Assignment 3

Assignment 3 has two variations: “The InfOli model” and “Nonuniform Fast Fourier Transform”. You only have to do one of them, i.e. you can pick the variation you like the most. Both variations consist of two sub-assignments. The due date for the first sub-assignment is **Friday, May 21, 2021, midnight 23:59**, and the due date for the second sub-assignment is **Friday, May 28, 2021, midnight 23:59**. Both sub-assignments must be uploaded on Nestor. Your grade for assignment 3 is the weighted average of your sub-assignment 1 grade (30%) and sub-assignment 2 grade (70%). Please note that the ways to get access to the basic code and the submission procedure are both subject to change and may not fully correspond to what is stated below.

The InfOli model (variation 1)

The inferior-olivary nucleus (ION)

A biological neural network is composed of interconnected neurons. Neurons are cells that process and transmit signals within the brain. The biological neuron generally comprises three parts: The Dendrites, the Soma, and the Axon.

- The **dendrite** represents the cell input stage. The dendrites pick up electrochemical stimuli from other cells and transfer them to the soma;
- The **soma** processes the stimuli and translates them into a cell membrane potential, which evokes a cell response called an action potential or, simply, a spike;
- This response is transferred through the **axon**, which is the cell output connection, to other cells' inputs. An electrochemical connection between two cells (axon-dendrite) is called a **synapse**.

The **inferior-olivary nucleus (ION)** is a biological neural network that is present in the brain. The purpose of the ION is to provide rhythm and coordination for motor functions. The ION's neurons are heavily interconnected to one another through **gap junctions (GJs),** i.e. direct electrical connections between their dendrites. These GJs are not just simple connections, but quite intricate. In this exercise, we assume an all-to-all interconnection: every neuron is connected to every other neuron. Furthermore, some of the **ION** neurons are connected to other parts of the brain.

The InfOli model of the inferior-olivary nucleus

InfOli is a computational model of the ION implemented in the C language. Each neuron is represented by the dendrite (also includes the GJs which implement the inter-neuron connectivity), soma, and axon.

The model consists of state parameters that describe the simulated state of the ION at a specific time. Each neuron is represented by a set of state parameters that describe both the state of the neuron as a whole as the individual states of the dendrites, soma, and axon which make up the neuron. By iteratively updating the state parameters, the ION can be simulated through time. All state parameters are updated concurrently at each simulation step. In each simulation step, the time at which the model is simulated is advanced by 50 μs.

Given the current state parameters of the model, the values of the state parameters in the next iteration depend on the following:

- The current state of the Dendrites, Soma or Axon that is being updated;
- The current state of the dendritic compartment of the neurons to which the updating neuron is connected through the GJs;
- The external input to the InfOli, representing the input coming from the rest of the brain into the ION.

Given a network with 96 neurons, we determined the total number of floating-point (FP) operations performed per neuron in a single simulation step. Furthermore, we determined the FP operations performed per compartmental task (e.g. soma or axon computations). The contributions of the different compartmental tasks to the total number of FP operations

performed per neuron in a single simulation step are listed below.

Compartmental Task	Percentage (%) of FP operations
Soma	13
Dendrite	10
Axon	8
Gap Junction	69

The GJs contribute significantly: 69% of the total FP operations. What's more: the GJ computations increase quadratically with the network size (as the computations are repeated in every neuron, once per simulation step, for every independent connection), as opposed to the linear fashion of the main neuron compartments, dictating also the overall complexity of the application. Additionally, the GJ computations themselves are demanding, as they are not just simple connections.

Sub-assignment 1:

Step 0: Login to peregrine in Interactive or Batch mode. Update (if needed) your local repository to obtain the code needed for the assignment. Warning: Do not accidentally overwrite the code you wrote for Assignment 1 and 2.

```
ssh <s/f_number>@pg-gpu.hpc.rug.nl          (interactive mode)
```
or
```
ssh <s/f_number>@peregrine.hpc.rug.nl          (batch mode)

cd <labs-directory>
git pull
```

Step 1: Study the implementation of the InfOli model that we provide in `assignment3/infoli`. The goal of the assignment is to accelerate the provided implementation (primarily the main-compartment and GJ computations) using CUDA. The most interesting part of the code is located between "`// MAIN LOOP - START`" and "`// MAIN LOOP - END`", and in the functions that are called therein.

The GJs are the main computational bottleneck and also the part that dominates the overall computational complexity. The main-compartment (dendrite, soma and axon) computations are also expensive. Although they scale linearly with the problem size, they still include a good deal of computation. The dataflow nature of the main-compartment computations gives great potential for parallelization.

Step 2: Come up with ways to implement a CUDA accelerated application. You do not have to implement these optimizations in sub-assignment 1. Instead, we ask you to write a report (preferably in latex, named `report1.pdf`) in which you include:

- A simple performance model of the sequential CPU program (Hint: you can use a CPU profiler to obtain an indication of the times spent by the different functions, however, also inspect and analyze the code and the data carefully in order to make reasonable assumptions about the Compute bound and the I/O bound performance of your accelerated application later);
- Simple performance models of the different CUDA accelerated versions/optimizations you are considering (Hint: the performance models consist of for example a system level diagram that shows where the data and the different functions “live”, how much data will be moved between the CPU, ACC, CPU_MEM and ACC_MEM, what computation will be offloaded on the CUDA device, which functions will be left on the CPU, and how much time you think the above movements and computations will require assuming fully overlapped computations and communications/data movements);
- For each possible optimization/version you should provide:
  - A description of the intended optimization and/or system model;
  - Your analysis of the speedup you expect to achieve after the implementation of the intended optimization.

Notes:

- If you have a hard time coming up with optimizations, don't worry. We will send everyone a list of optimizations that can be implemented in sub-assignment 2 right after sub-assignment 1 has been submitted by everyone. Furthermore, sub-assignment 1 contributes relatively little to your assignment 3 grade (see page 1).

- If you finish sub-assignment 1 before its deadline, we recommend to start working on sub-assignment 2.

Step 3: Submit your assignment. Compress the report and name it after your s-number:

```
tar -cf s<XYZ>_infoli_1.tar report1.pdf
```

, where `<XYZ>` is your student number. Upload the compressed folder to Nestor. Submissions with incorrect names may not be processed.

Sub-assignment 2:

Step 4:The following commands can be used to compile and run the code:

```
module load CUDA/9.0.176
make
./infoli_simple <Net_Size>
```

where `<Net_Size>`, the network size `<Net_Size> x <Net_Size>`, is <= 1000.

Optimize the provided code: implement as many optimizations as you can. Start with the optimizations you came up with in sub-assignment 1, and implement optimizations you came up with or learned about after submission of sub-assignment 1 afterwards. Feel free to restructure the code and add files (e.g., .cu and .h files).

Make sure that the output of the optimized code is close to that of the reference implementation. The outputs may deviate slightly due to floating point precision differences between CPU and GPU.

Step 5: Write a report (preferably in latex, named `report2.pdf`) which describes all optimizations you tried regardless of whether you committed them or abandoned them and whether they improved or hurt performance. For each optimization, include in your report:

- A description of the optimization;
- Any difficulties you had with implementing the optimization correctly;
- The change in execution time after the optimization was applied (use network sizes 10, 100, 250, 500, 750, and 1000);
- An explanation of why you think the optimization helped or hurt performance.
- Whether the obtained speedup is the same as the in the sub-assignment 1 report expected speedup. If it is not, an explanation of why you think they are not equal.

Step 6: Submit your assignment. Compress your report and the files you added or altered and name them after your s-number like so:

```
tar -cf s<XYZ>_infoli_2.tar report2.pdf <created/edited
files>
```

, where `<XYZ>` is your student number. Upload the compressed folder to Nestor. Submissions with incorrect names may not be processed.

Nonuniform Fast Fourier Transform (variation 2)

In `assignment3/nufft/main.cpp` you will find a loop in which complex hypercube *bb* is multiplied (element-wise) with a real cube *wu*, an adjoint transformation is applied to the result, the result is multiplied with complex cube *b1* (element-wise), a forward transformation is applied to the result, and finally, the result is again multiplied by real cube *wu* (element-wise).

The adjoint and forward transformations are executed on the GPU (using the cuFINUFFT library) by default. Nonetheless, there are opportunities to accelerate the code. Among others, there is a lot of unnecessary transformation of data between the host and device, and code that could run in parallel is actually run sequentially.

Sub-assignment 1:

Step 0: Login to peregrine in Interactive or Batch mode. Update (if needed) your local repository to obtain the code needed for the assignment. Warning: Do not accidentally overwrite the code you wrote for Assignment 1 and 2.

```
ssh <s/f_number>@pg-gpu.hpc.rug.nl          (interactive mode)
or
ssh <s/f_number>@peregrine.hpc.rug.nl              (batch mode)

cd <labs-directory>
git pull
```

Step 1: Study the code we provide in `assignment3/nufft`.

The functions `mcnufft_mtimes_adj()` and `mcnufft_mtimes()` are called in `main.cpp`. We provide reference implementations of these functions in `src/mcnufft.cpp`. The goal of the assignment is to optimize these functions and functions called thereby by accelerating them using CUDA.

Hints

- `wu` and `b1` are not changing, the only changing input is `data`. Can we reduce/optimise data movements?
- Can certain variables be stored in constant memory?
- There is a `many` version of the NUFFT library functions that allows you to execute multiple NUFFT at once. Can we reduce the amount of system calls by using this? Is the reduction in system calls the only speed-up factor? (please note that direct memory access (DMA) is faster when a lot of data is transferred at once instead of multiple small transfers);
- The NUFFT library is thread-safe. Can we parallelize over runs?

Step 2: Come up with ways to implement a CUDA accelerated application. You do not have to implement these optimizations in sub-assignment 1. Instead, we ask you to write a report (preferably in latex, named `report1.pdf`) in which you include:

- A simple performance model of the sequential CPU program (Hint: you can use a CPU profiler to obtain an indication of the times spent by the different functions, however, also inspect and analyze the code and the data carefully in order to make reasonable assumptions about the Compute bound and the I/O bound performance of your accelerated application later);
- Simple performance models of the different CUDA accelerated versions/optimizations you are considering (Hint: the performance models consist of for example a system level diagram that shows where the data and the different functions "live", how much data will be moved between the CPU, ACC, CPU_MEM and ACC_MEM, what computation will be offloaded on the CUDA device, which functions will be left on the CPU, and how much time you think the above movements and computations will require assuming fully overlapped computations and communications/data movements);
- For each possible optimization/version you should provide:
    - A description of the intended optimization and/or system model;
    - Your analysis of the speedup you expect to achieve after the implementation of the intended optimization.

Notes:

- If you have a hard time coming up with optimizations, don't worry. We will send everyone a list of optimizations that can be implemented in sub-assignment 2 right after sub-assignment 1 has been submitted by everyone. Furthermore, sub-assignment 1 contributes relatively little to your assignment 3 grade (see page 1).

- If you finish sub-assignment 1 before its deadline, we recommend to start working on sub-assignment 2.

Step 3: Submit your assignment. Compress the report and name it after your s-number:

```
tar -cf s<XYZ>_nufft_1.tar report1.pdf
```

, where `<XYZ>` is your student number. Upload the compressed folder to Nestor. Submissions with incorrect names may not be processed.

Sub-assignment 2:

Step 4: The following can be executed to compile and run the code (in interactive mode):

```
module load CMake/3.18.4-GCCcore-10.2.0
module load FFTW/3.3.8-gompic-2020b
module load flex/2.6.4

mkdir build
cd build

cmake .. -DGPU=ON          (use GPU)
```
or
```
cmake .. -DGPU=OFF         (do not use GPU)

make  -j
./nufft_exercise
```

Optimize the provided code: implement as many optimizations as you can. Start with the optimizations you came up with in sub-assignment 1, and implement optimizations you came up with or learned about after submission of sub-assignment 1 afterwards. Feel free to restructure the code and add files (e.g., .cu and .h files).

Run the loop in `main.cpp` which calls the optimized functions both with the reference and optimized functions, and make sure that the results, stored in the `results` variable, are close. The outputs may deviate slightly due to floating point precision differences between CPU and GPU.

Step 5: Write a report (preferably in latex, named `report2.pdf`) which describes all optimizations you tried regardless of whether you committed them or abandoned them and whether they improved or hurt performance. For each optimization, include in your report:

- A description of the optimization;
- Any difficulties you had with implementing the optimization correctly;
- The change in execution time after the optimization was applied;
- An explanation of why you think the optimization helped or hurt performance.
- Whether the obtained speedup is the same as the in the sub-assignment 1 report expected speedup. If it is not, an explanation of why you think they are not equal.

Step 6: Submit your assignment. Compress your report and the files you added or altered and name them after your s-number like so:

```
tar -cf s<XYZ>_nufft_2.tar report2.pdf <created/edited files>
```

, where `<XYZ>` is your student number. Upload the compressed folder to Nestor. Submissions with incorrect names may not be processed.