MANIPAL ACADEMY OF HIGHER EDUCATION

B.Tech VIth Semester Mid semester Examination March 2024

PARALLEL PROGRAMMING SCHEME [DSE 3254]

1) You're working on a real-time gray-scale image processing application. The processing involves a pre-processing step where your are supposed to determine the Cube of every pixel and then divide it by the total number of pixels in the given input image. Design a CUDA program to do the above task parallelly. Assume that the image dimension is 780x650. Also, consider that the streaming multi-processor can take maximum up to 2048 threads and it allows up to 1024 threads in each block. Assume that the Image is already read into variable I and size of the image is already estimated and stored in variable SIZE_I. (4 marks)

Solution:

```
__global__ void picturekernel(float* d_in, float* d_out, int n, int m)
{
int row=blockIdx.y * blockDim.y + threadIdx.y;
int col=blockIdx.x * blockDim.x + threadIdx.x;

if ((row<m) && (col <n))
{
d_out[row*n+col]=(d_in[row*n+col]* d_in[row*n+col]* d_in[row*n+col])/(n*m);
}
}

Int main()
{
        Int n=780, m=650
        dim3 dimBlock(32 , 32, 1);
        dim3 dimGrid( n/32,m/32 ,1 );
        #assumed that the data is read into variable I with size =SIZE_I
        cudaMalloc((void**)&d_in,SIZE_I);
        cudaMemcpy(d_in, h_in, SIZE_I, cudaMemcpyHostToDevice);

        cudaMalloc((void**) &d_out, SIZE_I)

        picturekernel<<<dimGrid, dimBlock>>>(d_in, d_out, n, m);

        cudaMemcpy(h_out, d_out, SIZE_I, cudaMemcpyDeviceToHost);
}
```

2) Design an MPI program to read an integer value in the root process. Root process sends this value to process 1, process 1 sends this value to process 2 and so on. Last process sends this value back to root process. When sending the value each process will first decrement the

received value by one. Desing the program using point to point communication routines. (4 marks)

Solution:

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

int main(int argc, char** argv) {
   int rank, size, value, tag = 0;

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

   if (rank == 0) {
      // Root process: Read initial value and send to process 1
      printf("Enter an integer value: ");
      scanf("%d", &value);
      MPI_Send(&value, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
   } else if (rank == size - 1) {
      // Last process: Receive, decrement, and send back to root
      MPI_Recv(&value, 1, MPI_INT, rank - 1, tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      value--;
      MPI_Send(&value, 1, MPI_INT, 0, tag, MPI_COMM_WORLD);
   } else {
      // Intermediate processes: Receive, decrement, and send to next process
      MPI_Recv(&value, 1, MPI_INT, rank - 1, tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      value--;
      MPI_Send(&value, 1, MPI_INT, rank + 1, tag, MPI_COMM_WORLD);
   }

   // Root process: Receive back the final value
   if (rank == 0) {
      MPI_Recv(&value, 1, MPI_INT, size - 1, tag, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Final value received by root process: %d\n", value);
   }

   MPI_Finalize();
   return 0;
}
```

3) Parallelize the following code using openMP pragmas. Be sure to explicitly specify the iterations to be divided into chunks of size C_SIZE which is based on N. For each rewrite the code. If necessary you can assume that the variable P represents the number of processors to be used. Assume that N is large (in the tens of thousands or more). You must explicitly list all

variables within the range of a parallel pragma that are private using the private() directive. (1.5+1.5= 3 Marks)

a)
```
for (i=0;i<N;i++){
    for (j=0;j<N;j++){
        A[i,j] = max(A[i,j],B[i,j]);
    }
}
```

Solution:

```
#pragma omp parallel for private(j) schedule(static,N/P)
for (i=0;i<N;i++){
 for (j=0;j<N;j++){
 A[i,j] = max(A[i,j],B[i,j]);
 }
}
```

b)
```
C[0] = 1;
for (i=1;i<N;i++){
    C[i] = C[i-1];
    for (j=0;j<N;j++){
        C[i] *= A[i,j] + B[i,j];
    }
}
```
(3 marks)

Solution:

```
C[i] = 1;
for (i=1;i<N;i++){
 C[i] = C[i-1];
#pragma omp parallel for schedule(static,N/P)
reduction(*:product)
 for (j=0;j<N;j++){
 product *= A[i,j] + B[i,j];
 }
 C[i] = product;
```

4) In the following code, one process sets array A and then uses it to update B; the other process sets array B and then uses it to update A. Argue that this code can deadlock. How could you fix this?

```
#pragma omp parallel shared(a, b, nthreads, locka, lockb)
#pragma omp sections nowait
{
#pragma omp section
{
omp_set_lock(&locka);
for (i=0; i<N; i++)
a[i] = ..
```

```
omp_set_lock(&lockb);
for (i=0; i<N; i++)
b[i] = .. a[i] ..
omp_unset_lock(&lockb);
omp_unset_lock(&locka);
}
#pragma omp section
{
omp_set_lock(&lockb);
for (i=0; i<N; i++)
b[i] = ...
omp_set_lock(&locka);
for (i=0; i<N; i++)
a[i] = .. b[i] ..
omp_unset_lock(&locka);
omp_unset_lock(&lockb);
}
}
} (3 marks)
```

Solution: 1.5+1.5 = 3

The code creates a potential deadlock due to the way locks locka and lockb are acquired and released. Here's the scenario:

Section 1 starts:

- Acquires locka.
- Updates a.

Section 2 starts:

- Acquires lockb.
- Updates b.

Now, both sections try to acquire the other lock:

- Section 1 attempts lockb:
- Section 2 attempts locka:

Deadlock:

- Section 1 holds locka and waits for lockb.
- Section 2 holds lockb and waits for locka.

Neither can proceed as they're both waiting for a lock held by the other.

```
#pragma omp section
{
omp_set_lock(&locka);
omp_set_lock(&lockb);  // Acquire locks in consistent order
// Update a and b
omp_unset_lock(&lockb);
omp_unset_lock(&locka);
}
```

```
#pragma omp section
{
omp_set_lock(&locka); // Same lock acquisition order
omp_set_lock(&lockb);
// Update a and b
omp_unset_lock(&lockb);
omp_unset_lock(&locka);
}
```

5) You're developing a parallel image processing application that filters large images. The application follows these steps:
    a. Image Loading: Each thread loads a specific tile (sub-section) of the image into its local memory.
    b. Filtering: Each thread applies a filter (e.g., blur, sharpen) to its assigned tile independently.
    c. Result Accumulation: The filtered tiles need to be combined to form the final filtered image. However, this requires ensuring all filtering is complete before any thread starts modifying the final image.

    At what point in the code would you strategically place the OpenMP Barrier directive to achieve proper synchronization and avoid race conditions? Explain your reasoning. Briefly discuss potential performance implications of using the Barrier directive in this scenario. Are there any alternative synchronization mechanisms you might consider (if applicable)? (3 marks)

    Solution:
    Optimal Barrier Placement: The Barrier directive should be placed after all threads finish filtering their assigned tiles (end of step 2). This guarantees that no thread attempts to modify the final image before all filtering is complete, preventing inconsistencies.
    Performance Considerations:
    Benefit: The Barrier ensures data consistency but introduces a synchronization overhead.
    Drawback: If filter times vary significantly across threads, faster threads might wait for slower ones at the barrier, impacting performance.
    Conditional synchronization (e.g., atomic operations or locks) could be explored if filtering has minimal data dependency between tiles. However, this might add complexity and might not be suitable for all filtering algorithms.

6) You're working on a physics simulation that involves calculating force exerted by large number of particles (N). Each particle has acceleration and mass. Assume that the acceleration and mass of all the particles are stored in two vectors ACC and MASS. The force exerted by each particle is calculated using the following formula: F=MA where F is force, M is mass and A is acceleration. Implement a CUDA program that efficiently calculates the forces exerted on all individual particles in parallel. Assume that, N=770 and a block can have 256 threads. (3 marks)

    Solution:

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

```
#define N 770
#define THREADS_PER_BLOCK 256

__global__ void calculateForces(float *forces, const float *accelerations, const float *masses)
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x;

   if (idx < N) {
      forces[idx] = masses[idx] * accelerations[idx];
   }
}

int main() {
   // Host arrays
   float *h_forces, *h_accelerations, *h_masses;

   // Device arrays
   float *d_forces, *d_accelerations, *d_masses;

   // Allocate memory on host
   h_forces = (float*)malloc(N * sizeof(float));
   h_accelerations = (float*)malloc(N * sizeof(float));
   h_masses = (float*)malloc(N * sizeof(float));

   // Initialize data on host
   // You can initialize accelerations and masses here

   // Allocate memory on device
   cudaMalloc(&d_forces, N * sizeof(float));
   cudaMalloc(&d_accelerations, N * sizeof(float));
   cudaMalloc(&d_masses, N * sizeof(float));

   // Copy data from host to device
   cudaMemcpy(d_accelerations, h_accelerations, N * sizeof(float),
cudaMemcpyHostToDevice);
   cudaMemcpy(d_masses, h_masses, N * sizeof(float), cudaMemcpyHostToDevice);

   // Calculate number of blocks needed
   int numBlocks = (N + THREADS_PER_BLOCK - 1) / THREADS_PER_BLOCK;

   // Launch kernel
   calculateForces<<<numBlocks,      THREADS_PER_BLOCK>>>(d_forces,      d_accelerations,
d_masses);

   // Copy result back to host
   cudaMemcpy(h_forces, d_forces, N * sizeof(float), cudaMemcpyDeviceToHost);
```

```
    // Free device memory
    cudaFree(d_forces);
    cudaFree(d_accelerations);
    cudaFree(d_masses);

    // Free host memory
    free(h_forces);
    free(h_accelerations);
    free(h_masses);

    return 0;
}
```

7) Compare and contrast MPI_BCAST and MPI_SCATTER operations with suitable examples. (3 marks)

Solution:

MPI_BCAST and MPI_SCATTER are two communication operations provided by the Message Passing Interface (MPI) library for parallel computing. While both operations involve data communication among processes in a parallel program, they serve different purposes and have different usage patterns.

MPI_BCAST (Broadcast):
MPI_BCAST is used to broadcast data from one process (often called the root process) to all other processes in a communicator. It's a one-to-many communication operation.
Broadcasts the same data from one process to all other processes.
Only the root process sends data to all other processes.

MPI_SCATTER: MPI_SCATTER is used to distribute data from one process (often called the root process) to all processes in a communicator, with each process receiving a unique portion of the data. It's a one-to-many communication operation.
Distributes data from one process to all processes, with each process receiving a unique portion of the data.
The root process sends different portions of data to all other processes.

8) Discuss the cache coherence problem and its significance in parallel programming. (2 marks)

Solution:
The cache coherence problem arises in multiprocessor systems with separate caches for each processor (core). It ensures that all processors have a consistent view of the shared memory, preventing data corruption and incorrect program behavior.

- Each processor has its own local cache memory to improve access speed for frequently used data.
- When multiple processors access the same data (stored in main memory) and potentially cache it locally, inconsistencies can occur.
- One processor might modify the data in its cache, but the other processors might still be using the outdated version from their caches.

Cache coherence is crucial for ensuring reliable and deterministic behavior in parallel programs. Without it, parallel programs can exhibit unexpected behavior, making debugging and maintaining them very difficult.

MCQ:

1) Given that the maximum number of threads in a thread block is 1600, which among the following distributions of thread is not allowed:
   a. (4,4,256)
   b. (4, 64, 4)
   c. (8, 8, 16)
   d. (32,32,1)

2) A CUDA device has 40 SMs and each SM can accommodate 8 thread blocks simultaneously. Each thread block has 4 wraps of size 24. The device can have up to _____ threads simultaneously residing on the device:
   a. 30,720
   b. 3,840
   c. 7,680
   d. 9,60

3) In the following function call, a message is sent to which process?
   MPI_Send(message, 2, MPI_CHAR, 4, 0, MPI_COMM_WORLD)
   1. Process 4
   2. Process 2
   3. Process 1
   4. Process 0

4) In MPI, what is the difference between blocking and non-blocking communication?
   a) Blocking communication waits for the message to be received/sent before proceeding, while non-blocking communication continues execution.
   b) Blocking communication involves copying data, while non-blocking uses references.
   c) Blocking communication is for inter-process communication, and non-blocking is for intra-process communication.
   d) There is no significant difference in practice.

5) What is a common approach to avoid deadlocks in MPI programs?
   a) Use non-blocking communication whenever possible.
   b) Employ synchronization with barriers and locks.
   c) Increase the buffer size for message passing.
   d) Reduce the number of processes involved in communication.

6) What is a potential challenge in OpenMP programming with shared memory?
   a) Limited communication overhead between threads.
   b) Race conditions when accessing shared data
   c) Difficulty in managing threads.
   d) Lack of portability across different compilers.

7) What is the advantage of using atomic operations in OpenMP?

a) They enable efficient communication between threads.
b) They guarantee thread-safe updates to shared variables.
c) They define a barrier for thread synchronization.
d) They improve performance for frequent read/write operations.

8) What is the difference between #pragma omp for and #pragma omp parallel for directives?
a) **#pragma omp for** implies a single thread iterating over the loop, while **#pragma omp parallel for** creates multiple threads iterating over the loop.
b) **#pragma omp for** is for data-parallel loops, while **#pragma omp parallel for** is for task-parallel loops.
c) **#pragma omp for** requires explicit synchronization, while **#pragma omp parallel for** handles it internally.
d) There is no significant difference; they achieve the same functionality.

9) In the vector addition kernel, the global threadID can be obtained by using the following formula
1. threadIdx.x + blockIdx.x*blockDim.x
2. threadIdx.y + blockIdx.y* blockDim.y
3. blockDim.x + blockIdx.x* threadIdx.x
4. blockIdx.x + blockIdx.x*blockDim.x

10) The rank of an MPI process in the MPI_COMM_WORLD communicator is a (0.5)
1. Floating point number starting from 1.0
2. Floating point number starting from 0.0
3. Integer number starting from 0
4. Integer number starting from 1