PCAP Assignment I

1.

A. **Why is there a large performance gap between many-core GPUs and general-purpose multicore CPUs. Discuss in detail.**

The multicore CPUs are designed to maximize the execution speed of sequential programs; whereas many-core GPUs focus more on the execution throughput of parallel applications.

GPUs have hundreds of cores each of which is a heavily multithreaded, in-order, single-instruction issue processor that shares its control and instruction cache with several other cores. Many-core processors, especially the GPUs, have led the race of floating-point performance.

The ratio between many-core GPUs and multicore CPUs for peak floating-point calculation throughput is about 20 to 1. This large performance gap has already motivated many applications developers to move the computationally intensive parts of their software to GPUs for execution.

B. **Write an MPI program to read a number N in root process. Broadcast this number to many other slaves. Calculate sum in each process (including root) up to N. Further the root process collects their computed sum to give the final sum (including its sum) and print final sum in root process.**

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

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

      int size, rank;
      MPI_Status status;

      MPI_Init(&argc, &argv);

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

      if (rank == 0) {

            int k;

            fprintf(stdout, "%d. Enter a number : ", rank);
            scanf_s("%d", &k, 1);

            int i;
            for (i = 1; i < size; ++i) {
                  MPI_Send(&k, 1, MPI_INT, i, i, MPI_COMM_WORLD);
            }

            int sum = 0;
            for (i = 1; i <= k; ++i) {
                  sum += i;
            }

            for (i = 1; i < size; ++i) {
                  int ssum;
                  MPI_Recv(&ssum, 1, MPI_INT, i, i, MPI_COMM_WORLD, &status);
                  sum += ssum;
            }

            fprintf(stdout, "Final sum = %d.\n", sum);
```

```
        } else {

                int k;
                MPI_Recv(&k, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);

                int i;
                int sum = 0;
                for (i = 1; i <= k; ++i) {
                        sum += i;
                }
                MPI_Send(&sum, 1, MPI_INT, 0, rank, MPI_COMM_WORLD);

        }

        MPI_Finalize();
}
```

2.

A. Explain the architecture of modern GPU with diagram.

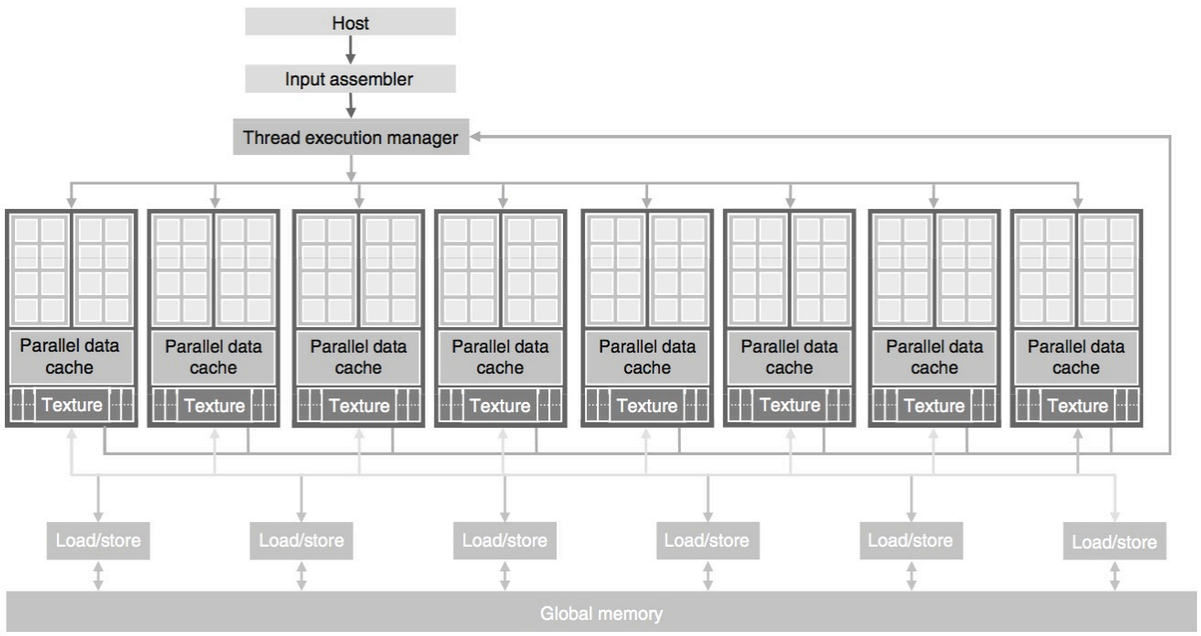
A modern GPU is organized into an array of highly threaded streaming multiprocessors (SMs).

GPUs are high latency high throughput processors. GPUs are designed for tasks that can tolerate latency.

GPUs can have more ALUs for the same sized chip and therefore run many more threads of computation. (Modern GPUs run 10,000s of threads concurrently). Threads are managed and scheduled by the hardware.

GPUs use data parallelism; SIMD (Single Instruction Multiple Data) streams. Same instruction is run on multiple data streams.

GPUs have higher memory bandwidth, and therefore are suited for display devices.



B. Explain and compare message passing programming with shared memory programming. Discuss on MPI Environment Management Routines with a general structure of MPI program.

In message passing, a set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines. Tasks exchange data through communications by sending and receiving messages. Data transfer usually requires cooperative operations to be performed by each process.

In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously. Various mechanisms such as locks / semaphores may be used to control access to the shared memory.

———————

`MPI_Init` initializes the MPI execution environment, must be called before any other MPI routine is called, and is invoked only once in an MPI program.

`MPI_Finalize` terminates the MPI execution environment and must be called at the end.

`MPI_Comm_size` determines the number of processes that are associated with a communicator. `MPI_Comm_rank` determines the number of processes within a communicator. `MPI_Wtime` is a timer routine

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

int main () {

      MPI_Init(NULL, NULL);

      // Do stuff.

      MPI_Finalize();
      return 0;
}
```

3.

A. **Do a detailed discussion on why applications will continue to demand increased speed?**

// Something to do with the need of more and more processing power.
// We need more speed while processing 3D models of DNA, Artificial Intelligence, Weather Forecast, Predictive analysis, Virtual Reality, 8K/60p gaming etc...

B. **Explain the varieties in blocking operations on MPI. Discuss any one of them with the program structure when the deadlock may or may not occur.**

MPI has a blocking send (`MPI_Send`) and a blocking receive (`MPI_Recv`), and blocking send and receive (`MPI_SendRecv`) operations.

The following code will deadlock.
Process with rank 0 and 1 cannot send data until they receive from the other.

```
// . . .
if (rank == 0) {
     MPI_Recv(&inmsg, 1, MPI_CHAR, 1, tag,
                                        MPI_COMM_WORLD, &Stat);

     MPI_Send(&outmsg, 1, MPI_CHAR, 1, tag, MPI_COMM_WORLD);

} else if (rank == 1) {

     MPI_Recv(&inmsg, 1, MPI_CHAR, 0, tag,
                                        MPI_COMM_WORLD, &Stat);

     MPI_Send(&outmsg, 1, MPI_CHAR, 0, tag, MPI_COMM_WORLD);
}
// . . .
```

Swap one Send and Recv operation and we avoid the deadlock.

4.

A. **What is a communicator in MPI? Discuss on Getting Communicator Information such as Rank and Size.**

A communicator is an opaque object that provides the environment for message passing among processes. The default communicator provided by the MPI is MPI_COMM_WORLD.

Process within the communicator are ordered. The rank a process is its position in the overall order. In a communicator with p processes, each process has a unique rank (10 number) between 0 and p - 1. A process may use its rank to determine which portion of a computation and/or a dataset it is responsible for.

A process calls function MPI_Comm_rank to determine its rank within Ii communicator. It calls MPI_Comm_size to determine the total number of processes in a communicator.

B. Write an MPI program using synchronous send. The sender process sends a message to the receiver. This receiver receives the message and sends it back.

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

#define SIZE sizeof(int)

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

      int size, rank;
      MPI_Status status;

      MPI_Init(&argc, &argv);

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

      char message[128];
      int len;

      int i;

      if (rank == 0) {

            message = "SEND HELP!";
            len = strlen(message);

            for (i = 1; i < size; ++i) {
                  MPI_Ssend(&len, 1, MPI_INT, i, i, MPI_COMM_WORLD);
                  MPI_Ssend(message, strlen(message), MPI_CHAR, i, size + i,
MPI_COMM_WORLD);
            }

      } else {

            MPI_Recv(&len, 1, MPI_INT, 0, rank, MPI_COMM_WORLD, &status);
            MPI_Recv(message, len, MPI_CHAR, 0, size + rank, MPI_COMM_WORLD,
&status);

            MPI_Ssend(message, len, MPI_CHAR, 0, 2 * size + rank,
MPI_COMM_WORLD, &status);

      }

      MPI_Finalize();
}
```

5.

A. Discuss the APIs with their arguments involved with point to point communication.

```
int MPI_Send ( void *buffer,
               int count,
               MPI_Datatype datatype,
               int dest,
               int tag,
               MPI_Comm comm )
```

- **buf** The data that is sent
- **count** Number of elements in buffer
- **datatype** Type of each element in buf (see later slides)
- **dest** The rank of the receiver
- **tag** An integer identifying the message
- **comm** Communicator
- *error*

```
int MPI_Recv ( void *buf,
               int count,
               MPI_Datatype datatype,
               int source,
               int tag,
               MPI_Comm comm,
               MPI_Status *status )
```

- **buf** Buffer for storing received data.
- **count** Number of elements in buffer, not the number of elements that are actually received.
- **datatype** Type of each element in buf
- **source** Sender of the message.
- **tag** Number identifying the message.
- **comm** Communicator
- **status** Information on the received message
- *error* The function returns an error value

B. Write an MPI program to read n*m elements in root process where n is the number processes and m is the number of elements expected to be scattered to each slaves including root. Send m elements to corresponding process using scatter.

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

using namespace std;

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

      int size, rank;
      MPI_Init(&argc, &argv);

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

      int m, n, i;
      n = size;

      int a[100],temp[100];

      if (rank == 0) {

            fprintf(stdout, "Enter m: \n");
            fflush(stdout);
            cin >> m;

            fprintf(stdout, "Enter n * m numbers: \n");
            for (i = 0; i < n*m; ++i) {
                  cin >> a[i];
            }
      }

      MPI_Bcast(&m, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Scatter(a, m, MPI_INT, temp, m, MPI_INT, 0, MPI_COMM_WORLD);

      fprintf(stdout, "Process %d got %d elements: ",rank, m);

      for(i = 0; i < m; i++) {
            cout << temp[i] << "\t";
      }


      MPI_Finalize();

      return 0;

}
```

EDIT: Updated with correct solutions.
Thanks Ishan.