MANIPAL INSTITUTE OF TECHNOLOGY
Manipal – 576 104

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
(A constituent unit of MAHE, Manipal)

CERTIFICATE

This is to certify that Ms./Mr. ……..…… Reg. No. …..…… Section: …… Roll No: .....has satisfactorily completed the lab exercises prescribed for Parallel Programming Lab [CSE 3241] of Third Year B. Tech. Degree at MIT, Manipal, in the academic year 2024-2025.

Date: …….....

Signature
Faculty in Charge

Signature
Head of the Department

CONTENTS

Course Objectives

- Learn different APIs used in MPI for point to point, collective communications and error handling
- Learn how to write host and kernel code in CUDA for NVIDIA GPU card
- To develop the skills of design and implement parallel algorithms using different parallel programming environment

Course Outcomes

At the end of this course, students will be able to

- Write MPI programs using point-to-point and collective communication primitives.
- Design and implement CUDA programs for different parallel applications on NVIDIA GPU architecture.
- Design optimized parallel solutions using thread and memory organization in CUDA.

Evaluation plan

- Internal Assessment Marks : 60%
  - ➢ Continuous Evaluation and Mid-term examination: 60%

    Continuous evaluation component (for each evaluation):10 marks

    The assessment will depend on punctuality, regularity, program execution, maintaining the observation notebook.

- End semester assessment of 2 hours duration: 40 %

INSTRUCTIONS TO THE STUDENTS

Pre- Lab Session Instructions

1. Students should carry the Lab Manual Book and the required stationery to every lab session.
2. Be in time and follow the institution dress code.
3. Must Sign in the log register provided.
4. Make sure to occupy the allotted seat and answer the attendance
5. Adhere to the rules and maintain the decorum.
6. Students must come prepared for the lab in advance.

In- Lab Session Instructions

- Follow the instructions on the allotted exercises.
- Show the program and results to the instructors on completion of experiments.
- On receiving approval from the instructor, copy the program and results in the Lab record
- Prescribed textbooks and class notes can be kept ready for reference if required.

General Instructions for the exercises in Lab

- Implement the given exercise individually and not in a group.
- Observation book should be complete with program, proper input output clearly showing the parallel execution in each process. Plagiarism (copying from others) is strictly prohibited and would invite severe penalty in evaluation.
- The exercises for each week are divided under three sets:
  - Solved example
  - Lab exercises - to be completed during lab hours
  - Additional Exercises - to be completed outside the lab or in the lab to enhance the skill
- In case a student misses a lab class, he/ she must ensure that the experiment is completed during the repetition class with the permission of the faculty concerned but credit will be given only to one day's experiment(s).
- Questions for lab tests and examination are not necessarily limited to the questions in the manual, but may involve some variations and / or combinations of the questions.

THE STUDENTS SHOULD NOT

- Bring mobile phones or any other electronic gadgets to the lab.
- Go out of the lab without permission.

Lab No 1: **Date:**

Introduction to execution environment of MPI

Objectives:

In this lab, student will be able to

1. Understand the execution environment of MPI programs
2. Learn the various concept of parallel programming
3. Learn and use the Basics API available in MPI

I. Introduction

In order to reduce the execution time work is carried out in parallel. Two types of parallel programming are:

- Explicit parallel programming
- Implicit parallel programming

Explicit parallel programming – These are languages where the user has full control and has to explicitly provide all the details. Compiler effort is minimal.

Implicit parallel programming – These are sequential languages where the compiler has full responsibility for extracting the parallelism in the program.

Parallel Programming Models:

- Message Passing Programming
- Shared Memory Programming

Message Passing Programming:

- In message passing programming, programmers view their programs (Applications) as a collection of co-operating processes with private (local) variables.
- The only way for an application to share data among processors is for programmer to explicitly code commands to move data from one processor to another.

Message Passing Libraries: There are two message passing libraries available. They are:

- PVM – Parallel Virtual Machine
- MPI – Message Passing Interface. It is a set of parallel APIs which can be used with languages such as C and FORTRAN.

Communicators and Groups:

- MPI assumes static processes.
- All the processes are created when the program is loaded.
- No process can be created or terminated in the middle of program execution.
- There is a default process group consisting of all such processes identified by **MPI_COMM_WORLD**.

III. MPI Environment Management Routines:

MPI_Init: Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

```
MPI_Init (&argc,&argv);
```

MPI_Comm_size: Returns the total number of MPI processes to the variable size in the specified communicator, such as MPI_COMM_WORLD.

```
MPI_Comm_size(Comm,&size);
```

MPI_Comm_rank: Returns the rank of the calling MPI process within the specified communicator. Each process will be assigned a unique integer rank between 0 and size - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a process ID.

```
MPI_Comm_rank Comm,&rank);
```

MPI_Finalize: Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program. No other MPI routines may be called after it.

```
MPI_Finalize ();
```

Solved Example:

Write a program in MPI to print total number of process and rank of each process.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
        int rank,size;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        printf("My rank is %d in total %d processes", rank, size);
        MPI_Finalize();
        return 0;
}
```

Steps to execute MPI program is provided in the form of video which is available in individual systems. However, the basic installation steps are given as follows.

// Follow the following steps to Install, Compile and Run MPI programs in Ubuntu O.S

// To install MPI in Ubuntu, execute the following command in command line

$sudo apt-get update; sudo apt-get install mpich

// To edit MPI program, use any text editor such as vim or gedit and create a file with .c extention.

//To Compile MPI program, execute the following command in command line

$mpicc filename.c -o filename.out

//To Run MPI program, execute the following command in command line

$mpirun -np 4 ./filename.out

Lab Exercises:

1. Write a simple MPI program to find out pow (x, rank) for all the processes where 'x' is the integer constant and 'rank' is the rank of the process.

2. Write a program in MPI where even ranked process prints "Hello" and odd ranked process prints "World".
3. Write a program in MPI to simulate simple calculator. Perform each operation using different process in parallel.
4. Write a program in MPI to toggle the character of a given string indexed by the rank of the process. Hint: Suppose the string is HELLO and there are 5 processes, then process 0 toggle 'H' to 'h', process 1 toggle 'E' to 'e' and so on.
5. Write a program in MPI where even ranked process prints factorial of the rank and odd ranked process prints ranks Fibonacci number.

Additional Exercises:

1. Write a program in MPI to reverse the digits of the following integer array of size 9 with 9 processes. Initialize the array to the following values.

   Input array: 18, 523, 301, 1234, 2, 14, 108, 150, 1928

   Output array: 81, 325, 103, 4321, 2, 41, 801, 51, 8291

2. Write a MPI program to find the prime numbers between 1 and 100 using two processes.

Lab No 2: **Date:**

Point to Point Communications in MPI

Objectives:

In this lab, student will be able to

1. Understand the different APIs used for point to point communication in MPI
2. Learn the different modes available in case of blocking send operation

Point to Point communication in MPI

- MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.
- MPI provides both blocking and non-blocking send and receive operations.

Sending message in MPI

- **Blocked Send** sends a message to another processor and waits until the receiver has received it before continuing the process. Also called as **Synchronous send**.
- **Send** sends a message and continues without waiting. Also called as **Asynchronous send.**

There are multiple communication modes used in blocking send operation:

- **Standard mode**
- **Synchronous mode**
- **Buffered mode**

Standard mode

This mode blocks until the message is buffered.

```
MPI_Send(&Msg, Count, Datatype, Destination, Tag, Comm);
```

- First 3 parameters together constitute message buffer. The **Msg** could be any address in sender's address space. The **Count** indicates the number of data elements of a particular type to be sent. The **Datatype** specifies the message type. Some Data types available in MPI are: MPI_INT, MPI_FLOAT, MPI_CHAR, MPI_DOUBLE, MPI_LONG
- Next 3 parameters specify message envelope. The **Destination** specifies the rank of the process to which the message is to be sent.
- **Tag:** The **tag** is an integer used by the programmer to label different types of messages and to restrict message reception.

- **Communicator:** Major problem with tags is that they are specified by users who can make mistakes. **Context** are allocated at run time by the system in response to user request and are used for matching messages. The notions of **context and group** are combined in a single object called a communicator (**Comm**).
- The default process group is **MPI_COMM_WORLD**.

Synchronous mode

This mode requires a send to block until the corresponding receive has occurred.

```
MPI_Ssend(&Msg, Count, Datatype, Destination, Tag,  Comm);
```

Buffered mode

```
MPI_Bsend(&Msg, Count, Datatype, Destination, Tag,  Comm);
```

In this mode a send assumes availability of a certain amount of buffer space, which must be previously specified by the user program through a routine call that allocates a user buffer.

```
MPI-Buffer_attach(buffer, size);
```

This buffer can be released by

```
MPI_Buffer_detach(*buffer, *size);
```

Receiving message in MPI

```
MPI_Recv(&Msg, Count, Datatype, Source, Tag, Comm, &status);
```

- Receive a message and block until the requested data is available in the application buffer in the receiving task.
- The **Msg** could be any address in receiver's address space. The **Count** specifies number of data items. The **Datatype** specifies the message type. The **Source** specifies the rank of the process which has sent the message. The **Tag** and **Comm** should be same as that is used in corresponding send operation. The status is a structure of type status which contains following information: Sender's rank, Sender's tag and number of items received

Finding execution time in MPI

MPI_Wtime: Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

- **MPI_Wtime ()**

Solved Example:

Write a MPI program using standard send. The sender process sends a number to the receiver. The second process receives the number and prints it.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
        int rank,size,x;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD,  &size);
        MPI_Status status;
        if(rank==0)
         {
          Printf("Enter a value in master process:");
          scanf("%d",&x);
          MPI_Send(&x,1,MPI_INT,1,1,MPI_COMM_WORLD);
          fprintf(stdout,"I have sent %d from process 0\n",x);
          fflush(stdout);
         }
        else
        {
         MPI_Recv(&x,1,MPI_INT,0,1,MPI_COMM_WORLD,&status);
         fprintf(stdout,"I have received %d in process 1\n",x);
         fflush(stdout);
         }
       MPI_Finalize();
       return 0;
}
```

Lab Exercises:

1) Write a MPI program using synchronous send. The sender process sends a word to the receiver. The second process receives the word, toggles each letter of the word and sends it back to the first process. Both processes use synchronous send operations.

2) Write a MPI program where the master process (process 0) sends a number to each of the slaves and the slave processes receive the number and prints it. Use standard send.

3) Write a MPI program to read N elements of the array in the root process (process 0) where N is equal to the total number of processes. The root process sends one value to each of the slaves. Let even ranked process finds square of the received element and odd ranked process finds cube of received element. Use Buffered send.
4) Write a MPI program to read an integer value in the root process. Root process sends this value to Process1, Process1 sends this value to Process2 and so on. Last process sends the value back to root process. When sending the value each process will first increment the received value by one. Write the program using point to point communication routines.

Additional Exercises:

1) Write a MPI program to read N elements of an array in the master process. Let N processes including master process check the array values are prime or not.
2) Write a MPI program to read value of N in the root process. Using N processes, including root, find out 1! + (1+2) +3! +(1+2+3+4) +5! +(1+2+3+4+5+6) and print the result in the root process.

3. Implement at least 2 programs to identify deadlock conditions in synchronous send and standard send with multiple point to point communications between two processes

Lab No 3: **Date:**

Collective Communications in MPI

Objectives:

In this lab, student will be able to

1. Understand the usage of collective communication in MPI
2. Learn how to broadcast messages from root
3. Learn and use the APIs for distributing values from root and gathering the values in the root

Collective Communication routines

When **all processes** in a group participate in a global communication operation, the resulting communication is called a **collective communication**.

MPI_Bcast:

```
MPI_Bcast (Address, Count, Datatype, Root, Comm);
```

The process ranked **Root** sends the same message whose content is identified by the triple (Address,Count,Datatype) to all processes(including itself) in the communicator **Comm**.

MPI_Scatter:

```
MPI_Scatter( SendBuff, Sendcount, SendDatatype, RecvBuff, Recvcount,
             RecvDatatype, Root, Comm);
```

Ensures that the **Root** process sends out personalized messages, which are in rank order in its send buffer, to all the N processes (including itself).

MPI_Gather:

```
MPI_Gather( SendAddress, Sendcount, SendDatatype, RecvAddress, RecvCount,
            RecvDatatype, Root, Comm);
```

The **root** process receives a personalized message from all N processes. These N received messages are concatenated in rank order and stored in the receive buffer of the root process.

Total Exchange:

In routine **MPI_Alltoall()** each process sends a personalized message to every other process including itself. This operation is equivalent to N gathers, each by a different process and in all N^2 messages are exchanged.

Solved Example:

Write a MPI program to read N values of the array in the root process. Distribute these N values among N processes. Every process finds the square of the value it received. Let every process return these values to the root and root process gathers and prints the result. Use collective communication routines.

```
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[])
{
        int rank,size,N,A[10],B[10], c, i;

        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);

        if(rank==0)
        {
            N=size;
            fprintf(stdout,"Enter  %d values:\n",N);
            fflush(stdout);
            for(i=0; i<N; i++)
               scanf("%d",&A[i]);
         }
       MPI_Scatter(A,1,MPI_INT,&c,1,MPI_INT,0,MPI_COMM_WORLD);
       fprintf(stdout,"I have received %d in process %d\n",c,rank);
       fflush(stdout);

       c=c*c;
       MPI_Gather(&c,1,MPI_INT,B,1,MPI_INT,0,MPI_COMM_WORLD);

       if(rank==0)
        {
           fprintf(stdout,"The Result gathered in the root \n");
           fflush(stdout);
           for(i=0; i<N; i++)
           fprintf(stdout,"%d \t",B[i]);
           fflush(stdout);
         }
```

```
        MPI_Finalize();
        return 0;
}
```

Lab Exercises:

1) Write a MPI program to read N values in the root process. Root process sends one value to each process. Every process receives it and finds the factorial of that number and returns it to the root process. Root process gathers the factorial and finds sum of it. Use N number of processes.
2) Write a MPI program to read an integer value M and NXM elements into an 1D array in the root process, where N is the number of processes. Root process sends M elements to each process. Each process finds average of M elements it received and sends these average values to root. Root collects all the values and finds the total average. Use collective communication routines.
3) Write a MPI program to read a string. Using N processes (string length is evenly divisible by N), find the number of non-vowels in the string. In the root process print number of non-vowels found by each process and print the total number of non-vowels.
4) Write a MPI Program to read two strings S1 and S2 of same length in the root process. Using *N* processes including the root (string length is evenly divisible by N), produce the resultant string as shown below. Display the resultant string in the root process. Use Collective communication routines.

   Example:

   String S1: string String S2: length Resultant String : slternigntgh

Additional Exercises:

1) Write a MPI program to read a value M and NXM number of elements into 1D array in the root, where N is the total number of processes. Find the square of first M numbers, the cube of next M numbers and so on. Print the results in the root.
2) Write a MPI program using collective communication functions, to replace all even elements of array A to 1 and replace all odd elements to 0 of size N. Display the resultant array A, count of all even and odd numbers in root process. Assume N is evenly divisible by number of processes.

   Example :
   Input Array (A): 1 2 3 4 5 6 7 8 9
   Resultant Array (A): 0 1 0 1 0 1 0 1 0
   Even (Count) = 4
   Odd (Count) = 5

Lab No 4: **Date:**

Collective Communications and Error Handling in MPI

Objectives:

In this lab, student will be able to

1. Understand the different aggregate functions used in MPI
2. Learn how to write MPI programs using both point to point and collective communication routines
3. Learn and use the APIs for handling errors in MPI

I. Aggregation Functions

MPI provides two forms of aggregation

- **Reduction**
- **Scan**

Reduction:

```
MPI_Reduce (SendAddress, RecvAddress, Count, Datatype, Op, Root, Comm);
```

This routine reduces the partial values stored in **SendAddress** of each process into a final result and stores it in **RecvAddress** of the **Root** process. The reduction operator is specified by the **Op** field. Some of the reduction operator available in MPI are: MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD

Scan:

```
MPI_Scan (SendAddress, RecvAddress, Count, Datatype, Op, Comm);
```

This routine combines the partial values into N final results which it stores in the **RecvAddress** of the N processes. Note that root field is absent here. The scan operator is specified by the **Op** field. Some of the scan operator available in MPI are: MPI_SUM, MPI_MAX, MPI_MIN, MPI_PROD

MPI_Barrier(Comm) :This routine synchronizes all processes in the communicator **Comm**. They wait until all N processes execute their respective MPI_Barrier.

Note: All collective communication routines except MPI_Barrier, employ a standard blocking mode of point-to-point communication.

Error Handling in MPI:

- An MPI *communicator* is more than just a group of process that belong to it. Amongst the items that the communicator hides inside is an *error handler*. The error handler is called every time an MPI error is detected within the communicator.
- The predefined default error handler, which is called **MPI_ERRORS_ARE_FATAL**, for a newly created communicator or for MPI_COMM_WORLD is to *abort the whole parallel program* as soon as any MPI error is detected. There is another predefined error handler, which is called **MPI_ERRORS_RETURN**.
- The default error handler can be replaced with this one by calling function **MPI_Errhandler_set**, for example:

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

- The only **error code** that MPI standard itself defines is **MPI_SUCCESS**, i.e., no error. But the meaning of an error code can be extracted by calling function **MPI_Error_string**. On top of the above MPI standard defines the so called ***error classes***. The **error class** for a given error code can be obtained by calling function **MPI_Error_class**.
- Error classes can be converted to comprehensible error messages by calling the same function that does it for error codes, i.e., **MPI_Error_string**. The reason for this is that error classes are implemented as a subset of error codes.

Solved Example:

Write a MPI program using N processes to find 1! + 2! +…..+N!. Use collective communication routines.

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char* argv[])
{
     int rank,size,fact=1, factsum, i;
```

```
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    for(i=1; i<=rank+1; i++)
        fact = fact * i;

   MPI_Reduce (&fact,&factsum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if(rank==0)
        printf("Sum of all the factorial=%d",factsum);

    MPI_Finalize();
    exit(0);
}
```

Lab Exercises:

1) Write a MPI program using N processes to find 1! + 2! +…..+N!. Use scan. Also, handle different errors using error handling routines.
2) Write a MPI program to read a 3 X 3 matrix. Enter an element to be searched in the root process. Find the number of occurrences of this element in the matrix using three processes.
3) Write a MPI program to read 4 X 4 matrix and display the following output using four processes.

I/p matrix:				O/p matrix:			
1	2	3	4	1	2	3	4
1	2	3	1	2	4	6	5
1	1	1	1	3	5	7	6
2	1	2	1	5	6	9	7

4) Write a MPI program to read a word of length N. Using N processes including the root get output word with the pattern as shown in example. Display the resultant output word in the root.
   Example: Input : PCAP Output : PCCAAAPPPP

Additional Exercises:

Lab No 5: **Date:**

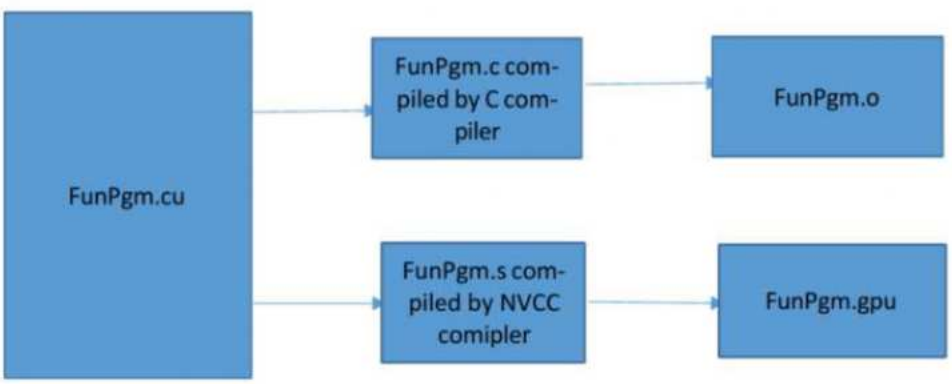
Programs on arrays in CUDA

Objectives:

In this lab, student will be able to

1. Know the basics of Computing Unified Device Architecture (CUDA).
2. Learn program structure of CUDA.
3. Learn about CUDA 1D blocks and threads
4. Write simple programs on one dimensional array
5. Learn mathematical functions in CUDA

About CUDA: CUDA is a platform for performing massively parallel computations on graphics accelerators. CUDA was developed by NVIDIA. It was first available with their G8X line of graphics cards. CUDA presents a unique opportunity to develop widely-deployed parallel applications. The CUDA programs are compiled as follows.



FunPgm.cu is compiled by both C compiler and Nvidia CUDA C compiler (NVCC compiler). If you have both main.c and Funpgm.cu then you can call cuda API's in main.c but keep in mind that you cannot call kernel from main.c. To call the kernel file extension must be .cu.

As in OpenCL CPU is the host and its memory the host memory and GPU is the device and its memory device memory. Serial code will be run on host and parallel code will be run on device.

1. Copy data from host memory to device memory.
2. Load device program and execute, caching data on chip for performance.
3. Copy result from device memory to host memory.

CUDA threads, blocks and grid

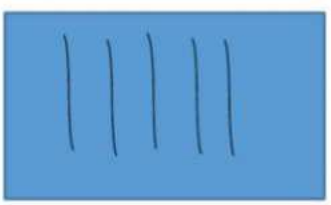
Thread – Distributed by the CUDA runtime. A single path of execution there can be multiple threads in a program.

(identified by threadIdx)

CUDA Thread

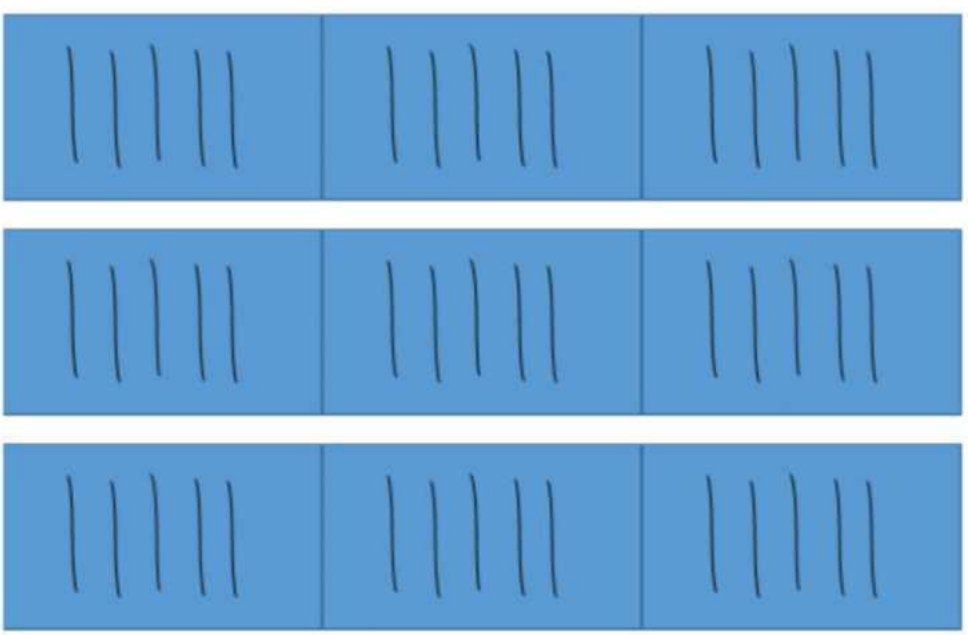
Block – A user defined group of 1 to 512 threads.

(identified by blockIdx)



CUDA Block

Grid – A group of one or more blocks. A grid is created for each CUDA kernel function



CUDA
GRID

Some of the calculations for indexing the thread is given below.

```
1D grid of 1D blocks
__device_
int getGlobalIdx_1D_1D(){
return blockIdx.x *blockDim.x + threadIdx.x;
}
1D grid of 2D blocks
__device_
int getGlobalIdx_1D_2D(){
return blockIdx.x * blockDim.x * blockDim.y
 + threadIdx.y * blockDim.x + threadIdx.x;
}
1D grid of 3D blocks
__device_
int getGlobalIdx_1D_3D(){
return blockIdx.x * blockDim.x * blockDim.y * blockDim.z
 + threadIdx.z * blockDim.y * blockDim.x
 + threadIdx.y * blockDim.x + threadIdx.x;
}
2D grid of 1D blocks
__device__ int getGlobalIdx_2D_1D(){
int blockId = blockIdx.y * gridDim.x + blockIdx.x;
int threadId = blockId * blockDim.x + threadIdx.x;
```

```
return threadId;
}
2D grid of 2D blocks
__device_
int getGlobalIdx_2D_2D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y)
 + (threadIdx.y * blockDim.x) + threadIdx.x;
return threadId;
}
2D grid of 3D blocks
__device_
int getGlobalIdx_2D_3D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x;
int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
 + (threadIdx.z * (blockDim.x * blockDim.y))
 + (threadIdx.y * blockDim.x) + threadIdx.x;
return threadId;
}
3D grid of 1D blocks
__device_
int getGlobalIdx_3D_1D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x
 + gridDim.x * gridDim.y * blockIdx.z;
int threadId = blockId * blockDim.x + threadIdx.x;
```

```
return threadId;
}
3D grid of 2D blocks
__device_
int getGlobalIdx_3D_2D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x
 + gridDim.x * gridDim.y * blockIdx.z;
int threadId = blockId * (blockDim.x * blockDim.y)
 + (threadIdx.y * blockDim.x) + threadIdx.x;
return threadId;
}
3D grid of 3D blocks
__device_
int getGlobalIdx_3D_3D(){
int blockId = blockIdx.x + blockIdx.y * gridDim.x
 + gridDim.x * gridDim.y * blockIdx.z;
int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
 + (threadIdx.z * (blockDim.x * blockDim.y))
 + (threadIdx.y * blockDim.x) + threadIdx.x;
return threadId;
}
```

Solved Exercise: Program to add two numbers.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

__global__ void add(int *a, int *b, int *c)
        {
                *c = *a + *b;
         }

int main(void) {
                int a, b, c;                    // host copies of variables a, b & c
                int *d_a, *d_b, *d_c;       // device copies of variables a, b & c
                int size = sizeof(int);

                // Allocate space for device copies of a, b, c
                cudaMalloc((void **)&d_a, size);
                cudaMalloc((void **)&d_b, size);
                cudaMalloc((void **)&d_c, size);
                // Setup input values
                a = 3;
                b = 5;
// Copy inputs to device
                cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
                cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
                // Launch add() kernel on GPU
                add<<<1,1>>>(d_a, d_b, d_c);
                // Copy result back to host
                cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
                printf("Result : %d",c);
                // Cleanup
                cudaFree(d_a);
                cudaFree(d_b);
                cudaFree(d_c);
                return 0;
        }
```

Explanation:

add is the function which runs on device.

```
cudaMalloc ((void **)&d_a, size);
```

cudaMalloc will allocate memory of size bytes given as second argument to variable passed as first argument.

```
cudaMemcpy (Destination,Source,Size,Direction);

cudaMemcpy (d_a, &a, size, cudaMemcpyHostToDevice);

cudaMemcpy (&c, d_c, size, cudaMemcpyDeviceToHost);
```

cudaMemcpy copies the variables from host to device or device to host based on the direction which is either cudaMemcpyHostToDevice or cudaMemcpyDeviceToHost. Value of size bytes long is copied from source to destination.

cudaFree frees the memory allocated by cudaMalloc.

The add function is called like this add<<<1,1>>>(d_a,d_b,d_c). The add is followed by three angular brackets then the number of blocks, threads per block then corresponding closing angular brackets then how many arguments the function add takes is enclosed within parenthesis. If you want to add N elements you can achieve it in two ways either having N blocks or having N threads.

That is pass an array with following function calls

add<<< N,1>>> (d_a,d_b,d_c) or add<<< 1, N>>> (d_a,d_b,d_c)

Few Mathematical functions in CUDA:

Major Single-Precision floating point functions: Single precision functions work on float value (32 bit). A float value is stored in IEEE 754 format.

Function	Description
sqrtf(x)	Square root function
expf(x)	Exponentiation function. Base = e
exp2f(x)	Exponentiation function. Base = 2
exp10f(x)	Exponentiation function. Base = 10
logf(x)	Logarithmic function. Base=e
log2f(x)	Logarithmic function. Base=2
log10f(x)	Logarithmic function. Base=10
sinf(x)	sine function

cosf(x)	cos function
tanf(x)	tan function
powf(x,y)	power function
truncf(x)	truncation function
roundf(x)	round function
ceilf(x)	ceil function
floorf(x)	floor function

Major Double-Precision floating point functions: Double precision functions work on double value (64 bit). A double value is stored in IEEE 754 format.

Function	**Description**
sqrt(x)	Square root function
exp(x)	Exponentiation function. Base = e
exp2(x)	Exponentiation function. Base = 2
exp10(x)	Exponentiation function. Base = 10
log(x)	Logarithmic function. Base=e
log2(x)	Logarithmic function. Base=2
log10(x)	Logarithmic function. Base=10
sin(x)	sine function
cos(x)	cos function
tan(x)	tan function
pow(x,y)	power function
trunc(x)	truncation function
round(x)	round function

ceil(x)	ceil function
floor(x)	floor function

Steps to execute a CUDA program is provided in the form of video which is made available in individual systems.

Lab Exercises:

1. Write a program in CUDA to add two vectors of length N using
   a) block size as N b) N threads
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.
3. Write a program in CUDA to process a 1D array containing angles in radians to generate sine of the angles in the output array. Use appropriate function.

Additional Exercises:

1. Write a program in CUDA to perform linear algebra function of the form $y = \alpha x + y$, where x and y are vectors and α is a scalar value.
2. Write a program in CUDA to sort every row of a matrix using selection sort.
3. Write a program in CUDA to perform odd even transposition sort in parallel.

Lab No 6: **Date:**

Programs on Arrays in CUDA

Objectives:

In this lab, student will be able to

1. Learn more about CUDA 1D blocks and threads
2. Write parallel program applications on one dimensional array
3. Learn parallel sorting applications in CUDA

1D Sequential Convolution

Convolution is a popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision.

Convolution is often performed as a filter that transforms signals and pixels into more desirable values. For example, Gaussian filters are convolution filters that can be used to sharpen boundaries and edges of objects in images.

Mathematically, convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements.

The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the convolution mask OR convolution kernel.

The same convolution mask is typically used for all elements of the array.

The following example shows a convolution example for 1D data where a five-element convolution mask array M is applied to a seven-element input array N.

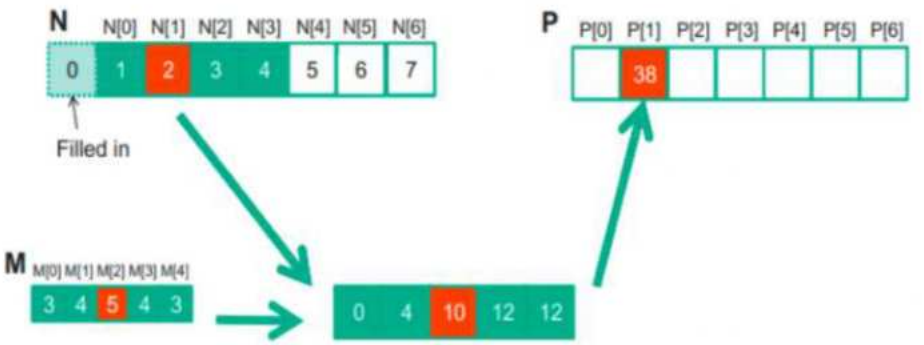
The fact that we use a five-element mask M means that each P element is generated by a weighted sum of the corresponding N element, up to two elements to the left and up to two elements to the right.

Each weight value is multiplied to the corresponding N element values before the products are summed together.

In general, the size of the mask tends to be an odd number, which makes the weighted sum calculation symmetric around the element being calculated.

Because convolution is defined in terms of neighbouring elements, *boundary conditions* naturally exist for output elements that are close to the ends of an array.

For example, when we calculate P[1], there is only one N element to the left of N[1]. That is, there are not enough N elements to calculate P[1] according to our definition of convolution. A typical approach to handling such a boundary condition is to define a default value to these missing N elements. For most applications, the default value is 0.



- These missing elements are typically referred to as *ghost elements* in literature.
- The calculation of all output *(P)* elements can be done in parallel in a 1D convolution.
- The first step is to define the major input parameters for the kernel. We assume that the 1D convolution kernel receives five arguments: *pointer to input array N*, *pointer to input mask M*, *pointer to output array P*, *size of the mask Mask_Width*, and *size of the input and output arrays Width*. Thus, we have the following set up:
- The second step is to determine and implement the mapping of threads to output elements. Since the output array is one dimensional, a simple and good approach is to organize the threads into a 1D grid and have each thread in the grid calculate one output element.

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {

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

  float 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 convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {

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

  float 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;

}
```


1. 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*.
2. Write a program in CUDA to perform selection sort in parallel.
3. Write a program in CUDA to perform odd even transposition sort in parallel.

Additional Exercises:

1. Write a program in CUDA which takes N integers as input. It converts these integers into their corresponding octal values and stores the result in another array in parallel.
2. Write a CUDA program which takes N binary numbers as input and stores the one's complement of each element in another array in parallel.

Lab No 7: **Date:**

Programs on strings in CUDA

Objectives:

In this lab, student will be able to

1. Write simple programs on Strings
2. Learn to compute time of kernel execution
3. Learn about atomic functions
4. Learn to handle errors in the kernel

Arithmetic functions:

In a multithreaded scenario, the issue of data inconsistency will arise, if multiple threads modify a single shared memory variable. To overcome this, atomic functions need to be used. List of atomic functions, their syntax and explanation are provided below.

atomicAdd():

```
int atomicAdd (int* address, int val);

unsigned int atomicAdd(unsigned int* address, unsigned int val);

float atomicAdd(float* address, float val);

double atomicAdd(double* address, double val);
```

Reads the 16-bit, 32-bit or 64-bit word old located at the address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

atomicSub():

```
int atomicSub(int* address, int val);

unsigned int atomicSub(unsigned int* address, unsigned int val);
```

Reads the 32-bit word old located at the address in global or shared memory, computes, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

atomicExch():

```
int atomicExch(int* address, int val);

unsigned int atomicExch(unsigned int* address,  unsigned int val);

float atomicExch(float* address, float val);
```

Reads the 32-bit word old located at the address address in global or shared memory and stores val back to memory at the same address. These two operations are performed in one atomic transaction. The function returns old.

atomicMin():

```
int atomicMin(int* address, int val);

unsigned int atomicMin(unsigned int* address,  unsigned int val);
```

Reads the 32-bitword old located at the address address in global or shared memory, computes the minimum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

atomicMax():

```
int atomicMax(int* address, int val);

unsigned int atomicMax(unsigned int* address,  unsigned int val);
```

Reads the 32-bit word old located at the address address in global or shared memory, computes the maximum of old and val, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

atomicInc():

```
unsigned int atomicInc(unsigned int* address,  unsigned int val);
```

Reads the 32-bit word old located at the address address in global or shared memory, computes ((old >= val) ? 0 : (old+1)), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

atomicDec():

unsigned int atomicDec(unsigned int* address, unsigned int val);

Reads the 32-bit word old located at the address address in global or shared memory, computes (((old == 0) | (old > val)) ? val : (old-1)), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.

Solved Example:

A CUDA program which takes a string as input and determines the number of occurences of a character 'a' in the string. This program uses atomicAdd() function.

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include  <stdio.h>
#include <stdlib.h>
#include<string.h>
#include <conio.h>
#define N 1024

__global__ void CUDACount(char* A, unsigned int *d_count){
        int i = threadIdx.x;
                if(A[i]=='a')
                        atomicAdd(d_count,1);
    }
int main() {
char A[N];
char *d_A;
unsigned int *count=0,*d_count,*result;
printf("Enter a string");
gets(A);
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
cudaMalloc((void**)&d_A, strlen(A)*sizeof(char));
cudaMalloc((void **)&d_count,sizeof(unsigned int));
cudaMemcpy(d_A, A, strlen(A)*sizeof(char), cudaMemcpyHostToDevice);
cudaMemcpy(d_count,count,sizeof(unsigned int),cudaMemcpyHostToDevice);
```

```
cudaError_t error =cudaGetLastError();
if (error != cudaSuccess) {
                    printf("CUDA Error1: %s\n", cudaGetErrorString(error));
        }
CUDACount<<<1,strlen(A)>>>(d_A,d_count);
error =cudaGetLastError();
if (error != cudaSuccess) {
                    printf("CUDA Error2: %s\n", cudaGetErrorString(error));

        }
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
cudaMemcpy(result, d_count, sizeof(unsigned int), cudaMemcpyDeviceToHost);
printf("Total occurences of a=%u",result);
printf("Time Taken=%f",elapsedTime);
cudaFree(d_A);
cudaFree(d_count);
printf("\n");
getch();
return 0;
}
```

Explanation:

The kernel uses *atomicAdd* function with 1 as the value each time a character 'a' occurs in the string. The instructions given in bold are present to find the time. As in OpenCL you need to declare an event, register the event and record the time before kernel execution and after kernel execution. You have to synchronize the event so that main thread can capture the time of execution of kernel. After that **cudaEventElapsedTime(&elapsedTime, start, stop)** will give the difference between the recorded stop and start time and store the value in the variable *elapsedTime* which is of type float. A negative time value means there is something wrong in the CUDA code. To find it out you use the code given in bold and italics. It will display the error message present in CUDA code. Call it once before calling the kernel and once after calling kernel. If first call throws an error message then error is present in CUDA API which precedes the kernel. If second call throws the error message then error is present in the kernel code.

Lab Exercises:

1. Write a program in CUDA to count the number of times a given word is repeated in a sentence. (Use Atomic function)

2. Write a CUDA program that reads a string ***S*** and produces the string ***RS*** as follows:
   Input string ***S***: PCAP Output string ***RS***: PCAP**PCA**PC**P**
   Note: Each work item copies required number of characters from *S* in *RS*.

Additional Exercises:

1) Write a CUDA program which reads a string consisting of N words and reverse each word of it in parallel.

2) Write a CUDA program that takes a string *Sin* as input and one integer value N and produces an output string , *Sout,* in parallel by concatenating input string *Sin*, N times as shown below.

   Input: Sin = "Hello" N = 3
   Ouput: Sout = "HelloHelloHello"

Note: Every thread copies the same character from the Input string S, N times to the required position.

3) Write a CUDA program which reads a string *Sin* and produces an output string T as shown below.

   Input: Sin: "Hai"
   Ouput: *T*: "Haaiii "

Note:Every thread stores a character from input string *Sin*, required number of times into output string *T*.

Lab No 8: **Date:**

Programs on Matrix using CUDA

Objectives:

In this lab, student will be able to

1. Understand how to write kernel code in CUDA to perform operations on matrix
2. Learn about CUDA 2D blocks/threads
3. Write simple programs on two dimensional arrays

Solved Exercise:

Write a program in CUDA to find transpose of a matrix in parallel.

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

__global__ void transpose(int *a, int *t)
        {
        int n=threadIdx.x, m=blockIdx.x, size=blockDim.x, size1=gridDim.x;
        t[n*size1+m]=a[m*size+n];
        }
int main(void)
        {
        int *a,*t, m,n,i,j;
        int *d_a,*d_t;
        printf("Enter the value of m: ");scanf("%d",&m);
        printf("Enter the value of n: ");scanf("%d",&n);
        int size=sizeof(int)*m*n;
        a=(int*)malloc(m*n*sizeof(int));
        c=(int*)malloc(m*n*sizeof(int));
        printf("Enter input matrix:\n");
        for(i=0;i<m*n;i++)
                scanf("%d",&a[i]);

        cudaMalloc((void**)&d_a,size);
```

```
cudaMalloc((void**)&d_t,size);

cudaMemcpy(d_a,a,size,cudaMemcpyHostToDevice);
transpose<<<m,n>>>(d_a,d_t);
cudaMemcpy(t,d_t,size,cudaMemcpyDeviceToHost);
printf("Result vector is:\n");
for(i=0;i<n;i++)
        {
        for(j=0;j<m;j++)
                printf("%d\t",t[i*m+j]);
        printf("\n");
        }

getchar();
cudaFree(d_a);
cudaFree(d_t);
return 0;
}
```

Lab Exercises:

1. Write a program in CUDA to add two Matrices for the following specifications:
   a. Each row of resultant matrix to be computed by one thread.
   b. Each column of resultant matrix to be computed by one thread.
   c. Each element of resultant matrix to be computed by one thread.

2. Write a program in CUDA to multiply two Matrices for the following specifications:
   a. Each row of resultant matrix to be computed by one thread.
   b. Each column of resultant matrix to be computed by one thread.
   c. Each element of resultant matrix to be computed by one thread.

Additional Exercises:

1 Write a CUDA program that reads a MXN matrix A and produces a resultant matrix B of same size as follows: Replace all the even numbered matrix elements with their row sum and odd numbered matrix elements with their column sum.

Example:

	A				B		
I/p:	**1**	**2**	**3**	O/p:	5	**6**	9
	4	5	6		15	7	15

2 Write a CUDA program to read a matrix A of size NXN. It replaces the principal diagonal elements with zero. Elements above the principal diagonal by their factorial and elements below the principal diagonal by their sum of digits.

Lab No 9: **Date:**

Programs on Matrix using CUDA(continued…)

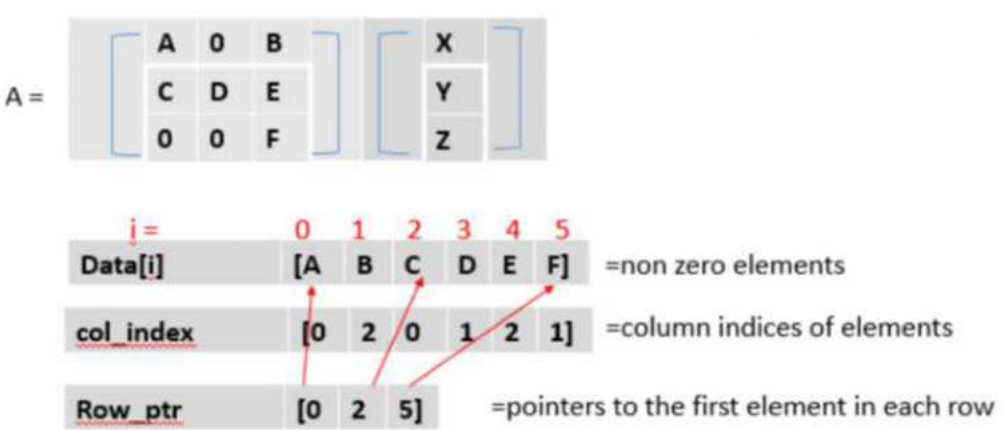
Objectives:

In this lab, student will be able to

1. Understand how to write kernel code in CUDA to perform operations on matrix.
2. Learn how to represent sparse matrix using CSR format and perform parallel operations using it.

Compressed Sparse Row (CSR) format:

- A sparse matrix is a matrix where the majority of the elements are zero. Sparse matrices are stored in a format that avoids storing zero elements.
- The compressed sparse row (CSR) format is a popular, general-purpose sparse matrix representation. CSR explicitly stores column indices and nonzero values in arrays col_index and data as shown below.
- A third array of row pointers, row_ptr, give the starting location of every row in the compressed storage.



Lab Exercises:

1. 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.

2. Write a program in CUDA to read MXN matrix A and replace 1st row of this matrix by same elements, 2nd row elements by square of each element and 3rd row elements by cube of each element and so on.

3. Write a CUDA program that reads a matrix A of size MXN and produce an output matrix B of same size such that it replaces all the non-border elements (numbers in bold) of A with its equivalent 1's complement and remaining elements same as matrix A.

A

1	2	3	4
6	**5**	**8**	3
2	**4**	**10**	1
9	1	2	5

B

1	2	3	4
6	**10**	**111**	3
2	**11**	**101**	1
9	1	2	5

Additional Exercises:

1. Write a CUDA program which reads an input matrix A of size MXN and produces an output matrix B of size MXN such that, each element of the output matrix is calculated in parallel. Each element, B[i][j], in the output matrix is obtained by adding the elements in i^{th} row and j^{th} column of the input matrix A.

Example:

A

1	**2**	**3**
4	5	6

B

O/p:

11	13	15
20	22	24

2. Write a CUDA program that reads a character type matrix A and integer type matrix B of size MXN. It produces an output string *STR* such that, every character of *A* is repeated *r* times (where *r* is the integer value in matrix B which is having the same index as that of the character taken in A). Write the kernel such that every value of input matrix must be produced required number of times by one thread.

Example:

A

p	C	a	P
e	X	a	M

B

1	2	4	3
2	4	3	2

Output String STR: pCCaaaaPPPeeXXXXaaaMM

Lab No 10: **Date:**

Programs using different CUDA Device memory types and synchronization

Objectives:

In this lab, student will be able to

1. Learn about CUDA 2D grid and 2D block
2. Implement the optimized parallel program applications using constant memory and shared memory

Synchronization: CUDA allows threads in the same block to coordinate their activities using a barrier synchronization function, **__syncthreads().** When a kernel function calls **__syncthreads**(), the thread that executes the function call will be held at the calling location until every thread in the block reaches the location. This ensures that all threads in a block have completed a phase of their execution of the kernel before any moves on to the next phase.

Shared Variables: Accessing shared memory is extremely fast and highly parallel. If a variable declaration is preceded by the keyword **__shared__** , it declares a shared variable in CUDA. Such declarations typically reside within a kernel function or a device function. The scope of a shared variable is within a thread block means all threads in a block see the same version of a shared variable. The lifetime of a shared variable is within the duration of the kernel. Shared variables are an efficient means for threads within a block to collaborate with each other. Shared memory is fast but it is small. A common strategy is partition the data into subsets called **tiles** so that each tile fits into the shared memory.

Constant Variables: If a variable declaration is preceded by the keyword **__constant__,** it de- clares a constant variable in CUDA. Declaration of constant variables must be outside any function body. The scope of a constant variable is all grids, meaning that all threads in all grids see the same version of a constant variable. The lifetime of a constant variable is the entire application execu- tion. Constant variables are stored in the global memory but are cached for efficient access. With appropriate access patterns, accessing constant memory is extremely fast and parallel. Currently, the total size of constant variables in an application is limited at 65,536 bytes. One may need to break up the input data volume to fit within this limitation.

Device Variables: A variable whose declaration is preceded only by the keyword **__device__** is a global variable and will be placed in global memory. Accesses to a global variable are slow. However, global variables are visible to all threads of all kernels. Their contents also persist through the entire execution. Thus, global variables can be used as a means for threads to collab- orate across blocks. Global variables are often used to pass information from one kernel invocation to another kernel invocation.

Solved Exercise:

Write a program in CUDA to perform tiled matrix multiplication using 2D Grid and 2D Block

```
//Matrix multiplication of 4x4 matrix
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <cuda_runtime.h>
#define BLOCK_WIDTH 2
#define TILE_WIDTH 2
#define WIDTH 4


_
__global__ void MatMulElementThreadShared(int *a, int *b, int *c) {
  __shared__ int MDs[TILE_WIDTH][TILE_WIDTH];
  _shared_ int NDs[TILE_WIDTH][TILE_WIDTH];
    int m;
    int bx=blockIdx.x; int by=blockIdx.y;
    int tx=threadIdx.x; int ty=threadIdx.y;

    int Row=by*TILE_WIDTH + ty;
    int Col= bx*TILE_WIDTH  + tx;

    int Pvalue=0;
    for(m=0; m<WIDTH/TILE_WIDTH; m++)
    {
          MDs[ty][tx]=a[Row*WIDTH+m*TILE_WIDTH+tx];
          NDs[ty][tx]=b[(m*TILE_WIDTH+ty)*WIDTH+Col];

    __syncthreads();

     for (int k = 0; k < TILE_WIDTH; k++)
     {
       Pvalue += MDs[ty][k]*NDs[k][tx];
     }
     __syncthreads();
   }
   c[Row*WIDTH+Col] = Pvalue;
}
```

```
int main() {
 int *matA, *matB, *matProd;
 int *da, *db, *dc;

   printf("\n== Enter elements of Matrix A (4x4) ==\n");

   matA = (int*)malloc(sizeof(int) * WIDTH * WIDTH);
   for(int i = 0; i < WIDTH * WIDTH; i++)
   {
      scanf("%d", &matA[i]);
    }

   printf("\n== Enter elements of Matrix B (4x4) ==\n");
   matB = (int*)malloc(sizeof(int) * WIDTH * WIDTH);
   for(int i = 0; i < WIDTH * WIDTH; i++)
   {
      scanf("%d", &matB[i]);
    }
   matProd  = (int*)malloc(sizeof(int) * WIDTH * WIDTH);

   cudaMalloc((void **) &da, sizeof(int) * WIDTH * WIDTH);
   cudaMalloc((void **) &db, sizeof(int) * WIDTH * WIDTH);
   cudaMalloc((void **) &dc, sizeof(int) * WIDTH * WIDTH);

 cudaMemcpy(da, matA, sizeof(int) * WIDTH *WIDTH, cudaMemcpyHostToDevice);
 cudaMemcpy(db, matB, sizeof(int) * WIDTH *WIDTH, cudaMemcpyHostToDevice);
   int NumBlocks = WIDTH / BLOCK_WIDTH;
   dim3 grid_conf (NumBlocks, NumBlocks);
   dim3 block_conf (BLOCK_WIDTH, BLOCK_WIDTH);

   MatMulElementThreadShared<<<grid_conf, block_conf>>>(da, db, dc);

cudaMemcpy(matProd,dc,sizeof(int)* WIDTH *WIDTH,cudaMemcpyDeviceToHost);
      printf("\n-=Result of Addition=-\n");
      printf("---------------\n");
      for (int i = 0; i < m; i++ ) {
         for (int j = 0; j < n; j++) {
            printf("%6d ", matProd[i * n + j]);
         }
```

```
            printf("\n");
        }
    cudaFree(da);
   cudaFree(db);
   cudaFree(dc);
   free(matA);
   free(matB);
   free(matProd);
   return 0;
}
```

Lab Exercises:

1. Write a program in CUDA to perform matrix multiplication using 2D Grid and 2D Block.
2. Write a program in CUDA to improve the performance of 1D parallel convolution using constant Memory.
3. Write a program in CUDA to perform inclusive scan algorithm.

Additional Exercises:

1. Write a program in CUDA which displays a shopping mall item menu with its price. The N number of friends are allowed to purchase as many items they want. Calculate the total purchase done by N friends.

4. Write a program in CUDA to perform tiled 1D convolution operation on the input array N of size *width* using the mask array, M of size *mask_width,* to produce the resultant array P of size *width.*

REFERENCES

1. Michael J. Quinn, "*Parallel Programming in C with MPI and OpenMP*", McGraw Hill Edition, 2003.
2. D. Kirk and W. Hwu , "*Programming Massively Parallel Processors –A Hands-on approach*", Elsevier Inc.,2nd Edition, 2013.