Lab 2: A "Monty" Carlo Option Pricer

Objective

- Implement a monte carlo pricer using the CU API

Quick Lesson on NumbaPro CU API

- CU = Compute Unit
- A OpenCL-like API to heterogeneous parallel computing
- Instantiate a CU for 'gpu' or 'cpu'
  - `cu = CU('gpu')`
- Transfer data to the CU
  - read only
    - `d_ary = cu.input(ary)`
  - write only
    - `d_ary = cu.output(ary)`
  - read+write
    - `d_ary = cu.inout(ary)`
  - scratchpad
    - `d_ary = cu.scratch(shape=arraylen, dtype=np.float32)`
    - `d_ary = cu.scratch_like(ary)`
- Enqueue kernels to the CU
  - `cu.enqueue(kernel, ntid=number_of_threads, args=(arg0, arg1))`
- The kernel runs asynchronously
- Wait for the kernel to complete
  - `cu.wait()`

A Numpy Implementation

```python
import numpy as np
from math import sqrt, exp
from timeit import default_timer as timer

def step(dt, prices, c0, c1, noises):
    return prices * np.exp(c0 * dt + c1 * noises)

def monte_carlo_pricer(paths, dt, interest, volatility):
    c0 = interest - 0.5 * volatility ** 2
    c1 = volatility * np.sqrt(dt)

    for j in xrange(1, paths.shape[1]):
```

```python
        prices = paths[:, j - 1]
        noises = np.random.normal(0., 1., prices.size)
        paths[:, j] = step(dt, prices, c0, c1, noises)

if __name__ == '__main__':
    from driver import driver
    driver(monte_carlo_pricer)
```

The Exercise

```python
'''
Implementation of the Monte Carlo pricer using numbapro.CU with GPU target
'''
from contextlib import closing
import numpy as np
from numbapro import CU
from numbapro.parallel.kernel import builtins

def step(tid, paths, dt, prices, c0, c1, noises):
    """
    paths  --- output array for the next prices
    dt     --- the discrete time step
    prices --- array of previous prices
    c0     --- scalar constant for the math
    c1     --- scalar constant for the math
    noises --- input array of random noises
    """
    # -------- Exercise --------
    # Complete this kernel.
    # Since a kernel does not return any value,
    # the output must be stored in the "paths" array.
    # The thread ID is passed in as "tid".
    # Hints: only "paths", "prices", "noises" are arrays; others are scalar.


def monte_carlo_pricer(paths, dt, interest, volatility):
    n = paths.shape[0]
    cu = CU('gpu')

    with closing(cu): # <--- auto closing the cu
        # seed the cuRAND RNG
        cu.configure(builtins.random.seed, 1234)

        c0 = interest - 0.5 * volatility ** 2
        c1 = volatility * np.sqrt(dt)

        # Step 1. prepare data

        # -------- Exercise --------
        # allocate scratchpad memory on the device for noises
        d_noises = # fill in the RHS

        # allocate a in-out memory on the device for the initial prices
```

```python
        # "paths" is a 2-D array with the 1st dimension as number of paths
        # the 2nd dimension as the number of time step.
        d_last_paths = cu.inout(paths[:, 0])

        # -- Step 2. simulation loop --
        # compute one step for all paths in each iteration
        for i in range(1, paths.shape[1]):
            # Allocate a in-out memory for the next batch of simulated prices
            d_paths = cu.inout(paths[:, i])

            # Use builtin kernel "builtins.random.normal"
            # to generate a sequence of normal distribution.
            cu.enqueue(builtins.random.normal,      # the kernel
                       ntid=n,                      # number of threads
                       args=(d_noises,))            # arguments

            # -------- Exercise --------
            # Enqueue the "step" kernel
            # Hints: The "tid" argument is automatically inserted.


            # prepare for next step
            d_last_paths = d_paths

        # wait the the task to complete
        cu.wait()

if __name__ == '__main__':
    from driver import driver
    driver(monte_carlo_pricer)
```

Expected Result

The result should be close to the following numbers:

```
StockPrice 22.6403957688
StandardError 0.000434370525451
PaidOff 1.14039936311
OptionPrice 1.04921806448
```

Performance

Numpy implementation

- 19.74 MStep per second

NumbaPro CU + GeForce GT 650M

- 101.78 MStep per second
- **5x** speedup

NumbaPro CU + Tesla C2075

- 188.84 MStep per second
- **9.5x** speedup