Lab 2: A “Monty” Carlo Option Pricer in CU API

- Implement a monte carlo pricer using the CU API
- Guidelines and hints will be provided as we go

Step 0

Instantiate a Compute Unit

```
cu = CU('gpu')    # or 'cpu' for multicore
```

Step 1

Prepare Data Memory

- 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)`

Exercise 1

```
d_noises = # fill in the RHS
# Hints: length of array is n
```

Step 2

Enqueue kernels

- `cu.enqueue(kernel, ntid=number_of_threads, args=(arg0, arg1))`
  - `tid` (1st argument of the kernel) is not automatically populated
- Kernels run asynchronously

Exercise 2

- Enqueue the "step" kernel

Step 3

Wait for the kernel to complete

- `cu.wait()`

Step 4

Fill in the kernel

Exercise 3

- Use the Numpy version as a reference.

A Numpy Implementation

```
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]):
        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)
```

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