## Lab Manual: Vector Addition using CUDA

**Instructor:** Dr. Nilesh Rathod

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

**1. Apparatus:**

* Computer with NVIDIA GPU and CUDA Toolkit installed.
* CUDA-enabled IDE (e.g., Visual Studio, NVIDIA Nsight)

**2. Theory:**

**Parallel Computing and CUDA:**

* Parallel computing involves dividing a computational task into smaller subtasks that can be executed simultaneously on multiple processing units.
* CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model created by NVIDIA for their GPUs. It allows programmers to utilize the massive parallelism of GPUs to accelerate computationally intensive tasks.
* **Threads and Blocks:** CUDA programs execute on a grid of thread blocks. Each block contains a number of threads that execute the same code concurrently.
* **Memory Management:** CUDA provides different memory spaces for data access:
  + **Global Memory:** The main memory accessible to all threads.
  + **Shared Memory:** Fast on-chip memory accessible by threads within a block.
  + **Constant Memory:** Read-only memory accessible by all threads.
  + **Texture Memory:** Memory optimized for spatial locality access.

**Vector Addition:**

* Vector addition is a fundamental operation in linear algebra, where corresponding elements of two vectors are added to produce a resulting vector.
* In parallel programming, the operation can be divided into subtasks, with each thread responsible for adding a pair of elements from the input vectors.

**3. Procedure:**

1. **Create a CUDA project:**
   * Start a new CUDA project in your chosen IDE.
   * Include the necessary CUDA headers (cuda\_runtime.h, device\_launch\_parameters.h).
2. **Define the vectors:**
   * Allocate memory for two input vectors (A and B) and the output vector (C) on the host (CPU).
   * Initialize the input vectors with random values.
3. **Allocate memory on the device (GPU):**
   * Use cudaMalloc() to allocate memory on the GPU for the input and output vectors.
4. **Copy data to the device:**
   * Use cudaMemcpy() to transfer the input vectors from the host to the device memory.
5. **Kernel function:**
   * Write a CUDA kernel function (e.g., vectorAddKernel) that performs the vector addition.
   * In the kernel function:
     + Use threadIdx and blockIdx to identify the current thread and block.
     + Calculate the index of the element being processed by the current thread.
     + Add the corresponding elements of the input vectors and store the result in the output vector.
6. **Launch the kernel:**
   * Determine the number of blocks and threads per block needed for the vector size.
   * Use cudaLaunchKernel() to launch the kernel on the GPU with the specified configuration.
7. **Copy data back to the host:**
   * Use cudaMemcpy() to transfer the output vector from the device to the host memory.
8. **Verify the results:**
   * Compare the elements of the calculated output vector (C) with the expected result from a sequential CPU implementation.

**4. Observations:**

* Record the time taken for the vector addition using the CUDA implementation.
* Compare the execution time with a sequential CPU implementation.
* Analyze the impact of different block and thread configurations on the performance.
* Investigate the memory usage of the CUDA program.

**5. Conclusion:**

* Summarize your observations and analyze the benefits of using CUDA for vector addition.
* Discuss the factors influencing the performance of the parallel implementation.
* Explain the concepts of parallelism, thread management, and memory allocation demonstrated in the experiment.

**6. Example Code:**

#include <cuda\_runtime.h>  
#include <device\_launch\_parameters.h>  
#include <iostream>  
  
#define SIZE 1024 // Size of the vectors  
  
// Kernel function for vector addition  
\_\_global\_\_ void vectorAddKernel(float\* A, float\* B, float\* C) {  
 int idx = blockIdx.x \* blockDim.x + threadIdx.x;  
 if (idx < SIZE) {  
 C[idx] = A[idx] + B[idx];  
 }  
}  
  
int main() {  
 // Allocate memory on the host  
 float\* A = new float[SIZE];  
 float\* B = new float[SIZE];  
 float\* C = new float[SIZE];  
  
 // Initialize input vectors  
 for (int i = 0; i < SIZE; i++) {  
 A[i] = rand() % 10;  
 B[i] = rand() % 10;  
 }  
  
 // Allocate memory on the device  
 float\* d\_A, \*d\_B, \*d\_C;  
 cudaMalloc(&d\_A, SIZE \* sizeof(float));  
 cudaMalloc(&d\_B, SIZE \* sizeof(float));  
 cudaMalloc(&d\_C, SIZE \* sizeof(float));  
  
 // Copy input vectors to the device  
 cudaMemcpy(d\_A, A, SIZE \* sizeof(float), cudaMemcpyHostToDevice);  
 cudaMemcpy(d\_B, B, SIZE \* sizeof(float), cudaMemcpyHostToDevice);  
  
 // Launch the kernel  
 int threadsPerBlock = 256;  
 int blocksPerGrid = (SIZE + threadsPerBlock - 1) / threadsPerBlock;  
 vectorAddKernel<<<blocksPerGrid, threadsPerBlock>>>(d\_A, d\_B, d\_C);  
  
 // Copy the output vector back to the host  
 cudaMemcpy(C, d\_C, SIZE \* sizeof(float), cudaMemcpyDeviceToHost);  
  
 // Verify the results  
 for (int i = 0; i < SIZE; i++) {  
 std::cout << "A[" << i << "] = " << A[i] << ", B[" << i << "] = " << B[i] << ", C[" << i << "] = " << C[i] << std::endl;  
 }  
  
 // Free device memory  
 cudaFree(d\_A);  
 cudaFree(d\_B);  
 cudaFree(d\_C);  
  
 // Free host memory  
 delete[] A;  
 delete[] B;  
 delete[] C;  
  
 return 0;  
}

This lab manual aims to provide a comprehensive guide for students to understand and implement parallel vector addition using CUDA. By performing this experiment, students will gain practical experience with thread management, memory allocation, and the benefits of parallel programming on GPUs.