to 16.
  + h\_b[i] is initialized to (i + 1) \* 2, resulting in values from 2 to 32 (even numbers).

**Allocate Memory on the Device**

cudaMalloc(&d\_a, N \* sizeof(int));

cudaMalloc(&d\_b, N \* sizeof(int));

cudaMalloc(&d\_sum, N \* sizeof(int));

cudaMalloc(&d\_diff, N \* sizeof(int));

cudaMalloc(&d\_prod, N \* sizeof(int));

cudaMalloc(&d\_quot, N \* sizeof(int));

* We allocate memory on the device (GPU) for the input and output arrays using cudaMalloc. Each array is allocated N \* sizeof(int) bytes, where N is the number of elements in the array.

**Copy Data from Host to Device**

cudaMemcpy(d\_a, h\_a, N \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, N \* sizeof(int), cudaMemcpyHostToDevice);

* We use cudaMemcpy to copy the input arrays h\_a and h\_b from the host (CPU) to the device (GPU).

**Launch the Kernel**

int threadsPerBlock = 8;

int numBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;

perform\_arithmetic\_operations<<<numBlocks, threadsPerBlock>>>(d\_a, d\_b, d\_sum, d\_diff, d\_prod, d\_quot);

* threadsPerBlock = 8: We use 8 threads per block.
* numBlocks = (N + threadsPerBlock - 1) / threadsPerBlock: This formula calculates the number of blocks required to process all N elements. We round up to ensure that all elements are processed even if N is not divisible by 8.
* perform\_arithmetic\_operations<<<numBlocks, threadsPerBlock>>>(d\_a, d\_b, d\_sum, d\_diff, d\_prod, d\_quot): This launches the kernel on the GPU. The kernel will run with numBlocks blocks, each with threadsPerBlock threads.

**Synchronize the Device**

cudaDeviceSynchronize();

* cudaDeviceSynchronize() ensures that the host waits for all threads to complete their work on the GPU before proceeding.

**Copy Results Back to Host**

cudaMemcpy(h\_sum, d\_sum, N \* sizeof(int), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_diff, d\_diff, N \* sizeof(int), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_prod, d\_prod, N \* sizeof(int), cudaMemcpyDeviceToHost);

cudaMemcpy(h\_quot, d\_quot, N \* sizeof(int), cudaMemcpyDeviceToHost);

* We copy the results from the device back to the host using cudaMemcpy for each of the output arrays (sum, diff, prod, and quot).

**Display the Results**

std::cout << "Index\tA\tB\tSum\tDiff\tProd\tQuot" << std::endl;

for (int i = 0; i < N; i++) {

std::cout << i << "\t" << h\_a[i] << "\t" << h\_b[i] << "\t" << h\_sum[i] << "\t"

<< h\_diff[i] << "\t" << h\_prod[i] << "\t" << h\_quot[i] << std::endl;

}

* We print the results in a tabular format, showing the index, values of a and b, and the results of the arithmetic operations (sum, diff, prod, and quot).

**Free Device Memory**

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_sum);

cudaFree(d\_diff);

cudaFree(d\_prod);

cudaFree(d\_quot);

* After the program finishes, we free the memory allocated on the device using cudaFree.

**Summary of Operations:**

* **Arrays:** We have two input arrays (a and b) and four output arrays (sum, diff, prod, quot).
* **Kernel Execution:** Each thread performs arithmetic operations on a corresponding element of arrays a and b.
* **Parallel Execution:** The arithmetic operations are carried out in parallel on the GPU, leveraging multiple threads and blocks for efficient computation.
* **Synchronization and Memory Management:** The program handles memory allocation, data transfer, synchronization, and the kernel execution in a structured manner to ensure efficient GPU processing.