

# Using Python to Speed-up Applications with GPUs

Travis Oliphant & Siu Kwan Lam

March 12, 2013

# Schedule

- ▶ Introduction
  - ▶ Python
  - ▶ Numba
  - ▶ NumbaPro
- ▶ Lab 1
- ▶ Lab 2

# Introduction

## Why Python?

### Pros

- ▶ A dynamic scripting language
- ▶ Rapid development
- ▶ Rich libraries — NumPy, SciPy
- ▶ Great glue language to connect native libraries

### Cons

- ▶ Hard to parallelize because of the GIL
- ▶ Slow execution speed

# Our Solution: Speedup Python with Numba

## Numba

- ▶ An opensource **JIT** compiler for **array-oriented programming** in CPython
- ▶ Turn numerical loops into fast native code
- ▶ Maximize hardware utilization
- ▶ Just add a decorator to your compute intensive function

# Mandelbrot in Numba

```
from numba import autojit
@autojit # <--- All we need to add
def mandel(x, y, max_iters):
    i = 0
    c = complex(x,y)
    z = 0.0j
    for i in range(max_iters):
        z = z*z + c
        if (z.real*z.real + z.imag*z.imag) >= 4:
            return i
    return 255
```

- Over **170x** speedup

# Need More Speed? Use NumbaPro “CUDA Python”

- Our commerical product enables parallel compute on GPU.

```
from numbapro import cuda, uint8, f8, uint32
# use CUDA jit
@cuda.jit(argtypes=[uint8[:, :], f8, f8, f8, f8, uint32])
def mandel_kernel(image, min_x, max_x,
                  min_y, max_y, iters):
    height = image.shape[0]
    width = image.shape[1]
    pixel_size_x = (max_x - min_x) / width
    pixel_size_y = (max_y - min_y) / height
    # access thread indices
    x = cuda.threadIdx.x + \
        cuda.blockIdx.x * cuda.blockDim.x
    y = cuda.threadIdx.y + \
        cuda.blockIdx.y * cuda.blockDim.y
    # truncated ...
```

- **1255x** faster on K20

# Lab 1: Saxpy in “CUDA Python”

- ▶ Implement saxpy in “CUDA Python”
- ▶ The lab is broken down into four small exercises
- ▶ We will provide guidelines and hints along the way

# Exercise 1

## Host -> Device

- ▶ `d_ary = cuda.to_device(ary)`
  - ▶ `cudaMalloc(size);`
  - ▶ `cudaMemcpy(devary, hstary, size, cudaMemcpyHostToDevice);`

## Host -> Device (allocate only, no copy)

- ▶ `d_ary = cuda.to_device(ary, copy=False)`
  - ▶ `cudaMalloc(size);`



# Exercise 2

## Kernel Launch

- ▶ griddim: tuple of 1-2 ints
- ▶ blockdim: tuple of 1-3 ints
  - ▶ 'dim3 griddim, blockdim;
- ▶ `a_kernel[griddim, blockddim](arg0, arg1)`
  - ▶ `a_kernel<<<griddim, blockdim>>>(arg0, arg1);`

# Exercise 3

## Device -> Host

- ▶ `d_ary.to_host()`
  - ▶ `cudaMemcpy(hstary, devary, size, cudaMemcpyHostToDevice);`

# Exercise 4

## Inside the Kernel

- ▶ `cuda.threadIdx, cuda.blockIdx, cuda.blockDim`
  - ▶ `threadIdx, blockIdx, blockDim`

```
i = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
```

# “CUDA Python” Too Low-Level?

- ▶ “CUDA Python” API may be too close to CUDA-C
- ▶ We want simpler API
- ▶ An API that is closer to array-expression

# NumbaPro Compute Unit (CU) API

- ▶ Parallel heterogeneous programming for GPU, CPU, ...
- ▶ Execute kernels asynchronously

## A Saxpy Example

```
def product(tid, A, B, Prod):  
    Prod[tid] = A[tid] * B[tid]  
  
def sum(tid, A, B, Sum):  
    Sum[tid] = A[tid] + B[tid]  
  
cu = CU('gpu') # or CU('cpu')  
... # prepare data  
cu.enqueue(product, ntid=dProd.size,  
            args=(dA, dB, dProd))  
cu.enqueue(sum, ntid=dSum.size,  
            args=(dProd, dC, dSum))  
... # do something while waiting?  
cu.wait()  
  
cu.close() # destroy the compute unit
```

## 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

Questions?