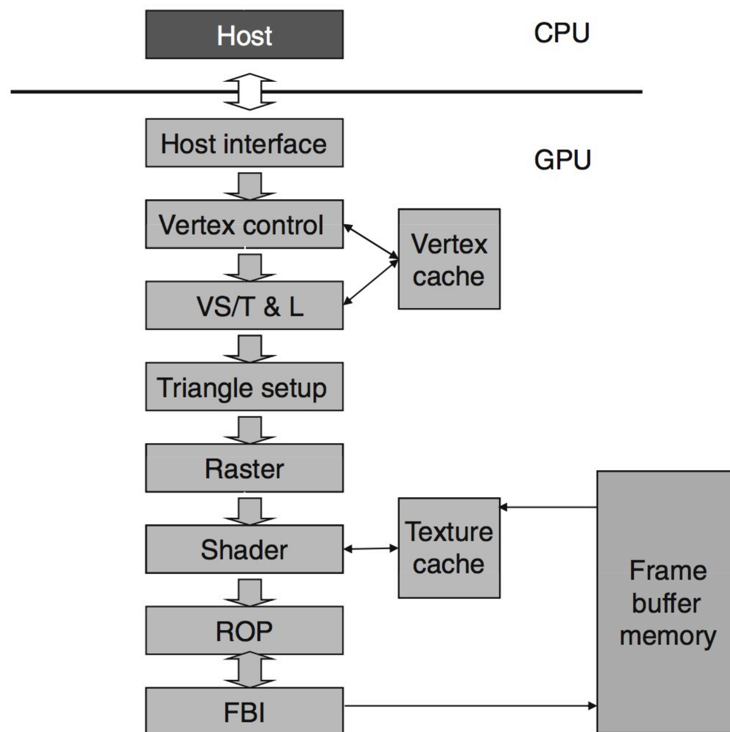
PCAP Assignment II

1. With a neat diagram, explain the various stages of fixed-function graphic pipeline.



The host interface receives graphics commands and data from the CPU. The commands are typically given by application programs by calling an API function.

The host interface typically contains a specialized direct memory access (DMA) hardware to efficiently transfer bulk data to and from the host system memory to the graphics pipeline.

The host interface also communicates back the status and result data of executing the commands.

The vertex control stage then converts the triangle data into a form that the hardware understands and places the prepared data into the vertex cache.

The vertex shading, transform, and lighting (VS/T&L) stage transforms vertices and assigns per-vertex values (e.g., colors, normals, texture coordinates, tangents)

The GeForce graphics pipeline is designed to render triangles. The finer the sizes of the triangles are, the better the quality of the picture typically becomes.

The shading is done by the pixel shader hard- ware. The vertex shader can assign a color to each vertex, but color is not applied to triangle pixels until later.

The raster operation (ROP) stage performs the final raster operations on the pixels. It performs color raster operations that blend the color of overlapping/adjacent objects for transparency and antialiasing effects. It also determines the visible objects for a given viewpoint and discards the occluded pixels.

Finally the frame buffer interface handles write to the frame buffer memory.

2.

A. **Write a MPI program to find out $1+(1\times2)+(1\times2\times3)+(1\times2\times3\times4)+\ldots\ldots\ldots\ldots\ldots+(1\times2\times3\times4\times\ldots\ldots\times n)$ using MPI_Scan and MPI_Reduce. Use n processes in your program.**

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

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

      int rank, size;
      int sum, prod;

      MPI_Init(&argc, &argv);

      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);

      int rank1 = rank + 1;

      // Scan and get the collective product of the numbers upto rank + 1

      MPI_Scan(&rank1, &prod, 1, MPI_INT, MPI_PROD, MPI_COMM_WORLD);

      // Reduce to get the sum of products

      MPI_Reduce(&prod, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

      if (rank == 0) {
            printf("The result is : %d\n", sum);
      }

      MPI_Finalize();

      return 0;
}
```

B. **Describe MPI_Alltoall() with appropriate example.**

MPI_Alltoall is a collective operation in which all processes send the same amount of data to each other, and receive the same amount of data from each other.

The operation of this routine can be represented as follows, where each process performs 2n (n being the number of processes in communicator *comm*) independent point-to-point communications (including communication with itself).

Example:

```
MPI_Comm_size(comm, &n);
   for (i = 0, i < n; i++)
       MPI_Send(sendbuf + i * sendcount * extent(sendtype),
           sendcount, sendtype, i, ..., comm);
   for (i = 0, i < n; i++)
       MPI_Recv(recvbuf + i * recvcount * extent(recvtype),
           recvcount, recvtype, i, ..., comm);
```

Each process breaks up its local *sendbuf* into n blocks - each containing *sendcount* elements of type *sendtype* - and divides its *recvbuf* similarly according to *recvcount* and *recvtype*.

Process j sends the k-th block of its local *sendbuf* to process k, which places the data in the j-th block of its local *recvbuf*. The amount of data sent must be equal to the amount of data received, pairwise, between every pair of processes

3.

A. **Write a program to read a value M and N*M number of elements in the root. Using N process 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. Use collective communication APIs.**

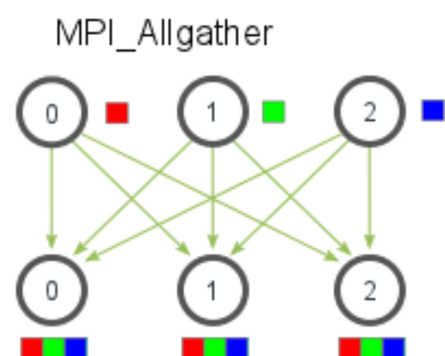
```
#include <stdio.h>
#include "mpi.h"

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

      int rank, size, arr[50], brr[10], crr[20], drr[10], m;

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

      if (rank == 0) {
            printf("Enter m: ");
            scanf("%d",&m);
            for(int i = 0; i < m * size; i++)
                  scanf("%d", &arr[i]);
      }

      MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Scatter(&arr, m, MPI_INT, &brr, m, MPI_INT, 0, MPI_COMM_WORLD);

      for (int i = 0; i < m; i++)
            drr[i]=brr[i];

      for (int i = 0; i < m; i++) {
            for(int j = 0; j < (rank + 1); j++) {
                  drr[i] = drr[i] * brr[i];
            }
      }

      MPI_Gather(&drr, m, MPI_INT, &crr, m, MPI_INT, 0, MPI_COMM_WORLD);

      if (rank == 0) {
            printf("ELEMENTS ARE: ");
            for(int i = 0; i < size * m; i++)
                  printf("%d ",crr[i]);
      }

      MPI_Finalize();
      return 0;
}
```

B. **Describe MPI_Allgather() with appropriate example.**

Given a set of elements distributed across all processes, MPI_Allgather will gather all of the elements to all the processes.

In the most basic sense, MPI_Allgather is an MPI_Gather followed by an MPI_Bcast.

Just like MPI_Gather, the elements from each process are gathered in order of their rank, except this time the elements are gathered to all processes.



4.

A. **Write an 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.**

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

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

      int rank, size;
      float area, area1;
      float x, y;

      MPI_Init(&a, &b);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);

      x = (float)rank/size;
      y = 4 / (1 + x * x);
      area = (1 / (float)size) * y;

      MPI_Reduce(&area, &area1, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

      if(rank == 0) {
            printf("The value of PI is: %f\n", area1);
      }

      MPI_Finalize();
      return 0;
}
```

B. **What is the use of MPI_Errhandler_set(). Discuss how to use it.**

MPI_Errhandler_set - Sets the error handler for a communicator. (PS. use of this routine is deprecated.)

Associates the new error handler errhandler with communicator comm at the calling process. Note that an error handler is always associated with the communicator.

The predefined error handler is called MPI_ERRORS_ARE_FATAL which has an inbuilt routine to abort the whole parallel program as soon as any error is detected.

There's another predefined error handler called MPI_ERRORS_RETURN, which can be set by using the function call;

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

The only error code that MPI defines is MPI_SUCCESS; i.e. no error.

5.

A. **Discuss the clGetPlatformIDs(), clGetDeviceIDs() functions of openCL.**

The API function clGetPlatformIDs() is used to discover the set of available platforms for a given system. The API function clGetDeviceIDs() is used to discover the set of available devices for a platform and device type.

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

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

clGetPlatformIDs() will often be called twice by an application.

The first call passes an unsigned int pointer as the num_platforms argument and NULL is passed as the platforms argument. The pointer is populated with the available number of platforms. The programmer can then allocate space to hold the platform information.

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

B.

i. **Define OpenCL kernel with an example code snippet.**

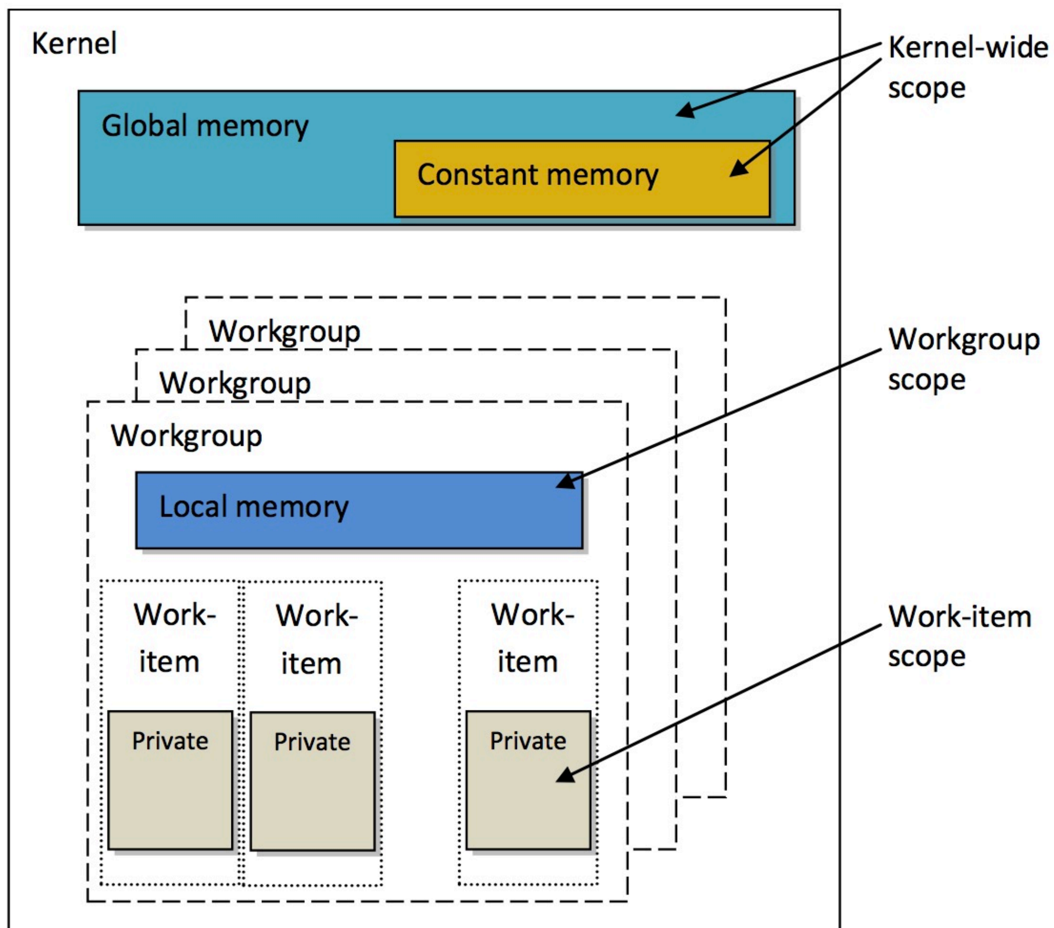
OpenCL C kernels are similar to C functions and can be thought of as instances of a parallel map operation. The function body, like the mapped function, will be executed once for every work-item created.

Kernels begin with the keyword __kernel and must have a return type of void. The argument list is as for a C function with the additional requirement that the address space of any pointer must be specified.

Example:

```
__kernel void vector_add (
      __global int *A,
      __global int *B,
      __global int *C) {

            int i = get_global_id(0);

            C[i] = A[i] + B[i];

}
```

ii. **With a neat diagram, explain the abstract memory model of OpenCL.**



OpenCL defines an abstract memory model that programmers can target when writing code and vendors can map to their actual memory hardware.

Global memory is visible to all compute units on the device. Whenever data is transferred from the host to the device, the data will reside in global memory. Any data that is to be transferred back from the device to the host must also reside in global memory.

Constant memory is not specifically designed for every type of read-only data but, rather, for data where each element is accessed simultaneously by all work-items. *(Kernel-wide scope)*

Local memory is a scratchpad memory whose address space is unique to each compute device. *(Work-group scope)*

Private memory is memory that is unique to an individual work-item. Local variables and nonpointer kernel arguments are private by default. *(Work-item scope)*

Special thanks to Ninad Shenoy for providing the solutions.