Assignment 4: Shallow Waters

Practical information

Deadline: Monday 3/11 23.59

Resources:

- ERDA for file storage
- Nvidia profiler to determine the parallelisation bottlenecks
- Jupyter for the Terminal to access DAG Nvidia vGPU instances

Handin:

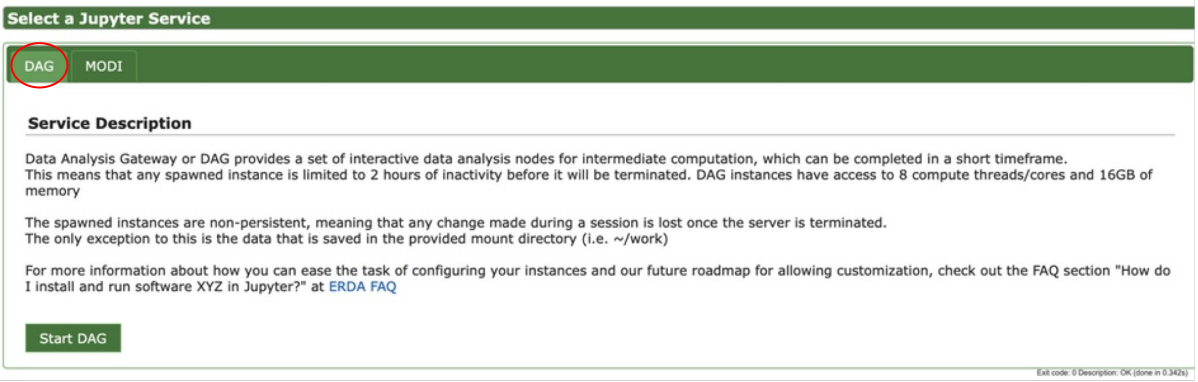
- Total assignment: a report of up to 3 pages in length (excluding the code)
- Use the template on Absalon to include your code in the report

Introduction

The Shallow Water (SW) model (see PHPC section 13.3 or the lecture notes on Absalon; first 3 pages are enough) is the simplest numerical representation of the ocean. Still, it has reasonable precision 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. For the python version of the assignment, we need to do "conda install `cupy`".



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`: not relevant.

Preparations

Start by updating the exercise folder in your storage area and enter in to the folder. You can write '`ls`' to get a file listing of the folder. Steps below are for C++ and Python. Fortran is very similar to C++.

```
cd erda_mount/ahpc
git pull
cd week4/cpp
cd week4/fortran
```

or

```
cd week4/python
```

You can write '`ls`' to get a file listing of the folder. For C++ it looks like

```
Makefile (only for C++ and Fortran)
sw_parallel.cpp/f90/py
sw_sequential.cpp/f90/py
visualize.ipynb
run_sw.sh
```

For C++ and Fortran, 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/f90/py` code is identical to `sw_parallel.cpp/f90/py` and is there (with produced corresponding binaries sw_sequential and sw_parallel) to give you a backup. The `visualize.ipynb` is for Shallow Waters model output visualisation and analysis. The `run_sw.sh` script can help you with launching the code on a restricted number of streaming multiprocesors to do weak scaling.

To run the code for 500 time-steps on the CPU on DAG and write the model output in ASCII file to your storage you can do:

```
./sw_sequential --iter 500 --out sw_output.txt
```

Or in Python

```
python ./sw_sequential --iter 500 --out sw_output.txt
```

To run the code for 500 time-steps on the GPU and write the model output in ASCII file to your storage you can do:

```
./sw_parallel --iter 500 --out sw_output_gpu.txt
```

Or in Python

```
python ./sw_parallel --iter 500 --out sw_output_gpu.txt
```

You should at least once try to visualize your output. Visualizing your output is also good for validation.

Nvidia profiler

NVIDIA profiler (nsys) enables you to understand and optimize the performance of your GPU application. An example of command-line nsys profiler output for parallelised SW model using CuPy and profiler decoraters is given below. In the different sections there are:

- [3/8]: Time spent, number of calls etc in the decorated functions. Directly related to source code
- [4/8]: Timing for OS and system calls, can normally be ignored
- [5/8]: Timing for CUDA specific functions
- [6/8]: Timing for individual kernels. Generated by CuPy / OpenACC
- [7/8]: Memory transfer statistics from host to device and device to host
- [8/8]: GPU memory usage by the program

All the information is quite useful for understanding what contributes to runtime, and can be sueful to e.g. detect transfer of small data done implicitly by either CuPy or OpenACC.

```
% nsys profile --stats=true /lustre/astro/troels/teaching/ahpc/venv/ahpc/bin/python sw_parallel_solution.py
checksum: 16470.993291673218
elapsed time: 0.19375168485566974 sec
Generating '/tmp/nsys-report-b74e.qdstrm'
[1/8] [========================100%] report3.nsys-rep
[2/8] [========================100%] report3.sqlite
[3/8] Executing 'nvtx_sum' stats report

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)   Style              Range
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  -------  -------------------------------
     62.5         46103239       1000   46103.2   40831.0     39018   3556885     112000.1  PushPop  integrate
     19.0         14024712       2000    7012.4    6109.0      5629    930805      21758.4  PushPop  exchange_horizontal_ghost_lines
     18.4         13606419       2000    6803.2    6079.0      5628    852759      20006.1  PushPop  exchange_vertical_ghost_lines
      0.1            74390          1   74390.0   74390.0     74390     74390          0.0  PushPop  CCCL:cub::DeviceReduce::Sum

[4/8] Executing 'osrt_sum' stats report

 Time (%)  Total Time (ns)  Num Calls   Avg (ns)     Med (ns)    Min (ns)   Max (ns)   StdDev (ns)          Name
 --------  ---------------  ---------  -----------  -----------  ---------  ---------  -----------  ----------------------
     38.5        566105651         15  37740376.7    9627771.0       2143  164757075   52899004.9  poll
.....output related to OS calls and libraries.....

[5/8] Executing 'cuda_api_sum' stats report

 Time (%)  Total Time (ns)  Num Calls   Avg (ns)    Med (ns)   Min (ns)  Max (ns)   StdDev (ns)             Name
 --------  ---------------  ---------  ----------  ----------  --------  ---------  -----------  ----------------------------
     50.3        146878293         12  12239857.8  15965730.5     34551   16220945    6820732.2  cudaMemcpyAsync
     44.2        129041961         12  10753496.8     98942.0      5869  128119089   36960595.4  cudaMalloc
      4.8         13897744       6027      2305.9      2233.0      2053      19379        656.7  cuLaunchKernel
      0.4          1275536         14     91109.7     78567.0     75352     247358      45061.5  cuModuleLoadData
      0.1           400376          9     44486.2     42744.0     31397      82612      16256.5  cuModuleUnload
      0.0           122901        440       279.3       230.0        60       4126        275.3  cuGetProcAddress_v2
      0.0            57285          2     28642.5     28642.5     25057      32228       5070.7  cudaLaunchKernel
      0.0            45998          3     15332.7      7602.0      6499      31897      14355.7  cudaMemsetAsync
      0.0            45298          1     45298.0     45298.0     45298      45298          0.0  cudaMemGetInfo
      0.0            39958         12      3329.8      3204.5      2463       5168        771.3  cudaStreamSynchronize
      0.0            13190         12      1099.2       936.0       531       2995        671.5  cudaStreamIsCapturing_v10000
      0.0             6260          2      3130.0      3130.0       160       6100       4200.2  cuModuleGetLoadingMode
      0.0             5278          2      2639.0      2639.0      2574       2704         91.9  cuInit
      0.0              721          1       721.0       721.0       721        721          0.0  cuLibraryGetKernel
      0.0              321          2       160.5       160.5       160        161          0.7  cuKernelGetName

[6/8] Executing 'cuda_gpu_kern_sum' stats report

 Time (%)  Total Time (ns)  Instances  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)                                                                                                  Name
 --------  ---------------  ---------  --------  --------  --------  --------  -----------  ----------------------------------------------------------------------------------------------------
------------------------------------------------------
     49.1         89292515       1000   89292.5   89824.0     84192     90240       1622.1  velocity_update
     46.3         84264150       1000   84264.2   84768.0     79520     85153       1537.5  elevation_update
      2.2          3995188       2000    1997.6    1984.0      1791      2272        104.7  exchange_horizontal_ghost_lines
      2.1          3824868       2000    1912.4    1920.0      1759      2144         44.4  exchange_vertical_ghost_lines
      0.2           330847         14   23631.9   25168.0     12672     26432       4525.3  cupy_copy__float64_float64
      0.0            65345          1   65345.0   65345.0     65345     65345          0.0  cupy_exp__float64_float64
      0.0            53472          2   26736.0   26736.0     26528     26944        294.2  cupy_multiply__float64_float64_float64
      0.0            38880          1   38880.0   38880.0     38880     38880          0.0  cupy_add__float64_float64_float64
      0.0            29119          3    9706.3    1408.0      1375     26336      14401.7  cupy_multiply__float_float64_float64
      0.0            16032          1   16032.0   16032.0     16032     16032          0.0  void
cub::CUB_200800_SM_750_800_860_890_900_1000_1200::DeviceReduceKernel<cub::CUB_200800_SM_750_80…
      0.0             3744          2    1872.0    1872.0      1856      1888         22.6  cupy_true_divide__float64_float_float64
      0.0             3040          2    1520.0    1520.0      1344      1696        248.9  cupy_subtract__float64_float_float64
      0.0             2944          2    1472.0    1472.0      1248      1696        316.8  cupy_arange__float_float_float64
      0.0             2432          1   2432.0    2432.0      2432      2432          0.0  void
cub::CUB_200800_SM_750_800_860_890_900_1000_1200::DeviceReduceSingleTileKernel<cub::CUB_200800…

[7/8] Executing 'cuda_gpu_mem_time_sum' stats report

 Time (%)  Total Time (ns)  Count  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)           Operation
 --------  ---------------  -----  --------  --------  --------  --------  -----------  ----------------------------
     99.6          9144887     12  762073.9  843314.0      1440    863362     241257.8  [CUDA memcpy Device-to-Host]
      0.4            38624      3   12874.7   13248.0     11936     13440        818.6  [CUDA memset]

[8/8] Executing 'cuda_gpu_mem_size_sum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)           Operation
 ----------  -----  --------  --------  --------  --------  -----------  ----------------------------
     92.275     12     7.690     8.389     0.000     8.389        2.422  [CUDA memcpy Device-to-Host]
     25.166      3     8.389     8.389     8.389     8.389        0.000  [CUDA memset]
```

Task 1: OpenACC / CuPy parallelise the program

The key challenge is to identify which parts of the code can reasonably be executed by the GPUs. For C++ and Fortran you should find suitable OpenACC directives and clauses for optimal parallelization. For Python you should use CuPy instead of NumPy. Use the profiler to determine the bottlenecks. For C++/Fortran, once the code is working, store it, and play around a bit with the #pragma and see if you can improve on your first try. For Python / CuPy once the code has been moved to the GPU consider if you can use fusing of operations or kernels to reduce the number of GPU invocations. In both cases streams may be useful for outputing data to I/O. To complete this task you should attach your code, and present your strategy for GPU paralellising the code. Add central parts of the nsys profiler output to show how your code performs and discuss the results.

Task 2: Weak scaling and asymptotic performance

Measure the weak scaling of your program using the `run_sw.sh` script. The script allows you to constrain the number of streaming multi-processors to an even number (from 2 to 14) using the CUDA Multi-Process Service daemon. Change the variables NX and NY to measure the weak scaling, you should think about and explain your choice of number of grid cells and how this map to the available vGPUs. Some key figures to note: You have 14 streaming multiprocessors (compute units or SMs) available, the maximum number of threads per thread-block is 1024 and each multiprocessor can handle at most 2048 threads simultaneously. Discuss briefly your figure.

What we often do when running on GPUs is to figure out what is the minimum size of the workload to make good use of the GPU. This can be determined by running on the full GPU while increasing the workload until the run-time increases linearly with the size of the workload. Make a plot of the asymptotic performance measured as the number of nano-seconds it takes to update a single grid-cell as a function of the number of grid cells. Based on your figure, what is a reasonable grid-size to run with on the ERDA GPU to make good use of it?