Name : Rajvardhan Reddy
Reg no : 180905093
Sec : B , Batch – B1
Roll No : 19

PP Lab - Week 8 : Programs on Parallel Patterns in CUDA

P1) 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*. Find the time taken by the kernel.

Code:

```
#include <cuda.h>
#include <stdlib.h>
#include <stdio.h>

#define MAX_WIDTH 7
#define MAX_MASK_WIDTH 5

__global__ void kernel_1d_conv_const_mem(int *N, int *M, int *P, int mask_width, int width)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    int Pvalue = 0;
    int N_start_point = i - (mask_width/2);

   for (int j = 0; j < mask_width; j++)
    {
        if (N_start_point + j >= 0 && N_start_point + j < width)
        {
            Pvalue += N[N_start_point + j] * M[j];
        }
    }

    P[i] = Pvalue;
```

```
}

int main()
{
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    int width = MAX_WIDTH;
    int mask_width = MAX_MASK_WIDTH;

    int *h_N = (int*) calloc(width, sizeof(int));
    int *h_P = (int*) calloc(width, sizeof(int));
    int *h_M = (int*) calloc(mask_width, sizeof(int));

    for (int i = 0; i < width; i++)
    {
        h_N[i] = i + 1;
    }

    h_M[0] = 25;
    h_M[1] = 50;
    h_M[2] = 75;
    h_M[3] = 100;
    h_M[4] = 125;

    int *d_N;
    int *d_M;
    int *d_P;

    int size = width * sizeof(int);

    cudaMalloc((void**) &d_N, size);
    cudaMalloc((void**) &d_M, size);
    cudaMalloc((void**) &d_P, size);
```

```
    cudaMemcpy(d_N, h_N, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_P, h_P, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_M, h_M, size, cudaMemcpyHostToDevice);

    cudaEventRecord(start);
    kernel_1d_conv_const_mem<<<1, MAX_WIDTH>>>(d_N, d_M, d_P, mask_width,
width);
   cudaEventRecord(stop);

    cudaMemcpy(h_P, d_P, size, cudaMemcpyDeviceToHost);

    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("P: ");
    for (int i = 0; i < width; i++)
    {
        printf("%d, ", h_P[i]);
    }
    printf("\n");

    printf("Time to taken for 1D convolution kernel %f ms\n", milliseconds);

}
```

Output :

```
[1] !nvcc ./0.cu -o 0.out
    !./0.out

    P: 474, 736, 1031, 1326, 1621, 1140, 810,
    Time to taken for 1D convolution kernel 0.022336 ms
```

```
[2] !nvcc ./0.cu -o 0.out
    !./0.out

    P: 650, 1000, 1375, 1750, 2125, 1500, 950,
    Time to taken for 1D convolution kernel 0.018560 ms
```

P2) Write a program in CUDA to perform parallel Sparse Matrix - Vector Multiplication using compressed sparse row (CSR) storage format. Represent the input sparse matrix in CSR format in the host code.

Code:

```
 %%cu

#include<stdio.h>
#include<stdlib.h>
#include"cuda_runtime.h"
#include"device_launch_parameters.h"

__global__ void SpMV_CSR(int num_rows,int *data,int *col_index,int *row_ptr,int *x,int *y)
{
  int row=threadIdx.x;
  if(row<num_rows)
  {
    int dot=0;
    int row_start=row_ptr[row];
    int row_end=row_ptr[row+1];
    for(int i=row_start;i<row_end;i++)
    {
      dot+= data[i]*x[col_index[i]];
    }
    y[row]=dot;
  }
}

int main()
{
    //declarations
    int n=4;
    int y[n],row_ptr[n+1];
    int ipmat[n][n]={{0,0,3,4},{0,0,0,0},{0,5,0,7},{0,2,6,0}};
    int x[]={7,8,9,10};
    int nonzerocount=0;
    //finding number of non zero elements and row ptr array
    for(int i=0;i<n;i++)
    {
      row_ptr[i]=nonzerocount;
      for(int j=0;j<n;j++)
      {
        if(ipmat[i][j]!=0)
```

```
    {
      nonzerocount++;
    }
    printf("%d\t",ipmat[i][j]);
  }
  printf("\n");
}
row_ptr[n]=nonzerocount;
int data[nonzerocount],col_index[nonzerocount];
int k=0;
//finding data and col_index array
for(int i=0;i<n;i++)
{
  for(int j=0;j<n;j++)
  {
    if(ipmat[i][j]!=0)
    {
      data[k]=ipmat[i][j];
      col_index[k++]=j;
    }
  }
}
printf("\ndata array\t");
for(int i=0;i<nonzerocount;i++)
{
  printf("%d\t",data[i]);
}
printf("\ncol_index array\t");
for(int i=0;i<nonzerocount;i++)
{
  printf("%d\t",col_index[i]);
}
printf("\nrow_ptr array\t");
for(int i=0;i<=n;i++)
{
  printf("%d\t",row_ptr[i]);
}
printf("\nvector X\t");
for(int i=0;i<n;i++)
{
  printf("%d\t",x[i]);
}
int *d_data,*d_col_index,*d_row_ptr,*d_x,*d_y;
//memory allocations
```

```
    cudaMalloc((void**)&d_data,nonzerocount*sizeof(int));
    cudaMalloc((void**)&d_col_index,nonzerocount*sizeof(int));
    cudaMalloc((void**)&d_row_ptr,(n+1)*sizeof(int));
        cudaMalloc((void**)&d_x,n*sizeof(int));
        cudaMalloc((void**)&d_y,n*sizeof(int));
    //copy from host to device
    cudaMemcpy(d_data,data,nonzerocount*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(d_col_index,col_index,nonzerocount*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(d_row_ptr,row_ptr,(n+1)*sizeof(int),cudaMemcpyHostToDevice);
        cudaMemcpy(d_x,x,n*sizeof(int),cudaMemcpyHostToDevice);
    //run kernel
    SpMV_CSR<<<1,n>>>(n,d_data,d_col_index,d_row_ptr,d_x,d_y);
    //copy from device to host
    cudaMemcpy(y,d_y,n*sizeof(int),cudaMemcpyDeviceToHost);

    printf("\nresult\t");
    for(int i=0;i<n;i++)
    {
      printf("%d\t",y[i]);
    }
    //free memory
    cudaFree(d_data);
    cudaFree(d_col_index);
    cudaFree(d_row_ptr);
        cudaFree(d_x);
        cudaFree(d_y);
    return 0;
}
```

Output:

```
0       0       3       4
0       0       0       0
0       5       0       7
0       2       6       0

data array      3       4       5       7       2       6
col_index array 2       3       1       3       1       2
row_ptr array   0       2       2       4       6
vector X        7       8       9       10
result  67      0       110     70
```

P3) Write a program in CUDA to perform matrix multiplication using 2D Grid and 2D Block.

<u>Code:</u>

```
%%cu
#include<stdio.h>
#include<stdlib.h>

__global__ void matrixMul(const int *a, const int *b, int *c, int m,int n,int o)
{
  //row and col calculations
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  c[row * o + col] = 0;
  //calculating one element
  for (int k = 0; k < n; k++) {
    c[row * o + col] += a[row * n + k] * b[k * o + col];
  }
}

int main()
{
  //declarations
  int size =sizeof(int);
  int m=4,n=2,o=4;
  int a[m][n];
  int b[n][o];
  int c[m][o];
  for(int i=0;i<m;i++)
  {
    for(int j=0;j<n;j++)
    {
      a[i][j]=33;
    }
  }
  for(int i=0;i<n;i++)
  {
    for(int j=0;j<o;j++)
    {
      b[i][j]=57;
    }
  }
```

```
int *d_a, *d_b, *d_c;
//memory allocations
cudaMalloc(&d_a,m*n*size);
cudaMalloc(&d_b,n*o*size);
cudaMalloc(&d_c,m*o*size);
//copy from host to device
cudaMemcpy(d_a,a,m*n*size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b,b,n*o*size, cudaMemcpyHostToDevice);
//dimensions for grid and block
int thread=2;
dim3 threads(thread,thread);
dim3 blocks((m*o)/(4*thread),(m*o)/(4*thread));

//run kernel
matrixMul<<<blocks, threads>>>(d_a, d_b, d_c,m,n,o);

// copy from device to host
cudaMemcpy(c, d_c,m*o*size, cudaMemcpyDeviceToHost);
printf("Matrix A\n");
for(int i=0;i<m;i++)
{
  for(int j=0;j<n;j++)
  {
    printf("%d\t",a[i][j]);
  }
  printf("\n");
}
printf("Matrix B\n");
for(int i=0;i<n;i++)
{
  for(int j=0;j<o;j++)
  {
    printf("%d\t",b[i][j]);
  }
  printf("\n");
}
printf("Matrix C\n");
for(int i=0;i<m;i++)
{
  for(int j=0;j<o;j++)
  {
    printf("%d\t",c[i][j]);
  }
  printf("\n");
```

```
  }
  //free memory
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  return 0;
}
```

Output:

```
Matrix A
12      12
12      12
12      12
12      12
Matrix B
2       2       2       2
2       2       2       2
Matrix C
48      48      48      48
48      48      48      48
48      48      48      48
48      48      48      48
```

```
Matrix A
33      33
33      33
33      33
33      33
Matrix B
57      57      57      57
57      57      57      57
Matrix C
3762    3762    3762    3762
3762    3762    3762    3762
3762    3762    3762    3762
3762    3762    3762    3762
```

P4) Write a CUDA program to perform 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* using ***constant Memory*** for Mask array. Add another kernel function to the same program to perform 1D convolution using ***shared memory***. Find and display the time taken by both the kernels.

Code:

```
#include <cuda.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#define MAX_WIDTH 7
#define MAX_MASK_WIDTH 5
__constant__ int M[MAX_MASK_WIDTH];

__global__ void kernel_1d_conv_const_mem(int *N, int *P, int mask_width, int width)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;

   int Pvalue = 0;
   int N_start_point = i - (mask_width/2);

   for (int j = 0; j < mask_width; j++)
   {
      if (N_start_point + j >= 0 && N_start_point + j < width)
      {
         Pvalue += N[N_start_point + j] * M[j];
      }
   }

   P[i] = Pvalue;
}

__global__ void kernel_1d_conv_shared_mem(int *N, int *P, int mask_width, int width)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;

   extern __shared__ int N_shared[];

   // copy to shared memory
   N_shared[i] = N[i];

   __syncthreads();

   int Pvalue = 0;
   int N_start_point = i - (mask_width/2);

   for (int j = 0; j < mask_width; j++)
   {
      if (N_start_point + j >= 0 && N_start_point + j < width)
      {
         Pvalue += N_shared[N_start_point + j] * M[j];
      }
   }
```

```
    P[i] = Pvalue;
}


int main()
{
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    int width = MAX_WIDTH;
    int mask_width = MAX_MASK_WIDTH;

    int *h_N = (int*) calloc(width, sizeof(int));
    int *h_P = (int*) calloc(width, sizeof(int));
    int *h_M = (int*) calloc(mask_width, sizeof(int));

    for (int i = 0; i < width; i++)
    {
        h_N[i] = i + 1;
    }

    h_M[0] = 7;
    h_M[1] = 5;
    h_M[2] = 9;
    h_M[3] = 8;
    h_M[4] = 6;

    int *d_N;
    int *d_P;

    int size = width * sizeof(int);

    cudaMalloc((void**) &d_N, size);
    cudaMalloc((void**) &d_P, size);

    cudaMemcpy(d_N, h_N, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_P, h_P, size, cudaMemcpyHostToDevice);
    cudaMemcpyToSymbol(M, h_M, mask_width * sizeof(int));

    cudaEventRecord(start);
    kernel_1d_conv_const_mem<<<1, MAX_WIDTH>>>(d_N, d_P, mask_width, width);
    cudaEventRecord(stop);
```

```
    cudaMemcpy(h_P, d_P, size, cudaMemcpyDeviceToHost);

    cudaEventSynchronize(stop);
    float milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("P: ");
    for (int i = 0; i < width; i++)
    {
        printf("%d, ", h_P[i]);
    }
    printf("\n");

    printf("Time to taken for 1D convolution kernel with constant memory for M is %f ms\n",
milliseconds);

    printf("= = = = = = = = = = \n");

    /* == Shared Memory == */
    h_P = (int*) calloc(width, sizeof(int));
    cudaMemcpy(d_P, h_P, size, cudaMemcpyHostToDevice);

    cudaEventRecord(start);
    kernel_1d_conv_shared_mem<<<1, MAX_WIDTH, MAX_WIDTH>>>(d_N, d_P,
mask_width, width);
    cudaEventRecord(stop);

    cudaMemcpy(h_P, d_P, size, cudaMemcpyDeviceToHost);

    cudaEventSynchronize(stop);
    milliseconds = 0;
    cudaEventElapsedTime(&milliseconds, start, stop);

    printf("P: ");
    for (int i = 0; i < width; i++)
    {
        printf("%d, ", h_P[i]);
    }
    printf("\n");

    printf("Time to taken for 1D convolution kernel with shared memory is %f ms\n",
milliseconds);

}
```

Output:

```
!nvcc ./1.cu -o 1.out
!./1.out
```

```
P: 22, 38, 57, 76, 95, 90, 74,
Time to taken for 1D convolution kernel with constant memory for M is 0.033120 ms
= = = = = = = = = =
P: 22, 38, 57, 76, 95, 90, 74,
Time to taken for 1D convolution kernel with shared memory is 0.011136 ms
```

```
[ ] !nvcc ./1.cu -o 1.out
    !./1.out
```

```
P: 43, 71, 106, 141, 176, 163, 128,
Time to taken for 1D convolution kernel with constant memory for M is 0.017120 ms
= = = = = = = = = =
P: 43, 71, 106, 141, 176, 163, 128,
Time to taken for 1D convolution kernel with shared memory is 0.010752 ms
```