MANIPAL INSTITUTE OF TECHNOLOGY

BENGALURU

(A constituent unit of MAHE, Manipal)

SESSIONAL RECORD DEPT.: COMPUTER SCIENCE ENGG

NAME: RUTVIK AVINASH BARBHAI REG.NO.: 225905222

SEMESTER: VI SECTION: CSE-CORE-A ROLL NO.: 39

SUBJECT: PARALLEL COMPUTING & ARCHITECTURE & PROGRAMMING

DATE OF TESTS: 9 | APRIL | 2025 MAX MARKS : 25/30

MARKS OF: I TEST II TEST

MARKS FOR VALUATION - I TEST							MARKS FOR VALUATION - II TEST						
Q. No.	FOR THE TEACHER TO AWARD MARKS					TOTAL MARKS	Q. No.	FOR THE TEACHER TO AWARD MARKS					TOTAL MARKS
	a	b	c	d	e			a	b	c	d	e	
1							1						
2							2						
3							3						
4							4						
5							5						
6							6						
TOTAL IN FIGURES							TOTAL IN FIGURES						

IN WORDS _____ IN WORDS _____

Signature of the Candidate Signature of the Invigilator Signature of the Teacher

PARALLEL COMPUTING & ARCHITECTURE & PROGRAMMING ASSIGNMENT

Topic: MCQS

Q 1) Given the code

```
#include <mpi.h>
int main (int argc, char** argv) {
 int rank, data[4], result;
 MPI_Init (&argc, &argv);
 MPI_comm_rank (MPI_COMM_WORLD, &rank);
 MPI_Reduce (data, &result, 4, MPI_INT,
 MPI_SUM, 0, MPI_COMM_WORLD);
      if (rank ==0) {
 // some code
 }
 MPI_Finalize();
 return 0;
 }
```

if each process has data initialized to [1,2,3,4] what will be the value of result on rank 0 after MPI_Reduce ?

A (1) option b) 10

Q 2) Given code

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char** argv) {
 int rank, size;
 int recvbuf[2];
```

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// Assume size is 4
:
if (rank == 0) {
printf("Process 0 gathered: ");
for (int i=0; i<8; i++) {
  printf("%d ", sendbuf[i]);
} printf("\n"); } MPI_Finalize();
return(0);
```

When run with 4 process, which of the following statements is true about output?

A 2) option d) Process 0 will print "Process 0 gathered: 1 2 3 4 5 6 7 8

Q 3) Identify deadlock condition in below code:-

```
int a, b, c;
int rank;
MPI_Status status
if (rank == 0) {
MPI_Send(&a, 1, MPI_INT, 2, 1, MPI_COMM_WORLD);
MPI_Recv(&b, 1, MPI_INT, 2, 1, MPI_COMM_WORLD, &status);
c=a+b/2; } else if (rank == 1) {
MPI_Send(&b, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Recv(&a, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
c=a+b/2 }
```

which of the following deadlock condition arises in this code?

A 3) option b) Rank mismatch

Q 4) Which of the following is a valid openCL API function specifically for buffer management

A 4) option a) clCreateBuffer()

Q 5) What does the CL_TRUE argument in the clEnqueueReadBuffer function specify?

cl_event waitEvent;

clEnqueueReadBuffer(queue, memBuffer, CL_TRUE, 0, dataSize, data, 0, NULL, &waitEvent);

A 5) option d) The operation is blocking, meaning it will wait until the data has been read into data before returning.

Q 6) Which of the following statements is true about OpenCL work-items & work-groups?

A 6) option a) Work-items execute the kernel code

Q 7) How would you declare a kernel function named my_kernel that takes a single argument, a pointer to an array of floats in global memory?

A 7) option b) __kernel void my_kernel(__global float* data)

Q 8) What is the primary purpose of cudaMemcpyToSymbol Function in CUDA?

A 8) option b) To copy data from host memory to a global or constant symbol (variable) on device.

Q 9) consider the following code

```
__global__ void kernel(int *data, int size) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < size) {
    atomicAdd(&data[0], data[tid]);
  }
}
```

What is the primary purpose of the atomicAdd() function used in the following CUDA kernel?

A 9) option c) To prevent data corruption when multiple threads try to update data[0] simultaneously.

Q 10) Which of the following is a common method to handle boundary conditions in tiled 1D convolution?

A 10) option a) Padding the input data with zero.

Q11a) Write MPI program to read matrix A of size 4x4. It produces a resultant matrix RES of size 4x4 by adding every row elements of A with a key value. The key value for first row is the product of elements in the last row and the key value for remaining row is the product of the elements from the previous row of matrix A. Uses 4 processes (including root) to solve this problem. Use only collective communication routines

sample I/O

A				Res			
1	1	1	1	5	5	5	5
2	2	2	2	3	3	3	3
3	3	3	3	19	19	19	19
1	2	1	2	82	83	82	83

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

#define SIZE 4

int main(int argc, char **argv) {
    int rank, size;
    int A[SIZE][SIZE];
    int RES[SIZE][SIZE];
    int key;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,
    &rank);
MPI_Comm_size(MPI_COMM_WORLD,
    &size);
```

```
if (size != 4) {
    if (rank == 0) {
fprintf (stderr, "This program must be run
        with 4 processes\n");
    }
  MPI_Abort (MPI_COMM_WORLD, 1);
}
if (rank == 0) {
    printf ("A\n");
  for (int i = 0 ; i < SIZE ; i++) {
      for (int j = 0 ; j < SIZE ; j++) {
          A[i][j] = (i+1) * (j+1) ; //sample matrix
                                      initialization
      if (i == 3) {
      A[i][j] = (j % 2 == 0) ? 1 : 2;
        }
     printf ("%d ", A[i][j]);
      }
  printf ("%d ", A "\n");
    }
  }
MPI_Bcast (A, SIZE * SIZE, MPI_INT, 0, MPI_
  COMM_WORLD);
    for (int i = 0 ; i < SIZE ; i++) {
        if (rank == 0) {
            key = 1;
          if (i == 0) {
            for (int j = 0 ; j < SIZE ; j++) {
                key *= A[i-1][j];
            }
          }
        }
  MPI_Bcast (&key, 1, MPI_INT, 0, MPI_COMM_
      WORLD);
```

```
for (int j=0 ; j < SIZE ; j++) {
        RES[i][j] = A[i][j] + key
    }
}
if(rank==0) {
printF("\n Res\n");
    for(int i=0; i<SIZE; i++){
        for (int j=0 ; j<SIZE ; j++) {
            printF("%d ", RES[i][j]);
        }
    printf("\n");
    }
}
MPI_Finalize();
  return(0);
}
```

Q11 b) What are the different ways to create a program object in openCL, and how are they used? Write an openCL kernel that performs matrix-vector multiplication. Represent the matrix in row-major order.

A11 b) create an openCL program:

- openCL ccode (written to run an open CL device) is called a program.
- A program is a collection of functions called kernels, which kernels are units of execution that can be scheduled to run on a device.

openCL programs are compiled at runtime through a series of API calls. There is no need for an openCL application to have been prebuilt against the AMD, NVIDIA, or Intel runtimes, for example, if it is to run on device produced by all these vendors.

- This runtime compilation gives the system an opportunity to optimize for a specific device.
- openCL software link only to common runtime layer (called ICD); all platform specific spk activity is delegated to a vendor runtime through dynamic library interface.
- The process of creating a kernel is as follows:

1) The OpenCL ~~software links~~ only a source code is stored in a character string. If the source code is stored in a file on disk, it must be read into memory and stored as a character array

2) The source code is turned into a program, object, cl_program, by calling clcreateProgram WithSource().

3) The program object is then compiled, for one or more openCL device, with clBuidProgram(). If there are compile errors, they will be ~~repro~~ reported here.

Kernel code

```
__kernel void matrix_vector_mult(__global
float *matrix, __global float *vector,
__global float *result, int rows, int cols) {

    int row = get_global_id(0);

if (row < rows) {
    float sum = 0.0f;
    for(int col=0; col<cols; col++) {
        sum += matrix[row*cols+col]*vector
                                    [col]
    }
```

result[row] = sum;
}
}

Q11 c) What are the two major shortcomings of parallel spMV/CSR kernel, despite its simplicity? Write the CSR format for the following sparse matrix. Also develop CUDA kernel code for parallel spMV using CSR.

5	0	0	1
0	0	2	0
0	3	0	0
4	0	0	0

A11 c) The simple spMV (sparse Matrix – Vector – Multiplication) kernel using the CSR (compressed sparse Row) format, while easy to understand, suffers from two major shortcomings-

1. First the kernel does not make ~~ea~~ coalesced (Formal) memory accesses. For figure it should be obvious that adjacent threads will be making simultaneous non adjacent memory accesses. In our small example thread 0, 1, 2 and 3 will access data[0], none, data[2], and data[5] in first iteration of their dot product loop. They will then access data[1], none, data[3], and data[6] in second iterations, and so on. It is obvious that these simultaneous accesses made by adjacent threads are not to adjacent locations. As a result, the parallel spMV/CSR kernel does not make efficient use of memory bandwidth.

2. The second shortcoming of the spMV/CSR kernel is that it can potentially have significant control flow divergence in all warps.

The number of iterations taken by a thread in the dot product loop depends on the number of nonzero elements in the row assigned to the thread. Since the distribution of nonzero elements among rows can be random, adjacent rows can have a very different number of nonzero elements. As a result, there can be widespread control flow divergence in most or even all warps.

CSR Format For given Sparse Matrix

The given sparse matrix

$$\begin{matrix} 5 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 0 & 0 \end{matrix}$$

The CSR format consist of three arrays:

1. val (values): stores the non-zero values.
2. col (column indices): store the column index of each non-zero element.
3. rowPtr (row pointers): stores the starting index of each row in the val and col arrays.

Here's the CSR representation

val : [5, 1, 2, 3, 4]

col : [0, 3, 2, 1, 0]

rowPtr : [0, 2, 3, 4, 5]

Explanation of rowPtr

rowPtr[0] = 0 : First row starts at index 0 in val and col.

rowPtr[1] = 2 : The second row starts at index 2.

rowPtr[2] = 3 : The third row starts at index 3.

rowPtr[3] = 4 : The fourth row start at index 4.

rowPtr[4] = 5 : rowPtr[4] is the total number of non-zero elements, used to determine the end of last row

1. ```
__global__ void SpMV_CSR(int num_rows, float *data, int *col_index, int *row_ptr, float *x, float *y) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < num_rows) {
        float dot = 0;
        int row_start = row_ptr[row];
        int row_end = row_ptr[row+1];
        for (int elem = row_start; elem < row_end; elem++) {
            dot += data[elem] * x[col_index[elem]];
        }
        y[row] = dot;
    }
}
```
```

Q12 a) Explain how constant memory can be used to increase the performance of the massively parallel processors? Write the kernel function for 1D convolution using constant memory in CUDA.

A 12 a) Like global memory variables, constant memory variables are also located in DRAM.

However, because the CUDA runtime knows that constant memory variables are not modified during kernel execution, it directs the hardware to aggressively cache the constant memory variables during kernel execution.

- To mitigate the effect of memory bottleneck, modern processors commonly employ on-chip cache memories, or caches, to reduce the number of variables that need to be accessed From DRAM.
- A major design issue with using cache in a massively parallel processor is cache coherence, which arises when one or more processor cores modify cached data.
- Since L1 caches are typically direct attached to one of the processor cores, changes in its contents are not easily observed by other processor cores. This causes a problem if the modified variable is shared among threads running on different processor cores.

We can make three interesting observations about the way the mask array M is used in convolution:

1. First, the size of the M array is typically small.
2. Second, the contents of M are not changed throughout the execution of the kernel.
3. Third all threads need to access the mask elements. Even better, all threads access the M elements in the same order, starting from M[0]. and move by one element a time through the iterations of the for loop in 1D parallel convolution.

These properties make the mask array an excellent candidate for constant memory and caching

code

```
__global__ void convolution_1D_basic_kernel(
float *N, float *P, int Mask_Width, int
Width) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
    for (int j = 0; j < Mask_Width; j++) {
  if (N_start_point + j >= 0 && N_start_point + j <
    Width) {
    Pvalue += N[N_start_point + j] * M[j];
  }
  P[i] = Pvalue;
}
```

Q12 b) What are two types of barrier operations for a command queue in openCL. Write an OpenCL kernel code that takes a string S as input and one integer value N. The program produces the output string N times as shown in parallel:

I/p S = Hello N = 3

O/p string : HelloHelloHello

Note : Each work item copies entire S

Two types of barrier operations for a command queue

✓ cl_int clFinish (cl_command_queue cmd queue);

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

This means that the commands will definetly be in-flight but will not necessarily have completed

Code

```
__kernel void repeatString(__global char *
inputString, int inputStringLength, int
repeatCount, __global char *outputString){
    int globalId = get_global_id(0);
    if (globalId < repeatCount){
        for (int i=0; i<inputStringLength;
            i++){
        outputString[globalId *inputStringLength+i]
            =inputString[i];
}}
```

Q12c) Describe the steps involved in using MPI_Bsend for sending a message, including buffer management.

Ans c)

```
MPI_Bsend (void *message,
           int count,
       MPI_Datatype datatype,
        int dest ,
        int tag,
        MPI_comm comm
```

- This routine permits the programmer to allocate the required amount of buffer space into which data can be copied until it is delivered.
- Insulates against the problems associated with insufficient system buffer space.
- Routine returns after the data has been copied from application buffer space to allocated send buffer.
- It must be used with the MPI_Buffer_attach() and MPI_Buffer_detach() routines :-

1 marks

MPI_Buffer_attach

MPI_Buffer_detach

```
MPI_Buffer_attach (void * buffer, int size)
MPI_Buffer_deatach (void * buffer,
                    int *size
```

used by programmes to allocate/deallocate message buffer space to be used by the MPI_Bsend() routine

The size argument is specified in actual data bytes :- not a count of data elements.

only one buffer can be attached to a process at a time.