Assignment 5: The Ocean

Practical information

Deadline: Sunday 13/3, 8pm

Resources:

- ERDA for file storage
- Jupyter for the Terminal to access DAG
- Nvidia profiler to determine the parallelisation bottlenecks
- Nvidia vGPUs for benchmarks on DAG

Handin:

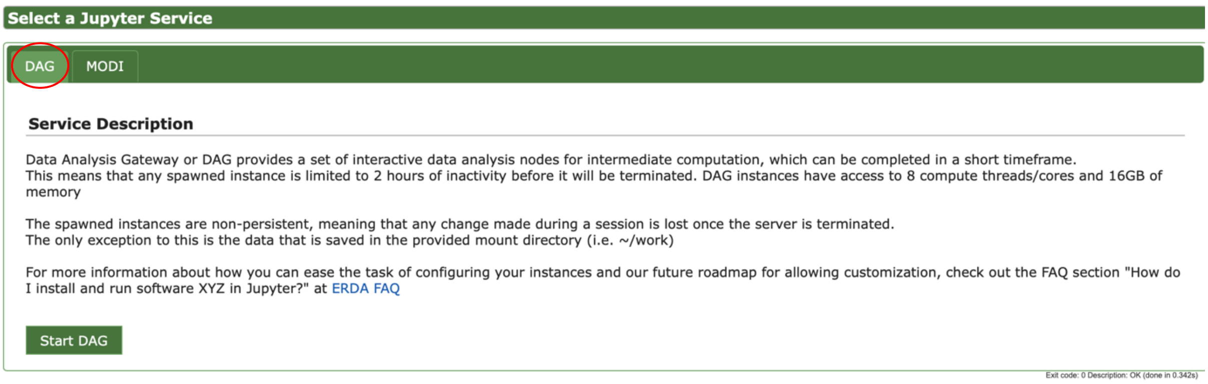
- Total assignment: a report of up to 3 pages in length (excluding the code)
- Use the template on Absalon to include your code in the report

Introduction

The Shallow Water (SW) model (section 13.3) is the simplest numerical representation of the ocean. Still, it has reasonable skills when used to predict the evolution of storm surges or Tsunamis. Moreover, it illustrates nicely the functioning and parallelization of stencil operations.

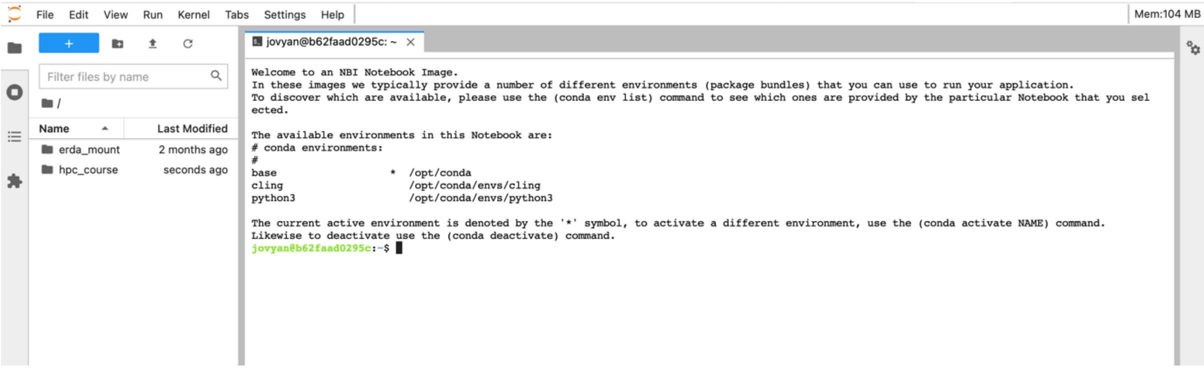
DAG

For this assignment we need `nvc++` to compile and DAG for source code profiling and running the benchmarks.



You can read more about DAG in the user guide: https://erda.dk/public/ucph-erda-user-guide.pdf

Spin up a Jupyter session on DAG selecting the “HPC GPU notebook” notebook image. In the terminal (or the folder view on the right side) you can see a number of folders.



The different folders contain:
`erda_mount`: your own files.
`hpc_course`: course files.

Preparations

Start by copying the exercise to your storage area and enter in to the folder. You can write '`ls`' to get a file listing of the folder.

```
cd erda_mount/HPPC
cp -a ~/hpc_course/module5 .
cd module5
ls
```

To be able to edit the files for the exercise navigate to the same folder in the file view. Here you can see 4 files:

```
Makefile
sw_parallel.cpp
sw_sequential.cpp
visualize.ipynb
```

Before you can run the code, you need to compile it. This can be done by running `make` in the terminal. The `sw_sequential.cpp` code is identical to `sw_parallel.cpp` there (with produced corresponding binaries `sw_sequential` and `parallel`) to give you a backup. The `visualize.ipynb` is for SW model output visualisation and analysis.

To run the code for the default 1000 time-steps on a grid of 512x512 on DAG and write the model output to a file, run

```
./sw_sequential
```

Nvidia profiler

NVIDIA profiler enables you to understand and optimize the performance of your OpenACC application. An example of command-line `nvprof` profiler output for parallelised SW model is given below. One can, for example, see a runtime of four compute kernels (`integrate_116_gpu`,`integrate_123_gpu`,`exchange_vertical_ghost_lines_100_gpu`, `exchange_horizontal_ghost_lines_87_gpu`) and time spent for copying data from Host to Device and back, lines with `[CUDA memcpy HtoD]` and `[CUDA memcpy DtoH]`, respectively.

```
jovyan@b62faad0295c:~/erda_mount/Teaching/hpc_course_private/module5$ nvprof ./sw_parallel --iter 500
==534== NVPROF is profiling process 534, command: ./sw_parallel --iter 500
checksum: 4117.75
elapsed time: 1.06693 sec
==534== Profiling application: ./sw_parallel --iter 500
==534== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   43.05%  16.155ms       500  32.310us  32.032us  33.183us  integrate_116_gpu(Water&)
                   34.99%  13.129ms       500  26.258us  25.984us  26.880us  integrate_123_gpu(Water&)
                   10.59%  3.9730ms      1000  3.9730us  3.9030us  14.720us  exchange_vertical_ghost_lines_100_gpu(std::array<std::array<float,
 unsigned long=512>, unsigned long=512>&)
                   10.02%  3.7595ms      1000  3.7590us  3.7110us  5.6640us  exchange_horizontal_ghost_lines_87_gpu(std::array<std::array<float
, unsigned long=512>, unsigned long=512>&)
                    0.68%  254.56us         1  254.56us  254.56us  254.56us  [CUDA memcpy HtoD]
                    0.67%  252.19us         1  252.19us  252.19us  252.19us  [CUDA memcpy DtoH]
      API calls:   69.07%  203.85ms         1  203.85ms  203.85ms  203.85ms  cuDevicePrimaryCtxRetain
                   18.15%  53.572ms      7002  7.6510us  1.0040us  635.72us  cuStreamSynchronize
                    7.98%  23.540ms         1  23.540ms  23.540ms  23.540ms  cuMemHostAlloc
                    4.36%  12.860ms      3000  4.2860us  3.4370us  533.67us  cuLaunchKernel
                    0.28%  838.18us         1  838.18us  838.18us  838.18us  cuMemAllocHost
                    0.08%  225.78us         2  112.89us  109.45us  116.33us  cuMemAlloc
                    0.05%  148.92us         1  148.92us  148.92us  148.92us  cuModuleLoadDataEx
                    0.01%  30.053us         1  30.053us  30.053us  30.053us  cuMemcpyHtoDAsync
                    0.00%  12.201us         3  4.0670us  2.1660us  7.5830us  cuEventRecord
                    0.00%  10.629us         1  10.629us  10.629us  10.629us  cuMemcpyDtoHAsync
                    0.00%  8.4790us         2  4.2390us  1.8640us  6.6150us  cuDeviceGetPCIBusId
                    0.00%  7.6780us         1  7.6780us  7.6780us  7.6780us  cuPointerGetAttributes
                    0.00%  6.4870us         3  2.1620us     826ns  3.1350us  cuEventCreate
                    0.00%  4.7920us         4  1.1980us     392ns  2.9810us  cuModuleGetFunction
                    0.00%  4.2280us        10     422ns     156ns  2.1590us  cuDeviceGetAttribute
                    0.00%  3.0310us         4     757ns     149ns  2.5130us  cuDeviceGet
                    0.00%  2.6640us         3     888ns     211ns  1.8230us  cuDeviceGetCount
                    0.00%  2.2560us         1  2.2560us  2.2560us  2.2560us  cuEventSynchronize
                    0.00%  2.1050us         1  2.1050us  2.1050us  2.1050us  cuCtxGetCurrent
                    0.00%  2.0000us         3     666ns     299ns     908ns  cuCtxSetCurrent
                    0.00%     526ns         2     263ns     175ns     351ns  cuDeviceComputeCapability
                    0.00%     231ns         1     231ns     231ns     231ns  cuDriverGetVersion
 OpenACC (excl):   19.91%  24.159ms         1  24.159ms  24.159ms  24.159ms  acc_enter_data@sw_parallel.cpp:145
                   16.83%  20.430ms      1500  13.619us  1.7520us  57.933us  acc_wait@sw_parallel.cpp:116
                   15.25%  18.508ms      1500  12.338us  1.7480us  637.24us  acc_wait@sw_parallel.cpp:123
                    8.48%  10.290ms      2000  5.1440us  1.8420us  25.397us  acc_wait@sw_parallel.cpp:100
                    8.45%  10.256ms      2000  5.1280us  1.8510us  25.614us  acc_wait@sw_parallel.cpp:87
                    4.91%  5.9630ms      1000  5.9630us  4.6990us  535.39us  acc_enqueue_launch@sw_parallel.cpp:87 (_Z38exchange_horizontal_gho
st_lines_87_gpuRSt5arrayIS_IfLm512EELm512EE)
                    4.37%  5.3077ms      1000  5.3070us  4.6310us  23.830us  acc_enqueue_launch@sw_parallel.cpp:100 (_Z37exchange_vertical_ghos
t_lines_100_gpuRSt5arrayIS_IfLm512EELm512EE)
                    2.71%  3.2845ms       500  6.5680us  4.8360us  592.57us  acc_enqueue_launch@sw_parallel.cpp:116 (_Z17integrate_116_gpuR5Wat
```

Task 1: OpenACC parallelise the program (points 5)

The key challenge is to identify which parts of the code can reasonably be executed by the GPUs and to find suitable OpenACC directives and clauses for optimal parallelization. With the help of a profiler determine the bottlenecks. Play around a bit with the `#pragma` and see if you can improve on your first try. Thus, you need to save the profiler output of your various experiments. To get all 5 points, you should experiment around and attempt different paths for optimization using OpenACC.

Task 2: Strong and weak scaling (points 3)

Measure the weak and strong scaling of your programs. You should adjust num_gangs() to change how much of the GPU is actually used, and explain what this means for how the work actually maps to the physical hardware. Note that num_gangs() controls number of blocks NOT number of SMs.

For the weak scaling, you should change the grid size such that the calculations per thread stays constant, and for the strong scaling the grid size should be set such that the scaling from 1 to 2 is approximately linear like last week. Some key figures to note: You have 14 SMs available, the maximum number of threads per thread-block is 1024 and each multiprocessor can handle at most 2048 threads.

Task 3: Physics (points 2)

Test if the theoretically predicted phase speed of $c = (gH)^{1/2}$ is correctly reproduced by your model. Quantify and explain the possible differences between theory and simulation.

Bonus

The group who has the fastest parallelized code at 4pm with default settings for Nx, Ny and iter=10000 is rewarded a six-pack of beer.