NTNU

Norwegian University of
Science and Technology

DEPARTMENT OF COMPUTER SCIENCE

TDT4200 - PARALLEL PROGRAMMING

Exercise 6

1 Introduction

This is the **graded** CUDA exercise and will potentially give you 15 points towards your final grade. You can get a maximum of 20 points combined with the MPI exercise. In this assignment, we will parallelise the 2D wave equation using an accelerator with CUDA. To get the maximum score, you need to utilise cooperative groups. However, a solution without using this model will give partial credit.

2 Tasks

2.1 Programming

Note that although the handout code looks like C, CUDA is a programming platform and model extending C++. To use cooperative groups, you must include the library and access it through the `cooperative_groups` namespace.

Threads in CUDA are organised as illustrated in Figure 1.

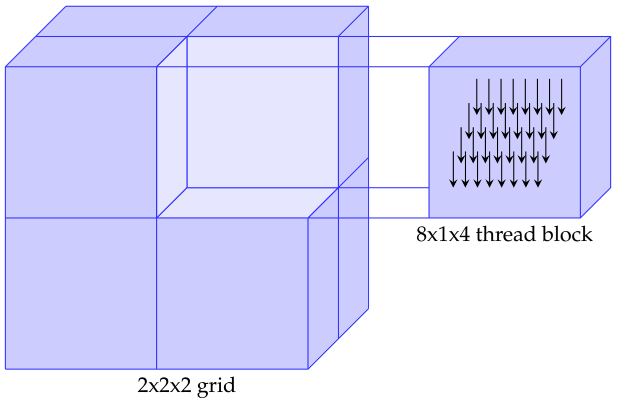
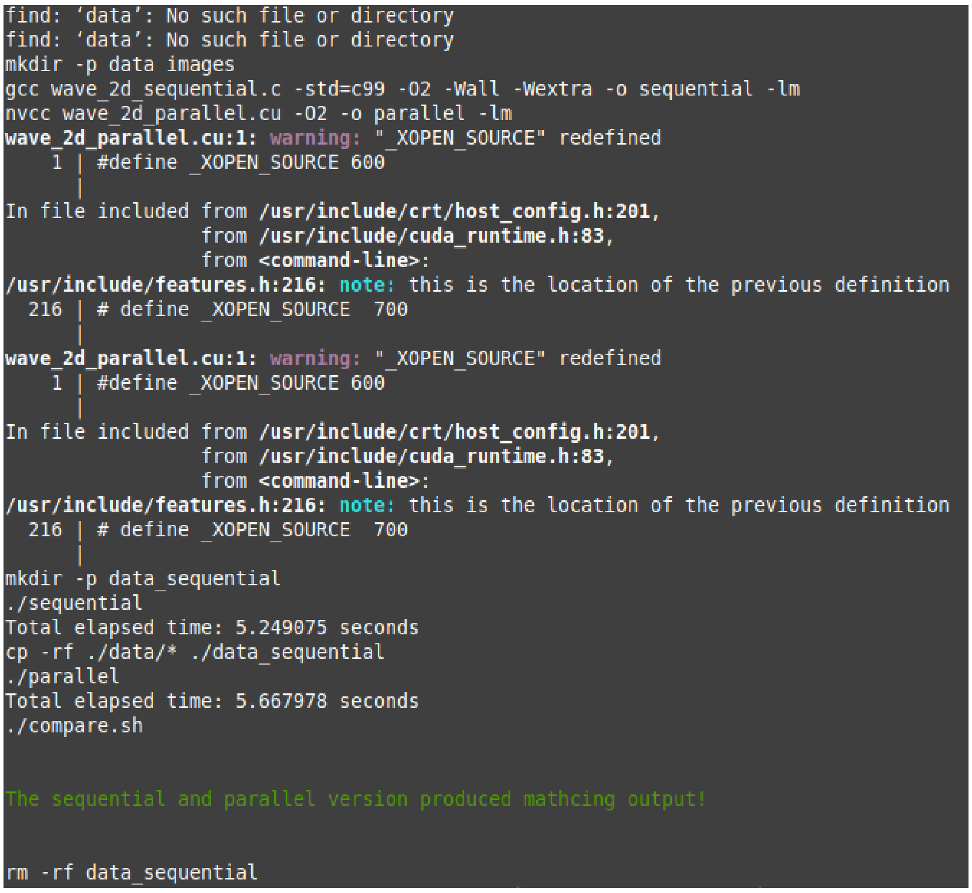


Figure 1: Threads are contained within **blocks** that are organised in **grids**.

For this problem set, we have set the default domain size to be small enough to fit in the GPUs available to you in the cluster (128x128), but do play around with M and N to see how it performs.

Before you start the exercise, it is recommended to run the `make check` command from the `Makefile` that will run the `compare.sh` script that compares the sequential and parallel solution. Running it on the handout files should produce a similar result as shown in Figure 2.



```
find: ‘data’: No such file or directory
find: ‘data’: No such file or directory
mkdir -p data images
gcc wave_2d_sequential.c -std=c99 -O2 -Wall -Wextra -o sequential -lm
nvcc wave_2d_parallel.cu -O2 -o parallel -lm
wave_2d_parallel.cu:1: warning: "_XOPEN_SOURCE" redefined
    1 | #define _XOPEN_SOURCE 600
      |
In file included from /usr/include/crt/host_config.h:201,
                 from /usr/include/cuda_runtime.h:83,
                 from <command-line>:
/usr/include/features.h:216: note: this is the location of the previous definition
  216 | # define _XOPEN_SOURCE  700
      |
wave_2d_parallel.cu:1: warning: "_XOPEN_SOURCE" redefined
    1 | #define _XOPEN_SOURCE 600
      |
In file included from /usr/include/crt/host_config.h:201,
                 from /usr/include/cuda_runtime.h:83,
                 from <command-line>:
/usr/include/features.h:216: note: this is the location of the previous definition
  216 | # define _XOPEN_SOURCE  700
      |
mkdir -p data_sequential
./sequential
Total elapsed time: 5.249075 seconds
cp -rf ./data/* ./data_sequential
./parallel
Total elapsed time: 5.667978 seconds
./compare.sh


The sequential and parallel version produced mathcing output!


rm -rf data_sequential
```

Figure 2: Running `make check` on a fresh project. The tests succeeded as we haven't implemented parallelisation yet; the parallel file is another sequential version.

1. **Include Cooperative Groups:**

   To access cooperative groups, we need to include the library.

   a Include the cooperative group header.

   b Declare variables that you find fitting for this exercise.

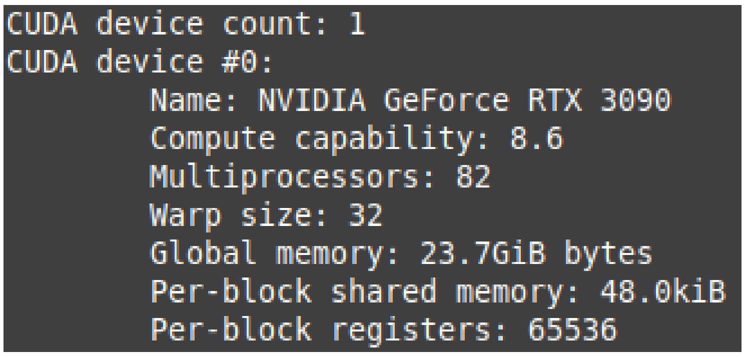
   Hint: It is helpful for readability to prefix variables on the host with `h_` and variables on the device with `d_`.

2. **Initialise CUDA:**

   We need to know that we have a CUDA-compatible GPU. With the function `cudaGetDeviceCount`, we can get the number of available devices. When we have decided which device to use, we can set it with `cudaSetDevice`. We can now print some information about the device by printing attributes of the struct `cudaDeviceProp` and use the function `cudaGetDeviceProperties`.

Implement the function `init_cuda`. The function should:

- Check whether a CUDA-compatible device is connected. If not, return `false`.
- Use a compatible device.
- Use `cudaDeviceProp` to print information similar to what's shown in Figure 3.
- If any errors occur while accessing the properties, return `false`; otherwise, return `true`.



```
CUDA device count: 1
CUDA device #0:
        Name: NVIDIA GeForce RTX 3090
        Compute capability: 8.6
        Multiprocessors: 82
        Warp size: 32
        Global memory: 23.7GiB bytes
        Per-block shared memory: 48.0kiB
        Per-block registers: 65536
```

Figure 3: After implementing T2, running the parallel version should print something similar.

3. **Initialise the Domain:**

   Our data needs to be transferred to the device to use on the GPU.

   Change the function `domain_initialize`, such that

   - It allocates space for at least one grid on the host, which can be written to a file.
   - It allocates space for three grids on the device (prv, current, and nxt).
   - It transfers suitable variables to the device for calculations.

4. **Finalise the Domain:**
   After allocating memory, we must also return it to the OS to prevent memory leaks.

   Change the function `domain_finalize` such that it frees all the memory allocated by `domain_initialize`.

5. **Time Step:**

   The central part we want to parallelise is calculating the space domain.

   Change the function `time_step` such that it can be launched as a kernel on the GPU from the CPU.

   Note: Depending on how you design your solution, you might want to change the `move_buffer_window` function as well.

6. **The Boundary Condition:**

   The boundary condition is embarrassingly parallel and suited to be calculated on the GPU.

   Change the function `boundary_condition` to calculate the ghost cells on the GPU.

   Hint: Does this kernel need to be launched from the CPU? Or can it be called from the `time_step` function?

7. **Combining it all Together:**

   Now, we must combine the updated simulation and launch the kernel from the CPU.

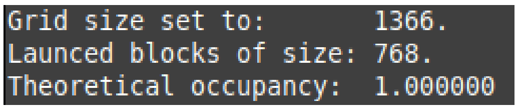
   Change the `simulate` function such that it prepares the grid and launches the cooperative groups. Remember to copy the answer to the CPU for data storage whenever the time iterations correspond with the snapshot frequency.

8. **Performance:**

   After our code is parallelised, we want to monitor the performance of the GPU. To do this, we use the occupancy metric, which is described as

$$\text{occupancy} = \frac{\text{Active Warps}}{\text{Maximum Warps per Multiprocessor}}. \tag{1}$$

   Implement the function `occupancy`. The function should use `cudaDeviceProp` and calculate occupancy on the GPU given by Equation 1.



```
Grid size set to:        1366.
Launced blocks of size: 768.
Theoretical occupancy:  1.000000
```

Figure 4: The `occupancy` function should print something similar to this after implementation.

2.2 Theory Questions

1. What is the difference between a function declared as `__global__` and a function declared as `__device__` in CUDA?

2. What are some advantages of parallelising with CUDA as opposed to MPI?

3. What are some pros and limitations when using cooperative groups?

4. How good occupancy did you achieve on the GPU? Can you improve it?

3 Deliverables

Please deliver `.zip` file with the following contents containing your changes to the code:

- `wave_2d_parallel.cu`
- A document with your answers to the theory questions.