SANSKAAR PATNI
180905134 CSE C 23
PP LAB 8

Programs on Parallel Patterns in CUDA

Qsn given to solve on 26th May:
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:
```
%%cu
#include <stdio.h>
#include <stdlib.h>

__global__ void convolution1DBasic(int *N,int *M, int *P, int
mask_width,int width) {
   int id=blockIdx.x * blockDim.x + threadIdx.x;
   int Pvalue=0;
   int start=id-(mask_width/2);
   for(int j=0;j<mask_width;j++){
       if(start+j>=0 && start+j<width){
         Pvalue+=N[start+j]*M[j];
       }
   }
 P[id]=Pvalue;
}

int main() {
 int width=7;
 int mask_width=5;
 int N[width]={1,2,3,4,5,6,7};
 int M[mask_width]={3,4,5,4,3};
 int P[width]={0,0,0,0,0,0,0};

 int *d_N,*d_M, *d_P;

 int maskArray_size=mask_width*sizeof(int);
 int inpArray_size=width*sizeof(int);
  cudaEvent_t start, stop;

 cudaEventCreate(&start);
```

```
 cudaEventCreate(&stop);

 cudaMalloc((void **)&d_N, inpArray_size);
 cudaMalloc((void **)&d_M, maskArray_size);
 cudaMalloc((void **)&d_P, inpArray_size);

 cudaMemcpy(d_N, &N, inpArray_size, cudaMemcpyHostToDevice);
 cudaMemcpy(d_M, &M, maskArray_size, cudaMemcpyHostToDevice);
  cudaEventRecord(start, 0);
 convolution1DBasic<<<1,width>>>(d_N, d_M,d_P, mask_width,width);
  cudaError err = cudaMemcpy(&P, d_P, inpArray_size,
cudaMemcpyDeviceToHost);
 if(err!=cudaSuccess) {
       printf("CUDA error copying to Host: %s\n",
cudaGetErrorString(err));
 }
 cudaEventRecord(stop, 0);
 cudaEventSynchronize(stop);
 float elapsedTime;
 cudaEventElapsedTime(&elapsedTime, start,stop);
 printf("1a. 1D Parallel Convolution Operation\n");
 printf("Time Taken: %f\n",elapsedTime);
 printf("Resultant Array P is\n");
 for(int k=0;k<width;k++){
     printf("%d ",P[k]);
 }
 // Cleanup
 cudaFree(d_N);
 cudaFree(d_M);
 cudaFree(d_P);
 return 0;
}
```

Screenshot:

```
1a. 1D Parallel Convolution Operation
Time Taken: 0.031168
Resultant Array P is
22 38 57 76 95 90 74
```

Today’s programs:

1.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:

```
%%cu
#include <stdio.h>
#include <stdlib.h>
#define MAX_MASK_WIDTH 10
#define TILE_SIZE 4
__constant__ int M[MAX_MASK_WIDTH];

__global__ void convolution1DConstant(int *N,int *P, int mask_width,int
width) {
   int id=blockIdx.x * blockDim.x + threadIdx.x;
   int Pvalue=0;
   int start=id-(mask_width/2);
   for(int j=0;j<mask_width;j++){
       if(start+j>=0 && start+j<width){
         Pvalue+=N[start+j]*M[j];
       }
   }
 P[id]=Pvalue;
}

__global__ void convolution1DShared(int *N, int *P, int mask_width, int
width)
{
   int id = blockIdx.x*blockDim.x+threadIdx.x;
   __shared__ float N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];

   int n = mask_width/2;
   int halo_index_left = (blockIdx.x-1)*blockDim.x + threadIdx.x;
   if(threadIdx.x >= blockDim.x - n)
   {
       N_ds[threadIdx.x - (blockDim.x-n)] = (halo_index_left <
0)?0:N[halo_index_left];
   }
   N_ds[n+threadIdx.x] = N[blockIdx.x*blockDim.x+threadIdx.x];
   int halo_index_right = (blockIdx.x+1)*blockDim.x+threadIdx.x;
   if(threadIdx.x < n)
   {
       N_ds[n+blockDim.x+threadIdx.x] =
(halo_index_right>=width)?0:N[halo_index_right];
```

```
    }
    __syncthreads();
    int Pvalue = 0;
    for(int j = 0;j<mask_width;j++)
    {
        Pvalue += N_ds[threadIdx.x+j]*M[j];
    }
    P[id]=Pvalue;
}

int main() {
 int width=7;
 int mask_width=5;
 int N[width]={1,2,3,4,5,6,7};
 int b[mask_width]={3,4,5,4,3};
 int P[width]={0,0,0,0,0,0,0};

 int *d_N, *d_P;

 int maskArray_size=mask_width*sizeof(int);
 int inpArray_size=width*sizeof(int);
  cudaEvent_t start1, stop1,start2,stop2;

 cudaEventCreate(&start1);
 cudaEventCreate(&stop1);
 cudaEventCreate(&start2);
 cudaEventCreate(&stop2);

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

 cudaMemcpy(d_N, &N, inpArray_size, cudaMemcpyHostToDevice);
 cudaMemcpyToSymbol(M,b,maskArray_size);
  cudaEventRecord(start1, 0);
 convolution1DConstant<<<1,width>>>(d_N, d_P, mask_width,width);
  cudaError err = cudaMemcpy(&P, d_P, inpArray_size,
cudaMemcpyDeviceToHost);
 if(err!=cudaSuccess) {
       printf("CUDA error copying to Host: %s\n",
cudaGetErrorString(err));
 }
 cudaEventRecord(stop1, 0);
 cudaEventSynchronize(stop1);
```

```
 float elapsedTime;
 cudaEventElapsedTime(&elapsedTime, start1,stop1);
 //constant memory
 printf("1a. 1D Parallel Convolution Operation\n");
 printf("Time Taken: %f\n",elapsedTime);
 printf("Resultant Array P is\n");
 for(int k=0;k<width;k++){
     printf("%d ",P[k]);
 }

 cudaEventRecord(start2, 0);
 convolution1DShared<<<1,width>>>(d_N, d_P, mask_width,width);
  err = cudaMemcpy(&P, d_P, inpArray_size, cudaMemcpyDeviceToHost);
 if(err!=cudaSuccess) {
       printf("CUDA error copying to Host: %s\n",
cudaGetErrorString(err));
 }
 cudaEventRecord(stop2, 0);
 cudaEventSynchronize(stop2);
 cudaEventElapsedTime(&elapsedTime, start2,stop2);
 //shared memory
 printf("\n\n1b.Tiled 1D Convolution with Halo Elements\n");
 printf("Time Taken: %f\n",elapsedTime);
 printf("Resultant Array P is\n");
 for(int k=0;k<width;k++){
     printf("%d ",P[k]);
 }

 // Cleanup
 cudaFree(d_N);
 cudaFree(d_P);
 return 0;
}
```

Screenshot:

```
1a. 1D Parallel Convolution Operation
Time Taken: 0.034560
Resultant Array P is
22 38 57 76 95 90 74

1b.Tiled 1D Convolution with Halo Elements
Time Taken: 0.024384
Resultant Array P is
22 38 57 76 95 90 74
```

2.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>

__global__ void SpMV_CSR(int num_rows,int *data,int *col_index,int
*row_ptr,int *x,int *y)
{
 int row= blockIdx.x * blockDim.x + 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()
{
   int n=4;
   int row_ptr[n+1];

//taking the slide sparse matrix
   int inputMatrix[n][n]={
       {3,0,1,0},
       {0,0,0,0},
       {0,2,4,1},
       {1,0,0,1}
       };
   int X[n]={1,2,3,4};
   int Y[n]={0,0,0,0};
// finding non zero elements because number of non zero elements
// are number of elements we will have in data and col_index arrays
```

```
  int nonZero=0;
printf("2. Parallel Sparse - Matrix Vector Multiplication:\n");
printf("Input Sparse Matrix:\n");
  for(int i=0;i<n;i++)
  {
    //also storing in row_ptr
    row_ptr[i]=nonZero;
    for(int j=0;j<n;j++)
    {
      if(inputMatrix[i][j]!=0)
      {
        nonZero++;
      }
      printf("%d\t",inputMatrix[i][j]);
    }
    printf("\n");
  }
  int data[nonZero],col_index[nonZero];
//storing last row_ptr value
  row_ptr[n]=nonZero;

  int k=0;
  for(int i=0;i<n;i++)
  {
    for(int j=0;j<n;j++)
    {
      if(inputMatrix[i][j]!=0)
      {
        data[k]= inputMatrix[i][j];
        col_index[k]= j;
        k++;
      }
    }
  }
  printf("\nX array:\n");
  for(int i=0;i<n;i++)
  {
    printf("%d\t",X[i]);
  }
  printf("\nY array initially:\n");
  for(int i=0;i<n;i++)
  {
    printf("%d\t",Y[i]);
```

```
    }
    printf("\n\nSparse Matrix Representation");
    printf("\ndata array\n");
    for(int i=0;i<nonZero;i++)
    {
      printf("%d\t",data[i]);
    }
    printf("\ncol_index array\n");
    for(int i=0;i<nonZero;i++)
    {
      printf("%d\t",col_index[i]);
    }
    printf("\nrow_ptr array\n");
    for(int i=0;i<=n;i++)
    {
      printf("%d\t",row_ptr[i]);
    }

    int *d_data,*d_col_index,*d_row_ptr,*d_X,*d_Y;
    cudaMalloc((void**)&d_data,nonZero * sizeof(int));
    cudaMalloc((void**)&d_col_index,nonZero * 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));

    cudaMemcpy(d_data,data,nonZero*sizeof(int),cudaMemcpyHostToDevice);

cudaMemcpy(d_col_index,col_index,nonZero*sizeof(int),cudaMemcpyHostToDe
vice);

cudaMemcpy(d_row_ptr,row_ptr,(n+1)*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(d_X,X,n*sizeof(int),cudaMemcpyHostToDevice);

    //1 block with row number of threads
    SpMV_CSR<<<1,n>>>(n,d_data,d_col_index,d_row_ptr,d_X,d_Y);

    cudaMemcpy(Y,d_Y,n * sizeof(int),cudaMemcpyDeviceToHost);

    printf("\n\nResult Y after Parallel Sparse - Matrix Vector
Multiplication:\n");
    for(int i=0;i<n;i++)
    {
        printf("%d\t",Y[i]);
```

```
    }
    cudaFree(d_data);
    cudaFree(d_col_index);
    cudaFree(d_row_ptr);
    cudaFree(d_X);
    cudaFree(d_Y);
    return 0;
}
```

Screenshot:

```
2. Parallel Sparse - Matrix Vector Multiplication:
Input Sparse Matrix:
3       0       1       0
0       0       0       0
0       2       4       1
1       0       0       1

X array:
1       2       3       4
Y array initially:
0       0       0       0

Sparse Matrix Representation
data array
3       1       2       4       1       1       1
col_index array
0       2       1       2       3       0       3
row_ptr array
0       2       2       5       7

Result Y after Parallel Sparse - Matrix Vector Multiplication:
6       0       20      5
```

3.Write a program in CUDA to perform matrix multiplication using 2D Grid and 2D Block. Taking square matrices

Code:

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

__global__ void matrixMulKernel(const int *a, const int *b, int *c, int
width) {
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 if ((Row < width) && (Col < width)){
   int pvalue=0;
   for (int k = 0; k < width; k++) {
     pvalue += a[Row * width + k] * b[k * width + Col];
   }
   c[Row * width + Col] = pvalue;
   }
}
```

```
int main()
{
 int width=4;
 int A[width][width]={{2,2,2,2},
             {2,2,2,2},
             {1,1,1,1},
             {2,2,2,2}};
  int B[width][width]={{4,1,4,1},
             {4,1,4,1},
             {4,1,4,1},
             {4,1,4,1}};
  int C[width][width]={{0,0,0,0},
            {0,0,0,0},
             {0,0,0,0},
            {0,0,0,0}};
  int *d_A, *d_B, *d_C;
  int size = sizeof(int);
 size= width*width*size;
 cudaMalloc(&d_A,size);
 cudaMalloc(&d_B,size);
 cudaMalloc(&d_C,size);

 cudaMemcpy(d_A,A, size, cudaMemcpyHostToDevice);
 cudaMemcpy(d_B,B, size, cudaMemcpyHostToDevice);
  int BLOCK_WIDTH=2;
 int NumBlocks = width/BLOCK_WIDTH;
 if (width % BLOCK_WIDTH)
 {
   NumBlocks++;
 }
 dim3 dimGrid(NumBlocks, NumBlocks);
 dim3 dimBlock(BLOCK_WIDTH,BLOCK_WIDTH);
 // Launch kernel
 matrixMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C,width);

 // Copy back to the host
 cudaMemcpy(C, d_C,size, cudaMemcpyDeviceToHost);
printf("3. Matrix Multiplication using 2D Grid and 2D Block\n");
 printf("\nMatrix A: \n");
 for(int i=0;i<width;i++)
 {
   for(int j=0;j<width;j++)
   {
```

```
      printf("%d\t",A[i][j]);
    }
    printf("\n");
  }
printf("\nMatrix B: \n");
for(int i=0;i<width;i++)
  {
    for(int j=0;j<width;j++)
    {
      printf("%d\t",B[i][j]);
    }
    printf("\n");
  }
 printf("\nResultant Matrix C:\n");
 for(int i=0;i<width;i++)
  {
    for(int j=0;j<width;j++)
    {
      printf("%d\t",C[i][j]);
    }
    printf("\n");
  }

 cudaFree(d_A);
 cudaFree(d_B);
 cudaFree(d_C);

 return 0;
}
```

Screenshot:

```
3. Matrix Multiplication using 2D Grid and 2D Block

Matrix A:
2       2       2       2
2       2       2       2
1       1       1       1
2       2       2       2

Matrix B:
4       1       4       1
4       1       4       1
4       1       4       1
4       1       4       1

Resultant Matrix C:
32      8       32      8
32      8       32      8
16      4       16      4
32      8       32      8
```

If A(ha,wa) dimensions

B(hb, wb) dimensions

Code:

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

__global__ void matrixMulKernel(const int *a, const int *b, int *c, int
ha,int wa,int wb) {
 int Row = blockIdx.y * blockDim.y + threadIdx.y;
 int Col = blockIdx.x * blockDim.x + threadIdx.x;

 int pvalue=0;
 if ((Row < ha) && (Col < wb)){
   for (int k = 0; k < wa; k++) {
     pvalue += a[Row * wa + k] * b[k * wb + Col];
   }
   c[Row * wb + Col] = pvalue;
   }
}

int main()
{
 int ha=4;
 int wa=3;
 int hb=3;
 int wb=4;
 int A[ha][wa]={{2,2,2},
             {2,2,2},
             {1,1,1},
```

```
            {2,2,2}};
  int B[hb][wb]={{4,1,4,1},
            {4,1,4,1},
            {4,1,4,1}};
  int C[ha][wb]={{0,0,0,0},
           {0,0,0,0},
            {0,0,0,0},
           {0,0,0,0}};
  int *d_A, *d_B, *d_C;
  int size = sizeof(int);
 int sizeA= ha*wa*size;
 int sizeB= hb*wb*size;
 int sizeC= ha*wb*size;
 cudaMalloc(&d_A,sizeA);
 cudaMalloc(&d_B,sizeB);
 cudaMalloc(&d_C,sizeC);

 cudaMemcpy(d_A,A, sizeA, cudaMemcpyHostToDevice);
 cudaMemcpy(d_B,B, sizeB, cudaMemcpyHostToDevice);
 dim3 dimGrid(ceil(wb/2.0), ceil(ha/2.0));
 dim3 dimBlock(2, 2);
 matrixMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C,ha,wa,wb);

 // Copy back to the host
 cudaMemcpy(C, d_C,sizeC, cudaMemcpyDeviceToHost);
printf("3. Matrix Multiplication using 2D Grid and 2D Block
matrices(Generalized case)\n");
 printf("\nMatrix A:\n");
 for(int i=0;i<ha;i++)
 {
   for(int j=0;j<wa;j++)
   {
     printf("%d\t",A[i][j]);
   }
   printf("\n");
 }
 printf("\nMatrix B:\n");
 for(int i=0;i<hb;i++)
 {
   for(int j=0;j<wb;j++)
   {
     printf("%d\t",B[i][j]);
   }
```

```
    printf("\n");
  }
  printf("\nResultant Matrix C:\n");
  for(int i=0;i<ha;i++)
  {
    for(int j=0;j<wb;j++)
    {
      printf("%d\t",C[i][j]);
    }
    printf("\n");
  }

  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);

  return 0;
}
```

```
3. Matrix Multiplication using 2D Grid and 2D Block matrices(Generalized case)

Matrix A:
2       2       2
2       2       2
1       1       1
2       2       2

Matrix B:
4       1       4       1
4       1       4       1
4       1       4       1

Resultant Matrix C:
24      6       24      6
24      6       24      6
12      3       12      3
24      6       24      6
```