## Lab Manual: Vector Addition with CUDA

**Instructor:** Dr. Nilesh Rathod

**Aim:** Implement vector addition using CUDA to introduce students to parallelism, thread management, and memory allocation in GPU programming.

**Apparatus:**

* Computer with NVIDIA GPU and CUDA toolkit installed
* Text editor for writing CUDA code
* CUDA compiler (nvcc)
* IDE (optional)

**Theory:**

**CUDA (Compute Unified Device Architecture)** is a parallel computing platform and programming model created by NVIDIA. It allows developers to utilize the massive parallel processing power of GPUs for general-purpose computing tasks. CUDA programs execute on the GPU, which consists of a large number of cores called Streaming Multiprocessors (SMs). Each SM contains multiple threads, allowing for highly parallel computations.

**Vector Addition** is a fundamental operation in linear algebra, involving adding corresponding elements of two vectors to produce a resultant vector. In CUDA, this operation can be efficiently implemented by dividing the vectors into segments and assigning each segment to a thread block. Each thread within a block then performs the addition for its assigned elements, resulting in parallel execution.

**Procedure:**

1. **Code Structure:**
   * Create a CUDA kernel function to perform the vector addition.
   * Allocate memory on both the host (CPU) and device (GPU) for the vectors.
   * Copy the vectors from the host to the device.
   * Launch a grid of thread blocks, with each block containing a certain number of threads.
   * Execute the kernel function on the device.
   * Copy the result vector from the device to the host.
   * Free the allocated memory on both the host and device.
2. **Kernel Function Implementation:**
   * Use \_\_global\_\_ keyword to declare the kernel function.
   * Each thread will access its corresponding elements in the input vectors using its thread ID and block ID.
   * Calculate the sum of the elements and store the result in the output vector.
3. **Memory Allocation and Transfer:**
   * Use cudaMalloc to allocate memory on the device for the vectors.
   * Use cudaMemcpy to copy data from the host to the device and vice versa.
4. **Thread Management:**
   * Use <<<gridDim, blockDim>>> to launch a grid of thread blocks.
   * Use blockIdx, threadIdx, and blockDim to determine the thread’s position and the size of the block.
5. **Compilation and Execution:**
   * Compile the CUDA code using the nvcc compiler.
   * Run the compiled executable to perform the vector addition.

**Observations:**

* Record the time taken for vector addition on the GPU.
* Compare the performance of GPU-based implementation with a CPU-based implementation (if available).
* Analyze the impact of different thread block sizes and grid dimensions on performance.

**Conclusion:**

* Discuss the benefits of using CUDA for vector addition.
* Explain how parallelism and thread management contribute to the improved performance.
* Analyze the factors affecting the efficiency of CUDA implementation.
* Suggest improvements or further experiments to optimize the code.

**Sample Code:**

#include <cuda\_runtime.h>  
#include <device\_launch\_parameters.h>  
#include <iostream>  
  
// Kernel function for vector addition  
\_\_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];  
 }  
}  
  
int main() {  
 // Define vector size  
 int n = 1024;  
  
 // Allocate memory on host  
 float \*a = new float[n];  
 float \*b = new float[n];  
 float \*c = new float[n];  
  
 // Initialize vectors  
 for (int i = 0; i < n; i++) {  
 a[i] = i;  
 b[i] = i \* 2;  
 }  
  
 // Allocate memory on device  
 float \*d\_a, \*d\_b, \*d\_c;  
 cudaMalloc(&d\_a, n \* sizeof(float));  
 cudaMalloc(&d\_b, n \* sizeof(float));  
 cudaMalloc(&d\_c, n \* sizeof(float));  
  
 // Copy data from host to device  
 cudaMemcpy(d\_a, a, n \* sizeof(float), cudaMemcpyHostToDevice);  
 cudaMemcpy(d\_b, b, n \* sizeof(float), cudaMemcpyHostToDevice);  
  
 // Define thread block and grid dimensions  
 int threadsPerBlock = 256;  
 int blocksPerGrid = (n + threadsPerBlock - 1) / threadsPerBlock;  
  
 // Launch kernel function  
 vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d\_a, d\_b, d\_c, n);  
  
 // Copy result from device to host  
 cudaMemcpy(c, d\_c, n \* sizeof(float), cudaMemcpyDeviceToHost);  
  
 // Free device memory  
 cudaFree(d\_a);  
 cudaFree(d\_b);  
 cudaFree(d\_c);  
  
 // Print result  
 std::cout << "Result vector: ";  
 for (int i = 0; i < n; i++) {  
 std::cout << c[i] << " ";  
 }  
 std::cout << std::endl;  
  
 // Free host memory  
 delete[] a;  
 delete[] b;  
 delete[] c;  
  
 return 0;  
}

**Note:** This is a basic example and can be further modified to explore different optimizations, error handling, and advanced CUDA features.