MANIPAL INSTITUTE OF TECHNOLOGY

MANIPAL

A Constituent Institution of Manipal University

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

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 3212] of Third Year B. Tech. (Computer Science and Engineering) Degree at MIT, Manipal, in the academic year 2018-2019.

Date: …….....

Signature
Faculty in Charge

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 for device neutral architecture using OpenCL
- Learn how to write host and kernel code in CUDA for nVIDIA GPU card
- To develop the skills of designing parallel algorithms and implement small projects 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
- Solve and test OpenCL programs using GPU architecture
- Develop CUDA programs for parallel applications
- Demonstrate the skill in parallel programming by developing mini projects

Evaluation plan

- Internal Assessment Marks : 60%
  - Continuous Evaluation : 40%

    Continuous evaluation component (for each evaluation):8 marks

    The assessment will depend on punctuality, program execution, maintaining the observation note and answering the questions in viva voce.
  - Project Evaluation : 20%
- End semester assessment of two hour 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.
- The programs should meet the following criteria:
    - Programs should be interactive with appropriate prompt messages, error messages if any, and descriptive messages for outputs.
    - Observation book should be complete with program, proper input output clearly showing the parallel execution in each process.
    - For comparing time of execution consider very large data size as input.
- 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 lab 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.
- A sample note preparation is given as a model for observation.

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 Visual Studio and Basics of MPI

Objectives:

In this lab, student will be able to

1. Understand the Visual Studio Environment and 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:

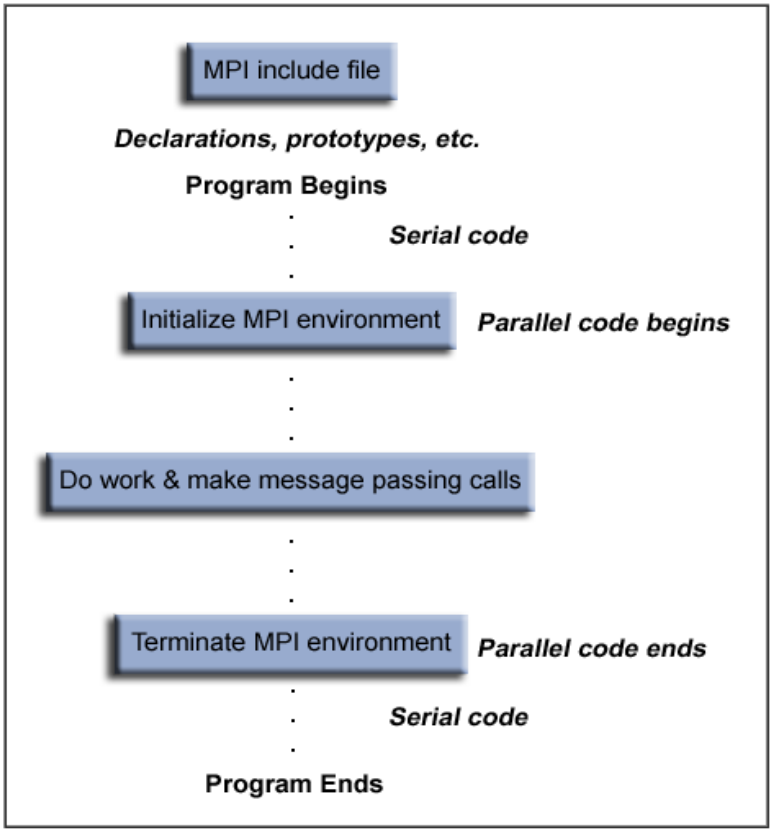
- In message passing programming, programmers view their programs (Applications) as a collection of co-operating processes with private (local) variables.
- The only way for an application to share data among processors is for programmer to explicitly code commands to move data from one processor to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

II. MPI Program Structure:



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 process",rank,size);
        MPI_Finalize();
        return 0;
}
```

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

Lab Exercises:

1. Write a simple C++ program to multiply two matrices of size M*x*N and N*x*P.
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.

Additional Exercises:

1. Write a program in C++ to count the words in a file and sort it in descending order of frequency of words i.e. highest occurring word must come first and least occurring word must come last.
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);

        if(rank==0)
         {
          scanf("%d",&x);
          MPI_Send(&x,1,MPI_INT,1,1,MPI_COMM_WORLD);
          fprintf(stdout,"I have send %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;
}
// Modify the above program. Use synchronous send and find out the difference
```

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 process 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 receives the number and prints it. Use standard send.
3) Write a MPI program to add an array of size N using two processes. Print the result in the root process. Investigate the amount of time taken by each process.
4) 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 process. The root process sends one value to each of the slaves. Let even ranked process find square of the received element and odd ranked process find cube of received element. Use Buffered send.
5) 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 root. Search a number in this array using root and another process. Print the result in the root.

2) Write a MPI program to read N elements of an array in the master process. Let N process including master process check the array values are prime or not.

3) 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)+……..+n! or (1+2+…..+n) depending on whether n is odd or even and print the result in the root process.

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 value 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 prints the factorial of that number. Use N number of processes.
2) Modify the above program such that every process returns the factorial to root process. Root process gathers the factorial and finds sum of it.
3) Write a MPI program to read a value M and N*x*M elements in the root process. 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. Use N number of processes.
4) 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.

5) Write a MPI Program to read two strings S1 and S2 of same length in the root process. Using *N* process including the root (string length is evenly divisible by N), produce the concatenated resultant string as shown below. Display the resultant string in the root process. Write the program using Collective communication routines.

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

Additional Exercises:

1) Write a MPI Program to read a string of length M in the root process. Using N processes (N evenly divides M) including the root toggle the characters and find the ASCII values of these toggled characters. Display the toggled characters and ASCII values in the root process.
2) Write a program to read a value M and N*x*M number of elements in the root. Using N processes do the following task. Find the square of first M numbers, Find the cube of next M numbers and so on. Print the results in the root.
3) Write a program to read a value M and N*x*M number of elements in the root. Using N number of processes find the sum 1+2…..+array element of each element and print the result in the root.

I/p: M=2 N=3

Array:	2	4	3	2	5	3
Result:	3	10	6	3	15	6

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.
2) Write a MPI program to calculate π-value by integrating $f(x) = 4 /(1+x^2)$. Area under the curve is divided into rectangles and the rectangles are distributed to the processors.
3) 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.
4) Write a MPI program to handle different errors using error handling routines.
5) Write a MPI program to read 4 *x* 4 matrix 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

Additional Exercises:

1) Write a MPI Program to read two 3 *x* 3 matrix in the root process. Using three processes including the root add the corresponding elements of A & B matrix. Display the added resultant matrix in the root.
2) Write a MPI program using collective communication that displays the sum of all prime numbers of 3 *x* 3 matrix in the root process. Use three processes.
3) Write a MPI Program to read a 3 *x* 3 matrix in the root process. Using three processes including the root find the maximum in each row and minimum in each column of this matrix. Display the maximum and minimum values in the root. Use collective communication routines.
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.
   Eg: Input : PCAP Output : PCCAAAPPPP

Lab No:5 **Date:**

OpenCL Introduction and programs on Vectors

Objectives:

In this lab, student will be able to

1. Understand how to write kernel code in OpenCL
2. Learn how to use different APIs in writing the host code in OpenCL
3. Study the execution environment necessary to execute OpenCL programs

I.OpenCL Introduction

- Open Computing Language is a heterogeneous programming framework that is managed by Khronos group.
- OpenCL is a framework for developing applications that execute across a range of device type made by different vendors.

OpenCL Specification: It is defined in four parts called Models.

Platform Model: Specifies that there is one processor coordinating the execution (the host) and one or more processors capable of executing OpenCL C code (the devices).

Execution Model: Defines how the **OpenCL environment** is configured on the host and how kernels are executed on the device. This includes

- setting up an **OpenCL context** on the host
- providing **mechanisms for host-device interaction**
- Defining a **concurrency model** used for execution on devices

Memory Model: Defines the abstract memory hierarchy that kernels use regardless of the actual underlying memory architecture.

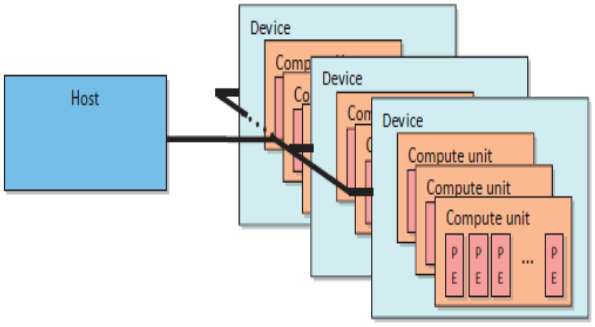
Programming Model: Defines how the concurrency model is mapped to physical hardware

Platform and Devices

Defines the **roles of the host and devices** and provides an abstract hardware model for devices. Devices that a platform can target are limited to those with which a vendor knows how to interact. If company A's platform is chosen, it can't communicate with company B's GPU.

- Platform model defines a **device as an array of compute units**, with each compute unit functionally independent from the rest.
- These compute units are further divided into **processing elements**.

Host and Device Interaction



II.Different APIs used for writing OpenCL programs

Discover & initialize platforms

- CLGetPlatformIDs() – It is used to discover the set of available platforms for a given system.
- This function will be called twice by an application.
- The **first call** passes an unsigned int pointer as the num_platforms argument and NULL is passed as a platform's argument.
- The pointer is **populated with the available number of platforms.**
- Then the programmer can allocate space to hold the platform information.

```
cl_int   clGetPlatformIDs ( cl_uint num_entries, cl_platform_id *platforms,
                            cl_uint *num_platforms);
```

- For the second call, cl_platform_id pointer is passed to function with enough space allocated for num_entries platforms.

```
cl_int   clGetPlatformIDs (cl_uint num_entries, cl_platform_id *platforms,
                           cl_uint *num_platforms);
```

Discover & initialize Devices

cl_GetDeviceIDs() call works very similar to cl_GetPlatformIDs(). It takes the additional arguments of a **platform** and a **device type**.

Device type argument can be used to limit the devices to GPUs only (**CL_DEVICE_TYPE_GPU**), CPUs only (**CL_DEVICE_TYPE_CPU**), all devices (**CL_DEVICE_TYPE_ALL**)

```
cl_int  clGetDeviceIDs ( cl_platform_id platform, cl_device_type device_type,
                         cl_uint num_entries, cl_device_id *devices, cl_uint
                         *num_devices);
```

```
cl_int  clGetDeviceIDs (cl_platform_id platform, cl_device_type device_type,
                        cl_uint num_entries, cl_device_id *devices,
                        cl_uint  *num_devices);
```

Execution Environment- Context

- Context is an abstract container that exists on the host.
- A context coordinates
  - the mechanisms for **host-device interactions**
  - **Manages the memory objects** that are available to the devices
  - **Keep track of the programs and kernels** that are created for each device

```
cl_context clCreateContext ( const cl_context_properties *properties,
                             cl_uint num_devices, const cl_device_id *devices, void
                             (CL_CALLBACK *pfn_notify)( const char *errinfo, const void
                             *private_info, size_t cb, void *user_data),void *user_data,
                             cl_int *errcode_ret);
```

Create command queue

- **Communication with a device** occurs by submitting commands to a command queue.
- Once the host decides which devices to work with and a context is created, **one command queue needs to be created per device**.
- Whenever the host needs an action to be performed by a device, it will submit commands to the proper command queue.

```
cl_command_queue clCreateCommandQueue( cl_context context,
                                       cl_device_id device,
                                       cl_command_queue_properties
                                        properties,
                                       cl_int* errcode_ret);
```

- The **parameter** properties by default is **INORDER** ie command queue commands are pulled from the queue in the order they were received.

Memory objects

- In OpenCL the **data needs to be physically present on a device** before execution can begin. In order for data to be transferred to a device it must first be **encapsulated as a memory object**.

Two types of Memory objects available in OpenCL are:

- buffers
- images

Buffers

- Whenever a **memory object** is created, it is **valid** only **within a single context**.

- Creating a buffer requires supplying the size of the buffer and a context in which the buffer will be allocated.
- It is visible for **all devices associated with the context**.

Create device buffers

cl_mem clCreateBuffer (**cl_context context, cl_mem_flags flags, size_t size,** void *host_ptr, **cl_int *errcode_ret**);

flags can be:

CL_MEM_READ_ONLY or CL_MEM_WRITE_ONLY or CL_MEM_READ_WRITE

Write host data to device buffers

- Data contained in host memory is transferred to and from an OpenCL buffer using the commands **CLEnqueueWriteBuffer() and CLEnqueueReadBuffer()** respectively.

cl_int clEnqueueWriteBuffer (**cl_command_queue command_queue**, **cl_mem buffer**, **cl_bool blocking_write**, **size_t offset, size_t cb, const void *ptr,** cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event);

- **blocking_write** is set to **CL_TRUE** if the transfer into an OpenCL buffer should complete before function returns.
- **offset** The offset in bytes in the buffer object to write to.
- **cb** The size in bytes of data being written.
- **ptr** The pointer to buffer in host memory where data is to be written from
- If **event_wait_list** is NULL, then this particular command does not wait on any event to complete. If event_wait_list is NULL, num_events_in_wait_list must be 0.
- If event_wait_list is not NULL, the list of events pointed to by event_wait_list must be valid and num_events_in_wait_list must be greater than 0.
- **event** Returns an event object that **identifies this particular write command** and can be used to query or queue a wait for this particular command to complete.

Creating a OpenCL program Object

- OpenCL C (kernel)code is called a program
- OpenCL programs are compiled at runtime through a series of API calls

The process of creating a kernel is as follows:

- The OpenCL C source code (kernel) is stored in a character stream/string.
- The source code is turned into a **program object** cl_program by calling **clCreateProgramWithSource()**
- The program object is then compiled for one of OpenCL devices with **clBuildProgram()**

Kernels & OpenCL execution Model

- **Kernels** are the part of OpenCL program that actually executed on a device. Return type must be void

```
__kernel void vecadd(__global int *C, __global int* A, __global int *B)
{
        int tid = get_global_id(0); // OpenCL intrinsic function
        C[tid] = A[tid] + B[tid];
}
```

- The unit of concurrent execution in OpenCL is a **work item**.
- Single iteration of a loop is mapped to a work-item.
- We tell OpenCL runtime to generate as many work-items as elements in the input and output arrays and allow the runtime to map those work-items to the underlying hardware.
- **get_global_id()** will give the **position of the current work item**.

Transfer kernel code to a character array

```
char *source_str;

size_t source_size;
fp = fopen("Hello.cl", "r");
   if (!fp) {
```

```
        fprintf(stderr, "Failed to load kernel.\n");
        getchar();
        exit(1);
    }
  source_str = (char*)malloc(MAX_SOURCE_SIZE);
  source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
  fclose( fp );
```

Create and compile the program

cl_program clCreateProgramWithSource(**cl_context context**, **cl_uint count**, **const char** **
strings, **const size_t *lengths**, **cl_int *errcode_ret**);

cl_int clBuildProgram(**cl_program program, cl_uint num_devices, cl_device_id * device_list**, const char *options, void (ptr_notify *) (cl_program, void *user_data), void *user_data);

If device_list is **NULL** value, the program executable is built for all devices associated with program for which a source or binary has been loaded.

If device_list is a **non-NULL** value, the program executable is built for devices specified in this list for which a source or binary has been loaded.

Create the Kernel

- The final stage to obtain a CL kernel object that can be used to execute kernels on a device is to extract the kernel from the **program object** (cl_program).
- The name of the kernel is passed to **clCreateKernel** function along with the program object.
- The kernel object will be retuned if the program object was valid and particular kernel is formed.

cl_kernel clCreateKernel(**cl_program program, const char* kernel_name , cl_int * errcode_ret**);

- kernel_name - A function name in the program declared with the __kernel qualifier

Set the kernel arguments

cl_int clSetKernelArg(**cl_kernel kernel, cl_uint arg_index , size_t arg_size,const void * arg_value**);

- Arg_index-Arguments to the kernel are referred by indices that go from 0 for the leftmost argument to n - 1, where n is the total number of arguments declared by a kernel.
- We must specify each kernel argument individually using the function **clSetKernelArg().**
- When the kernel is executed this information is used to transfer arguments to the device.

OpenCL execution Model

- When a kernel is executed, the programmer specifies the number of work items that should be created as an N dimensional range, ie NDRange. NDRange is 1 or 2 or 3 dimensional index space of work items that will often map to the dimension of either the input or output data.

Set global & local work size

- Scalability comes from dividing the work items of an NDRange into smaller equally sized workgroups using the same N dimensions.

- For example Work Dimension is 1 and if we set

  size_t global_work_size = 1024;

  size_t local_work_size=128;

Since there are 1024 total work-items and 128 work-items / work-group, a simple division of 1024 / 128 = **8 work-groups** are created.

- The global_work_size is the **total number of work-items (WI)**
- The local_work_size is the **number of work-items per work-group (WI/WG)**
- The number of work-groups is **the** global_work_size **/** local_work_size **or WG**

Enqueue the kernel for execution:

```
cl_int  clEnqueueNDRangeKernel( cl_command_queue command_queue, cl_kernel kernel,
                                cl_uint work_dim, const size_t *global_work_offset, const
                                size_t *global_work_size, const size_t  *local_work_size,
                                cl_uint num_events_in_wait_list, const cl_event
                                *event_wait_list, cl_event *event);
```

4 fields are related to work-item creation:

- **work_dim:** specifies the number of dimensions in which work-item will be created
- **global_work_size:** specifies the number of work items in each dimension of the ND range
- **local_work_size:** specifies the number of work items in each dimension of the workgroup
- **global_work_offset:** used to provide global IDs to the work items that do not start from zero

Read the output buffer back to the host

```
cl_int  clEnqueueReadBuffer ( cl_command_queue command_queue, cl_mem
                              buffer, cl_bool blocking_write, size_t offset, size_t
                              cb, const void *ptr, cl_uint num_events_in_wait_list,
                              const  cl_event *event_wait_list, cl_event *event);
```

Barrier Operations for a command queue

- Two types of barrier operations for a command queue:

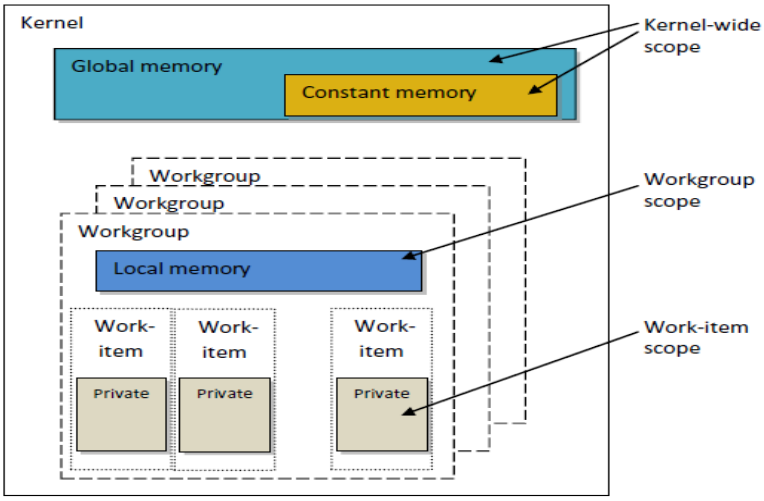
```
cl_int clFinish (cl_command_queue cmdQueue);
```

This function blocks until all of the commands in a command queue have completely executed.

```
cl_int clFlush (cl_command_queue cmdQueue);
```

This function blocks until all of the commands in a command queue have been removed from the command queue.

The abstract memory model



- **Global Memory(__global)** – Is **visible to all compute units on the device**, similar to main memory on a CPU based system.
- Whenever data is transferred from host to device and transferred back from device to host, it must reside in global memory.
- **Constant Memory(__constant)** – It is modeled as a part of global memory, so memory objects that are transferred to global memory can be specified as constant variables whose values never change fall into this category.
- **Local Memory (__local)** – It is modeled as being **shared by a workgroup**.
- **Private Memory** – It is **unique to an individual work item**. Local variables and non_pointer kernel arguments are private by default.

Release OpenCL resources:

```
// Free OpenCL resources
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(cmdQueue);
```

```
    clReleaseMemObject(bufferA);
    clReleaseMemObject(bufferB);
    clReleaseMemObject(bufferC);
    clReleaseContext(context);
// Free host resources
    free(A);  //arrays
    free(B);
    free(C);
    free(platforms);
    free(devices);
```

Solved Example:

Write an OpenCL program to read two arrays A and B of same size N. Find the sum of corresponding array elements. Store the result in array C.

```
// A COMPLETE PROGRAM FOR VECTOR-VECTOR ADDITION

#include <stdio.h>
#include <CL/cl.h>
#include<stdlib.h>



//Max source size of the kernel string
#define MAX_SOURCE_SIZE (0x100000)

int main(void)
{
   // Create the two input vectors
   int i;

   int LIST_SIZE;
   printf("Enter how many elements:");
   scanf("%d",&LIST_SIZE);

   int *A = (int*) malloc (sizeof (int) * LIST_SIZE);
```

```
//Initialize the input vectors
  for(i = 0; i < LIST_SIZE; i++)
  {
     A[i] = i; //if LIST_SIZE is very large
   }

int *B = (int*)malloc(sizeof(int)*LIST_SIZE);
  //Initialize the input vectors
  for(i = 0; i < LIST_SIZE; i++)
  {
     B[i] = i+10;
   }


  // Load the kernel source code into the array source_str
  FILE *fp;
  char *source_str;
  size_t source_size;

  fp = fopen("vectorCLKernel.cl", "r");

  if (!fp)
  {
     fprintf(stderr, "Failed to load kernel.\n");
     getchar();
     exit(1);
  }
  source_str = (char*)malloc(MAX_SOURCE_SIZE);
  source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);

 fclose( fp );

  // Get platform and device information
  cl_platform_id platform_id = NULL;
  cl_device_id device_id = NULL;
  cl_uint ret_num_devices;
```

```
    cl_uint ret_num_platforms;


    cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
     ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1,&device_id,
     &ret_num_devices);

    // Create an OpenCL context
    cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL,
&ret);

    // Create a command queue
    cl_command_queue command_queue = clCreateCommandQueue(context,
device_id,
    NULL,   &ret);

    // Create memory buffers on the device for each vector A, B and C
 cl_mem a_mem_obj = clCreateBuffer(context,
CL_MEM_READ_ONLY,LIST_SIZE * sizeof(int), NULL, &ret);
 cl_mem b_mem_obj = clCreateBuffer(context,
CL_MEM_READ_ONLY,LIST_SIZE * sizeof(int), NULL, &ret);

cl_mem c_mem_obj = clCreateBuffer(context,
CL_MEM_WRITE_ONLY,LIST_SIZE * sizeof(int), NULL, &ret);

    // Copy the lists A and B to their respective memory buffers
 ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE,
0,LIST_SIZE * sizeof(int), A, 0, NULL, NULL);

ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE,
0,LIST_SIZE * sizeof(int), B, 0, NULL, NULL);

    // Create a program from the kernel source
 cl_program program = clCreateProgramWithSource(context, 1, (const
char**)&source_str, (const size_t *)&source_size, &ret);

    // Build the program
```

```
    ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

    // Create the OpenCL kernel object
    cl_kernel kernel = clCreateKernel(program, "vector_add", &ret);

    // Set the arguments of the kernel
    ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
    ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
    ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);

    // Execute the OpenCL kernel on the array
    size_t global_item_size = LIST_SIZE;
    size_t local_item_size = 1;

//Execute the kernel on the device
cl_event event;
ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
&global_item_size, &local_item_size, 0, NULL, NULL);

ret = clFinish(command_queue);

    // Read the memory buffer C on the device to the local variable C
int *C = (int*)malloc(sizeof(int)*LIST_SIZE);
ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0,LIST_SIZE
* sizeof(int), C, 0, NULL, NULL);

   // Display the result to the screen
    for(i = 0; i < LIST_SIZE; i++)
       printf("%d + %d  = %d\n", A[i], B[i], C[i]);

    // Clean up
    ret = clFlush(command_queue);
    ret = clReleaseKernel(kernel);
    ret = clReleaseProgram(program);
    ret = clReleaseMemObject(a_mem_obj);
    ret = clReleaseMemObject(b_mem_obj);
    ret = clReleaseMemObject(c_mem_obj);
```

```
    ret = clReleaseCommandQueue(command_queue);
    ret = clReleaseContext(context);

    free(A);
    free(B);
    free(C);
    getchar();
    return 0;
}


// vectorCLKernel.cl
__kernel void vector_add(__global int *A, __global int *B, __global int *C)
{
    // Get the index of the current work item
    int i = get_global_id(0);
    // Do the operation
          C[i] = A[i] + B[i];
}
```

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

Lab Exercises:

1) Write an OpenCL program which takes N number of decimal values as input. It converts these values into their corresponding octal values and stores the result in another array in parallel.
2) Write an OpenCL program which takes N number of binary values as inputs and stores the one's complement of each element in another array in parallel.
3) Write an OpenCL program which takes an integer array with N number of values and it modifies that array by swapping alternative elements in that same array in parallel.

Additional Exercises:

1) Write an OpenCL program which takes N number of binary values as input. It converts these values into their corresponding decimal values and stores the result in another array in parallel.

2) Write an OpenCL program which reads an array A with N number of values and checks each element is prime or not. If prime, store the number as it is else store the square of the number in another array B.

Lab No:6 **Date:**

OpenCL programs on Strings

and to check the execution time in OpenCL

Objectives:

In this lab, student will be able to

1. Understand how to write kernel code in OpenCL to perform operations on strings
2. Learn how to use different APIs in writing the host code in OpenCL in order to deal with strings
3. Learn different APIs used in OpenCL to check the execution time taken by kernel

APIs used to find the time taken by kernel

- Execution time taken by the kernel can be extracted by using the function **clGetEventProfilingInfo()** after the execution of the kernel by the function clEnqueueNDRangeKernel().
- **clFinish()** function must be used after clEnqueueNDRangeKernel() so that finish time is calculated after the execution of kernel.

```
cl_int clGetEventProfilingInfo ( cl_event event, cl_profiling_info param_name,
                                 size_t param_value_size, void *param_value,
                                 size_t  *param_value_size_ret);
```

- **cl_profiling_info** can have following types:
  CL_PROFILING_COMMAND_START: (cl_ulong): A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event starts execution on the device.

  CL_PROFILING_COMMAND_END: (cl_ulong): A 64-bit value that describes the current device time counter in nanoseconds when the command identified by event has finished execution on the device.

- Profiling of OpenCL commands can be enabled by using a command-queue created with CL_QUEUE_PROFILING_ENABLE flag set in properties argument to clCreateCommandQueue().

Solved Exercise:

Write an OpenCL program which reads a string as input and it toggles every character of this string in parallel. Store the resultant string in another character array. Also find the execution time taken by kernel.

```
#include <stdio.h>
#include<time.h>
#include <CL/cl.h>
#include<string.h>
#include<stdlib.h>
#include<conio.h>
#define MAX_SOURCE_SIZE (0x100000)

int main(void)
{
   time_t start, end;
   start = clock();

    char tempstr[20486];

   //Initialize the input string
   int i;

    for(i=0; i<20485; i++)
      tempstr[i]='A';
    tempstr[20485]='\0';

 int len= strlen(tempstr);
 len++;

  char *str = (char*)malloc(sizeof(char)*len);

         strcpy(str,tempstr);
```

```
        // Load the kernel source code into the array source_str
    FILE *fp;
    char *source_str;
    size_t source_size;

    fp = fopen("strtoggle.cl", "r");

    if (!fp) {
                    fprintf(stderr, "Failed to load kernel.\n");
                    getchar();
                    exit(1);
                }
    source_str = (char*)malloc(MAX_SOURCE_SIZE);
    source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
    fclose( fp );

   cl_platform_id platform_id = NULL;
    cl_device_id device_id = NULL;
    cl_uint ret_num_devices;
    cl_uint ret_num_platforms;


    cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);

ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1,&device_id,
&ret_num_devices);

    // Create an OpenCL context
   cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);

    // Create a command queue
    cl_command_queue command_queue = clCreateCommandQueue(context,
device_id, CL_QUEUE_PROFILING_ENABLE, &ret);


    // Create memory buffers on the device for input and output string
```

```
cl_mem s_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,len*
sizeof(char), NULL, &ret);

        cl_mem t_mem_obj = clCreateBuffer(context,
CL_MEM_WRITE_ONLY,len* sizeof(char), NULL, &ret);

   // Copy the input string into respective memory buffer
   ret = clEnqueueWriteBuffer(command_queue, s_mem_obj, CL_TRUE, 0,len *
sizeof(char), str, 0, NULL, NULL);



   // Create a program from the kernel source
   cl_program program = clCreateProgramWithSource(context, 1,(const char
**)&source_str, (const size_t *)&source_size, &ret);

   // Build the program
   ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

   // Create the OpenCL kernel
   cl_kernel kernel = clCreateKernel(program, "str_chgcase", &ret);

  // Set the arguments of the kernel
   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&s_mem_obj);
   ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&t_mem_obj);

   // Set the global work size as string length
   size_t global_item_size = len; // Process the entire lists
   size_t local_item_size =1;

//Execute the OpenCL kernel for entire string in parallel
cl_event event;
   ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
&global_item_size, &local_item_size, 0, NULL, &event);
time_t stime=clock();

//kernel execution must be finished before calculating time
ret = clFinish(command_queue);
```

```
cl_ulong time_start, time_end;
double total_time;

//Find the kernel execution  start time
        clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
sizeof(time_start), &time_start, NULL);

//Find the kernel execution end time
clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
sizeof(time_end), &time_end, NULL);
total_time = double (time_end - time_start);

// Read the result in memory buffer on the device to the local variable strres
char *strres = (char*)malloc(sizeof(char)*len);

   ret = clEnqueueReadBuffer(command_queue, t_mem_obj, CL_TRUE, 0,len*
sizeof(char), strres, 0, NULL, NULL);
 printf("\nDone");
strres[len-1]='\0';
printf("\nResultant toggled string :%s",strres);
getchar();



   ret = clReleaseKernel(kernel);
   ret = clReleaseProgram(program);
   ret = clReleaseMemObject(s_mem_obj);
   ret = clReleaseMemObject(t_mem_obj);
   ret = clReleaseCommandQueue(command_queue);
   ret = clReleaseContext(context);

   end = clock();

        printf("\n\n Time taken to execute the KERNEL in milliseconds = %0.3f
msec\n\n", total_time/1000000);
        printf( "\nTime taken to execute the whole program in seconds: %0.3f
Seconds\n", (end-start)/(double)CLOCKS_PER_SEC );
```

```
free(str);
free(strres);
getch();
return 0;
}

// strtoggle.cl
__kernel void str_chgcase (__global char *A, __global char *B)
{

   int i = get_global_id(0);
 if(A[i] >='A' && A[i] <='Z')
    B[i] = A[i] + 32;
else
      B[i] = A[i] - 32;
}
```

Lab Exercises:

1) Write an OpenCL program which reads a string as input and produces resultant string which is having characters obtained by reversing ASCII value of corresponding character in the original string.
2) Write an OpenCL program that takes a string S as input and one integer value N. Produces output string N times as follows in parallel:
   I/p: S = Hello N = 3
   O/p String: HelloHelloHello (Each work item copies entire S)
3) Write an OpenCL program that reads a string and reverse entire string in parallel.
4) Write an OpenCL program which reads a string consisting of N words and reverse each word of it in parallel.

Additional Exercises:

1) Write an OpenCL program that takes a string S as input and one integer value N. Produces string N times as follows in parallel:

I/p: S = Hello N = 3

O/p String: HelloHelloHello (Each work item copies same character from the Input N times to the required position)

2) Write an OpenCL program which reads a string S and produces output string T as follows:

S: Hai T: Haaiii

(Each work item stores a character from input string S required number of times in T)

3)Write an OpenCL program which reads a string S as input. It produces an output string T such that if a character is vowel in S convert the character to uppercase else convert the character to lower case and store in T.

Lab No:7 **Date:**

OpenCL programs on Matrix

Objectives:

In this lab, student will be able to

1. Understand how to write kernel code in OpenCL to perform operations on matrix
2. Learn how to deal with two dimensional range in the host code in OpenCL
3. Learn about work items and work groups for two dimensional arrays

Work items and Work groups for Two Dimensional arrays

For example for a two Dimensional array, global and local work size can be set as follows:

```
size_t global_work_size[2]={64,32};

size_t local_work_size[2]={64,1};
```

This specifies:

- A global size of 64 work-items in dimension 0 and 32 work-items in dimension 1, for a total of 64 * 32 = 2,048 total work-items (WI).
- It also specifies the local size to be 64 WI/WG in dimension 0 and 1 WI/WG in dimension 1. This results in 64/64 = 1 work-group in dimension 0 and 32/1 work-groups in dimension 1 for a total of 2,048 work-groups (WG).

Solved Exercise:

Write an OpenCL program to read matrix A of size M*x*N, read matrix B of size N*x*P. Produce a resultant matrix C of size MxP in parallel.

```
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>

//Max source size of the kernel string
#define MAX_SOURCE_SIZE (0x100000)
```

```
//Width and height of matrices
#define WA  3
#define  HA  2
#define  WB 3
#define  HB  3
#define  WC 3
#define  HC 2
#define BLOCK_SIZE 1

int main(void)
{
        unsigned int size_A = WA * HA;
        unsigned int size_B = WB * HB;
        int* a = (int*) malloc(size_A*sizeof(int));
        int* b = (int*) malloc(size_B*sizeof(int));

        FILE *fp;
        char *source_str;
        size_t source_size;

        fp = fopen("matrixKernel.cl", "r");
        if (!fp) {
                        fprintf(stderr, "Failed to load kernel.\n");
                        getchar();
                        exit(1);
                }

        source_str = (char*)malloc(MAX_SOURCE_SIZE);
        source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);

        fclose( fp ); //Close file pointer
        int i;
        for(i=0; i<size_A; i++)
                scanf("%d",&a[i]);

        for(i=0; i<size_B; i++)
                scanf("%d",&b[i]);
```

```
        unsigned int size_C = WC * HC;
        int* c = (int*) malloc(size_C*sizeof(int));

   // Get platform and device information
   cl_platform_id platform_id = NULL;
   cl_device_id device_id = NULL;
   cl_uint ret_num_devices;
   cl_uint ret_num_platforms;


   cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
    ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1,&device_id,
    &ret_num_devices);

   // Create an OpenCL context
   cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);

   // Create a command queue
   cl_command_queue command_queue = clCreateCommandQueue(context,
device_id, NULL, &ret);

   // Create memory buffers on the device for each vector
cl_mem a_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
size_A *sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_READ_ONLY,
size_B *sizeof(int), NULL, &ret);
cl_mem c_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
size_C*sizeof(int), NULL, &ret);

   // Copy the lists A and B to their respective memory buffers
   ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE,
0,size_A*sizeof(int),a , 0, NULL, NULL);
        ret = clEnqueueWriteBuffer(command_queue, b_mem_obj, CL_TRUE,
0,size_B*sizeof(int),b , 0, NULL, NULL);

   // Create a program from the kernel source
```

```
   cl_program program = clCreateProgramWithSource(context, 1,(const char **)&source_str, (const size_t *)&source_size, &ret);

   // Build the program
   ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

   // Create the OpenCL kernel
   cl_kernel kernel = clCreateKernel(program, "mat_mul", &ret);

         // Set the arguments of the kernel
   int widthA = WA;
   int widthB = WB;
   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
   ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);
   ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&c_mem_obj);
   ret = clSetKernelArg(kernel, 3, sizeof(cl_int), (void *)&widthA);
   ret = clSetKernelArg(kernel, 4, sizeof(cl_int), (void *)&widthB);
   size_t localWorkSize[2]={BLOCK_SIZE,BLOCK_SIZE};
   size_t globalWorkSize[2]= {WC,HC};

   ret = clEnqueueNDRangeKernel(command_queue, kernel, 2, NULL,globalWorkSize, localWorkSize, 0, NULL,NULL);
 clFinish(command_queue);

ret = clEnqueueReadBuffer(command_queue, c_mem_obj, CL_TRUE, 0, size_C*sizeof(int), c, 0, NULL, NULL);

printf("\nMatrix C\n");
 for(int i = 0; i < size_C; i++)
          {
            printf("%d\t", c[i]);
           if(((i + 1) % WC) == 0)
                    printf("\n");
          }

      free(source_str);
      clReleaseContext(context);
```

```
    clReleaseKernel(kernel);
    clReleaseProgram(clProgram);
    clReleaseCommandQueue(command_queue);
    getchar();
}

//matrixKernel.cl
__kernel void mat_mul( __global int* Aelements, __global int* Belements,__global
int* Celements,int widthA, int widthB)
{
 int row=get_global_id(1);
 int col=get_global_id(0);
 int sum=0;

 for(int i=0; i<widthA; i++)
  {
        sum+=Aelements[row*widthA+i]*Belements[i*widthB+col];
   }
  Celements[row*widthB+col] = sum;
 }
```

Lab Exercises:

1) Write an OpenCL program to read M*x*N matrix. 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.
2) Write an OpenCL program that reads a matrix A of size M*x*N and produces a 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

3) Write an OpenCL program to read a matrix A of size M*x*N and compute the transpose of this matrix into B.
4) Write an OpenCL program which reads an input matrix A of size M*x*N. It produces an output matrix B of size M*x*N such that, each element of the output matrix is calculated in parallel. Each element in the output matrix is a total sum of row sum and column sum of those elements that lies in the same row and same column index of that element in the input matrix.

Example:

A

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

B

O/p:

11	13	15
20	22	24

Additional exercises:

1) Write an OpenCL program that reads a M*x*N 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.
2) Write an OpenCL program to read a matrix A of size N*x*N. 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.
3) Write an OpenCL program to read a matrix A of size N*x*N where N is even. Produce a resultant matrix B of size N**x**N such that: consider elements of matrix A in to 4 equal parts. First part elements should be incremented by 1, Second part elements should be incremented by 2, Third part elements should be incremented by 3 and last part elements should be incremented by 4.Write the kernel code which does this in parallel for all the elements.

Example : Input : N=4

A

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

B

4	**9**	4	7
3	**4**	7	8
5	7	7	5
6	5	5	9

Lab No:8 **Date:**

OpenCL programs on Sorting and searching

Objectives:

In this lab, student will be able to

1. Understand how to write kernel code in OpenCL to perform sorting operations
2. Understand how to write kernel code in OpenCL to perform searching operations

Solved exercise:

Write an OpenCL program to read an array of N integer values. Sort each element of this array in parallel and store the result in another array.

```
//A COMPLETE PROGRAM FOR PARALLEL SORTING

#include <stdio.h>
#include<stdlib.h>
#include <CL/cl.h>

#define MAX_SOURCE_SIZE (0x100000)

int main(void)
{
   // Create the two input vector
   int i, LIST_SIZE;
   printf("Enter how many elements:");
   scanf("%d",&LIST_SIZE);

   int *A = (int*)malloc(sizeof(int)*LIST_SIZE);

  //Initialize the input vector
   for(i = 0; i < LIST_SIZE; i++)
   {
     printf("Enter %d values:\n",LIST_SIZE);
      scanf("%d",&A[i]);
```

```
    }

   // Load the kernel source code into the array source_str
   FILE *fp;
   char *source_str;
   size_t source_size;

   fp = fopen("sortCLKernel.cl", "r");

   if (!fp)
   {
      fprintf(stderr, "Failed to load kernel.\n");
      getchar();
      exit(1);
   }
   source_str = (char*)malloc(MAX_SOURCE_SIZE);
   source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);

  fclose( fp );

   // Get platform and device information
   cl_platform_id platform_id = NULL;
   cl_device_id device_id = NULL;
   cl_uint ret_num_devices;
   cl_uint ret_num_platforms;


   cl_int ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
    ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_GPU, 1,&device_id,
    &ret_num_devices);

   // Create an OpenCL context
   cl_context context = clCreateContext( NULL, 1, &device_id, NULL, NULL,
&ret);

   // Create a command queue
```

```
   cl_command_queue command_queue = clCreateCommandQueue(context,
device_id,     NULL,   &ret);

   // Create memory buffers on the device for array A and resultant array
 cl_mem a_mem_obj = clCreateBuffer(context,
CL_MEM_READ_ONLY,LIST_SIZE * sizeof(int), NULL, &ret);
cl_mem b_mem_obj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
LIST_SIZE * sizeof(int), NULL, &ret);



   // Copy the lists A to their respective memory buffer
 ret = clEnqueueWriteBuffer(command_queue, a_mem_obj, CL_TRUE,
0,LIST_SIZE * sizeof(int), A, 0, NULL, NULL);

   // Create a program from the kernel source
   cl_program program = clCreateProgramWithSource(context, 1,(const char**)
&source_str, (const size_t *)&source_size, &ret);

   // Build the program
   ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

   // Create the OpenCL kernel
   cl_kernel kernel = clCreateKernel(program, "sort_vector", &ret);

   // Set the arguments of the kernel
   ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&a_mem_obj);
   ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&b_mem_obj);



   // Execute the OpenCL kernel on the array
   size_t global_item_size = LIST_SIZE;
   size_t local_item_size = 1;



         // Read the memory buffer C on the device to the local variable C
         int *RES = (int*)malloc(sizeof(int)*LIST_SIZE);
```

```
        cl_event event;
        ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
&global_item_size, &local_item_size, 0, NULL, NULL);
        ret = clFinish(command_queue);



        ret = clEnqueueReadBuffer(command_queue, b_mem_obj, CL_TRUE,
0,LIST_SIZE * sizeof(int), RES, 0, NULL, NULL);

  // Display the result to the screen
        printf("Sorted Array\n");
   for(i = 0; i < LIST_SIZE; i++)
     printf("%d = %d\n", A[i], RES[i]);

   // Clean up
   ret = clFlush(command_queue);
   ret = clReleaseKernel(kernel);
   ret = clReleaseProgram(program);
   ret = clReleaseMemObject(a_mem_obj);
   ret = clReleaseMemObject(b_mem_obj);
   ret = clReleaseCommandQueue(command_queue);
   ret = clReleaseContext(context);

   free(A);
   free(RES);
   getchar();
   return 0;
}
// sortCLKernel.cl
__kernel void sort_vector(__global int *A, __global int *RES)
{
   // Get the index of the current work item
   int id = get_global_id(0);
   int data = A[id];

   int n=get_global_size(0);
   int i,pos=0;
```

```
    for(i=0; i<n; i++)
     {
       if((A[i]<data) || (A[i] == data && i<id))
             pos++;
     }

     RES[pos] = data;
}
```

Lab Exercises:

1. Write an OpenCL program to read a string consisting of N words. Sort the entire string characterwise using selection sort in parallel.
2. Write an OpenCL program to read an integer array of N numbers. Sort this array using odd-even transposition sorting. (Use 2 kernels)
3. Write an OpenCL program to find the occurrences of a word in the string consisting of N words in parallel.

Additional Exercises:

1) Implement bitonic merge sort in parallel.
2) Implement quick sort in parallel.

Lab No:9 **Date:**

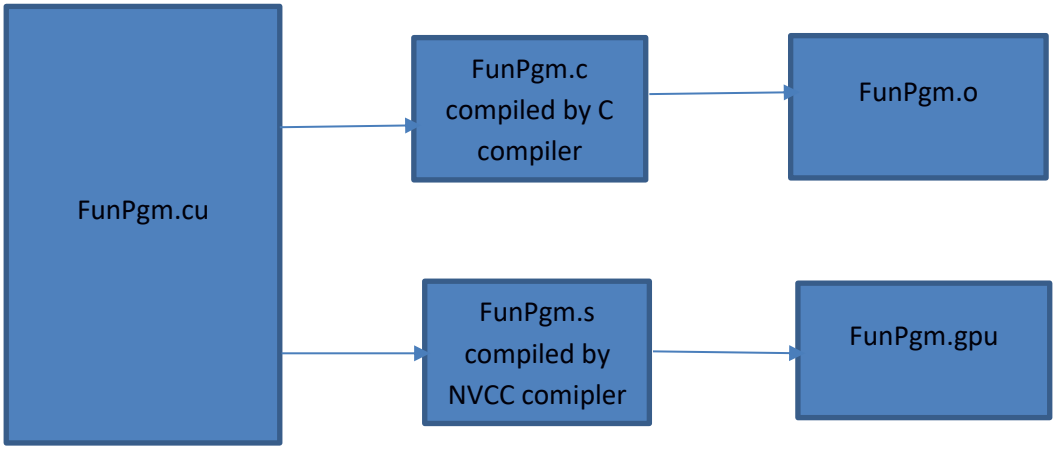
CUDA Programs on arrays and matrices

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. Write simple programs on Arrays and Matrices

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.

Type the following program save it as .cu extension observe the output.

```
int main(void) {

        printf("Hello World!\n");

        return 0;

        }
```

NVCC compiler has successfully compiled your code and produced the output there is no parallel code to be run on device in the above code.

Solved Exercise: Program to add two numbers.

```
__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);
         //print result
        // 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)

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 using a)block size as N b)N threads
2. Write a program in CUDA to add two Matrices using
   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.
3. Repeat the above exercise for matrix multiplication.
4. Write a program in CUDA to find transpose of a matrix in parallel.

Additional Exercises:

1. Write a program in CUDA to perform odd even transposition sort in parallel.
2. Write a program in CUDA to perform selection sort in parallel.
3. Write a program in CUDA to swap upper and lower diagonal matrices in parallel.

Lab No 10: **Date:**

CUDA Programs on strings

Objectives:

In this lab, student will be able to

1. Write simple programs on Strings
2. Learn about CUDA blocks and threads
3. Learn to compute time of kernel execution
4. Learn to handle errors in the kernel

CUDA threads, blocks and grid

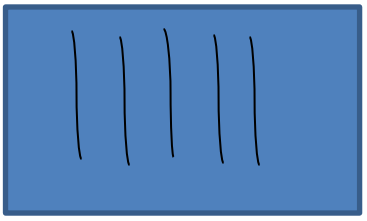
Thread – Distributed by the CUDA runtime. A single path of execution there can be multiple threads in a program.

(identified by threadIdx)

CUDA Thread

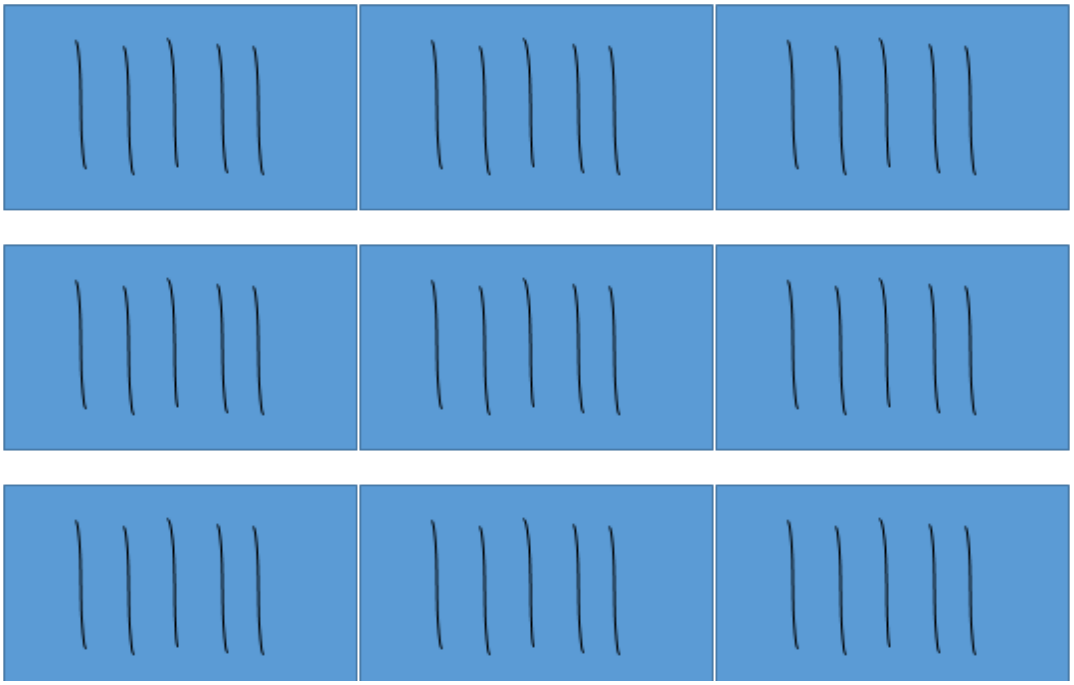
Block – A user defined group of 1 to 512 threads.

(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 Example:

A CUDA program which takes a string as input and converts lower case character into its equivalent upper case character.

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

```
#define N 1024

__global__ void CUDAStrCopy(char* A, char C[N]){
        int i = threadIdx.x;
                C[i]=A[i]-32;
                printf("%s\n",C[i]);
    }
int main(){

char A[N];;
char C[N];
char *pA,*pC;
for(int i=0;i<N;i++)
{
        A[i]='a';
}
printf("C = \n");
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
cudaMalloc((void**)&pA, N*sizeof(char));
cudaMalloc((void**)&pC, N*sizeof(char));
cudaMemcpy(pA, A, N*sizeof(char), cudaMemcpyHostToDevice);
cudaError_t error =cudaGetLastError();
if (error != cudaSuccess)
                {
                        printf("CUDA  Error1:  %s\n",  cudaGetErrorString(error));

                }
CUDAStrCopy<<<1,N>>>(pA,pC);
error =cudaGetLastError();
if (error != cudaSuccess)
                {
                        printf("CUDA  Error2:  %s\n",  cudaGetErrorString(error));

                }
```

```
cudaMemcpy(C, pC, N*sizeof(char), cudaMemcpyDeviceToHost);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
int i;
/* printf("VALUE OF C IN HOST AFTER KERNEL EXECUTION\n");
for(int i=0;i<N;i++)
        printf("%c\n",C[i]);*/
printf("Time Taken=%f",elapsedTime);
cudaFree(pA);
cudaFree(pC);
printf("\n");
getch();
return 0;
}
```

Explanation:

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.

Do the following with solved exercise code:

1. Type the program and note down the error which occurs during execution.

2. Change the %s in printf statement of kernel code to %c and note down the time of execution.
3. Comment out printf statement present in kernel code and uncomment the multi-line comment. Note down the time taken for execution of the program. Which version is faster and why?
4. Change the value of #DEFINE N to 1025 Note down the error.

Lab Exercises:

1. Write a program in CUDA to find substring in main string.
2. Write a program in CUDA to reverse a string in parallel.
3. Write a program in CUDA to display the binary values of each character of given string.
4. Write a program in CUDA to count the number of times a given word is repeated in a sentence.
5. Write a program in CUDA to implement matrix addition and multiplication using 2D grids and 2D blocks.

Additional Exercises:

1. Write a program in CUDA to sort a given string using selection sort.
2. Write a program in CUDA to sort each word of a given string using selection sort.

Mini Project

Objective:

To implement a parallel application in MPI and CUDA or OpenCL.

Project Details

1. Student must do a mini project in MPI. He / She has to implement the same mini project in either OpenCL or CUDA.
2. Student must submit the synopsis in 7th lab.
3. Complete the MPI mini project and demonstrate by 10th lab.
4. Complete the OpenCL/CUDA mini project and demonstrate by 12th lab.
5. Student must submit the report in 12th lab.

Project Synopsis format

1. Students can form a group of at most 2 members.
2. Synopsis should contain the following
    a. Project title.
    b. Abstract.
    c. Team members name, Section and roll number.

Project Report format for research projects

1. Abstract
2. Motivation
3. Objectives
4. Introduction
5. Literature review
6. Methodology
7. Results
8. Limitations and Possible Improvements
9. Conclusion
10. References

Other type of projects can exclude literature review.

Project Examples

1. Implement Dijkstra algorithm in parallel.
2. Implement image editing in parallel. Image editing includes finding the edge, brightening, blurring, embossing of an image.
3. Implement image reconstruction in parallel. When you bisect the image to find the features of an image when you recombine the image the image will be distorted if not reconstructed correctly.
4. Implement red eye problem in an image editing using parallel approach.

References:

1. V. Rajaraman, C. Siva Ram Murthy, “Parallel Computers Architecture and Programming” Prentice-Hall India, 2000.
2. Michael J Quinn, “Parallel Computing: Theory and Practice” Tata McGraw Hill, 2nd Edition,1994.
3. Michael J Quinn, “Parallel Programming in C with MPI and OpenMP”, Tata McGraw Hill, 2011.
4. Benedict R. Gaster, Lee Howes, David R, Perhaad Mistry, Dana Schaa, “Heterogeneous Computing with OpenCL” Morgan Kaufmann, 2012.
5. David B. Kirk, Wen-Mei. W. Hwu, “Programming Massively Parallel Processors”, Morgan Kaufmann, second Edition, 2013
6. https://computing.llnl.gov/tutorials/mpi/