NumbaPro

- Enables parallel programming in Python
- Support various entry points:
  - Low-level (CUDA-C like) programming language
  - High-level array oriented interface
  - CUDA library bindings
- Also support multicore CPU
  - And more hardware architectures in the future.

NumbaPro "CUDA Python"

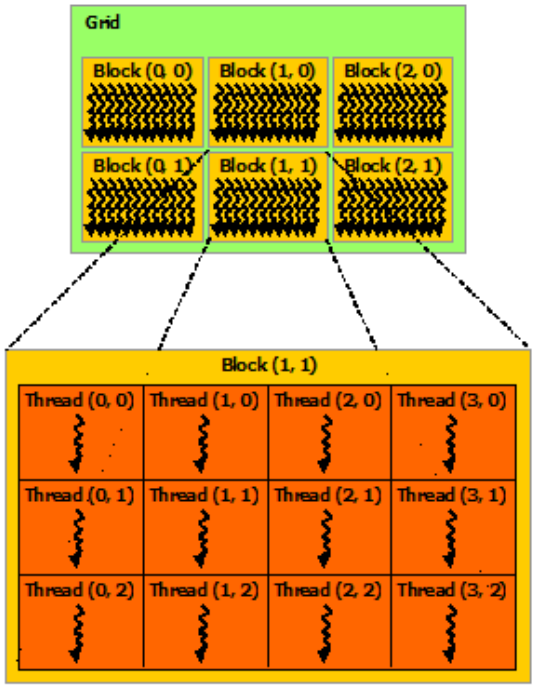
```
from numbapro import cuda, float32, void

@cuda.jit(void(float32[:,:], float32[:,:], float32[:,:]))
def square_matrix_mult(A, B, C):
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    bw = cuda.blockDim.x
    bh = cuda.blockDim.y

    x = tx + bx * bw
    y = ty + by * bh
    n = C.shape[0]

    if x >= n or y >= n:
        return

    cs = 0
    for i in range(n):
        cs += A[y, i] * B[i, x]
    C[y, x] = cs
```

Square matrix multiplication

NumbaPro "CUDA Python"

```
from numbapro import cuda, float32, void

@cuda.jit(void(float32[:,:], float32[:,:], float32[:,:]))
def square_matrix_mult(A, B, C):
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    bw = cuda.blockDim.x
    bh = cuda.blockDim.y

    x = tx + bx * bw
    y = ty + by * bh
    n = C.shape[0]

    if x >= n or y >= n:
        return

    cs = 0
    for i in range(n):
        cs += A[y, i] * B[i, x]
    C[y, x] = cs
```

Determine thread Identity



NumbaPro "CUDA Python"

```
from numbapro import cuda, float32, void

@cuda.jit(void(float32[:,:], float32[:,:], float32[:,:]))
def square_matrix_mult(A, B, C):
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    bw = cuda.blockDim.x
    bh = cuda.blockDim.y

    x = tx + bx * bw
    y = ty + by * bh
    n = C.shape[0]

    if x >= n or y >= n:
        return

    cs = 0
    for i in range(n):
        cs += A[y, i] * B[i, x]
    C[y, x] = cs
```

Map threads to matrix coordinate

NumbaPro "CUDA Python"

```
from numbapro import cuda, float32, void

@cuda.jit(void(float32[:,:], float32[:,:], float32[:,:]))
def square_matrix_mult(A, B, C):
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    bw = cuda.blockDim.x
    bh = cuda.blockDim.y

    x = tx + bx * bw
    y = ty + by * bh
    n = C.shape[0]

    if x >= n or y >= n:
        return

    cs = 0
    for i in range(n):
        cs += A[y, i] * B[i, x]
    C[y, x] = cs
```

Thread inside matrix?

NumbaPro "CUDA Python"

```
from numbapro import cuda, float32, void

@cuda.jit(void(float32[:,:], float32[:,:], float32[:,:]))
def square_matrix_mult(A, B, C):
    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y
    bx = cuda.blockIdx.x
    by = cuda.blockIdx.y
    bw = cuda.blockDim.x
    bh = cuda.blockDim.y

    x = tx + bx * bw
    y = ty + by * bh
    n = C.shape[0]

    if x >= n or y >= n:
        return

    cs = 0
    for i in range(n):
        cs += A[y, i] * B[i, x]
    C[y, x] = cs
```

Compute one element.

Launch NxN threads for NxN matrix

Launch CUDA Kernel

```
@cuda.jit(void(float32[:,:], float32[:,:], float32[:,:]))
def square_matrix_mult(A, B, C):
    ...

griddim = 100, 100
blockdim = 32, 32
square_matrix_mult[griddim, blockdim](A, B, C)
```

Launch (100 x 32)^2 = 3200^2 threads for 3200 x 3200 matrix

Equivalent CUDA-C

```
dim3 griddim(100, 100);
dim3 blockdim(32, 32);
square_matrix_mult<<<griddim, blockdim>>>(A, B, C);
```

Higher-level Entry Points

So far, the API is quite low-level.

We will go through some higher-level entry points in the lessons.

Lesson 1

SAXPY with Vectorize

@vectorize

- Creates elementwise operation from a scalar function
- Produces a NumPy universal function (ufunc).
- numpy.add is a ufunc
- Eliminate most of CUDA specific info
  - griddim, blockdim are computed for you

The Scalar Function Core

- All arguments are scalar
- Returns a scalar value as the output

Writing a SAXPY function

SAXPY computes

$$a\,X + Y$$

where X and Y are vectors of equal length.

```
@vectorize([float32(float32, float32, float32)],
           target='gpu')
def vec_saxpy(a, x, y):
    return a * x + y
```

@vectorize

```
@vectorize([float32(float32, float32, float32)],
           target='gpu')
def vec_saxpy(a, x, y):
    return
```

List of function type signatures

@vectorize

```
@vectorize([float32(float32, float32, float32)],
           target='gpu')
def vec_saxpy(a, x, y):
    return
```

Code generation target:
“cpu”, “parallel”, “gpu”

@vectorize

```
@vectorize([                                    float32)],
           target='gpu')
def vec_saxpy(a, x, y):
    return a * x + y
```

A scalar function

Args: a, x, y are float32
Returns a float32

Calling a vectorize function

- Use as regular NumPy ufunc
  - Applies to regular NumPy arrays
  - Auto host->device and device->host transfer
  - Auto calculate griddim and blockdim

```
x = numpy.arange(NELEM, dtype='float32')
y = numpy.arange(NELEM, dtype='float32')
vecout = vec_saxpy(a, x, y)
```

SAXPY in CUDA Python

```
@cuda.jit(void(float32, float32[:], float32[:], float32[:]))
def saxpy(a, x, y, out):
    # Short for cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
    i = cuda.grid(1)
    # Map i to array elements
    if i >= out.size:
        # Out of range?
        return
    # Do actual work
    out[i] = a * x[i] + y[i]
```

Memory transfer

Explicit memory transfer is optional.

Host->Device:

```
device_array = cuda.to_device(host_array)
```

Device Allocation:

```
device_array =
        cuda.device_array_like(device_or_host_array)
```

Note: behaves like numpy.empty_like

Device->Host:

```
host_array = device_array.copy_to_host()
```

Controlling Memory Transfer

```
def task2():
    a = numpy.float32(2.)                    # Force value to be float32
    x = numpy.arange(NELEM, dtype='float32')
    y = numpy.arange(NELEM, dtype='float32')

    ### Task2 ###
    # a) Complete the memory transfer for x -> dx, y -> dy
    # b) Allocate device memory for dout
    # c) Transfer for out <- dout
```

host -> device

```
    griddim = NUM_BLOCKS
    blockdim = NUM_THREADS
    saxpy[griddim, blockdim](a, dx, dy, dout)
```

device -> host

```
    print "out =", out
```

Controlling Memory Transfer

```
def task2():
    a = numpy.float32(2.)                # Force value to be float32
    x = numpy.arange(NELEM, dtype='float32')
    y = numpy.arange(NELEM, dtype='float32')

    ### Task2 ###
    # a) Complete the memory transfer for x -> dx, y -> dy
    # b) Allocate device memory for dout
    # c) Transfer for out <- dout
    dx = cuda.to_device(x)
    dy = cuda.to_device(y)
    dout = cuda.device_array_like(x)

    griddim = NUM_BLOCKS
    blockdim = NUM_THREADS
    saxpy[griddim, blockdim](a, dx, dy, dout)

    device -> host

    print "out =", out
```

Controlling Memory Transfer

```
def task2():
    a = numpy.float32(2.)                  # Force value to be float32
    x = numpy.arange(NELEM, dtype='float32')
    y = numpy.arange(NELEM, dtype='float32')

    ### Task2 ###
    # a) Complete the memory transfer for x -> dx, y -> dy
    # b) Allocate device memory for dout
    # c) Transfer for out <- dout
    dx = cuda.to_device(x)
    dy = cuda.to_device(y)
    dout = cuda.device_array_like(x)

    griddim = NUM_BLOCKS
    blockdim = NUM_THREADS
    saxpy[griddim, blockdim](a, dx, dy, dout)

    out = dout.copy_to_host()
    print "out =", out
```

Why manual transfer?

- As an optimization
- Control device memory usage
- Allow reusing of memory

Lesson 2

cuFFT convolution

FFT Convolution

Image filter using FFT convolution with cuFFT.

*convolved = IFFT(FFT(image) * FFT (response))*

cuFFT API

The cuFFT object (`cufft` in the code) has:

Forward FFT

```
cufft.fft(in_array, out_array)
cufft.fft_inplace(inout_array)
```

Inverse FFT

```
cufft.ifft(in_array, out_array)
cufft.ifft_inplace(inout_array)
```

Doing a Inplace Convolution

```
@vectorize(['complex64(complex64, complex64)'], target='gpu')
def vmult(a, b):
    """Element complex64 multiplication
    """
    return a * b

def task1(cufft, d_image_complex, d_response_complex):
```

Forward FFT of image and response arrays

Elementwise image and response arrays in frequency domain

Inverse FFT the product

```
    # At this point, we have applied the filter onto d_image_complex
    return  # Does not return anything
```

Doing a Inplace Convolution

```
@vectorize(['complex64(complex64, complex64)'], target='gpu')
def vmult(a, b):
    """Element complex64 multiplication
    """
    return a * b

def task1(cufft, d_image_complex, d_response_complex):
    cufft.fft_inplace(d_image_complex)
    cufft.fft_inplace(d_response_complex)
```

Elementwise image and response arrays in frequency domain

Inverse FFT the product

```
    # At this point, we have applied the filter onto d_image_complex
    return  # Does not return anything
```

Doing a Inplace Convolution

```
@vectorize(['complex64(complex64, complex64)'], target='gpu')
def vmult(a, b):
    """Element complex64 multiplication
    """
    return a * b

def task1(cufft, d_image_complex, d_response_complex):
    cufft.fft_inplace(d_image_complex)
    cufft.fft_inplace(d_response_complex)

    vmult(d_image_complex, d_response_complex, out=d_image_complex)
```

Inverse FFT the product

```
    # At this point, we have applied the filter onto d_image_complex
    return  # Does not return anything
```

Doing a Inplace Convolution

```
@vectorize(['complex64(complex64, complex64)'], target='gpu')
def vmult(a, b):
    """Element complex64 multiplication
    """
    return a * b

def task1(cufft, d_image_complex, d_response_complex):
    cufft.fft_inplace(d_image_complex)
    cufft.fft_inplace(d_response_complex)

    vmult(d_image_complex, d_response_complex, out=d_image_complex)

    cufft.ifft_inplace(d_image_complex)

    # At this point, we have applied the filter onto d_image_complex
    return  # Does not return anything
```

Lesson 3

JIT Linking

CUDA JIT Linking

- Use CUDA-C code inside NumbaPro
- Compile CUDA-C code into relocatable device code
- NumbaPro use CUDA JIT Linker to combine its generated code with a precompiled library

Use of JIT Linking

- Connect to missing features
  - NumbaPro is still young
- Connect to CUDA-C only features
- Reusing existing CUDA-C code

NumbaPro Python code

```
bar = cuda.declare_device('bar', 'int32(int32, int32)')
linkfile = "../data/jitlink.o"

@cuda.jit('void(int32[:], int32[:])', link=[linkfile])
def foo(inp, out):
    i = cuda.grid(1)
    out[i] = bar(inp[i], 2)
```

NumbaPro Python code

```
bar = cuda.declare_device('bar', 'int32(int32, int32)')
linkfile = "../data/jitlink.o"

@cuda.jit(argtypes=[int32[:], int32[:]], link=[linkfile])
def foo(inp, out):
    i = cuda.grid(1)
    out[i] = bar(inp[i], 2)
```

Declare external device function in Python

NumbaPro Python code

```
bar = cuda.declare_device('bar', 'int32(int32, int32)')
linkfile = "../data/jitlink.o"

@cuda.jit('void(int32[:], int32[:])', link=[linkfile])
def
    i = cuda.grid(1)
    out[i] = bar(inp[i], 2)
```

Precompiled object file

NumbaPro Python code

```
bar = cuda.declare_device('bar', 'int32(int32, int32)')
linkfile = "../data/jitlink.o"

@cuda.jit('void(int32[:], int32[:])', link=[linkfile])
def foo(inp, out):
    i = cuda.grid(1)
    out[i] = bar(inp[i, 2)
```

Add library dependencies to the CUDA kernel

NumbaPro Python code

```
bar = cuda.declare_device('bar', 'int32(int32, int32)')
linkfile = "...

@cuda.jit('void(int32[:], int32[:])', link=[linkfile])
def foo(inp, out):
    i = cuda.grid(1)
    out[i] = bar(inp[i], 2)
```

Use external function

CUDA-C code

```
extern "C" {

__device__
int bar(int* retval, int a, int b){



    return 0;
}

}
```

CUDA-C code

```
extern "C" {

__device__
int bar(int* retval, int a, int b){

}

}
```

NumbaPro expects return value to be passed as the first argument

CUDA-C code

```
extern "C" {

__device__
int bar(int* retval, int a, int b){

    return
}

}
```

Actual arguments follows

CUDA-C code

```
extern "C" {

__device__
int bar(int* retval, int a, int b){

    return 0;
}

}
```

Return value indicates status.

Return 0 for success.

Other return codes are possible to indicate builtin errors.

How to compile

nvcc -arch=sm_20 -dc yourcode.cu

- Support only CC 2.0 or above
- -dc flag triggers relocatable device code

Example

```
#include <cstdio>

extern "C" {

__device__
int bar(int* retval, int a, int b){
    /* Fill this function with anything */
    printf("inside foo: thread=%d a=%d b=%d\n",
           threadIdx.x, a, b);
    *retval = a * b;
    /* Return 0 to indicate success */
    return 0;
}

}
```

Q & A

Thank You

NumbaPro is Part of
Anaconda Accelerate.
Visit **continuum.io**

Anaconda