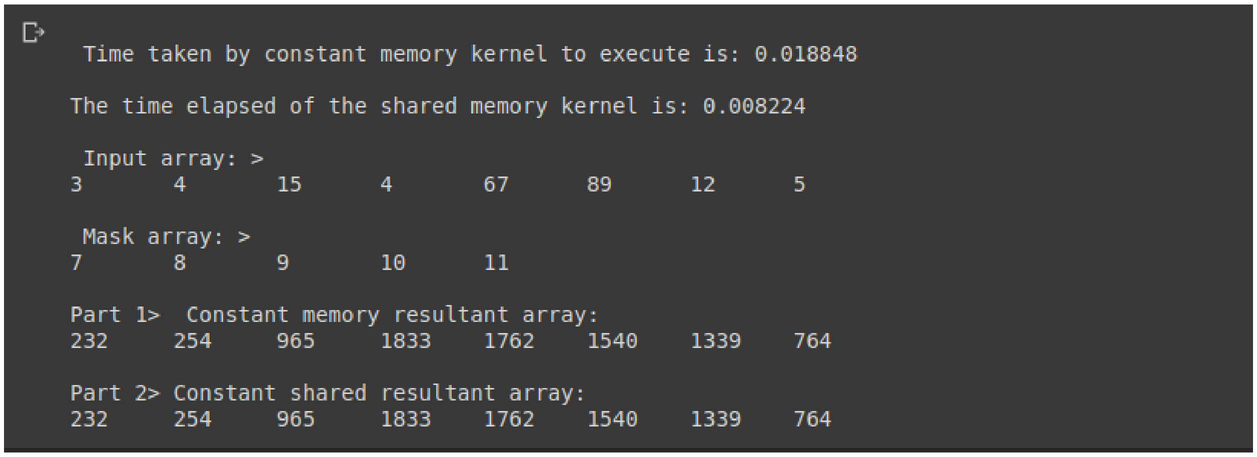
WEEK 9 PCAP_LAB Parallel Patterns In CUDA

Name: Sagnik Chatterjee
Reg: 180905478
Roll No: 61
Sem : VI B

Q1. 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.

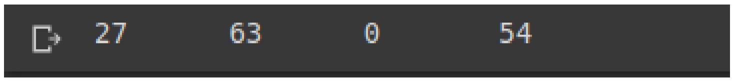
```
%%cu

#include<stdio.h>
#include<stdlib.h>
#define Mask_Width 5
#define TILE_SIZE 4
__constant__ int d_mask[Mask_Width];
__global__ void Convolution_constant(int *src, int *res, int
m_width, int src_length){
   //taking the threadid
   int id =  threadIdx.x;
   if(id < src_length){
       //declaring the start point
       int start = id - (m_width / 2);
       int pval = 0;

      for(int i = 0; i < m_width; i++){
           if((start + i) >= 0 && (start + i) < src_length){
              pval += (src[start + i] * d_mask[i]);
              // printf("PVal = %d\n", pval);
           }
       }
       //store the answer in the resultant array
res[id] = pval;
     }
}
__global__ void Convolution_shared(int *eles, int *ans, int
eles_length){
   int id = blockIdx.x * blockDim.x + threadIdx.x;

     //Initializing the shared memory for elements
   __shared__ int shared_Src[TILE_SIZE + Mask_Width - 1];
   int s = Mask_Width / 2;

   if(id < eles_length){
           //Populating the shared memory
   //calculating the left halo indices
   int halo_left = (blockIdx.x - 1) * blockDim.x + threadIdx.x;
   if(threadIdx.x >= (blockDim.x - s)){
       shared_Src[threadIdx.x - (blockDim.x - s)] = (halo_left <
0) ? 0: eles[halo_left];
   }
   shared_Src[s + threadIdx.x] = eles[blockIdx.x * blockDim.x +
threadIdx.x];
   //Calculating the right halo indices
   int halo_right = (blockIdx.x + 1) * blockDim.x + threadIdx.x;
   if(threadIdx.x < s){
       shared_Src[s + blockDim.x + threadIdx.x] = (halo_right >=
eles_length) ? 0 : eles[halo_right];
   }
   __syncthreads();
   //Calculating the resultant array
   int pval = 0;
   for(int i = 0; i < Mask_Width; i++){
       pval += (shared_Src[threadIdx.x + i] * d_mask[i]);
   }
   //Storing the result
   ans[id] = pval;

   }
}
int main(){
   //Initializing the input array and the mask array
   int n = 8;
   int input [] = {3, 4, 15, 4, 67, 89, 12, 5};
   int mask[] = {7, 8, 9, 10, 11};
   int size_input = sizeof(int) * n;
   int size_mask = sizeof(int) * Mask_Width;
   int h_output[n], h_output_s[n];

   //to mark the time taken by the kernel to execute
   float et;
   cudaEvent_t start, stop;
   cudaEventCreate(&start); cudaEventCreate(&stop);
   //Allocating space in the device
   int *d_input, *d_output, *d_output_s;
   cudaMalloc((void **)&d_input, size_input);
   cudaMalloc((void **)&d_output, size_input);
   cudaMalloc((void **)&d_output_s, size_input);
   //Copying the mask directly to constant memory
   cudaMemcpyToSymbol(d_mask, mask, size_mask);
   //Copying the input to the device memory
   cudaMemcpy(d_input, input, size_input,
cudaMemcpyHostToDevice);
   //Calling the kernel function
   int threads = 16;
   int blocks = (threads + n - 1) / threads;

   //starting the recorder to calc the execution time
   cudaEventRecord(start);
   Convolution_constant<<<blocks, threads>>>(d_input, d_output,
5, n);

   cudaEventRecord(stop);
   cudaDeviceSynchronize();

   //Calculating the elapsed time for constant memory kernel
   cudaEventElapsedTime(&et, start, stop);
   printf("\n Time taken by constant memory kernel to execute
is: %f\n", et);
   //calling the shared memory kernel
   cudaEvent_t start_s, stop_s;
   cudaEventCreate(&start_s); cudaEventCreate(&stop_s);
   threads = 4;
   blocks = (threads + n - 1) / threads;

   //similarly as before starting the timer to calcualte the
execution time
   // this is for the shared memory kernel
   cudaEventRecord(start_s);
   Convolution_shared<<<blocks, threads>>>(d_input, d_output_s,
n);
   cudaEventRecord(stop_s);
   cudaDeviceSynchronize();
   cudaEventElapsedTime(&et, start_s, stop_s);
   //printing the elapsed time of shared kernel
   printf("\nThe time elapsed of the shared memory kernel is:
%f\n", et);

   //Copying the constant memory result to host
   cudaMemcpy(h_output, d_output, size_input,
cudaMemcpyDeviceToHost);
   //Copying the shared memory result to host
   cudaMemcpy(h_output_s, d_output_s, size_input,
cudaMemcpyDeviceToHost);
   //printing the result
   printf("\n Input array: > \n");
   for(int i = 0; i < n; i++){
       printf("%d\t", input[i]);
   }
   printf("\n");
   printf("\n Mask array: > \n");
   for(int i = 0; i < Mask_Width; i++){
       printf("%d\t", mask[i]);
   }
   printf("\n");
   printf("\nPart 1>  Constant memory resultant array:\n");
   for(int i = 0; i < n; i++){
       printf("%d\t", h_output[i]);
   }
   printf("\n");
   printf("\nPart 2> Constant shared resultant array:\n");
   for(int i = 0; i < n; i++){
       printf("%d\t", h_output_s[i]);
   }
   //Freeing the allocated device memory
   cudaFree(d_output);
   cudaFree(d_output_s);
   cudaFree(d_input);
   cudaFree(d_mask);

   return 0;
}
```



```
 Time taken by constant memory kernel to execute is: 0.018848

The time elapsed of the shared memory kernel is: 0.008224

 Input array: >
3       4       15      4       67      89      12      5

 Mask array: >
7       8       9       10      11

Part 1>  Constant memory resultant array:
232     254     965     1833    1762    1540    1339    764

Part 2> Constant shared resultant array:
232     254     965     1833    1762    1540    1339    764
```

Q2 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.

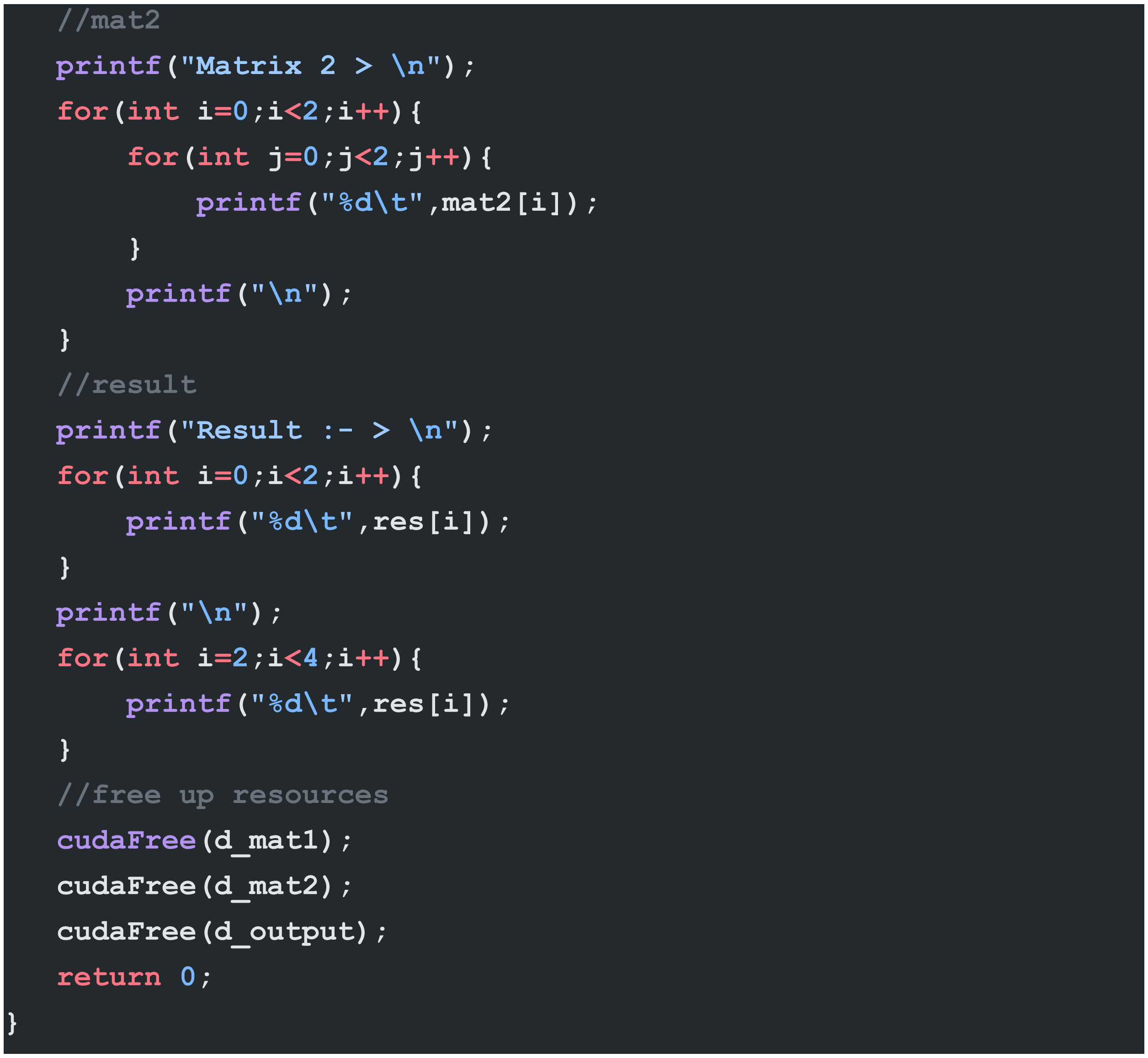
```
%%cu
#include <stdio.h>
#include <stdlib.h>

__global__ void csr_mul(int n_rows,int *mat,int *ci,int *rp,int
*mulvec,int *ans){
   int id =threadIdx.x;
   if(id < n_rows){
       int element =0;
       int start = rp[id];
       int end= rp[id+1];
       for(int i=start;i<end;i++){
          int temp = mat[i] * mulvec[ci[i]];
          element += temp;
       }
       ans[id] = element;
   }
}

int main(){
   int data[] = {4, 5, 6, 7, 8, 1, 2};
   int col_index [] = {0, 3, 1, 2, 0, 3, 3};
   int row_ptr[] = {0, 2, 5, 3, 8};
   int vector [] = {3, 3, 3, 3};
   int res[4];
   int size_data = sizeof(int) * 7;
   int size_col_id = sizeof(int) * 7;
   int size_row_ptr = sizeof(int) * 5;

   int *d_data, *d_col_ind, *d_row_ptr, *d_res, *d_vector;
   cudaMalloc((void **)&d_data, size_data);
   cudaMalloc((void **)&d_col_ind, size_col_id);
   cudaMalloc((void **)&d_row_ptr, size_row_ptr);
   cudaMalloc((void **)&d_res, sizeof(int)*4);
   cudaMalloc((void **)&d_vector, sizeof(int)*4);
   cudaMemcpy(d_data, data, size_data, cudaMemcpyHostToDevice);
   cudaMemcpy(d_col_ind, col_index, size_col_id,
cudaMemcpyHostToDevice);
   cudaMemcpy(d_row_ptr, row_ptr, size_row_ptr,
cudaMemcpyHostToDevice);
   cudaMemcpy(d_vector, vector, sizeof(int)*4,
cudaMemcpyHostToDevice);
   //kernel function call
   csr_mul <<<1, 8>>>(4, d_data, d_col_ind, d_row_ptr, d_vector,
d_res);
   cudaMemcpy(res, d_res, sizeof(int)*4,
cudaMemcpyDeviceToHost);
   for(int i = 0; i < 4; i++){
       printf("%d\t", res[i]);
   }
   cudaFree(d_data);
   cudaFree(d_col_ind);
   cudaFree(d_row_ptr);
   cudaFree(d_res);
   cudaFree(d_vector);
   return 0;
}
```



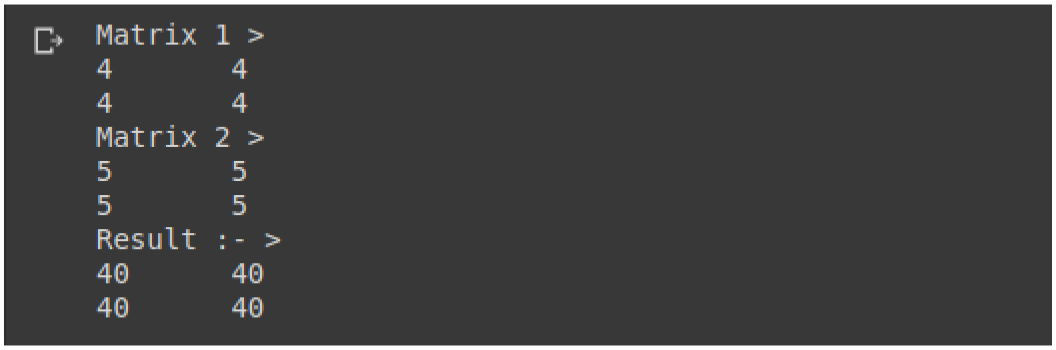
```
27      63      0      54
```

Q3 **Write a program in CUDA to perform matrix multiplication using 2D Grid and 2D Block.**

```
%%cu

#include <stdio.h>
#include <stdlib.h>
__global__ void mat_mul(int *src_mat, int *mul_mat, int *res,
int r){
   int row = blockIdx.y * blockDim.y + threadIdx.y;
   int col = blockIdx.x * blockDim.x + threadIdx.x;
   if(row < r && col < r){
       //Accumulating a partial result
       int temp = 0;
       for(int i = 0; i < r; i++){
           temp += (src_mat[row * r + i] * mul_mat[i * r +
col]);
       }
       //Writing back the result
       res[row * r + col] = temp;
   }
}




int main(){
   int mat1[] = {4,4,4,4};
   int mat2[] = {5,5,5,5};
   int res[4];
   int n =2 ;
   int size_val = sizeof(int)*4;

   int *d_mat1 , *d_mat2 ,*d_output;

   cudaMalloc((void**)&d_mat1,size_val);
   cudaMalloc((void**)&d_mat2,size_val);
   cudaMalloc((void**)&d_output,size_val);

   cudaMemcpy(d_mat1,mat1,size_val,cudaMemcpyHostToDevice);
   cudaMemcpy(d_mat2,mat2,size_val,cudaMemcpyHostToDevice);

   //threads and blocks set up
   int thread_count =4;
   int blocks = (thread_count + n-1) /thread_count;
   dim3 THREADS(thread_count,thread_count);
   dim3 BLOCKS(blocks,blocks);

   //calling kernel
   mat_mul <<<BLOCKS,THREADS>>>(d_mat1,d_mat2,d_output,2);

   //copy back to host
   cudaMemcpy(res,d_output,size_val,cudaMemcpyDeviceToHost);

   //input print
   printf("Matrix 1 > \n");
   for(int i=0;i<2;i++){
       for(int j=0;j<2;j++){
           printf("%d\t",mat1[i]);
       }
       printf("\n");
   }
```



```
    //mat2
    printf("Matrix 2 > \n");
    for(int i=0;i<2;i++){
        for(int j=0;j<2;j++){
            printf("%d\t",mat2[i]);
        }
        printf("\n");
    }
    //result
    printf("Result :- > \n");
    for(int i=0;i<2;i++){
        printf("%d\t",res[i]);
    }
    printf("\n");
    for(int i=2;i<4;i++){
        printf("%d\t",res[i]);
    }
    //free up resources
    cudaFree(d_mat1);
    cudaFree(d_mat2);
    cudaFree(d_output);
    return 0;
}
```

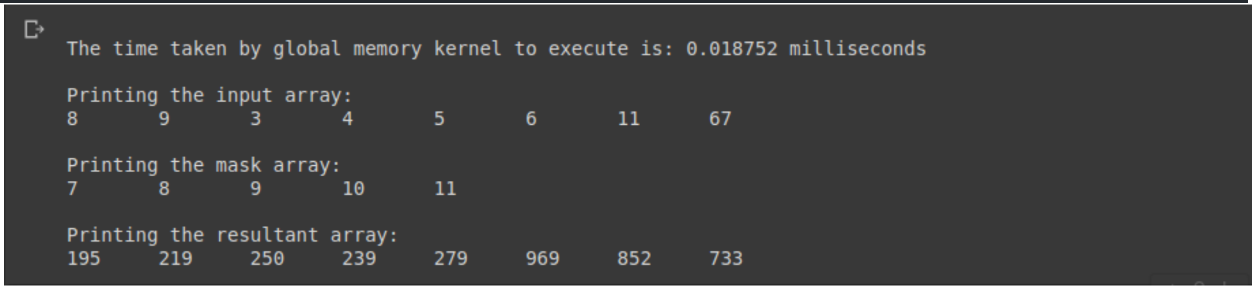


```
Matrix 1 >
4       4
4       4
Matrix 2 >
5       5
5       5
Result :- >
40      40
40      40
```

4. **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.**

```
%%cu
#include<stdio.h>
#include<stdlib.h>
#define Mask_Width 5
__global__ void Convolution_global(int *src, int *res, int
*d_mask, int src_length){
   //taking the threadid
   int id =  blockIdx.x * blockDim.x + threadIdx.x;
   if(id < src_length){
       //declaring the start point
       int start = id - (Mask_Width / 2);
             int pval = 0;

      //Looping throught the array and multiplying with the mask
array
       for(int i = 0; i < Mask_Width; i++){
           if((start + i) >= 0 && (start + i) < src_length){
              // printf("elements being multiplied are: src = %d
mask = %d\n", src[start + i], d_mask[i]);
               pval += (src[start + i] * d_mask[i]);
              // printf("pval = %d\n", pval);
           }
       }
       //storing the answer in the resultant array
       res[id] = pval;
     }
}
int main(){
   //Initializing the input array and the mask array
   int n = 8;
   int input [] = {8, 9, 3, 4, 5, 6, 11, 67};
   int mask[] = {7, 8, 9, 10, 11};
   int size_input = sizeof(int) * n;
   int size_mask = sizeof(int) * Mask_Width;
   int h_output[n];
   //Allocating space in the device
   int *d_input, *d_output, *d_mask_s;
   cudaMalloc((void **)&d_input, size_input);
   cudaMalloc((void **)&d_output, size_input);
   cudaMalloc((void **)&d_mask_s, size_mask);
   //Copying to the device memory
   cudaMemcpy(d_input, input, size_input,
cudaMemcpyHostToDevice);
   cudaMemcpy(d_mask_s, mask, size_mask,
cudaMemcpyHostToDevice);
   //Creating event to calculate the time elapsed
   float et;
   cudaEvent_t start, stop;
   cudaEventCreate(&start); cudaEventCreate(&stop);
   //Calling the kernel along with time calculation
   int threads = 4;
   int blocks = (threads + n - 1) / threads;
   cudaEventRecord(start);
   Convolution_global<<<blocks, threads>>>(d_input, d_output,
d_mask_s, n);
   cudaEventRecord(stop);
   cudaDeviceSynchronize();
   //Calculating the elapsed time of first kernel
   cudaEventElapsedTime(&et, start, stop);
   printf("\nThe time taken by global memory kernel to execute
is: %f milliseconds\n", et);
   //Copying the shared memory result to host
   cudaMemcpy(h_output, d_output, size_input,
cudaMemcpyDeviceToHost);
   //printing the result
   printf("\nPrinting the input array:\n");
   for(int i = 0; i < n; i++){
       printf("%d\t", input[i]);
   }
   printf("\n");
   printf("\nPrinting the mask array:\n");
   for(int i = 0; i < Mask_Width; i++){
       printf("%d\t", mask[i]);
   }
   printf("\n");
   printf("\nPrinting the resultant array:\n");
   for(int i = 0; i < n; i++){
       printf("%d\t", h_output[i]);
   }
   //Freeing the cuda resources
   cudaFree(d_input);
   cudaFree(d_output);
   cudaFree(d_mask_s);
}
```



```
The time taken by global memory kernel to execute is: 0.018752 milliseconds

Printing the input array:
8       9       3       4       5       6       11      67

Printing the mask array:
7       8       9       10      11

Printing the resultant array:
195     219     250     239     279     969     852     733
```