**PP LAB WEEK-8**

# DSE VI-A2 Divansh Prasad 210968140

1) Write a program in CUDA to add two vectors of length N using

a) block size as N

b) N threads

a) Block Size as N

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// CUDA kernel to add two vectors

\_\_global\_\_ void vecAddKernel(int\* A, int\* B, int\* C, int n) {

// Get the global thread ID

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

// Check if the thread is within the vector range

if (id < n) {

// Add the corresponding elements of A and B

C[id] = A[id] + B[id];

}

}

// Function to add two vectors using CUDA

void vecAdd(int\* A, int\* B, int\* C, int n) {

// Allocate device memory for the vectors

int\* d\_A, \* d\_B, \* d\_C;

cudaMalloc(&d\_A, n \* sizeof(int));

cudaMalloc(&d\_B, n \* sizeof(int));

cudaMalloc(&d\_C, n \* sizeof(int));

// Copy the vectors from host to device

cudaMemcpy(d\_A, A, n \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, B, n \* sizeof(int), cudaMemcpyHostToDevice);

// Define the block size and grid size

int blockSize = 1; // Number of threads per block

int gridSize = n; // Number of blocks per grid

// Launch the kernel with the specified configuration

vecAddKernel << <gridSize, blockSize >> > (d\_A, d\_B, d\_C, n);

// Copy the result vector from device to host

cudaMemcpy(C, d\_C, n \* sizeof(int), cudaMemcpyDeviceToHost);

// Free the device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

}

// Main function to test the vector addition

int main() {

// Get the vector length from the user

int n;

printf("Enter the vector length: ");

scanf("%d", &n);

// Allocate host memory for the vectors

int\* A = (int\*)malloc(n \* sizeof(int));

int\* B = (int\*)malloc(n \* sizeof(int));

int\* C = (int\*)malloc(n \* sizeof(int));

// Initialize the vectors with random values

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

A[i] = rand() % 100;

B[i] = rand() % 100;

C[i] = 0;

}

// Add the vectors using CUDA

vecAdd(A, B, C, n);

printf("\nThe first vector is:\n");

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

printf("%d\t", A[i]);

}

printf("\nThe second vector is:\n");

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

printf("%d\t", B[i]);

}

// Print the result vector

printf("\n\nThe result vector is:\n");

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

printf("%d\t", C[i]);

}

printf("\n");

// Free the host memory

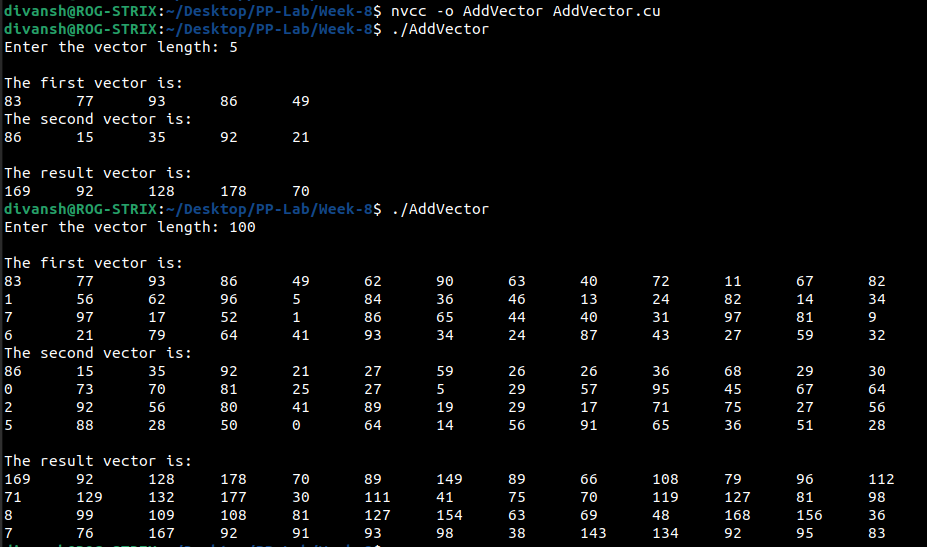
free(A);

free(B);

free(C);

return 0;

}



b) N Threads

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// CUDA kernel to add two vectors

\_\_global\_\_ void vecAddKernel(int\* A, int\* B, int\* C, int n) {

// Get the global thread ID

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

// Check if the thread is within the vector range

if (id < n) {

// Add the corresponding elements of A and B

C[id] = A[id] + B[id];

}

}

void vecAdd(int\* A, int\* B, int\* C, int n) {

int\* d\_A, \* d\_B, \* d\_C;

cudaMalloc(&d\_A, n \* sizeof(int));

cudaMalloc(&d\_B, n \* sizeof(int));

cudaMalloc(&d\_C, n \* sizeof(int));

// Copy the vectors from host to device

cudaMemcpy(d\_A, A, n \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, B, n \* sizeof(int), cudaMemcpyHostToDevice);

// Define the block size and grid size

int blockSize = n; // Number of threads per block

int gridSize = 1; // Number of blocks per grid

vecAddKernel << <gridSize, blockSize >> > (d\_A, d\_B, d\_C, n);

cudaMemcpy(C, d\_C, n \* sizeof(int), cudaMemcpyDeviceToHost);

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

}

int main() {

int n;

printf("Enter the vector length: ");

scanf("%d", &n);

int\* A = (int\*)malloc(n \* sizeof(int));

int\* B = (int\*)malloc(n \* sizeof(int));

int\* C = (int\*)malloc(n \* sizeof(int));

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

A[i] = rand() % 100;

B[i] = rand() % 100;

C[i] = 0;

}

vecAdd(A, B, C, n);

printf("\nThe first vector is:\n");

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

printf("%d\t", A[i]);

}

printf("\nThe second vector is:\n");

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

printf("%d\t", B[i]);

}

printf("\n\nThe result vector is:\n");

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

printf("%d\t", C[i]);

}

printf("\n");

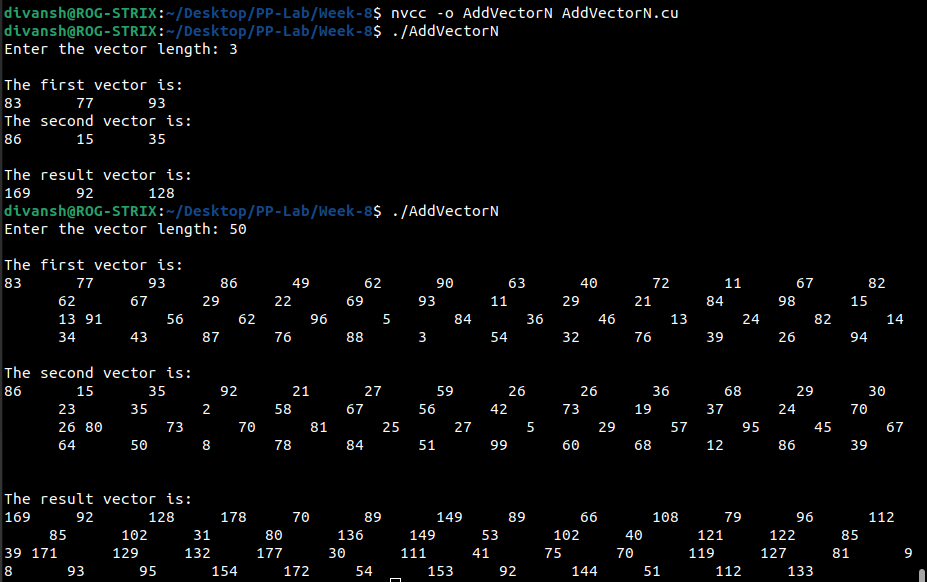
free(A);

free(B);

free(C);

return 0;

}



2) Implement a CUDA program to add two vectors of length N by keeping the number of threads per block as 256 (constant) and vary the number of blocks to handle N elements.

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// CUDA kernel to add two vectors

\_\_global\_\_ void vecAddKernel(int\* A, int\* B, int\* C, int n) {

// Get the global thread ID

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

// Check if the thread is within the vector range

if (id < n) {

// Add the corresponding elements of A and B

C[id] = A[id] + B[id];

}

}

// Function to add two vectors using CUDA

void vecAdd(int\* A, int\* B, int\* C, int n) {

// Allocate device memory for the vectors

int\* d\_A, \* d\_B, \* d\_C;

cudaMalloc(&d\_A, n \* sizeof(int));

cudaMalloc(&d\_B, n \* sizeof(int));

cudaMalloc(&d\_C, n \* sizeof(int));

// Copy the vectors from host to device

cudaMemcpy(d\_A, A, n \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, B, n \* sizeof(int), cudaMemcpyHostToDevice);

// Define the block size and grid size

int blockSize = 256; // Number of threads per block

int gridSize = (n + 256 - 1) / n; // Number of blocks per grid

// Launch the kernel with the specified configuration

vecAddKernel << <gridSize, blockSize >> > (d\_A, d\_B, d\_C, n);

// Copy the result vector from device to host

cudaMemcpy(C, d\_C, n \* sizeof(int), cudaMemcpyDeviceToHost);

// Free the device memory

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

}

// Main function to test the vector addition

int main() {

// Get the vector length from the user

int n;

printf("Enter the vector length: ");

scanf("%d", &n);

// Allocate host memory for the vectors

int\* A = (int\*)malloc(n \* sizeof(int));

int\* B = (int\*)malloc(n \* sizeof(int));

int\* C = (int\*)malloc(n \* sizeof(int));

// Initialize the vectors with random values

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

A[i] = rand() % 100;

B[i] = rand() % 100;

C[i] = 0;

}

// Add the vectors using CUDA

vecAdd(A, B, C, n);

printf("\nThe first vector is:\n");

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

printf("%d\t", A[i]);

}

printf("\nThe second vector is:\n");

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

printf("%d\t", B[i]);

}

// Print the result vector

printf("\n\nThe result vector is:\n");

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

printf("%d\t", C[i]);

}

printf("\n");

// Free the host memory

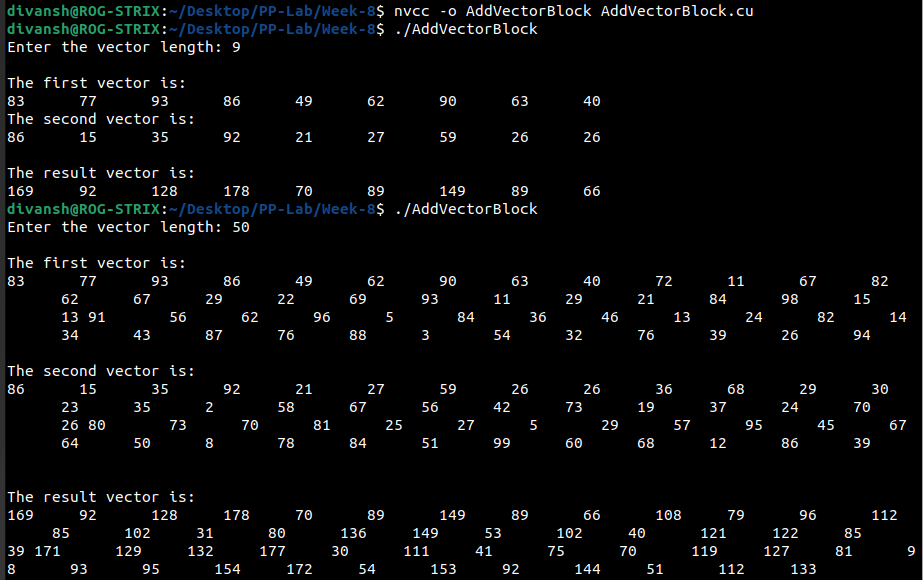
free(A);

free(B);

free(C);

return 0;

}



3) Write a program in CUDA which performs convolution operation on one dimensional input array N of size width using a mask array M of size mask\_width to produce the resultant one-dimensional array P of size width.

#include <stdio.h>

#include <stdlib.h>

#include <cuda.h>

#define TILE\_WIDTH 16 // number of threads per block

// kernel function for convolution

\_\_global\_\_ void convolve(float\* N, float\* M, float\* P, int width, int mask\_width) {

// calculate global thread index

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

// initialize output element to zero

float P\_val = 0;

// loop over the mask array

for (int j = 0; j < mask\_width; j++) {

// calculate the index of the input element

int k = i - (mask\_width / 2) + j;

// check if the index is within bounds

if (k >= 0 && k < width) {

// accumulate the product of input and mask elements

P\_val += N[k] \* M[j];

}

}

// store the output element in the output array

P[i] = P\_val;

}

int main() {

// initialize input array N

float N[] = { 1, 2, 3, 4, 5 };

// initialize mask array M

float M[] = { 0.2, 0.2, 0.2, 0.2, 0.2 };

// get the sizes of the arrays

int width = sizeof(N) / sizeof(float);

int mask\_width = sizeof(M) / sizeof(float);

// allocate memory for output array P on host

float\* P = (float\*)malloc(width \* sizeof(float));

// allocate memory for arrays on device

float\* d\_N, \* d\_M, \* d\_P;

cudaMalloc((void\*\*)&d\_N, width \* sizeof(float));

cudaMalloc((void\*\*)&d\_M, mask\_width \* sizeof(float));

cudaMalloc((void\*\*)&d\_P, width \* sizeof(float));

// copy arrays from host to device

cudaMemcpy(d\_N, N, width \* sizeof(float), cudaMemcpyHostToDevice);

cudaMemcpy(d\_M, M, mask\_width \* sizeof(float), cudaMemcpyHostToDevice);

// calculate number of blocks needed

int num\_blocks = ceil((float)width / TILE\_WIDTH);

// launch kernel function

convolve << <num\_blocks, TILE\_WIDTH >> > (d\_N, d\_M, d\_P, width, mask\_width);

// copy output array from device to host

cudaMemcpy(P, d\_P, width \* sizeof(float), cudaMemcpyDeviceToHost);

printf("Input array P:\n");

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

printf("%f ", N[i]);

}

printf("\n");

printf("Mask array P:\n");

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

printf("%f ", P[i]);

}

printf("\n");

// print output array

printf("Output array P:\n");

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

printf("%f ", P[i]);

}

printf("\n");

// free memory on host and device

free(P);

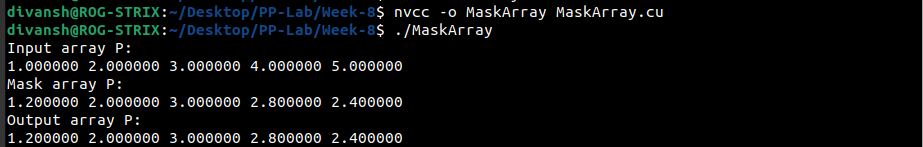
cudaFree(d\_N);

cudaFree(d\_M);

cudaFree(d\_P);

return 0;

}



4) Write a program in CUDA to process an ID array containing angles in radians to generate sine of the angles in the output array. Use appropriate functions.

#include <stdio.h>

#include <math.h>

\_\_global\_\_ void computeSine(float \*input, float \*output, int size) {

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

if (tid < size) {

output[tid] = sinf(input[tid]);

}

}

int main() {

int size = 10; // Size of the input array

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

// Allocate memory for the host arrays

float \*h\_input = (float\*)malloc(bytes);

float \*h\_output = (float\*)malloc(bytes);

// Initialize the input array with angles in radians

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

h\_input[i] = 100\*i; // Increment angle by 0.01 radians

}

// Allocate memory for the device arrays

float \*d\_input, \*d\_output;

cudaMalloc(&d\_input, bytes);

cudaMalloc(&d\_output, bytes);

// Copy the input array from host to device

cudaMemcpy(d\_input, h\_input, bytes, cudaMemcpyHostToDevice);

// Define grid and block dimensions

int threadsPerBlock = 256;

int blocksPerGrid = (size + threadsPerBlock - 1) / threadsPerBlock;

// Launch the kernel

computeSine<<<blocksPerGrid, threadsPerBlock>>>(d\_input, d\_output, size);

// Copy the result array from device to host

cudaMemcpy(h\_output, d\_output, bytes, cudaMemcpyDeviceToHost);

// Print the result

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

printf("sin(%f radians) = %f\n", h\_input[i], h\_output[i]);

}

// Free device memory

cudaFree(d\_input);

cudaFree(d\_output);

// Free host memory

free(h\_input);

free(h\_output);

return 0;

}

