TDT4200 Exam (example)
Autumn 2023

1 True/False (10%)

1. The CUDA `__syncthreads()` operation synchronizes all active threads on the GPU
   False; it only synchronizes threads in the same thread block.

2. An MPI_Allreduce call will use more total bandwidth than the corresponding MPI_Reduce call
   True; the final result must be communicated to all participants, so the total bandwidth requirement increases by at least the size of the result times $p - 1$ messages for a system with p active ranks.

3. CUDA context switches can increase occupancy
   True; CUDA threads that block while pending a memory operation can be context-switched for threads that are ready to execute, thereby increasing GPU utilization.

4. Simultaneous Multithreading allows any pair of independent threads to run simultaneously
   False; SMT only runs threads simultaneously when they require distinct parts of available ALUs

5. Any collective MPI operation be replaced with a collection of point-to-point operations
   True; any MPI program can be implemented strictly in terms of point-to-point operations.

6. The operational intensity of a program increases when it is run on a faster processor
   False; the operational intensity of a program is determined by its number of operations combined with the number and size of data elements it accesses.

7. Worksharing directives in OpenMP end with an implicit barrier by default
   True; threads synchronize at the end of worksharing directives unless this is explicitly disabled with the 'nowait' clause.

8. A mutual exchange with standard mode Send and Recv operations can not deadlock
   False; message sizes that are too large to be buffered by the network interface cause Send and Recv to behave as blocking operations, which makes deadlock possible.

9. Strong scaling results suggest that an algorithm is suitable for use on higher processor counts than weak scaling results
   False

10. Pthreads operations on a cond variable associate it with a mutex variable
    True

2 Message passing and MPI (15%)

2.1

In MPI terminology, what distinguishes the Ready and Synchronized communication modes from each other?

Ready mode assumes that the recipient has already posted the matching Recv call at the time when a Send call is made, while Synchronized mode blocks the Send call until the recipient's Recv call has been made (and acknowledged).

2.2

Using pseudo-code, write a barrier implementation for a message-passing program.

```
if ( rank = 0 )
{
    arrived = 0
    while ( arrived < number_of_ranks ):
        receive id from any rank
        arrived = arrived + 1
    for all id > 0:
        send ack to rank id
}
else
{
    send id to rank 0
    receive ack from rank 0
}
```

3 GPGPU programming (10%)

3.1

Briefly describe two differences between POSIX threads and CUDA threads.

POSIX threads run with entirely separate instruction counters, and can directly make calls that access the operating system. CUDA threads run in SIMT mode where threads in a thread block share the same instruction, and can only directly access memory on the graphics device that executes them.

3.2

What differentiates shared and global memory spaces in CUDA programming?

Shared memory is a small amount of low-latency memory that is only shared among threads within a warp. Global memory is a much larger amount of higher latency memory, which is accessible to every thread in the entire grid.

4 Performance analysis (5%)

In a strong scaling study, what is theoretically the maximal speedup attainable by a program with 8% inherently sequential run time?

Amdahl's law gives the maximal speedup as $\frac{1}{f}$ *for a sequential fraction* f*. Thus,* $f = \frac{8}{100}$ *gives a maximal speedup of* $\frac{25}{2}$*, or* $12.5\times$*.*

5 Threads (10%)

5.1

Give an example of a race condition.

A race condition occurs when separate threads attempt to use an exclusive resource at the same time, for example if they simultaneously try to write data to a shared, open file.

5.2

Describe two ways to prevent race conditions using OpenMP.

Any section of code can require exclusive access using the `#pragma omp critical` *directive. Another way to do this is to declare an* `omp_lock_t` *variable for the section, require entering threads to obtain it with* `omp_set_lock` *operations, and release it with* `omp_unset_lock` *operations.*

6 15%

The C program on the following page calculates whether each point in a 640×480 array lies inside a unit circle scaled to the array, and saves the array in a file where values are scaled to their point's distance from the origin. The program partitions its domain along the vertical axis, and assumes that the number of ranks evenly divides the domain's height.

Modify the program so that it will also work with rank counts that are not divisors of the height.

```c
#define _XOPEN_SOURCE 600
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

#define width 640
#define height 480

int main ( int argc, char **argv ) {
    int size, rank;
    MPI_Init ( &argc, &argv );
    MPI_Comm_size ( MPI_COMM_WORLD, &size );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );
    int local_origin = (rank)*(height / size);
    int local_height = height / size;

    float *local_map = malloc ( local_height*width*sizeof(float) );
    #define MAP(i,j) local_map[(i)*width+(j)]
    for ( int i=0; i<local_height; i++ ) {
        for ( int j=0; j<width; j++ ) {
            float y = (2.0*(local_origin+i)-height) / (float) height;
            float x = (2.0*j-width) / (float) width;
            float rad = sqrt(y*y+x*x);
            if ( rad < 1.0 )
                MAP(i,j) = 1.0 - rad;
            else
                MAP(i,j) = 0.0;
        }
    }
    if ( rank == 0 ) {
        FILE *output = fopen ( "circle.dat", "w" );
        fwrite ( local_map, sizeof(float), local_height*width, output );
        for ( int r=1; r<size; r++ ) {
            MPI_Recv (
                local_map, local_height*width, MPI_FLOAT, r, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE
            );
            fwrite ( local_map, sizeof(float), local_height*width, output );
        }
        fclose ( output );
    } else {
        MPI_Send (
            local_map, local_height*width, MPI_FLOAT, 0, 0, MPI_COMM_WORLD
        );
    }
    free ( local_map );
    MPI_Finalize();
    exit ( EXIT_SUCCESS );
}
```

```
/* Solution: */
int main ( int argc, char **argv ) {
    /* Omitting initialization, size, rank */

    int local_origin = (rank)*(height / size);
    int local_height = height / size;

    if ( rank < (height%size) )
    {
        local_origin += rank;
        local_height += 1;
    }
    else
        local_origin += (height%size);

    /* Omitting computational kernel, as it remains identical */

    if ( rank == 0 ) {
        FILE *output = fopen ( "circle.dat", "w" );
        fwrite ( local_map, sizeof(float), local_height*width, output );
        for ( int r=1; r<size; r++ ) {
            int rank_local_height = height / size;
            if ( r < (height%size) )
                rank_local_height += 1;
            MPI_Recv (
                local_map, rank_local_height*width, MPI_FLOAT, r, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE
            );
            fwrite (
                local_map, sizeof(float), rank_local_height*width, output
            );
        }
        fclose ( output );
    } else {
        MPI_Send (
            local_map, local_height*width, MPI_FLOAT, 0, 0, MPI_COMM_WORLD
        );
    }
    free ( local_map );
    MPI_Finalize();
    exit ( EXIT_SUCCESS );
}
```

7 15%

The C program on the following page calculates whether each point in a 640×480 array lies inside a unit circle scaled to the array, and writes an image file where points inside are shaded according to their distance from the origin.

7.1

Create a version of the `write_map` function in the form of a CUDA kernel which can be called with thread blocks of size 4×4.

```
#define BLOCK 4
__global__ void write_map ( float *map ) {
    int i = blockIdx.y*BLOCK+threadIdx.y;
    int j = blockIdx.x*BLOCK+threadIdx.x;

    float y = (2*i-height) / (float)height;
    float x = (2*j-width) / (float)width;
    float rad = sqrt(y*y+x*x);
    if ( rad < 1.0 )
        MAP(i,j) = rad;
    else
        MAP(i,j) = 1.0;
}
```

7.2

Create a version of the `main()` function which

- allocates the floating-point array in GPU device memory instead,
- calls your `write_map` kernel using 4×4 thread blocks, and
- copies the result into a host memory buffer, before passing it to `save_image`

```
#define BLOCK 4
int main () {
    dim3 gridBlock(640/BLOCK,480/BLOCK);
    dim3 threadBlock(BLOCK,BLOCK);
    float *map;
    cudaMalloc ( (void **)&map, width*height*sizeof(float) );
    write_map<<<gridBlock,threadBlock>>>(map);

    float host_map[height*width];
    cudaMemcpy (
        host_map, map, 640*480*sizeof(float), cudaMemcpyDeviceToHost
    );
    cudaDeviceSynchronize();
    save_image ( host_map );
    cudaFree ( map );
    exit ( EXIT_SUCCESS );
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <math.h>

#define width 640
#define height 480
#define MAP(i,j) map[(i)*width+(j)]

void save_image ( float *map );
void write_map ( float *map );


int main () {
    float *map = malloc ( width*height*sizeof(float) );
    write_map ( map );
    save_image ( map );
    free ( map );
    exit ( EXIT_SUCCESS );
}


void write_map ( float *map ) {
    for ( int i=0; i<height; i++ ) {
        for ( int j=0; j<width; j++ ) {
            float y = (2*i-height) / (float)height;
            float x = (2*j-width) / (float)width;
            float rad = sqrt(y*y+x*x);
            if ( rad < 1.0 )
                MAP(i,j) = rad;
            else
                MAP(i,j) = 1.0;
        }
    }
}


void save_image ( float *map ) {
    uint8_t image[height][width][3];
    for ( int i=0; i<height; i++ )
        for ( int j=0; j<width; j++ )
            image[i][j][0] = image[i][j][1] = image[i][j][2] = (1.0-MAP(i,j))*255;
    FILE *out = fopen ( "circle.ppm", "w" );
    fprintf ( out, "P6 %d %d 255\n", width, height );
    fwrite ( image, 3*sizeof(uint8_t), width*height, out );
    fclose ( out );
}
```