# 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;
}
```

```
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ nvcc -o AddVector AddVector.cu
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ ./AddVector
Enter the vector length: 5

The first vector is:
83      77      93      86      49
The second vector is:
86      15      35      92      21

The result vector is:
169     92      128     178     70
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ ./AddVector
Enter the vector length: 100

The first vector is:
83      77      93      86      49      62      90      63      40      72      11      67      82
1       56      62      96      5       84      36      46      13      24      82      14      34
7       97      17      52      1       86      65      44      40      31      97      81      9
6       21      79      64      41      93      34      24      87      43      27      59      32
The second vector is:
86      15      35      92      21      27      59      26      26      36      68      29      30
0       73      70      81      25      27      5       29      57      95      45      67      64
2       92      56      80      41      89      19      29      17      71      75      27      56
5       88      28      50      0       64      14      56      91      65      36      51      28

The result vector is:
169     92      128     178     70      89      149     89      66      108     79      96      112
71      129     132     177     30      111     41      75      70      119     127     81      98
8       99      109     108     81      127     154     63      69      48      168     156     36
7       76      167     92      91      93      98      38      143     134     92      95      83
```

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;
}
```

```
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ nvcc -o AddVectorN AddVectorN.cu
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ ./AddVectorN
Enter the vector length: 3

The first vector is:
83      77      93
The second vector is:
86      15      35

The result vector is:
169     92      128
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ ./AddVectorN
Enter the vector length: 50

The first vector is:
83      77      93      86      49      62      90      63      40      72      11      67      82
      62      67      29      22      69      93      11      29      21      84      98      15
      13 91       56      62      96      5       84      36      46      13      24      82      14
      34      43      87      76      88      3       54      32      76      39      26      94

The second vector is:
86      15      35      92      21      27      59      26      26      36      68      29      30
      23      35      2       58      67      56      42      73      19      37      24      70
      26 80       73      70      81      25      27      5       29      57      95      45      67
      64      50      8       78      84      51      99      60      68      12      86      39


The result vector is:
169     92      128     178     70      89      149     89      66      108     79      96      112
     85      102     31      80      136     149     53      102     40      121     122     85
39 171      129     132     177     30      111     41      75      70      119     127     81      9
8       93      95      154     172     54      153     92      144     51      112     133
```

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;
}
```

```
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ nvcc -o AddVectorBlock AddVectorBlock.cu
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ ./AddVectorBlock
Enter the vector length: 9

The first vector is:
83      77      93      86      49      62      90      63      40
The second vector is:
86      15      35      92      21      27      59      26      26

The result vector is:
169     92      128     178     70      89      149     89      66
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ ./AddVectorBlock
Enter the vector length: 50

The first vector is:
83      77      93      86      49      62      90      63      40      72      11      67      82
      62      67      29      22      69      93      11      29      21      84      98      15
      13 91       56      62      96      5       84      36      46      13      24      82      14
      34      43      87      76      88      3       54      32      76      39      26      94

The second vector is:
86      15      35      92      21      27      59      26      26      36      68      29      30
      23      35      2       58      67      56      42      73      19      37      24      70
      26 80       73      70      81      25      27      5       29      57      95      45      67
      64      50      8       78      84      51      99      60      68      12      86      39


The result vector is:
169     92      128     178     70      89      149     89      66      108     79      96      112
     85      102     31      80      136     149     53      102     40      121     122     85
39 171      129     132     177     30      111     41      75      70      119     127     81      9
8       93      95      154     172     54      153     92      144     51      112     133
```

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;
}
```

```
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ nvcc -o MaskArray MaskArray.cu
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ ./MaskArray
Input array P:
1.000000 2.000000 3.000000 4.000000 5.000000
Mask array P:
1.200000 2.000000 3.000000 2.800000 2.400000
Output array P:
1.200000 2.000000 3.000000 2.800000 2.400000
```

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;
}
```

```
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ nvcc -o Sine Sine.cu
divansh@ROG-STRIX:~/Desktop/PP-Lab/Week-8$ ./Sine
sin(0.000000 radians) = 0.000000
sin(100.000000 radians) = -0.506366
sin(200.000000 radians) = -0.873297
sin(300.000000 radians) = -0.999756
sin(400.000000 radians) = -0.850919
sin(500.000000 radians) = -0.467772
sin(600.000000 radians) = 0.044182
sin(700.000000 radians) = 0.543971
sin(800.000000 radians) = 0.893970
sin(900.000000 radians) = 0.997803
```