SINTEF

CONSERVATION LAWS ON GPUS

EXERCISES

André R. Brodtkorb
Researcher, Department of Mathematics and Cybernetics, SINTEF Digital
Associate Professor, OsloMet – Oslo Metropolitan University

Exercise 1: Double precision

1. Make a directory that is your own, call it <your-username>
2. Make a copy of the Jupyter notebook "<your-username>/11 MatrixAddition.ipynb"
3. Move the copy to <your-username>/MatrixAdditionDouble.ipynb
4. Change the code to add two matrices of double precision
   - Hint: double in numpy is astype(np.float64), but typically all numbers are double already
     Hint: You need to change both the kernel and the code calling the kernel
     Hint: Look at the DoublePrecision notebook for guidance
5. Measure the performance using
   ```
   import time
   tic = time.time()
   <operation>
   elapsed = time.time() - tic
   ```
6. Repeat the exercise with PyOpenCL

Exercise 2: Unit testing

1. Add unit testing to your notebooks so far
   Hint: Look at "PyTest.ipynb" to see how testing is implemented there
   Hint: you need to include the following:
   import pytest
   from ipytest import run_pytest, clean_tests
2. Make sure you test different kinds of cases
   - Default case (what usually happens)
   - Corner cases (what happens with special arguments)
3. When testing floating point operations, how low a relative and absolute tolerance to you need for the test to pass?
   Hint: Look at the documentation for approx(expected, rel=None, abs=None, nan_ok=False)
   Hint: The GPU and CPU results will not be identical: Why?

Exercise 3: Logging

- Implement logging in your notebooks
  Hint: Start with including the logging and replacing print statements with log statements
  Hint: Look at "Logging.ipynb" to see how the logger can be initialized and works

- Implement logging to file also
  - How does the output differ from file and console?
  - Make the file output have more information than the console

- Log important information such as Python version, CUDA / OpenCL version in all of your notebooks
  Hint: You can look at https://documen.tician.de/pyopencl/ for getting the OpenCL version and https://documen.tician.de/pycuda/ for CUDA

SINTEF

Exercise 4: Measuring performance

- Create a new file called Timer.py, and implement the following timer

```python
"""
Class which keeps track of time spent for a section of code
"""
class Timer(object):
    def __init__(self, tag, log_level=logging.DEBUG):
        self.tag = tag
        self.log_level = log_level
        self.logger = logging.getLogger(__name__)

    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, *args):
        self.end = time.time()
        self.secs = self.end - self.start
        self.msecs = self.secs * 1000 # millisecs
        self.logger.log(self.log_level, "%s: %f ms", self.tag, self.msecs)
```

Exercise 4: Measuring performance

- The timer can be used as follows:

```
with Timer("timer tag") as t:
        callPythonFunction(arguments)
print("The function took " + str(t.secs) + " seconds")
```

- Use the timer to time the functions you have made so far on the GPU.
  - What takes the longest time? Memory copy? Kernel execution? Upload or download?
  - How does your code scale? Does it take twice as long to run twice as large a problem?
  - How large does the problem need to be before your timing results are reproducible?

Exercise 4: Measuring performance

- GPU time can be measured using events!
- Try using the following:

```
start = cuda.Event()
end = cuda.Event()
start.record(0)
<kernel launch here>
end.record(0)
gpu_elapsed = end.time_since(start)*1.0e-3
```

- Is there a large differende between the GPU elapsed and the CPU elapsed time?

Exercise 5: Kahan summation on the GPU

- Start by making a copy of the "Kahan sum.ipynb" notebook and implement parallel Kahan summation on the GPU
  Hint: Assume the data is divisible by the number of threads
  Hint: Let each thread handle an equal portion of the data and compute a partial sum on the GPU
  Hint: Compute the total sum from the partial sums on the CPU

Example: 32 threads and 4096 data elements to sum.

- Each thread will sum 4096/32 = 128 elements.
- Then we have 32 partial sums.
- Transfer these to the CPU, and perform the final Kahan summation on the CPU

Exercise 6: Make prepared calls

- Make your kernels initialize faster by using prepared calls

```
cuda_kernel = """
__global__ void vectorAddKernel(float* c, float* a, float* b) {
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}
"""
module = cuda_compiler.SourceModule(cuda_kernel)
kernel = module.get_function("vectorAddKernel")
kernel.prepare("PPP")

grid = (n, 1, 1)
block = (1, 1, 1)
kernel.prepared_call(grid, block, c_g.gpudata, a_g.gpudata, b_g.gpudata)
```

How does this affect the performance?

SINTEF

Exercise 7: Optimizing CUDA code

1. Try using a different block size
   Try a power of 2, and a non-multiple of 2
   Example (13,3) versus (16, 4)
   What is the performance for the two versions?
2. Implement asynchronous memory transfers
   Hint: Look at set_async and get_async,
   https://documen.tician.de/pycuda/array.html#pycuda.gpuarray.GPUArray.set_async
3. Implement asynchronous kernel launches
   Hint: This requires a CUDA stream, see
   https://documen.tician.de/pycuda/driver.html#pycuda.driver.Stream and
   https://documen.tician.de/pycuda/driver.html#pycuda.driver.Function and
   https://documen.tician.de/pycuda/driver.html#pycuda.driver.Function.prepared_async_call
   Hint: look at the argument stream in the __call__ documentation
   Is the CPU time now different from the GPU time?

Exercise 8: Compilation flags

- Try experimenting with the following compiler flags:
  --maxrregcount=10
  --use_fast_math
  --gpu-architecture=compute_50 --gpu-code=sm_50,sm_52
- How do these parameters affect the accuracy and performance?

- [Full overview: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html](https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html)

Exercise 9: Mandelbrot set

- Experiment with the Mandelbrot notebook
- Change parameters and see how they affect the performance
  - Domain size
  - Block size
  - Iterations

- Change the kernel so that it handles an arbitrary domain size and block size
  Hint: Look at matrix vector product on how it handles threads which are "out of bounds"

SINTEF