




CUSIGNAL - GPU ACCELERATED SCIPY SIGNAL

Adam Thompson | Senior Solutions Architect | adamt@nvidia.com

 @adamlikesai



cuSignal is built as a GPU accelerated version of the popular SciPy Signal library

Most of the coding has leveraged CuPy - GPU accelerated NumPy

In certain cases, we have implemented custom CUDA kernels using Numba - more on this (pros and cons!) later



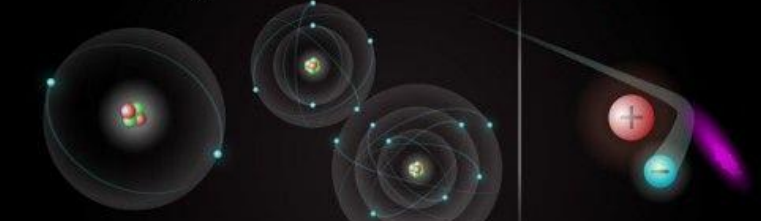

GitHub Repo:

<https://github.com/rapidsai/cusignal>


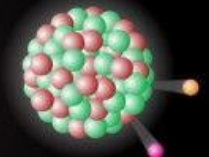

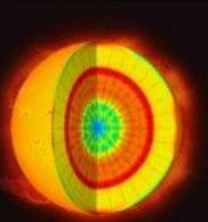



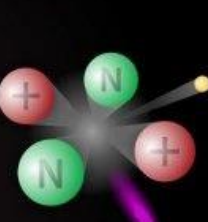


BACKGROUND AND MOTIVATIONS

THE FOUR FUNDAMENTAL FORCES OF THE UNIVERSE

| | |
|--|--|
| <h3>Weak Nuclear Force</h3>  <p>Converting protons into neutrons When two protons collide and fuse, a disruption in the weak nuclear force emits a positron and neutrino, which converts one of the positively charged proton to a neutrally charged Neutron. Without the weak nuclear force converting protons into neutrons, certain complex nuclei cannot form.</p> <p>Releasing radiation Heavy atoms have an imbalance of protons and neutrons, so the weak nuclear force converts protons to neutrons releasing radiation.</p> | <h3>Gravity</h3>  <p>Adding motion to the Universe Gravity forms stars, planets, and moons, and forces these objects to spin on an axis and move along an orbital path. The planets appear to be orbiting the center of the Sun, but the Sun and planets all orbit a shared center of mass. Planets with enough mass can develop orbiting moons or rings of debris.</p> <p>Creating energy Gravity is the force that creates pressure and fusion energy in the core of stars allowing them to burn for millions of years.</p> |
| <h3>Electromagnetic Force</h3>  <p>Forming atoms and molecules The electromagnetic force pulls negatively charged electrons into bound orbits around positively charged nuclei to form atoms and molecules. As a gas cools, electrons will find their way into the presence of atomic nuclei. Larger nuclei with a greater positive charge pull in more electrons until atoms and molecules have a balance of charges.</p> <p>Generating light When a negative electron interacts with a positive proton, the electromagnetic force adds energy to the electron generating a photon.</p> | <h3>Strong Nuclear Force</h3>  <p>Binding protons in atomic nuclei Positively charged particles naturally repel each other, it takes an extreme amount of force to hold protons together. The strong nuclear force overcomes the repulsion between protons to hold together atomic nuclei. Without the strong nuclear force, complex nuclei cannot form.</p> <p>Breaking the bond Enormous energy is released as gamma rays and neutrinos when the strong nuclear force is broken between protons and neutrons.</p> |

THE FOUR FUNDAMENTAL FORCES OF THE UNIVERSE

| | |
|--|---|
| <h3>Weak Nuclear Force</h3>  <p>Converting protons into neutrons</p> <p>When two protons collide and fuse, a disruption of the weak nuclear force releases a positron and a neutrino, which converts one of the protons into a neutrally charged Neutron. Without the weak nuclear force, certain complex nuclei cannot form.</p>  <p>Releasing radiation</p> <p>Heavy atoms have an imbalance of protons and neutrons, so the weak nuclear force converts protons to neutrons releasing radiation.</p> | <h3>Gravity</h3>  <p>Adding motion to the Universe</p> <p>Gravity forms stars, planets, and moons, and forces all objects to spiral inward toward an axis and move along an orbital path. The planets appear to orbit the center of the Sun, but the Sun and planets all orbit a shared center of mass. Planets with enough mass can develop orbiting moons of their own.</p>  <p>Creating energy</p> <p>Gravity is the force that creates pressure and fusion energy in the core of stars allowing them to burn for millions of years.</p> |
| <h3>Electromagnetic Force</h3>  <p>Forming atoms and molecules</p> <p>The electromagnetic force pulls negatively charged electrons into bound orbits around positively charged nuclei to form atoms and molecules. As a gas cools, electrons will find their way into the presence of atomic nuclei. Larger nuclei with a greater positive charge pull in more electrons until atoms and molecules have a balance of charges.</p>  <p>Generating light</p> <p>When a negative electron interacts with a positive proton, the electromagnetic force adds energy to the electron generating a photon.</p> | <h3>Strong Nuclear Force</h3>  <p>Binding protons in atomic nuclei</p> <p>Positively charged particles naturally repel each other. It takes an extremely strong force of force to hold protons together. The strong nuclear force overcomes the repulsion between protons to hold together atomic nuclei. Without the strong nuclear force, complex nuclei cannot form.</p>  <p>Breaking the bond</p> <p>Enormous energy is released as gamma rays and neutrinos when the strong nuclear force is broken between protons and neutrons.</p> |

THE FOUR FUNDAMENTAL FORCES OF THE UNIVERSE



Remember everyone, of the four fundamental forces of the universe, only one is safe to manipulate at home! Grab an SDR, @gnuradio, and have fun!

Electromagnetic Force

Forming atoms and molecules
The electromagnetic force pulls negatively charged electrons into bound orbits around positively charged nuclei to form atoms and molecules. As a gas cools, electrons will find their way into the presence of atomic nuclei. Larger nuclei with a greater positive charge pull in more electrons until atoms and molecules have a balance of charges.

Generating light
When a negative electron interacts with a positive proton, the electromagnetic force adds energy to the electron generating a photon.

Strong Nuclear Force

Binding protons in atomic nuclei
Positively charged particles naturally repel each other, it takes an extreme amount of force to hold protons together. The strong nuclear force overcomes the repulsion between protons to hold together atomic nuclei. Without the strong nuclear force, complex nuclei cannot form.

Breaking the bond
Enormous energy is released as gamma rays and neutrinos when the strong nuclear force is broken between protons and neutrons.

A Mystery Frequency Disrupted Car Fobs in an Ohio City, and Now Residents Know Why



By Saturday afternoon, City Councilman Chris Glassburn announced that the mystery had been solved: The source of the problem was a homemade battery-operated device designed by a local resident to alert him if someone was upstairs when he was working in his basement. It did so by turning off a light.

“He has a fascination with electronics,” Mr. Glassburn said, adding that the resident has special needs and would not be identified to protect his privacy.

The inventor and other residents of his home had no idea that the device was wreaking havoc on the neighborhood, he said, until Mr. Glassburn and a volunteer with expertise in radio frequencies knocked on the door.

“The way he designed it, it was persistently putting out a 315 megahertz signal,” Mr. Glassburn said. That is the frequency many car fobs and garage door openers rely on.

UNITED STATES FREQUENCY ALLOCATIONS

THE RADIO SPECTRUM



ALLOCATION USAGE DESIGNATION

| SERVICE | EXAMPLE | DESCRIPTION |
|-----------|---------|------------------------------------|
| Primary | F1D2 | Upper Label |
| Secondary | M4B | Not Capital with lower case letter |

The data in this report is prepared in accordance with the Table of Frequency Allocations and the FCC's 47 CFR, 2.106, which sets out the table of frequency allocations for the United States. This data is provided for informational purposes only and is not intended to be used for any other purpose. For more information, visit www.fcc.gov.

U.S. DEPARTMENT OF COMMERCE
National Telecommunications and Information Administration
Office of Spectrum Management

NTIA
JANUARY 2014



UNITED STATES FREQUENCY ALLOCATIONS

THE RADIO SPECTRUM

RADIO SERVICES COLOR LEGEND

- AIR NAUTICAL MOBILE
- FIXED SATELLITE
- RADIO AERONAUTICS
- AIR NAUTICAL MOBILE SATELLITE
- LAND MOBILE
- RADIO EXTENSION SATELLITE
- AIR NAUTICAL RADIO NAVIGATION
- LAND MOBILE SATELLITE
- RADIOLOCATION
- AMATEUR
- MARITIME MOBILE
- RADIOLOCATION SATELLITE
- AMATEUR SATELLITE
- MARITIME MOBILE SATELLITE
- RADIO NAVIGATION
- BROADCASTING
- MARITIME RADIO NAVIGATION
- RADIO NAVIGATION SATELLITE
- BROADCASTING SATELLITE
- METEOROLOGICAL
- SPACE OPERATION
- EARTH EXPLORATION SATELLITE
- METEOROLOGICAL SATELLITE
- SPACE RESEARCH
- FIXED
- MOBILE
- STANDARD FREQUENCY AND TIME SIGNAL
- FIXED SATELLITE
- MOBILE SATELLITE
- STANDARD FREQUENCY AND TIME SIGNAL SATELLITE

ACTIVITY CODE

- FEDERAL EXCLUSIVE
- FEDERAL/NON-FEDERAL SHARED
- NON-FEDERAL EXCLUSIVE

ALLOCATION USAGE DESIGNATION

| SERVICE | EXAMPLE | DESCRIPTION |
|-----------|---------|-----------------------------------|
| Primary | F1D2 | Upper/Lower |
| Secondary | M4B | Not Capital with Same case letter |

The table is a partial representation of the information contained in the Table of Frequency Allocations and is for informational purposes only. It is not intended to be used as a legal document. For more information, please refer to the Table of Frequency Allocations. Further, it is not intended to be used as a legal document for any other purpose.

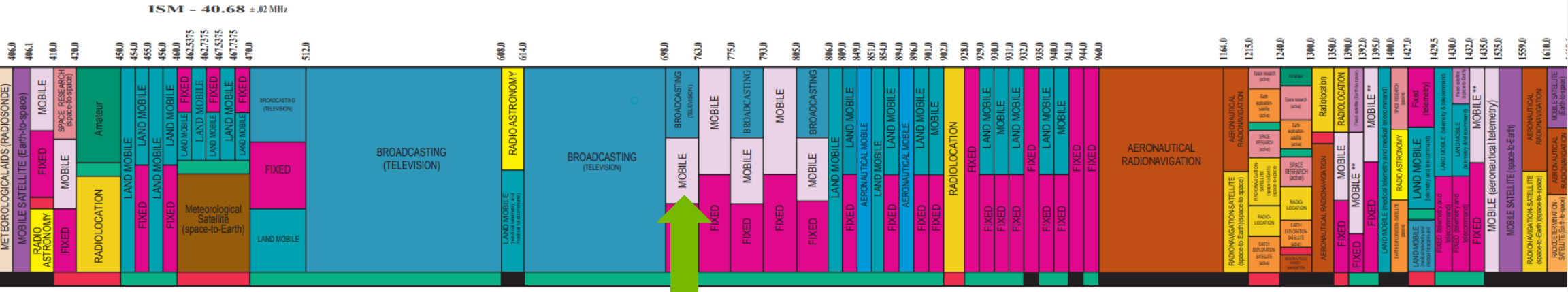
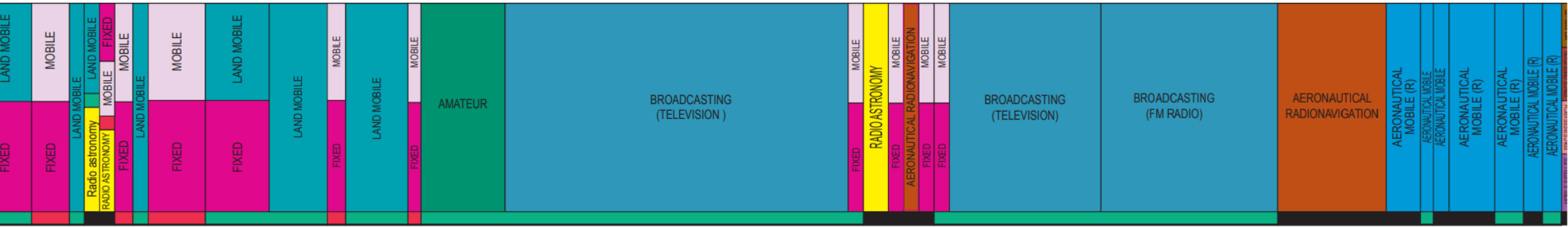
U.S. DEPARTMENT OF COMMERCE
National Telecommunications and Information Administration
Office of Spectrum Management
JANUARY 2014



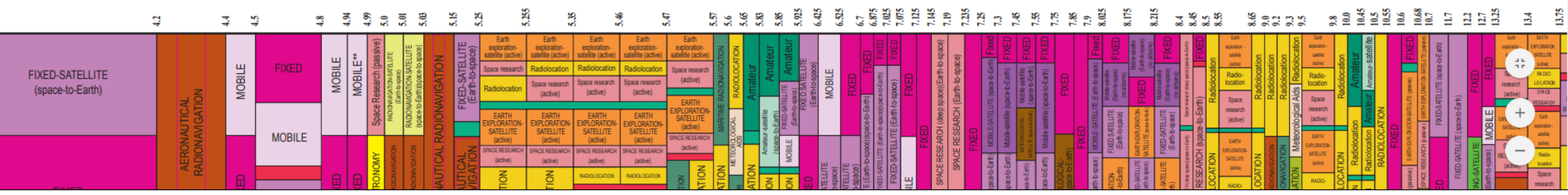
300 MHz - 3 GHz

30 GHz - 300 GHz

300 GHz



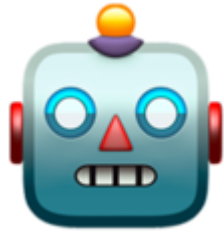
700MHz, Band 13 - Active Main LTE Band for Verizon (4G)



TWO FUNDAMENTAL NEEDS



Fast filtering, FFTs, correlations, convolutions, resampling, etc to process increasingly larger bandwidths of signals at increasingly fast rates and do increasingly cool stuff we couldn't do before



Artificial Intelligence techniques applied to spectrum sensing, signal identification, spectrum collaboration, and anomaly detection

SIGNAL PROCESSING ON GPUS: A HISTORY

home sample applications

GPU VSIPL

GPU VSIPL is an implementation of [Vector Signal Image Processing Library](#) that targets Graphics Processing Units (GPUs) supporting NVIDIA's CUDA platform. By leveraging processors capable of 900 GFLOP/s or more, your application may achieve considerable speedup without any specialized development for GPUs. Our [range-Doppler map](#) application achieved a **75x** speedup on the GPU simply by linking it with GPU VSIPL.

Distribution

GPU VSIPL is currently released as a binary-only static library with the restriction that the library not be redistributed. This should enable internal development and testing to see if GPU VSIPL meets your needs. If you wish to distribute applications developed with GPU VSIPL, please contact us to arrange a separate licensing agreement. Email gpu-vsip@gtri.gatech.edu

For announcements on new updates to GPU VSIPL, and discussion about the software, please subscribe to the [GPU VSIPL Mailing List](#).

Validation

All releases are verified with the [VSIPL Core Lite Test Suite](#).

GPU VSIPL was presented to the [High Performance Embedded Computing Workshop 2008](#). Read the [GPU VSIPL extended abstract](#) [PDF].



cuFFT

GPU-accelerated library for Fast Fourier Transforms



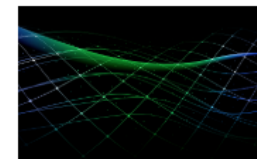
cuSPARSE

GPU-accelerated BLAS for sparse matrices



cuBLAS

GPU-accelerated standard BLAS library



cuSOLVER

Dense and sparse direct solvers for Computer Vision, CFD, Computational Chemistry, and Linear Optimization applications



Free and Open Source signal processing from Python (BSD 3 license)



CPU performance optimizations for various computationally intensive operations (e.g. linear filtering)



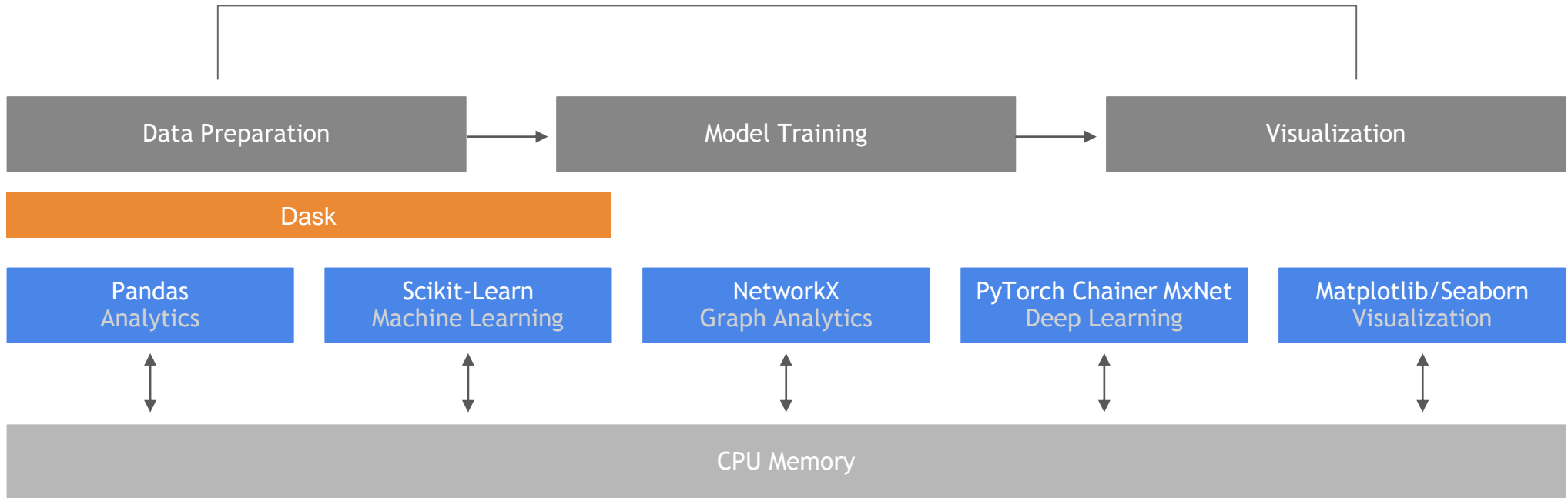
Extensive functionality: convolution, filtering and filter design, peak finding, spectral analysis among others

LET'S TALK ABOUT RAPIDS FOR A SECOND



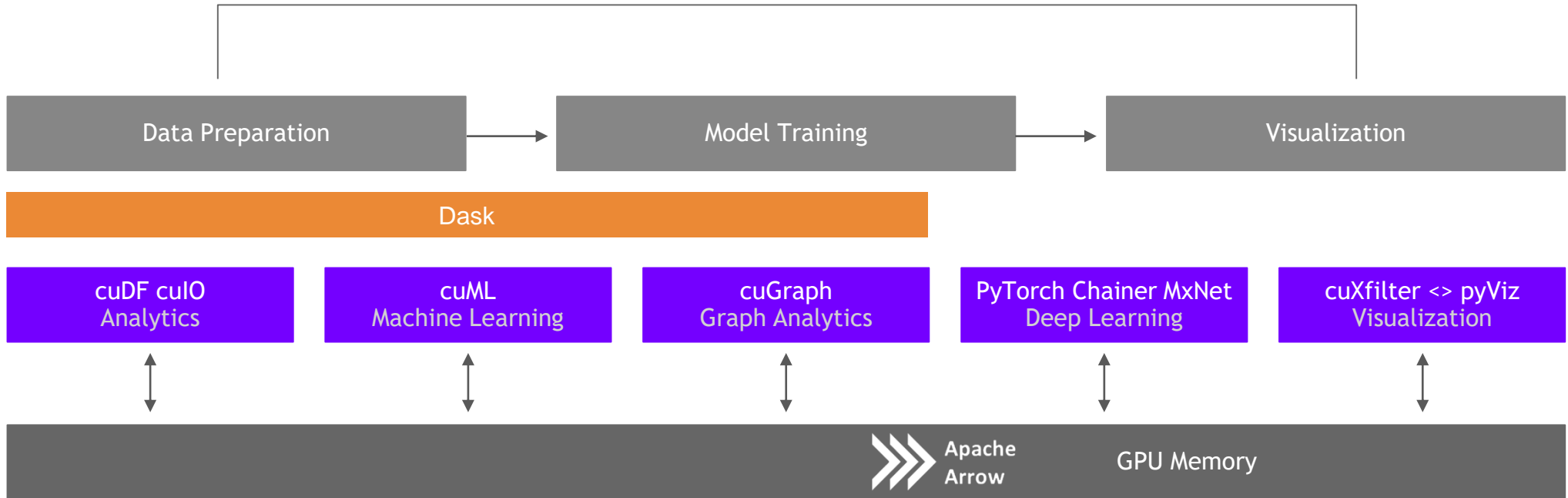
Open Source Data Science Ecosystem

Familiar Python APIs



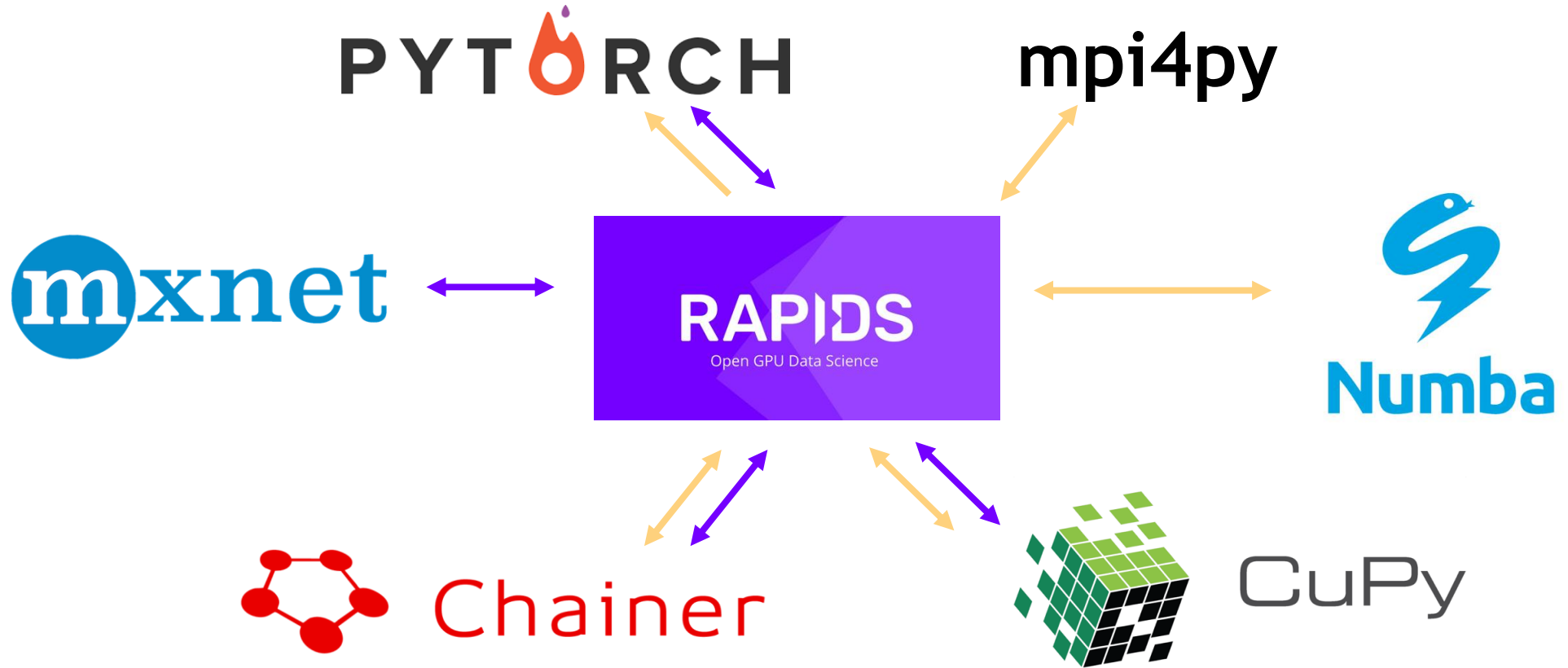
RAPIDS

End-to-End Accelerated GPU Data Science



Interoperability for the Win

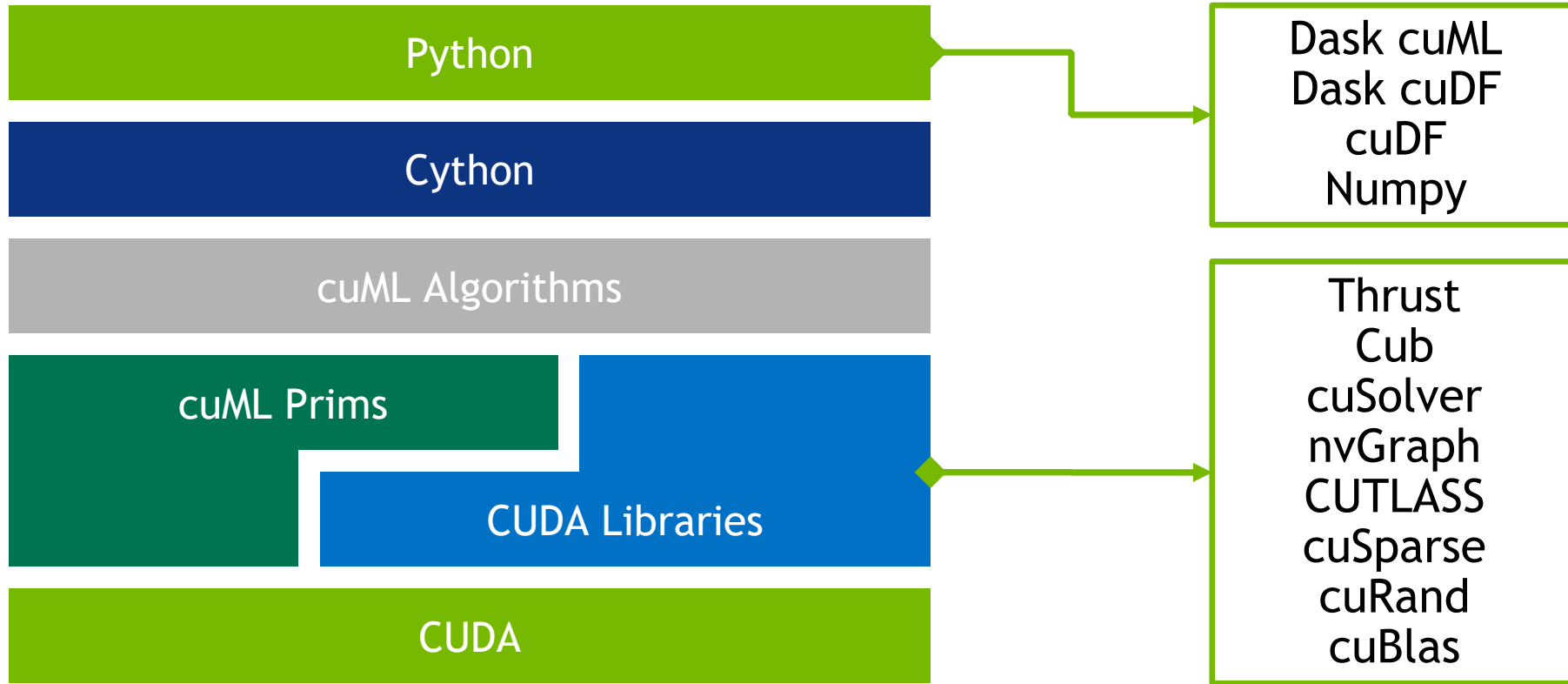
DLPack and `__cuda_array_interface__`



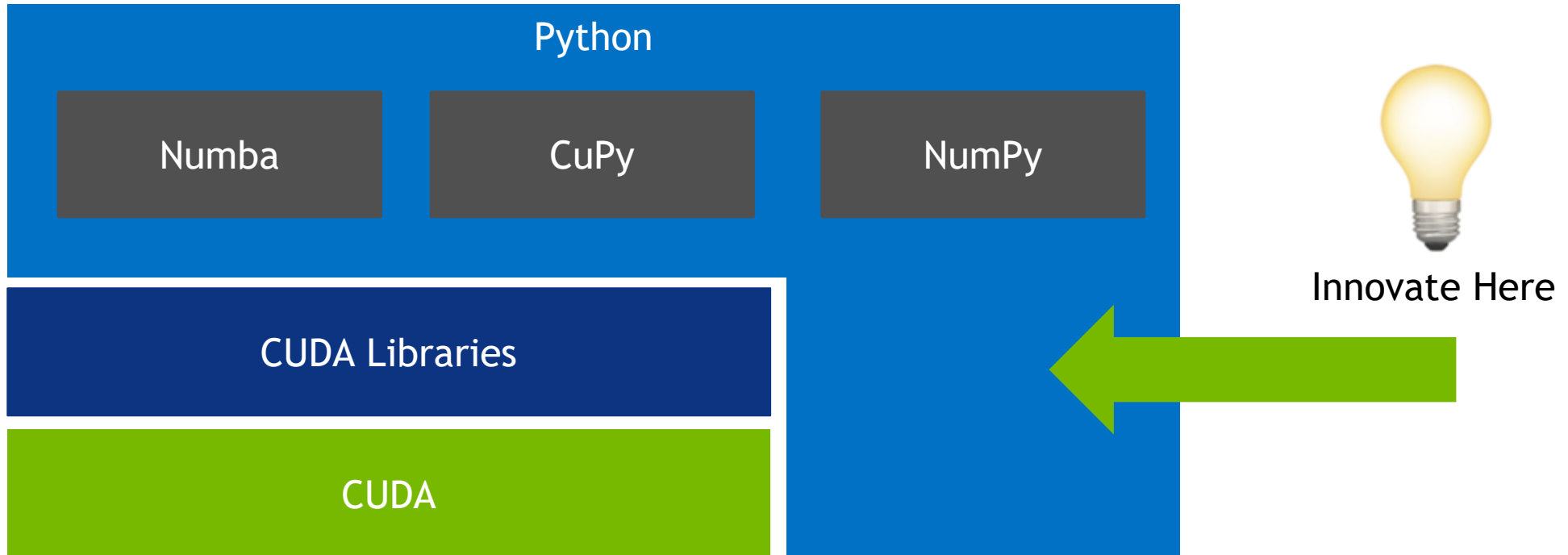


CUSIGNAL CORE

REFERENCE RAPIDS TECHNOLOGY STACK



CUSIGNAL TECHNOLOGY STACK



ALGORITHMS

GPU-accelerated SciPy Signal



Wavelets

Peak Finding

More to come!

Convolution

Filtering and Filter Design

Waveform Generation

Window Functions

Spectral Analysis

Convolve/Correlate
FFT Convolve
Convolve/Correlate 2D

Resampling - Polyphase, Upfirdn, Resample
Hilbert/Hilbert 2D
Wiener
Firwin

Chirp
Square
Gaussian Pulse

Kaiser
Blackman
Hamming
Hanning

Periodogram
Welch
Spectrogram

PERFORMANCE

As Always, YMMV. Benchmarked with ~1e8 sample signals on a P100 GPU using *time* around Python calls

| Method | Scipy Signal (ms) | cuSignal (ms) | Speedup (xN) |
|---------------|-------------------|---------------|--------------|
| fftconvolve | 34173 | 450 | 76.0 |
| correlate | 20580 | 390 | 52.8 |
| resample | 18834 | 372 | 50.7 |
| resample_poly | 4182 | 291 | 14.3 |
| welch | 7015 | 270 | 25.9 |
| spectrogram | 4061 | 271 | 15.0 |
| cwt | 56035 | 628 | 89.2 |

Learn more about cuSignal functionality and performance by browsing the [notebooks](#)

SPEED OF LIGHT PERFORMANCE - P100

timeit (7 runs) rather than *time*. Benchmarked with ~1e8 sample signals on a P100 GPU

| Method | Scipy Signal (ms) | cuSignal (ms) | Speedup (xN) |
|---------------|-------------------|---------------|--------------|
| fftconvolve | 33200 | 130.0 | 255.4 |
| correlate | 19900 | 72.6 | 274.1 |
| resample | 15100 | 70.2 | 215.1 |
| resample_poly | 4250 | 52.3 | 81.3 |
| welch | 6730 | 79.5 | 84.7 |
| spectrogram | 4120 | 37.7 | 109.3 |
| cwt | 56200 | 272 | 206.6 |

Learn more about cuSignal functionality and performance by browsing the [notebooks](#)

SPEED OF LIGHT PERFORMANCE - V100

timeit (7 runs) rather than *time*. Benchmarked with ~1e8 sample signals on a DGX Station

| Method | Scipy Signal (ms) | cuSignal (ms) | Speedup (xN) |
|---------------|-------------------|---------------|--------------|
| fftconvolve | 28400 | 92.2 | 308.0 |
| correlate | 16800 | 48.4 | 347.1 |
| resample | 14700 | 51.1 | 287.7 |
| resample_poly | 3110 | 13.7 | 227.0 |
| welch | 4620 | 53.7 | 86.0 |
| spectrogram | 2520 | 28 | 90.0 |
| cwt | 46700 | 277 | 168.6 |

Learn more about cuSignal functionality and performance by browsing the [notebooks](#)

“Using the cuSignal library we were able to speed-up a long running signal processing task from ~14 hours to ~3 hours with minimal drop-in code replacements.”

EXPEDITION
TECHNOLOGY



2019 SECAF Government Contractor of the Year, \$7.5-15M Revenue Category

DIVING DEEPER



Much of the cuSignal codebase has been written by simply swapping out NumPy functionality for CuPy and fixing errors as they appear



resample_poly is different, however, and includes a custom Numba CUDA kernel implementing *upfirdn*



Not all memory is created equal, and it doesn't always originate on the GPU

DIVING DEEPER



Much of the cuSignal codebase has been written by simply swapping out NumPy functionality for CuPy and fixing errors as they appear



resample_poly is different, however, and includes a custom Numba CUDA kernel implementing *upfirdn*



Not all memory is created equal, and it doesn't always originate on the GPU



A NumPy-Compatible Matrix Library Accelerated by CUDA



Free and open source software developed under the Chainer project and Preferred Networks (MIT License)



Includes CUDA libraries: cuBLAS, cuDNN, cuRand, cuSolver, cuSparse, cuFFT, and NCCL



Typically a drop-in replacement for NumPy



Ability to write custom kernel for additional performance, requiring a bit of C++

HILBERT TRANSFORM: NUMPY ↔ CUPY

```
hilbert_cpu.py x
C: > Users > adamt > Desktop > hilbert_cpu.py > ...
1 from scipy import fft as sp_fft
2 from numpy import asarray, zeros
3
4 def hilbert(x, N=None, axis=-1):
5     x = asarray(x)
6     if iscomplexobj(x):
7         raise ValueError("x must be real.")
8     if N is None:
9         N = x.shape[axis]
10    if N <= 0:
11        raise ValueError("N must be positive.")
12
13    Xf = sp_fft.fft(x, N, axis=axis)
14    h = zeros(N)
15    if N % 2 == 0:
16        h[0] = h[N // 2] = 1
17        h[1:N // 2] = 2
18    else:
19        h[0] = 1
20        h[1:(N + 1) // 2] = 2
21
22    if x.ndim > 1:
23        ind = [newaxis] * x.ndim
24        ind[axis] = slice(None)
25        h = h[tuple(ind)]
26    x = sp_fft.ifft(Xf * h, axis=axis)
27    return x

hilbert_gpu.py •
C: > Users > adamt > Desktop > hilbert_gpu.py > ...
1 from cupy.scipy import fftpack
2 from cupy import asarray, zeros
3
4 def hilbert(x, N=None, axis=-1):
5     x = asarray(x)
6     if iscomplexobj(x):
7         raise ValueError("x must be real.")
8     if N is None:
9         N = x.shape[axis]
10    if N <= 0:
11        raise ValueError("N must be positive.")
12
13    Xf = fftpack.fft(x, N, axis=axis)
14    h = zeros(N)
15    if N % 2 == 0:
16        h[0] = h[N // 2] = 1
17        h[1:N // 2] = 2
18    else:
19        h[0] = 1
20        h[1:(N + 1) // 2] = 2
21
22    if x.ndim > 1:
23        ind = [newaxis] * x.ndim
24        ind[axis] = slice(None)
25        h = h[tuple(ind)]
26    x = fftpack.ifft(Xf * h, axis=axis)
27    return x
```

DIVING DEEPER



Much of the cuSignal codebase has been written by simply swapping out NumPy functionality for CuPy and fixing errors as they appear



resample_poly is different, however, and includes a custom Numba CUDA kernel implementing *upfirdn*



Not all memory is created equal, and it doesn't always originate on the GPU



NUMBA

JIT Compiler for Python with LLVM

- **Write Python function**
 - Use C/Fortran style for loops
 - Large subset of Python language
 - Mostly for numeric data
- Wrap it in `@numba.jit`
 - Compiles to native code with LLVM
 - JIT compiles on first use with new type signatures
- Runs at C/Fortran speeds

```
def sum(x):  
    total = 0  
    for i in range(x.shape[0]):  
        total += x[i]  
    return total
```

```
>>> x = numpy.arange(10_000_000)  
>>> %time sum(x)  
1.34 s ± 8.17 ms
```

See also: Cython, Pythran, pybind, f2py



NUMBA

JIT Compiler for Python with LLVM

- Write Python function
 - Use C/Fortran style for loops
 - Large subset of Python language
 - Mostly for numeric data
- **Wrap it in @numba.jit**
 - Compiles to native code with LLVM
 - JIT compiles on first use with new type signatures
- Runs at C/Fortran speeds

```
import numba
```

```
@numba.jit
```

```
def sum(x):  
    total = 0  
    for i in range(x.shape[0]):  
        total += x[i]  
    return total
```

```
>>> x = numpy.arange(10_000_000)
```

```
>>> %time sum(x)
```

```
55 ms
```

See also: Cython, Pythran, pybind, f2py



NUMBA

JIT Compiler for Python with LLVM

- Write Python function
 - Use C/Fortran style for loops
 - Large subset of Python language
 - Mostly for numeric data
- **Wrap it in @numba.jit**
 - Compiles to native code with LLVM
 - JIT compiles on first use with new type signatures
- Runs at C/Fortran speeds

```
import numba
```

```
@numba.jit
```

```
def sum(x):  
    total = 0  
    for i in range(x.shape[0]):  
        total += x[i]  
    return total
```

```
>>> x = numpy.arange(10_000_000)
```

```
>>> %time sum(x)
```

```
55 ms # mostly compile time
```

See also: Cython, Pythran, pybind, f2py



NUMBA

JIT Compiler for Python with LLVM

- Write Python function
 - Use C/Fortran style for loops
 - Large subset of Python language
 - Mostly for numeric data
- Wrap it in @numba.jit
 - Compiles to native code with LLVM
 - JIT compiles on first use with new type signatures
- **Runs at C/Fortran speeds**

```
import numba
```

```
@numba.jit  
def sum(x):  
    total = 0  
    for i in range(x.shape[0]):  
        total += x[i]  
    return total
```

```
>>> x = numpy.arange(10_000_000)  
>>> %time sum(x)  
5.09 ms ± 110 µs # subsequent runs
```

See also: Cython, Pythran, pybind, f2py



NUMBA

JIT Compiler for Python with LLVM

- Write Python function
 - Use C/Fortran style for loops
 - Large subset of Python language
 - Mostly for numeric data
- Wrap it in `@numba.jit`
 - Compiles to native code with LLVM
 - JIT compiles on first use with new type signatures
- Runs at C/Fortran speeds
- **Supports**
 - Normal numeric code
 - Dynamic data structures
 - Recursion
 - CPU Parallelism (thanks Intel!)
 - CUDA, AMD ROCm, ARM
 - ...

```
import numba
```

```
@numba.jit
def sum(x):
    total = 0
    for i in range(x.shape[0]):
        total += x[i]
    return total
```

```
>>> x = numpy.arange(10_000_000)
>>> %time sum(x)
5.09 ms ± 110 µs
```

COMBINE NUMBA WITH CUPY

Write custom CUDA code from Python

Stencil computations on CPU

```
In [1]: import numpy as np
import numba

@numba.stencil
def _smooth(x):
    return (x[-1, -1] + x[-1, 0] + x[-1, 1] +
            x[ 0, -1] + x[ 0, 0] + x[ 0, 1] +
            x[ 1, -1] + x[ 1, 0] + x[ 1, 1]) // 9

@numba.njit
def smooth_cpu(x):
    return _smooth(x)
```

```
In [2]: x_cpu = np.ones((10000, 10000), dtype='int8')

%timeit smooth_cpu(x_cpu)

621 ms ± 15.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

COMBINE NUMBA WITH CUPY

Write custom CUDA code from Python

Stencil computations on GPU

Using the `numba.cuda` module I'm able to get about a 200x increase with a modest increase in code complexity.

```
In [3]: from numba import cuda

@cuda.jit
def smooth_gpu(x, out):
    i, j = cuda.grid(2)
    n, m = x.shape
    if 1 <= i < n - 1 and 1 <= j < m - 1:
        out[i, j] = (x[i - 1, j - 1] + x[i - 1, j] + x[i - 1, j + 1] +
                    x[i, j - 1] + x[i, j] + x[i, j + 1] +
                    x[i + 1, j - 1] + x[i + 1, j] + x[i + 1, j + 1]) // 9
```

```
In [4]: import cupy, math

x_gpu = cupy.ones((10000, 10000), dtype='int8')
out_gpu = cupy.zeros((10000, 10000), dtype='int8')

# I copied the four lines below from the Numba docs
threadsperblock = (16, 16)
blockspergrid_x = math.ceil(x_gpu.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(x_gpu.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)

%timeit smooth_gpu[blockspergrid, threadsperblock](x_gpu, out_gpu)

2.87 ms ± 90.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Note: the GPU solution here cheats a bit because it pre-allocates the output array

CUSTOM NUMBA KERNELS FOR IN CUSIGNAL

upfirdn

correlate2d

convolve2d

lombscargle

...and more on the way (*lfilter* of particular interest)

DIVING DEEPER



Much of the cuSignal codebase has been written by simply swapping out NumPy functionality for CuPy and fixing errors as they appear



resample_poly is different, however, and includes a custom Numba CUDA kernel implementing *upfirdn*



Not all memory is created equal, and it doesn't always originate on the GPU

CASE STUDY - POLYPHASE RESAMPLING

Scipy Signal (CPU)

```
import numpy as np
from scipy import signal

start = 0
stop = 10
num_samps = int(1e8)
resample_up = 2
resample_down = 3

cx = np.linspace(start, stop, num_samps, endpoint=False)
cy = np.cos(-cx**2/6.0)

cf = signal.resample_poly(cy, resample_up, resample_down, window=('kaiser', 0.5))
```

This code executes on 2x Xeon E5-2600 in 2.36 sec.

CASE STUDY - POLYPHASE RESAMPLING

cuSignal with Data Generated on the GPU with CuPy

```
import cupy as cp
import cusignal

start = 0
stop = 10
num_samps = int(1e8)
resample_up = 2
resample_down = 3

gx = cp.linspace(start, stop, num_samps, endpoint=False)
gy = cp.cos(-cx**2/6.0)

gf = cusignal.resample_poly(gy, resample_up, resample_down, window=('kaiser', 0.5))
```

This code executes on an NVIDIA P100 in 258 ms.

CASE STUDY - POLYPHASE RESAMPLING

cuSignal with Data Generated on the CPU and Copied to GPU [AVOID THIS FOR ONLINE SIGNAL PROCESSING]

```
import cupy as cp
import numpy as np
import cusignal

start = 0
stop = 10
num_samps = int(1e8)
resample_up = 2
resample_down = 3

# Generate Data on CPU
cx = np.linspace(start, stop, num_samps, endpoint=False)
cy = np.cos(-cx**2/6.0)

gf = cusignal.resample_poly(cp.asarray(cy), resample_up, resample_down, window=('kaiser', 0.5))
```

This code executes on an NVIDIA P100 in 728 ms.

CASE STUDY - POLYPHASE RESAMPLING

cuSignal with Data Generated on the CPU with Mapped, Pinned (zero-copy) Memory

```
import cupy as cp
import numpy as np
import cusignal

start = 0
stop = 10
num_samps = int(1e8)
resample_up = 2
resample_down = 3

# Generate Data on CPU
cx = np.linspace(start, stop, num_samps, endpoint=False)
cy = np.cos(-cx**2/6.0)

# Create shared memory between CPU and GPU and load with CPU signal (cy)
gpu_signal = cusignal.get_shared_mem(num_samps, dtype=np.complex128)
gpu_signal[:] = cy

gf = cusignal.resample_poly(gpu_signal, resample_up, resample_down, window=('kaiser', 0.5))
```

This code executes on an NVIDIA P100 in 154 ms.

WHAT'S GOING ON HERE?



Software Defined Radios (SDR) often transfer a “small” number of samples from the local buffer to host to avoid dropped packets



Frequent, small data copies will cripple GPU performance; the GPU will be underutilized, and we'll be handcuffed by CPU controlled data transfers from SDR to CPU to GPU



We are making use of pinned and mapped memory (zero-copy) from Numba to provide a dedicated memory space usable by both the CPU and GPU, reducing the data copy overhead

- `_arraytools.get_shared_mem` - mapped, pinned memory, similar to `np.zeros`
- `_arraytools.get_shared_array` - mapped, pinned memory loaded with given data of a given type

FFT BENCHMARKING

N = 32768 complex128 samples

FFT speed with NumPy: 0.734 ms

FFT speed with CuPy and asarray call (CPU->GPU movement): 210* ms

FFT speed with CuPy and memory already on GPU with CuPy: 0.397 ms

FFT speed with mapped array and Numba (create array and load data): 0.792 ms

FFT speed if context came in as mapped (just load data in zero-copy space): 0.454 ms



We want to create some mapped, pinned memory space of a given size and load data here.

* includes FFT plan creation that is ultimately cached; in an online signal processing application, you can do this before you start executing streaming FFTs. More details [here](#)

A network diagram with green nodes and lines on a dark background. The nodes are represented by small green circles of varying sizes, and they are interconnected by thin, light green lines. The overall appearance is that of a complex, interconnected network or data structure. The background is dark, with some faint, larger green circles scattered throughout, suggesting a vast or multi-layered network.

FROM CUSIGNAL TO APPLICATIONS OF AI

MARRIAGE OF DEEP LEARNING AND RF DATA

SIGNAL IDENTIFICATION

Learn features specific to a desired emitter

Fits into many existing RF dataflows

Success in high noise, high interference environments

ANOMALY DETECTION

Facilitates in discovery

Early warning system for defense and commercial applications

Enforce FCC regulations

SCHEDULING

Automatic recognition of free communication channels

Provide a basis for effective signal transmission or reception

MOVE SEAMLESSLY FROM CUSIGNAL TO PYTORCH

```
[12]: import numpy as np
import cupy as cp
from numba import cuda
```

```
[13]: N = 2**18
```

```
[14]: def get_shared_mem(shape, dtype=np.float32, strides=None, order='C', stream=0, portable=False, wc=True):
return cuda.mapped_array(shape, dtype=dtype, strides=strides, order=order, stream=stream, portable=portable, wc=wc)
```

```
[15]: # Allocate known memory size before processing. This is accessible to the CPU or GPU
shared_sig = get_shared_mem(N, dtype=np.complex128)
print('CPU Pointer: ', shared_sig.__array_interface__['data'])
print('GPU Pointer: ', shared_sig.__cuda_array_interface__['data'])
```

```
CPU Pointer: (140400685744128, False)
GPU Pointer: (140400685744128, False)
```

```
[16]: %%time
shared_sig[:] = np.random.rand(N) + 1j*np.random.rand(N)
```

```
CPU times: user 11.5 ms, sys: 2.91 ms, total: 14.4 ms
Wall time: 12.6 ms
```


MOVE SEAMLESSLY FROM CUSIGNAL TO PYTORCH

sig value can be called by a CPU function or GPU one. Here's we'll take the mean via both NumPy and CuPy, comparing performance

```
[17]: %%time
      cpu_fft = np.abs(np.fft.fft(shared_sig))
      CPU times: user 18.8 ms, sys: 4.68 ms, total: 23.5 ms
      Wall time: 21.4 ms
```

```
[18]: %%time
      gpu_fft = cp.abs(cp.fft.fft(cp.asarray(shared_sig)))
      CPU times: user 2.63 ms, sys: 8.66 ms, total: 11.3 ms
      Wall time: 9.86 ms
```

```
[19]: # Prove cp.asarray() just gives cupy context - same pointer is used
      shared_sig.__cuda_array_interface__['data']
      cp.asarray(shared_sig).__cuda_array_interface__['data']
```

```
[19]: (140400685744128, False)
```

MOVE SEAMLESSLY FROM CUSIGNAL TO PYTORCH

Move sig to PyTorch via DLPack

```
[20]: from torch.utils.dlpack import to_dlpack
      from torch.utils.dlpack import from_dlpack

[21]: # Enforce cupy array, still zero copy
      sig = cp.asarray(gpu_fft).astype(cp.float64)
      torch_sig = from_dlpack(sig.toDlpack())
      torch_sig

[21]: tensor([[1.8519e+05, 9.9990e+01, 2.2987e+02, ..., 2.4414e+02, 2.2538e+02,
              3.0817e+02], device='cuda:0', dtype=torch.float64)
```



As of PyTorch 1.2, `__cuda_array_interface__` is officially supported, and one no longer has to move data to PyTorch via DLPack

END-TO-END EXAMPLE

Predict the Number of Carriers in a Signal

Generate 2000 signals that are each 2^{15} samples in length; each signal has between 1 and 5 carriers spaced at one of 10 different center frequencies

Use polyphase resampler to upsample by 2

Run periodogram with flattop filter over each signal

Use a simple multi-layer linear neural network to train and predict the number of carriers in an arbitrary signal



**WHERE TO GO FROM
HERE?**

WHAT'S NEXT FOR CUSIGNAL?



Integrate GPU CI/CD and add Conda packaging



Add test scripts to ensure integrity of cuSignal functionality, especially compared with SciPy Signal



Please help profile performance, optimize the code, and add new features!



Further SDR integration via SoapySDR, pyrtlsdr, etc



Examine GPU acceleration of common RF recording specifications (SigMF, MIDAS Blue/Platinum, Vita 49)

ACKNOWLEDGEMENTS

SciPy Signal Core Development Team, Particularly Travis Oliphant

Matthew Nicely - NVIDIA - Numba/CUDA optimization

Ryan Crawford - Expedition Technology - API/Performance Feedback

Deepwave Digital - API/Performance Feedback, Online Signal Processing

John Murray - Fusion Data Science

Jeff Shultz - CACI

LPS/BAH

