TDT4200 Exam (example)
Autumn 2023

1 True/False (10%)

1. The CUDA `__syncthreads()` operation synchronizes all active threads on the GPU
2. An MPI_Allreduce call will use more total bandwidth than the corresponding MPI_Reduce call
3. CUDA context switches can increase occupancy
4. Simultaneous Multithreading allows any pair of independent threads to run simultaneously
5. Any collective MPI operation be replaced with a collection of point-to-point operations
6. The operational intensity of a program increases when it is run on a faster processor
7. Worksharing directives in OpenMP end with an implicit barrier by default
8. A mutual exchange with standard mode Send and Recv operations can not deadlock
9. Strong scaling results suggest that an algorithm is suitable for use on higher processor counts than weak scaling results
10. Pthreads operations on a cond variable associate it with a mutex variable

2 Message passing and MPI (15%)

2.1

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

2.2

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

3 GPGPU programming (10%)

3.1

Briefly describe two differences between POSIX threads and CUDA threads.

3.2

What differentiates shared and global memory spaces in CUDA programming?

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?

5 Threads (10%)

5.1

Give an example of a race condition.

5.2

Describe two ways to prevent race conditions using OpenMP.

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.

```
#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 );
}
```

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.

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`

```
#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 );
}
```