Parshvanath Charitable Trust's
A. P. SHAH INSTITUTE OF TECHNOLOGY
(Approved by AICTE New Delhi & Govt. of Maharashtra, Affiliated to University of Mumbai)
(Religious Jain Minority)

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING (DATA SCIENCE & AIML)

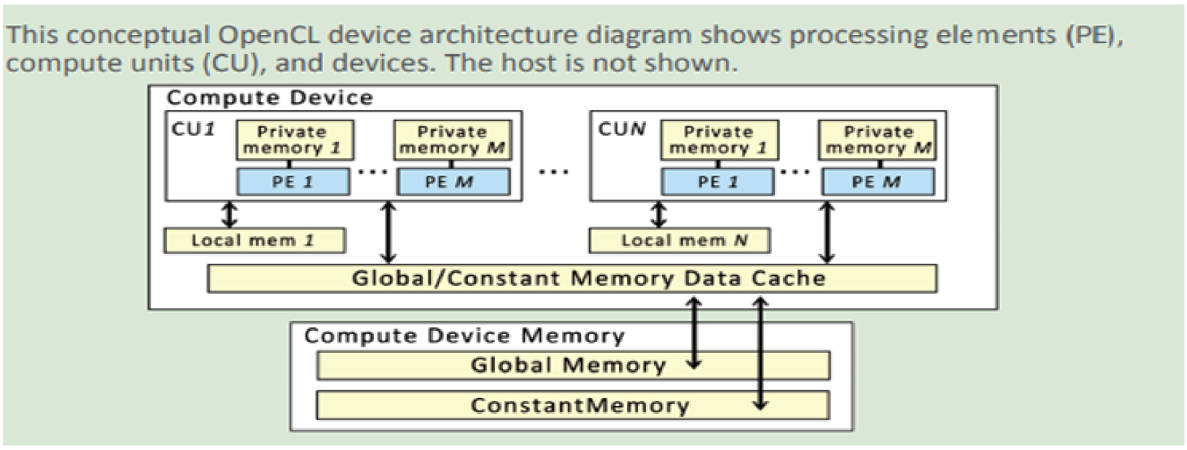
UNIT TEST - II SOLUTION

Class: TE **Semester: VI** **Subject: CSDLO6011 High Performance Computing**

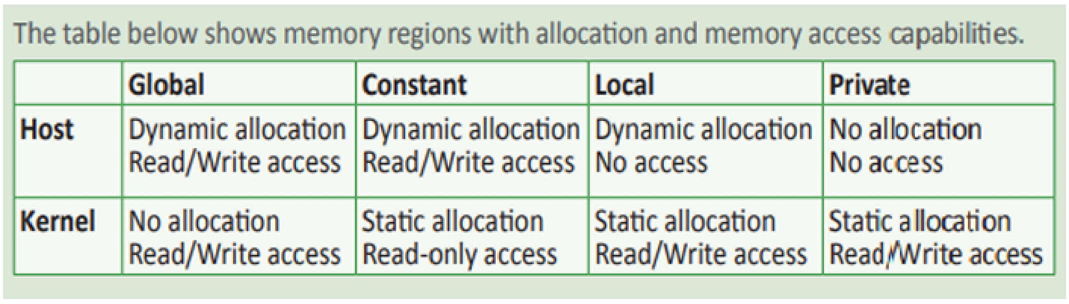
Date: 19-04-2024 **Time: 2:00 TO 3:30 PM** **Max marks: 40**

Note the following instructions

1. **Attempt all questions.**
2. **Draw neat diagrams wherever necessary.**
3. **Write everything in ink (no pencil) only.**
4. **Assume data, if missing, with justification.**

Q. N.	**Questions**
Q.1.	**Attempt any two.**
1 **Solution:**	**Sketch and Explain** OpenCL Device Architecture Diagram.  This conceptual OpenCL device architecture diagram shows processing elements (PE), compute units (CU), and devices. The host is not shown. Compute Device CU1 Private memory 1 Private memory M PE 1 PE M CUN Local mem 1 Local mem N Global/Constant Memory Data Cache Compute Device Memory Global Memory ConstantMemory Figure above zooms in further and gives us another perspective look at the same model we've seen previously but with additional memory model layout along with their corresponding capabilities. Each PE has a private memory for fast access before requiring it to communicate with local memory, and global/constant cache. Constant memory is a fixed (read-only) memory type that is expected not to change during the course of kernel execution. Higher up from cache memory is the main memory

of the device itself which requires more latency to get access there. Also as seen, there are similar global/constant memory type in the main memory of the device.



The table below shows memory regions with allocation and memory access capabilities.

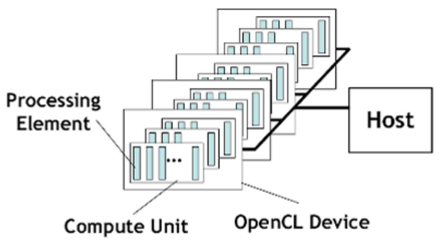
	Global	Constant	Local	Private
Host	Dynamic allocation Read/Write access	Dynamic allocation Read/Write access	Dynamic allocation No access	No allocation No access
Kernel	No allocation Read/Write access	Static allocation Read-only access	Static allocation Read/Write access	Static allocation Read/Write access

2

Solution:

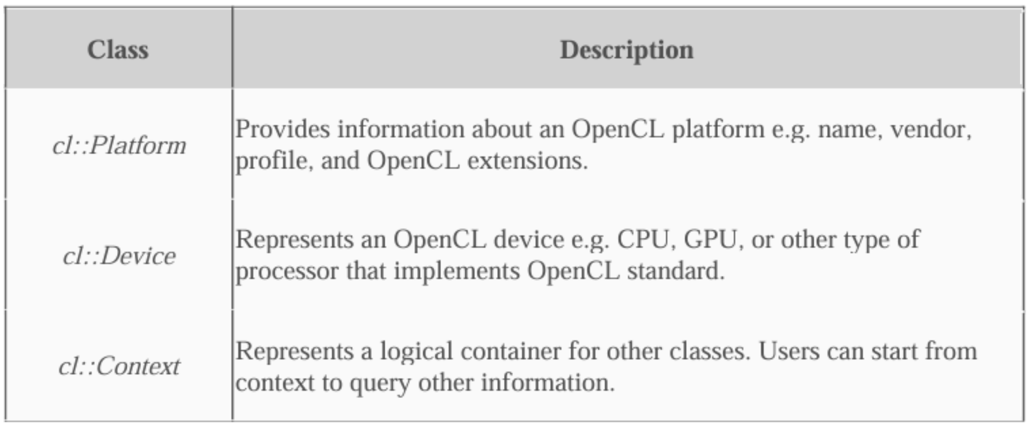
Draw and Explain OpenCL Platform Model.

OpenCL Platform Model

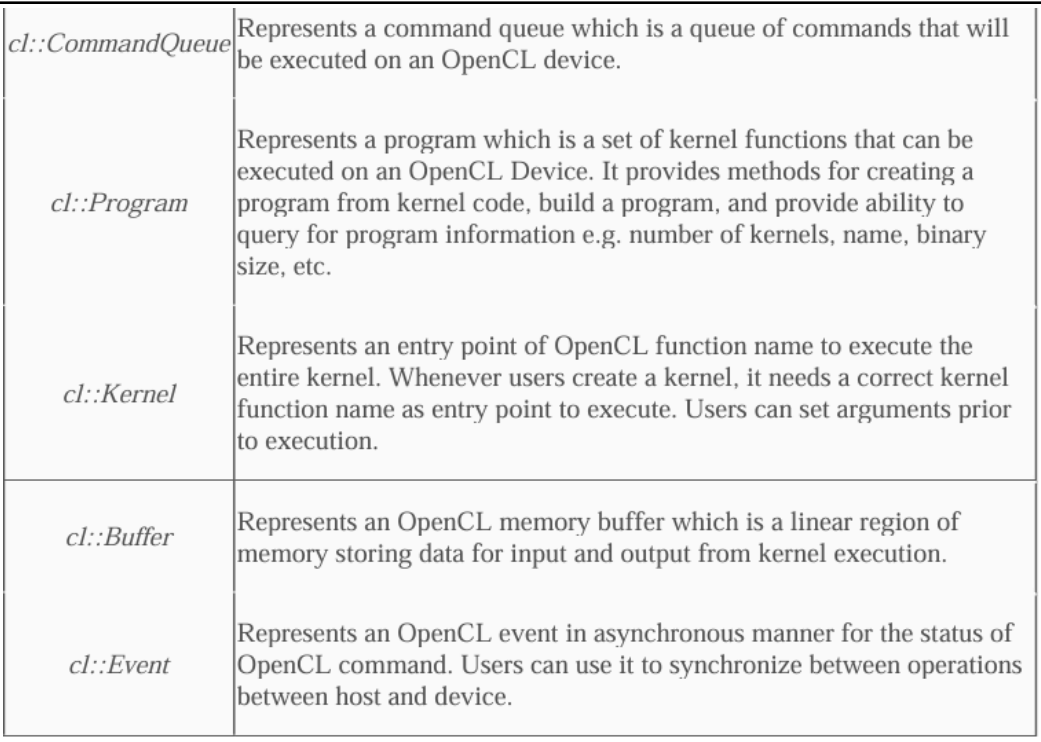


- One ***Host*** and one or more ***OpenCL Devices***
  - Each OpenCL Device is composed of one or more ***Compute Units***
    - Each Compute Unit is divided into one or more *Processing Elements*
- Memory divided into ***host memory*** and ***device memory***

The platform model of OpenCL is similar to the one of the CUDA programming model. In short, according to the OpenCL Specification, "The model consists of a host (usually the CPU) connected to one or more OpenCL devices (e.g., GPUs, FPGAs). An OpenCL device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs). Computations on a device occur within the processing element" An OpenCL program consists of

	two parts: host code and device code. As the name suggests, the host code is executed by the host and also "submits the kernel code as commands from the host to OpenCL devices". Finally, such as in the CUDA programming model, the host communicates with the device(s) through the global memory of the device(s). As in the CUDA programming model, there is a memory hierarchy on the device. However, we have omitted these details for the sake of greater simplicity. As we can see from the figure, OpenCL device (think GPU) consists of bunch of Compute unit. Each of them consists of dozens of Processing Elements (PE). In the big picture, memory is divided into host memory, and device memory.
3 **Solution:**	**List and Explain** any five classes of OpenCL. 

Class	Description
cl::Platform	Provides information about an OpenCL platform e.g. name, vendor, profile, and OpenCL extensions.
cl::Device	Represents an OpenCL device e.g. CPU, GPU, or other type of processor that implements OpenCL standard.
cl::Context	Represents a logical container for other classes. Users can start from context to query other information.



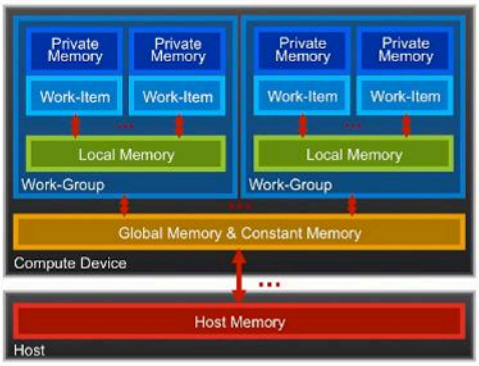
cl::CommandQueue	Represents a command queue which is a queue of commands that will be executed on an OpenCL device.
cl::Program	Represents a program which is a set of kernel functions that can be executed on an OpenCL Device. It provides methods for creating a program from kernel code, build a program, and provide ability to query for program information e.g. number of kernels, name, binary size, etc.
cl::Kernel	Represents an entry point of OpenCL function name to execute the entire kernel. Whenever users create a kernel, it needs a correct kernel function name as entry point to execute. Users can set arguments prior to execution.
cl::Buffer	Represents an OpenCL memory buffer which is a linear region of memory storing data for input and output from kernel execution.
cl::Event	Represents an OpenCL event in asynchronous manner for the status of OpenCL command. Users can use it to synchronize between operations between host and device.

4 **Sketch and Explain** OpenCL Memory Model.

Solution: See the below figure for another clearer overview of memory model with some interchangeably terms. Similarly, but with notable key terms on work-item, and work-group. We can see PE as work-item. There are lots of work-items as per single work-group. Memory model as previously mentioned dispersed all across the whole architecture working from work-item to work-group, and interconnecting between device and host (think PC). Notable note as seen at the bottom of the figure is that we as a user of OpenCL would be responsible for moving data back and forth between host and device. We will see why this is the case when we get involved with the code. But in short, because data on both ends need to be synchronized for consistency in consuming result from computation or feeding data for kernel execution.

OpenCL Memory model

- *Private Memory*
  - Per work-item
- *Local Memory*
  - Shared within a work-group
- *Global Memory*
  Constant Memory
  - Visible to all work-groups
- *Host memory*
  - On the CPU



Memory management is explicit:
You are responsible for moving data from
host → global → local *and* back

Q.2.	**Attempt any two**
1	**Develop** an MPI program for addition of two arrays.

Solution:

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

#define MASTER 0
#define ARRAY_SIZE 8000

int main(int argc, char *argv[])
{

        int *a, *b, *c;
        int *ap, *bp, *cp;
        int total_proc, rank, n_per_proc, n = ARRAY_SIZE, i;
        MPI_Status status;
        double start_time, end_time;

  MPI_Init(&argc, &argv);

   MPI_Comm_size(MPI_COMM_WORLD, &total_proc);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
        if (rank == MASTER)
        {
        a = (int *)malloc(sizeof(int) * n);
        b = (int *)malloc(sizeof(int) * n);
        c = (int *)malloc(sizeof(int) * n);

        for (i = 0; i < n; i++)
        a[i] = i;

        for (i = 0; i < n; i++)
        b[i] = i;
        }

        n_per_proc = n / total_proc;

        ap = (int *)malloc(sizeof(int) * n_per_proc);
        bp = (int *)malloc(sizeof(int) * n_per_proc);
        cp = (int *)malloc(sizeof(int) * n_per_proc);

  MPI_Scatter(a, n_per_proc, MPI_INT, ap, n_per_proc, MPI_INT, MASTER,
MPI_COMM_WORLD);


        MPI_Scatter(b, n_per_proc, MPI_INT, bp, n_per_proc, MPI_INT, MASTER,
MPI_COMM_WORLD);


        for (i = 0; i < n_per_proc; i++)
        cp[i] = ap[i] + bp[i];

        MPI_Gather(cp, n_per_proc, MPI_INT, c, n_per_proc, MPI_INT, MASTER,
MPI_COMM_WORLD);

        if (rank == MASTER)
        {
        int good = 1;
        for (i = 0; i < n; i++)
        {
        if (c[i] != a[i] + b[i])
        {
           printf("problem at index %d\n", i);
```

```
                        good = 0;
                        break;
            }
            }

            if (good)
            {
            printf("Added %d Elements Successfully\n", n);
            end_time = MPI_Wtime();
            printf("Wallclock time elapsed: %lf seconds\n", end_time - start_time);
            }
            }

            if (rank == MASTER)
            {
            free(a);
            free(b);
            free(c);
            }

            free(ap);
            free(bp);
            free(cp);

            MPI_Finalize();
            return 0;
}
```

2

Develop an MPI program for sum of n natural numbers.

Solution:

```
#include <stdio.h>
#include <mpi.h>
 int main(int argc, char *argv[])
{
            int myRank;
            int size;
            int sum;
            int lower, upper;
            int i;
            double local_result = 1.0;
            double total; //final sum value
            double start_time, end_time;

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

	```
	   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
	   MPI_Comm_size(MPI_COMM_WORLD, &size);
	         if (myRank == 0)
	         {
	      printf("Enter a number : ");
	      scanf("%d", &sum);
	         }
	MPI_Bcast(&sum, 1, MPI_INT, 0, MPI_COMM_WORLD);
	    if (myRank == 0)
	         lower = 1;
	         else
	         lower = myRank * (sum / size) + 1;

	         if (myRank == (size - 1))
	         upper = sum;
	         else
	         upper = (myRank + 1) * (sum / size);
	         for (i = lower; i <= upper; i++)
	         {
	         local_result = local_result + (double)i;
	         }
	MPI_Reduce(&local_result, &total, 1, MPI_DOUBLE, MPI_PROD, 0,
	MPI_COMM_WORLD);
	         printf("The sum of %d natural numbers : %lf \nCalculated using %d processes\n",
	sum, total, size);
	         MPI_Finalize();
	         return 0;
	}
	```

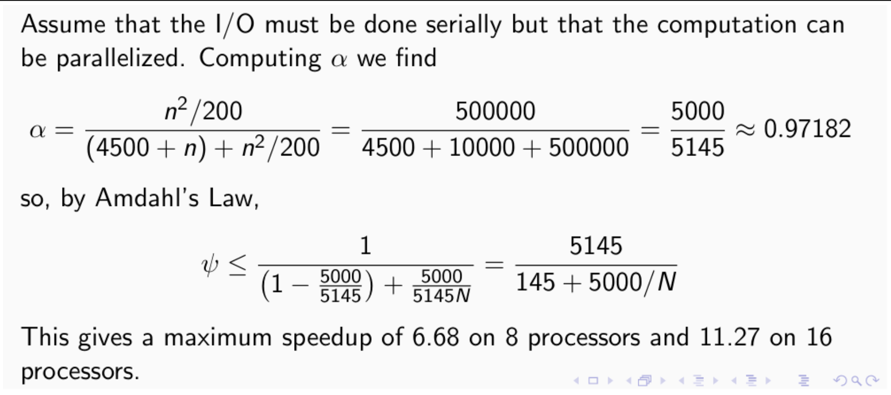
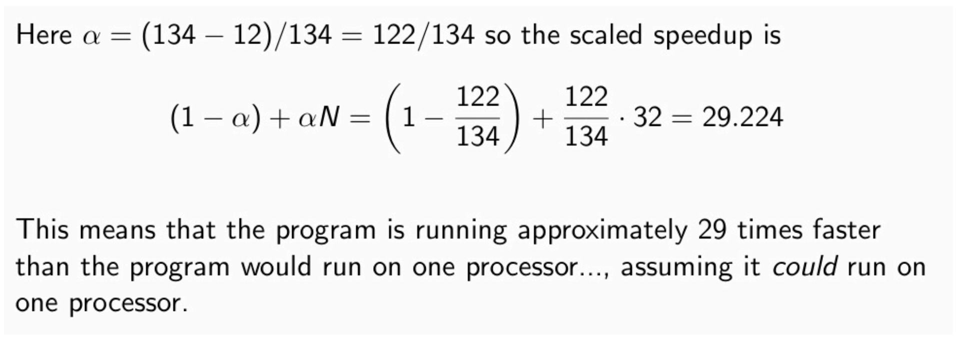
3
 	
Solution:
	Develop an MPI program for calculating the factorial of a number.
	```
	#include <stdio.h>
	#include <mpi.h>
	 int main(int argc, char *argv[])
	{
	         int myRank;
	         int size;
	         int fact;
	         int lower, upper;
	         int i;
	         double local_result = 1.0;
	         double total; //final factorial value
	```

```
        double start_time, end_time;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
        if (myRank == 0)
        {
      printf("Enter a number : ");
      scanf("%d", &fact);
        }
MPI_Bcast(&fact, 1, MPI_INT, 0, MPI_COMM_WORLD);
   if (myRank == 0)
        lower = 1;
        else
        lower = myRank * (fact / size) + 1;

        if (myRank == (size - 1))
        upper = fact;
        else
        upper = (myRank + 1) * (fact / size);
        for (i = lower; i <= upper; i++)
        {
        local_result = local_result * (double)i;
        }
MPI_Reduce(&local_result, &total, 1, MPI_DOUBLE, MPI_PROD, 0,
MPI_COMM_WORLD);
        printf("The factorial of %d : %lf \nCalculated using %d processes\n", fact, total,
size);
        MPI_Finalize();
        return 0;
}
```

Q.3.	**Attempt any one.**
1 **Solution:**	**State and Explain** Amdahl's Law. Suppose a serial program reads n data from a file, performs some computation and then writes n data back out to another file. The I/O time is measured and found to be 4500+n sec. If the computation portion takes $n^2/200$ µsec. **Apply** Amdahl's law to calculate the maximum speed up we can expect when n=10,000 and N processors are used.

	 Assume that the I/O must be done serially but that the computation can be parallelized. Computing α we find $\alpha = \frac{n^2/200}{(4500+n)+n^2/200} = \frac{500000}{4500+10000+500000} = \frac{5000}{5145} \approx 0.97182$ so, by Amdahl's Law, $\psi \le \frac{1}{(1-\frac{5000}{5145})+\frac{5000}{5145N}} = \frac{5145}{145+5000/N}$ This gives a maximum speedup of 6.68 on 8 processors and 11.27 on 16 processors.
2 **Solution:**	**State and Explain** Gustafson's Law. A parallel program takes 134 seconds to run on 32 processors. The total time spent in the sequential part of the program was 12 seconds. **Apply** Gustafson's law to calculate the scaled speedup.  Here $\alpha = (134-12)/134 = 122/134$ so the scaled speedup is $(1-\alpha)+\alpha N = \left(1-\frac{122}{134}\right)+\frac{122}{134}\cdot 32 = 29.224$ This means that the program is running approximately 29 times faster than the program would run on one processor..., assuming it *could* run on one processor.

********************All the best*****************