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

- Global Interpretter Lock (GIL)
- Slow execution speed

Our Solution: Numba

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

Why Numba?

- Works with existing CPython
- Goal: Integration with scientific software stack
  - Numpy/SciPy/Blaze

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
    y = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim
    # truncated ...
```

- **1255x** faster on K20

NumbaPro CU API

- Compute Unit (CU) API
- Similar to OpenCL
- Portable parallel 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
```

CUDA C Memory transfer

- cudaMalloc(devptr, size)
  - devptr: output device pointer
  - size: allocation size
- cudaFree(devptr)
- cudaMemcpy(dst, src, sz, flag)
  - dst: destination pointer
  - src: souce pointer
  - sz: transfer byte size
  - flag: cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost

CUDA C Kernel Launch

```
dim3 griddim;
dim3 blockdim;
a_kernel<<<griddim, blockdim>>>(arg0, arg1, ..., argN);
```

CUDA C Putting Everything Together

```
float host_ary[n];
// host -> device
float device_ary;
cudaMalloc(&device_ary, n * sizeof(float));
cudaMemcpy(device_ary, host_ary, cudaMemcpyHostToDevice);
// launch kernel
dim3 blockdim(256);
dim3 griddim(n / 256); // assume n is multiple of 256
inplace_square<<<griddim, blockdim>>>(device_ary);
// device -> host
cudaMemcpy(host_ary, device_ary, cudaMemcpyDeviceToHost);
cudaFree(device_ary);
```

Lab 1

Saxpy in “CUDA Python”

Quick Lesson to CUDA Python

- Similar to CUDA-C

Memory Transfer

- Host->Device
  - `d_ary = cuda.to_device(ary)`
- Host->Device (allocate only, no copy)
  - `d_ary = cuda.to_device(ary, copy=False)`
- Device->Host
  - `d_ary.to_host()`

Decorate kernel

- `cuda.autojit`, `cuda.jit`

Kernel launch

- `a_kernel[griddim, blockddim](arg0, arg1)`
- `griddim`: tuple of 1-2 ints
- `blockdim`: tuple of 1-3 ints

Inside the kernel

- `threadIdx`, `blockIdx`, `blockDim` -> `cuda.threadIdx`, `cuda.blockIdx`, `cuda.blockDim`
- `i = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x`

Lab 2

A “Monty” Carlo Option Pricer in NumbaPro 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
  - `d_ary = cu.input(ary)`
  - `d_ary = cu.output(ary)`
  - `d_ary = cu.inout(ary)`
  - `d_ary = cu.scratch(shape=arraylen, dtype=np.float32)`
  - `d_ary = cu.scratch_like(ary)`

Continue. . .

- 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

```
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)
```