Extending Python for High-Performance Data-Parallel Programming

Siu Kwan Lam

March 24, 2014

Python for Data Analytics

Why Python?

- High-level scripting language
  - Dynamic-typed, Garbage Collected
- Rapid development
- Rich libraries
  - Array: NumPy, Blaze
  - Science: SciPy, Scikit-Learn
  - Visualization: Matplotlib, Boken
- Great glue language

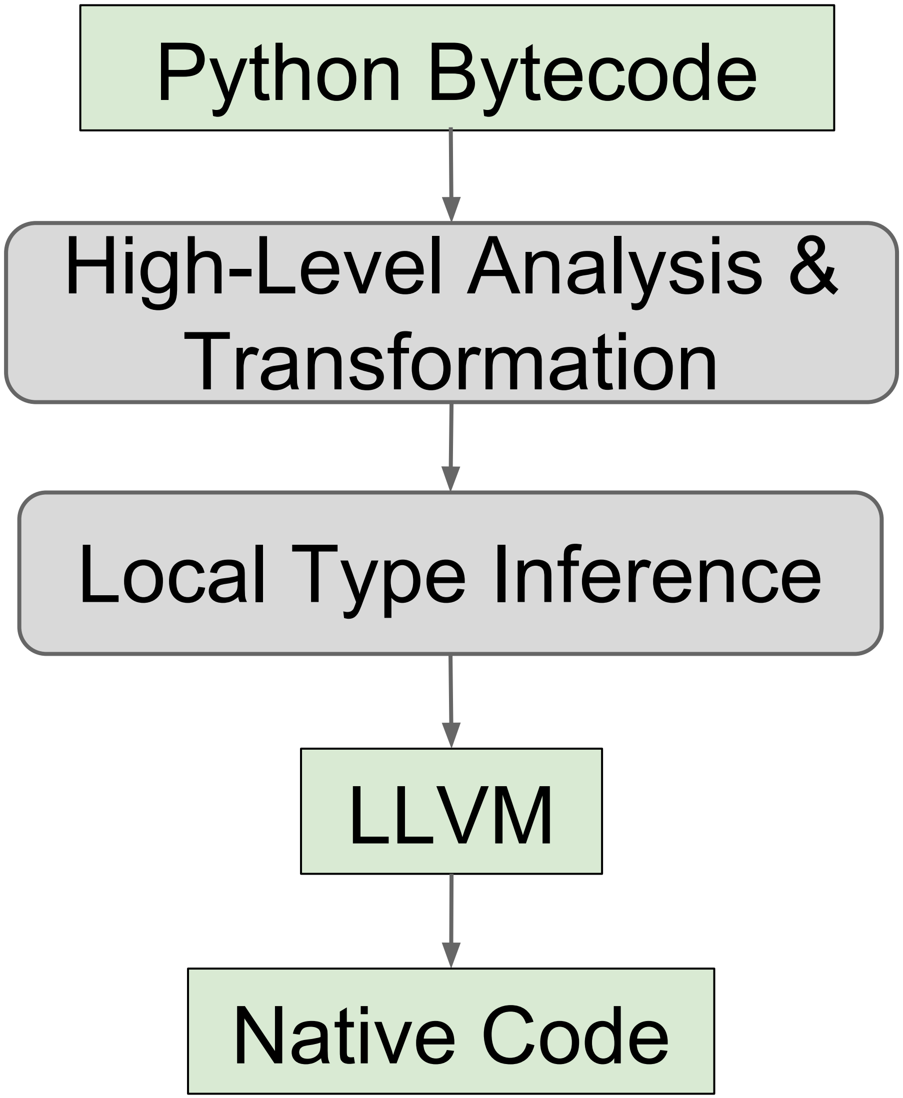
But...

- Hard to parallelize
  - Global Interpreter Lock
- Slow execution

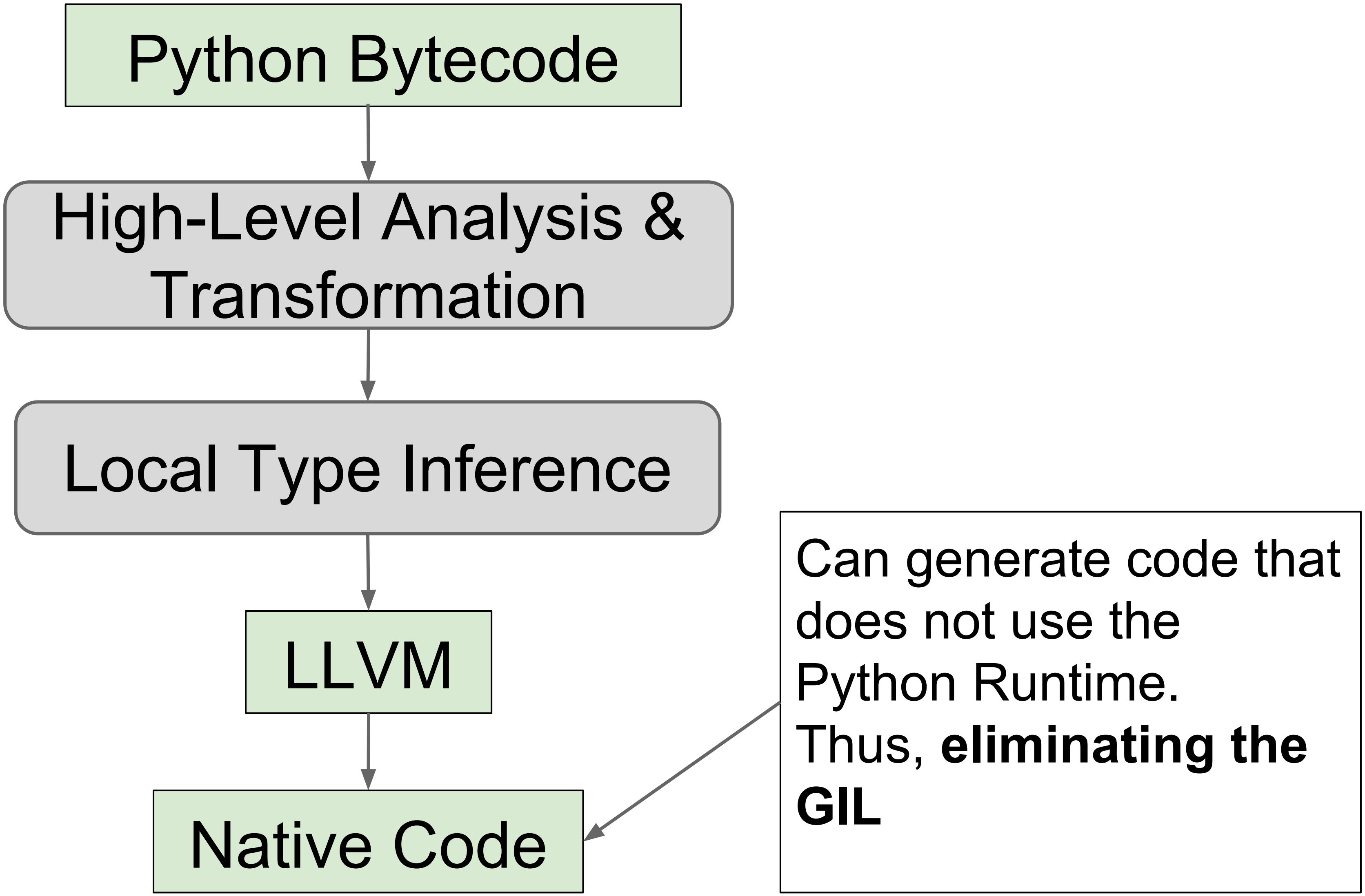
Our Solution: Numba

- Open-source JIT compiler for CPython
- Numerical loop to fast native code
- Work seamlessly with NumPy arrays

Numba Compilation Pipeline



Numba Compilation Pipeline



Numba Example: Sum 2D Array

```python
from numba import jit
from numpy import arange

@jit
def sum2d(arr):
    M, N = arr.shape
    result = 0.0
    for i in range(M):
        for j in range(N):
            result += arr[i,j]
    return result

a = arange(9).reshape(3,3)
print(sum2d(a))
```

Specialize parameter type for var `a`

NumbaPro

- Enables parallel programming in Python
- Support various entry points:
  - Low-level CUDA Python
    - Just released an open-source version to Numba
  - 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

High-Level APIs

```
@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
```

High-Level APIs

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

def task1(cufft, d
    cufft.fft_inpl
    cufft.fft_inpl

    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
```

@vectorize turns a **scalar function** to an elementwise **array functions**

High-Level APIs

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

def task1(cufft, d_image_comple
    cufft.fft_inplace(d_image_c
    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
```

Support multiple targets:
cpu, parallel, gpu

High-Level APIs

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

    cufft.ifft_inplace(d_image_complex)

    # At this point, we have applied the filter o                   ex
    return  # Does not return anything
```

CUDA library support
This uses cuFFT

Also, supporting: cuBlas, cuRand, cuSparse

We can do better!

- Still need CUDA specific knowledge
- Needs higher-level abstraction

DARPA GPU Project (STTR-D13B-004)

- Started about a month ago
- Develop high-level easy to use programming language for GPUs
- Partner with Dr. Alex Dimakis at UT Austin

Project Goals

- Provide new language features as an extension to NumbaPro
- Portable parallel algorithms
- Especially for sparse problems:
    - graphs, sparse matrices

What we did...

- Try to implement a Sparse PCA in NumbaPro
- Identify
  - common patterns
  - shortcomings
  - missing features

Sparse PCA (CPU)

```
def spca_unopt(Vd, epsilon=0.1, d=3, k=10):
    p = Vd.shape[0]
    numSamples = (4. / epsilon) ** d

    opt_x = np.zeros((p, 1))
    opt_v = -np.inf

    C = np.random.randn(d, numSamples)

    for i in np.arange(1, numSamples + 1):
        c = C[:, i - 1:i]
        c = c / np.linalg.norm(c)
        a = Vd.dot(c)

        I = np.argsort(a, axis=0)
        val = np.linalg.norm(a[I[-k:]])

        if val > opt_v:
            opt_v = val
            opt_x = np.zeros((p, 1))
            opt_x[I[-k:]] = a[I[-k:], :] / val

    return opt_x
```

Sparse PCA (CPU)

```
def spca_unopt(Vd, epsilon=0.1, d=3, k=10):
    p = Vd.shape[0]
    numSamples = (4. / epsilon) ** d

    opt_x = np.zeros((p, 1))
    opt_v = -np.inf

    C = np.random.randn(d, numSamples)

    for i in np.arange(1, numSamples + 1):
        c = C[:, i - 1:i]
        c = c / np.linalg.norm(c)
        a = Vd.dot(c)

        I = np.argsort(a, axis=0)
        val = np.linalg.norm(a[I[-k:]])

        if val > opt_v:
            opt_v = val
            opt_x = np.zeros((p, 1))
            opt_x[I[-k:]] = a[I[-k:], :] / val

    return opt_x
```

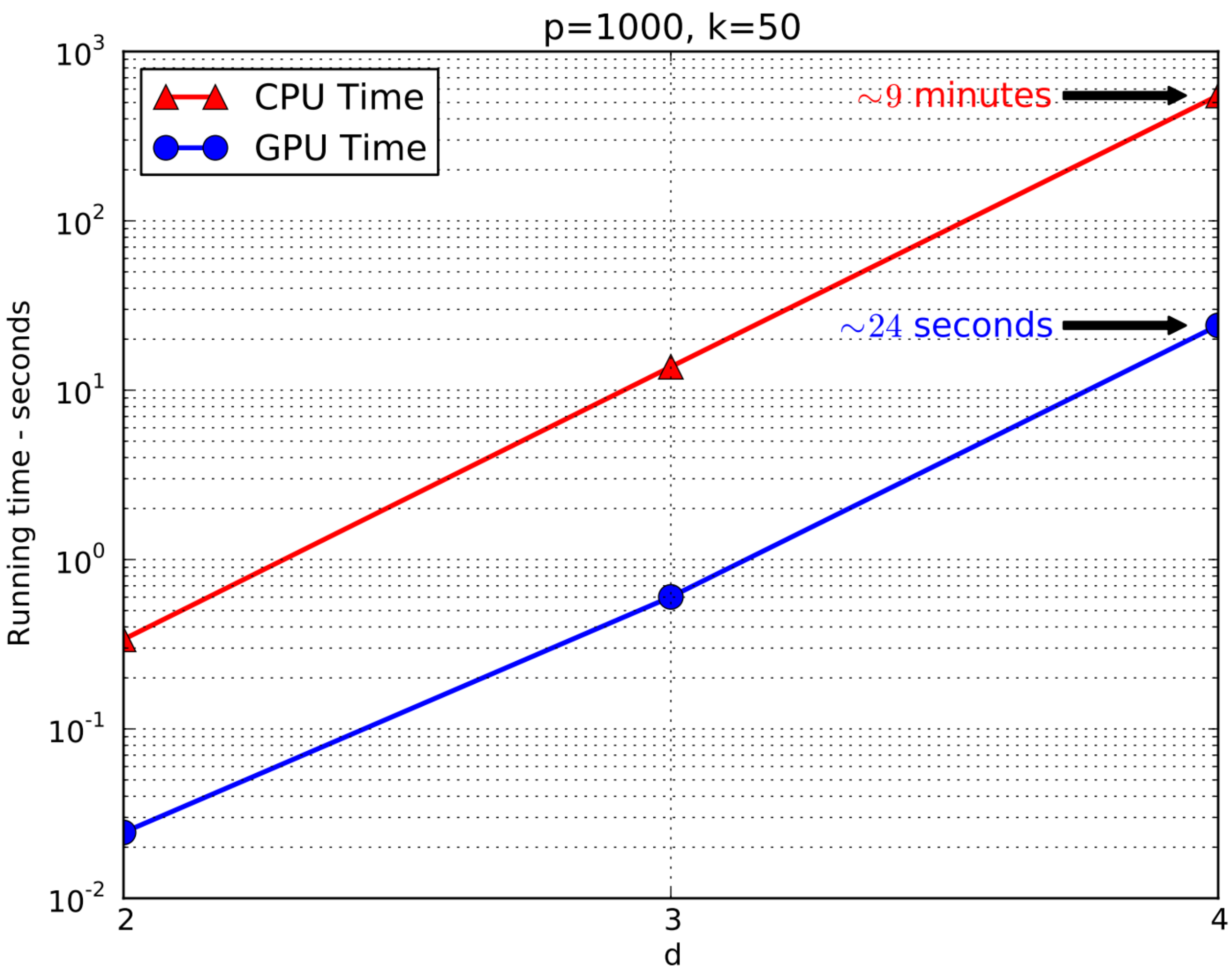
Embarrassingly Parallel

Sparse PCA (GPU)

```python
def spca(Vd, epsilon=0.1, d=3, k=10):
    p = Vd.shape[0]
    initNumSamples = int((4. / epsilon) ** d)
    maxSize = 32000
    opt_x = np.zeros((p, 1))
    opt_v = -np.inf

    dVd = cuda.to_device(Vd)

    remaining = initNumSamples
    custr = cuda.stream()
    prng = curand.PRNG(stream=custr)

    while remaining:
        numSamples = min(remaining, maxSize)
        remaining -= numSamples

        dA = cuda.device_array(shape=(Vd.shape[0], numSamples), order='F')
        dI = cuda.device_array(shape=(k, numSamples), dtype=np.int16, order='F')
        daInorm = cuda.device_array(shape=numSamples, dtype=np.float64)
        dC = cuda.device_array(shape=(d, numSamples), order='F')

        prng.normal(dC.reshape(dC.size), mean=0, sigma=1)
        norm_random_nums[calc_ncta1d(dC.shape[1], 512), 512, custr](dC, d)
        batch_matmul[numSamples, 512, custr](dVd, dC, dA)
        batch_k_selection[numSamples, Vd.shape[0], custr](dA, dI, k)
        batch_scatter_norm[calc_ncta1d(numSamples, 512), 512, custr](dA, dI,
                                                                      daInorm)

        aInorm = daInorm.copy_to_host(stream=custr)
        custr.synchronize()
        for i in xrange(numSamples):
            val = aInorm[i]
            if val > opt_v:
                opt_v = val
                opt_x.fill(0)
                a = gpu_slice(dA, i).reshape(p, 1)
                Ik = gpu_slice(dI, i).reshape(k, 1)
                aIk = a[Ik]
                opt_x[Ik] = (aIk / val)

        del dA, dI, daInorm, dC
    return opt_x
```

- Longer code
- Complicated
- Not scalable
- Uses
  - cuRAND
  - Batch matrix mult
  - **K-selection**
  - Scatter
  - Slicing
  - Custom elementwise functions

Sparse PCA Benchmark (GTX 780Ti)



Realizations...

We need:

- Need more generic high-level array functions
  - map, reduce, zipwith
- Need builtin library functions
  - k-select, sort, scatter, random

Can Learn from...

- Nvidia Nova
- Halide
- Haskell Accelerate
- C++ Thrust

Can Learn from...

- Nvidia Nova
- Halide
- Haskell Accelerate
- C++ Thrust

They all have a functional/dataflow style

Potentially...

- Build dataflow graph at runtime
  - at runtime, the imperative control-flow is flattened
  - *map(f, map(g, array))*

- Optimize by fusion
  - Function fusion
    - *map(f, map(g, array)) == map(f.g, array)*
  - Storage fusion
    - remove & reuse temporaries

Parallel Primitives

- map
- zipwith
- reduce
- scan
- scatter
- sort
- k-select
- random
- *(enough?)*

Parallel Primitives

- map
- zipwith
- reduce
- scan
- scatter
- sort
- k-select
- random
- *(enough?)*

And, library calls as extensions?

Manual Tuning?

- Leave room for manual tuning
  - Require expressing optimization and scheduling.
- Can we do compiler optimization in a reasonable time?
- Is tuning by expert still better?
- *f.g == fuse(f, g)*

Q & A

Thank You

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

Anaconda

Backup Slides

@vectorize

```
@vectorize([float32(float32, float32, float32)],
           target='gpu')
def vec_saxpy(a, x, y):
    ### Task 1 ###
    # Comple
    # Hint: this is a scalar function of
    #       float32(float32 a, float32 x, float32 y)
```

List of function type signatures

@vectorize

```
@vectorize([float32(float32, float32, float32)],
           target='gpu')
def vec_saxpy(a, x, y):
    ### Task
    # Comple
    # Hint:
    #                               , float32 y)
```

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

@vectorize

A scalar function

```
@vectorize([f                                    at32)],
           target='gpu')
def vec_saxpy(a, x, y):
    ### Task 1 ###
    # Complete the vectorize version
    # Hint: this is a scalar function of
    #        float32(float32 a, float32 x, float32 y)
```

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

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"

@cu                                                    le])
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"

@cu                                32[:])', 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.