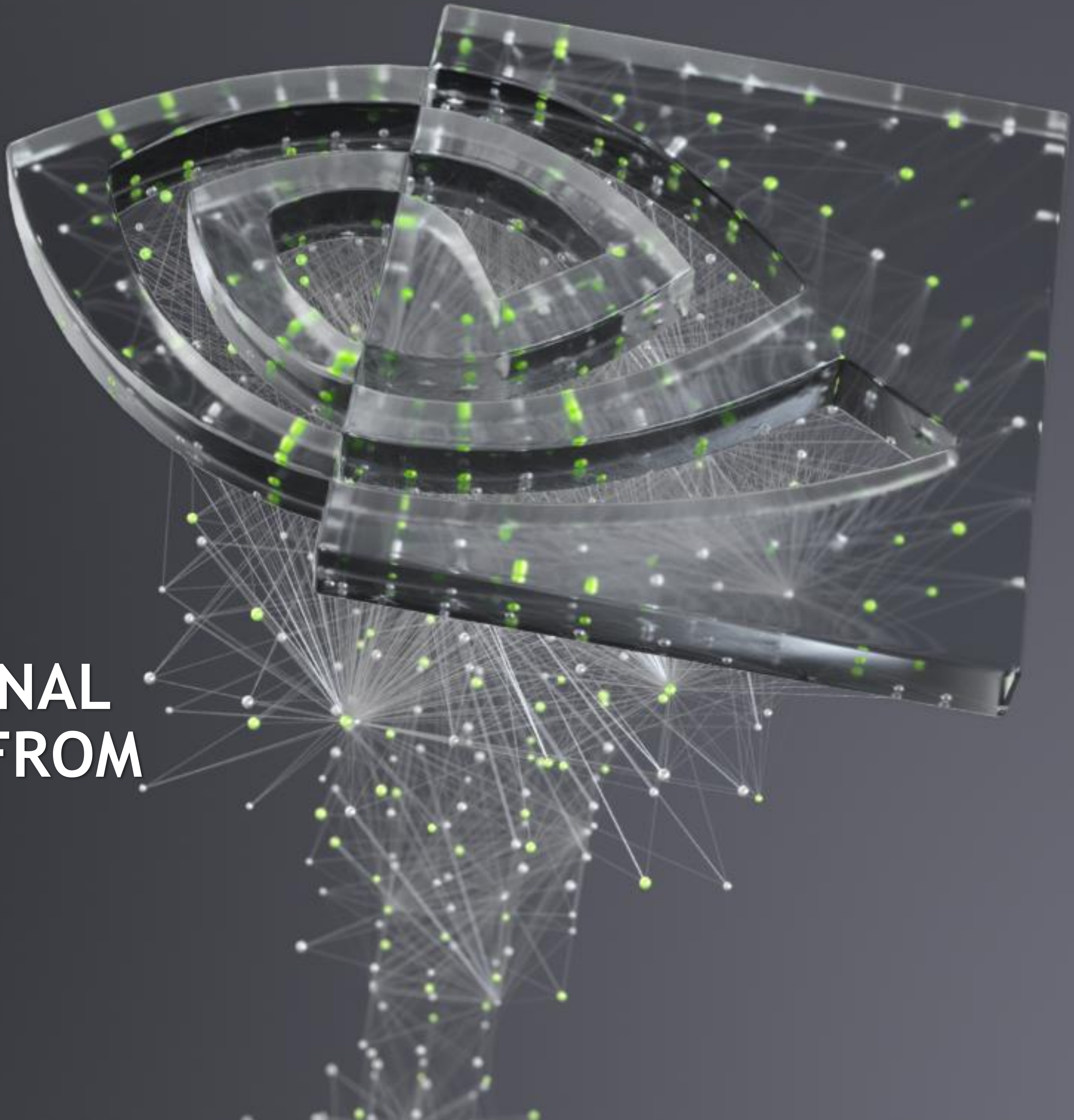


GPU-ACCELERATION OF SIGNAL PROCESSING WORKFLOWS FROM PYTHON

IEEE ICASSP ‘21 – Toronto, Canada

Adam Thompson, Matthew Nicely, Zoe Ryan

AGENDA

cuSignal

cuSignal Overview and Design Philosophy

cuSignal Installation and Example Walkthrough

GPU Computing in Python

GPU Basics and Introduction to Numba and CuPy

How to Use GPU Profiling Tools

Acceleration of Elementwise Kernels

How to Build Custom CUDA Kernels with Numba and CuPy

The New York Times

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

Virginia Avenue in North Olmsted, Ohio, where residents complained that their car key fobs and garage door openers had stopped working. Dustin Franz for The New York Times

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.

CONSIDERATIONS

As Signal and Sensor Processing Developers, We Care About:

🔥 High-performance signal processing operations (e.g FFT, convolution, correlation, spectrum estimation) via an easy to program and modify API

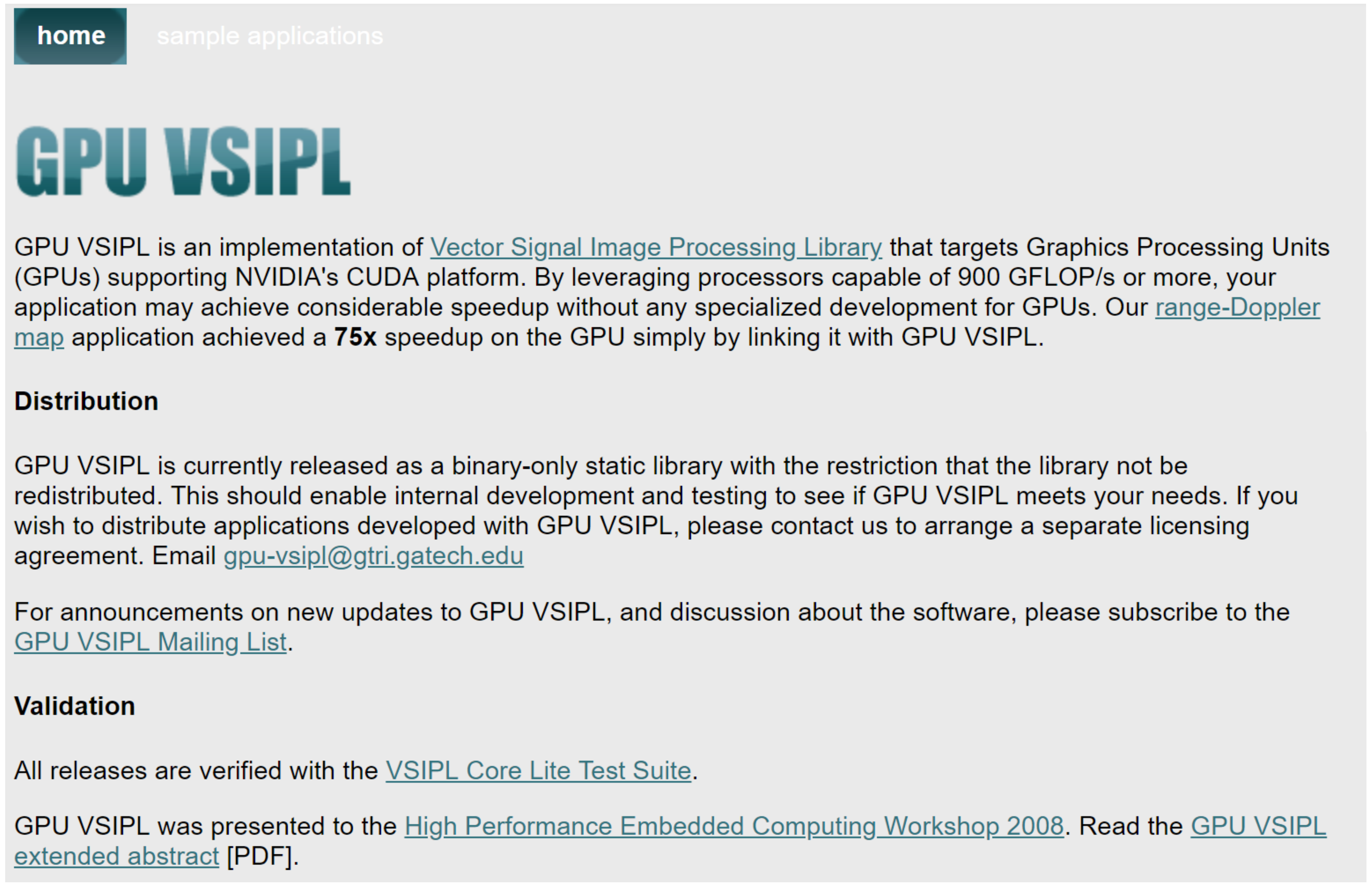
💾 Handling high-throughput, low-latency I/O with the ability to quickly respond to changes from the sensor

💪 Future proofing for AI: Enable zero-copy data movement connections to AI/ML frameworks like PyTorch and TensorFlow or be auto-differentiable themselves

🔢 Complex valued numerical support

SIGNAL PROCESSING ON GPU: A HISTORY

Abstract Away the Complexity



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

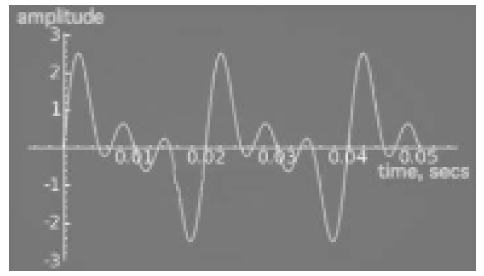
GPU VSIPL is currently released as a binary-only static library with the restriction that the library not be redistributed. This should enable internal development and testing to see if GPU VSIPL meets your needs. If you wish to distribute applications developed with GPU VSIPL, please contact us to arrange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

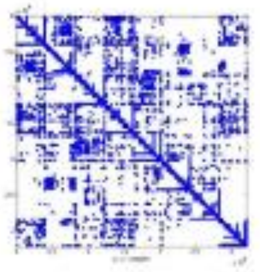
All releases are verified with the VSIPL Core Lite Test Suite.

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

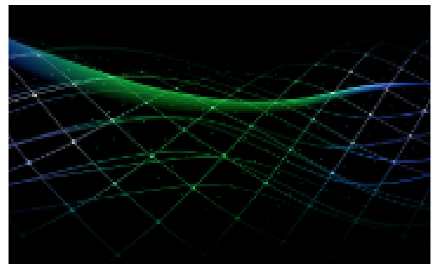


cuSPARSE

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

CUSIGNAL – SELECTED ALGORITHMS

GPU-accelerated SciPy Signal (Python)

Convolution	Convolve/Correlate FFT Convolve Convolve/Correlate 2D
Filtering and Filter Design	Resampling – Polyphase, Upfirdn, Resample Hilbert/Hilbert 2D Wiener Firwin, FIR Filter
Waveform Generation	Chirp Square Gaussian Pulse
Window Functions	Kaiser Blackman Hamming Hanning
Spectral Analysis	Periodogram Welch Spectrogram

Phased Array

Peak Finding

Full List of Supported Functions – cuSignal Docs

SCIPY SIGNAL – POLYPHASE RESAMPLER

```python
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)

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

2x Xeon E5-2600: 2.36 seconds

CUSIGNAL – POLYPHASE RESAMPLER

```
import cupy as cp
import cusignal

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

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

%%timeit
cf = cusignal.resample_poly(cy, resample_up, resample_down, window=('kaiser', 0.5))
```

NVIDIA V100: 8.29 milliseconds, **285x SciPy Signal (CPU)**

SPEED OF LIGHT PERFORMANCE – V100

timeit (7 runs); Benchmarked with ~1e8 sample signals, float64

Method	SciPy Signal (ms)	cuSignal (ms)	Speedup (xN)
fftconvolve	27300	85.1	320.8
correlate	4020	47.4	84.8
resample	14700	45.9	320.2
resample_poly	2360	8.29	284.6
welch	4870	45.5	107.0
spectrogram	2520	23.3	108.1
convolve2d	8410	9.92	847.7

Learn more about cuSignal functionality and performance by browsing the notebooks

SPEED OF LIGHT PERFORMANCE – V100 VS A100

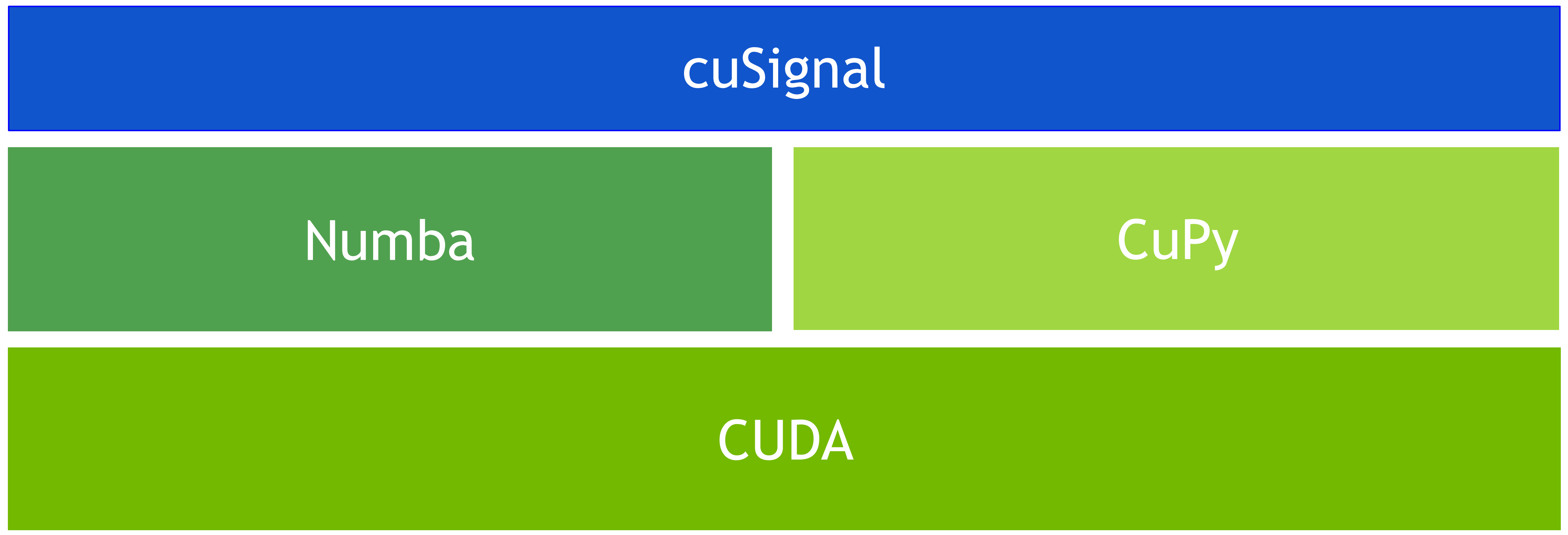
timeit (7 runs); Benchmarked with ~1e8 sample signals, float64

Method	cuSignal: V100 (ms)	cuSignal: A100 (ms)	Speedup (xN)
fftconvolve	85.1	46.6	1.83
correlate	47.4	28.3	1.67
resample	45.9	15.4	2.98
resample_poly	8.29	4.6	1.80
welch	45.5	23.5	1.94
spectrogram	23.3	13.2	1.77
convolve2d	9.92	6.04	1.64

Learn more about cuSignal functionality and performance by browsing the notebooks

CUSIGNAL TECHNOLOGY STACK

An Innovative Approach to Pure Python GPU Library Development



Unlike other RAPIDS libraries, cuSignal is purely developed in Python
with custom CUDA Kernels written with Numba and CuPy (notice no Cython layer).

cuSignal is designed to quickly prototype Signal Processing applications and give baseline GPU performance for a given workflow. We stress data generality and ease of use.

CUPY

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 along with lightweight Python wrappers to each
- Typically a drop-in replacement for NumPy
- Ability to write custom kernel for additional performance, requiring a bit of C++

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

See also: Cython, Pythran, pybind, f2py

```
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
```

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**

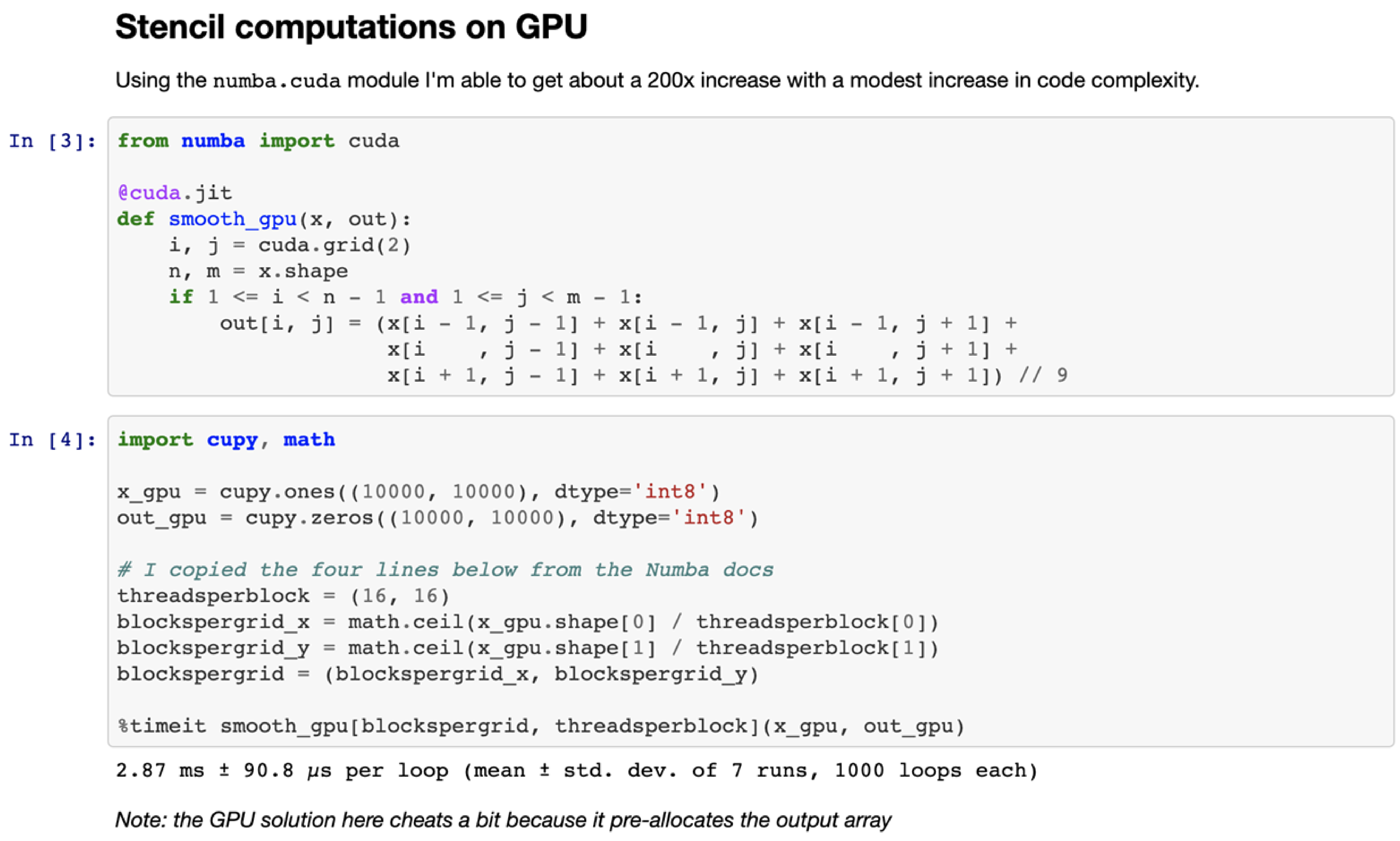
See also: Cython, Pythran, pybind, f2py

```
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
```

NUMBA SUPPORT CUDA AND 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

HOW WE BUILT CUSIGNAL (AND HOW TO DO IT YOURSELF!)

Drop-in GPU Library Replacements

NumPy -> CuPy
Pandas -> cuDF
Scikit-Learn -> cuML
Network-X -> cuGraph

Pros:
Trivial code change
“Free” Performance

Cons:
Potentially sub-optimal
Limited control

Custom Numba CUDA Kernels

Leverage JIT compilation and Numba’s CUDA support to quickly build and test custom CUDA kernels with a Pythonic API

Pros:
Quickly build custom features
Boilerplate code

Cons:
JIT compilation overhead
Excess register pressure

Custom Raw CUDA Kernels

To match native CUDA speeds, wrap raw CUDA kernels in CuPy; precompile and cache kernel to avoid JIT overhead

Pros:
Matches CUDA C++ speed
No excess SW layer

Cons:
Limited debugging tools
Basically C/C++
Support multiple dtypes

DROP-IN REPLACEMENTS: 1D HILBERT TRANSFORM

SciPy Signal

```
import numpy as np
from scipy import fft as sp_fft

def hilbert(x, N=None, axis=-1):
    x = np.asarray(x)
    if np.iscomplexobj(x):
        raise ValueError("x must be real.")
    if N is None:
        N = x.shape[axis]
    if N <= 0:
        raise ValueError("N must be positive.")

    Xf = sp_fft.fft(x, N, axis=axis)
    h = np.zeros(N)
    if N % 2 == 0:
        h[0] = h[N // 2] = 1
        h[1:N // 2] = 2
    else:
        h[0] = 1
        h[1:(N + 1) // 2] = 2

    if x.ndim > 1:
        ind = [np.newaxis] * x.ndim
        ind[axis] = slice(None)
        h = h[tuple(ind)]
    x = sp_fft.ifft(Xf * h, axis=axis)
    return x
```

cuSignal

```
import cupy as cp
from cupy import fft as sp_fft

def hilbert(x, N=None, axis=-1):
    x = cp.asarray(x)
    if cp.iscomplexobj(x):
        raise ValueError("x must be real.")
    if N is None:
        N = x.shape[axis]
    if N <= 0:
        raise ValueError("N must be positive.")

    Xf = sp_fft.fft(x, N, axis=axis)
    h = cp.zeros(N)
    if N % 2 == 0:
        h[0] = h[N // 2] = 1
        h[1:N // 2] = 2
    else:
        h[0] = 1
        h[1:(N + 1) // 2] = 2

    if x.ndim > 1:
        ind = [cp.newaxis] * x.ndim
        ind[axis] = slice(None)
        h = h[tuple(ind)]
    x = sp_fft.ifft(Xf * h, axis=axis)
    return x
```

NVIDIA V100, 1e8 float64 samples, **230x speedup**

CUSTOM NUMBA CUDA KERNELS: LOMBSCARGLE

SciPy Signal (_spectral.pyx)

```
import numpy as np
cimport numpy as np
cimport cython

@cython.boundscheck(False)
def _lombscargle(np.ndarray[np.float64_t, ndim=1] x,
                 np.ndarray[np.float64_t, ndim=1] y,
                 np.ndarray[np.float64_t, ndim=1] freqs):

    for i in range(freqs.shape[0]):

        # Code not shown

        for j in range(x.shape[0]):

            # Code not shown

        pgram[i] = 0.5 * (((c_tau * xc + s_tau * xs)**2 /
            (c_tau2 * cc + cs_tau * cs + s_tau2 * ss)) +
            ((c_tau * xs - s_tau * xc)**2 /
            (c_tau2 * ss - cs_tau * cs + s_tau2 * cc)))
```

cuSignal

```
import cupy as cp
from numba import cuda

@cuda.jit(fastmath=True)
def _numba_lombscargle(x, y, freqs, pgram, y_dot):

    F = cuda.grid(1)
    strideF = cuda.gridsize(1)

    if not y_dot[0]:
        yD = 1.0
    else:
        yD = 2.0 / y_dot[0]

    for i in range(F, freqs.shape[0], strideF):

        # Code not shown

        for j in range(x.shape[0]):

            # Code not shown

        pgram[i] = 0.5 * (((c_tau * xc + s_tau * xs)**2 /
            (c_tau2 * cc + cs_tau * cs + s_tau2 * ss)) +
            ((c_tau * xs - s_tau * xc)**2 /
            (c_tau2 * ss - cs_tau * cs + s_tau2 * cc)))
```

NVIDIA V100, 1e4 float64 samples, **322x speedup**

CUSTOM RAW CUDA KERNELS: LOMBSCARGLE

SciPy Signal (_spectral.pyx)

```
import numpy as np
cimport numpy as np
cimport cython

@cython.boundscheck(False)
def _lombscargle(np.ndarray[np.float64_t, ndim=1] x,
                np.ndarray[np.float64_t, ndim=1] y,
                np.ndarray[np.float64_t, ndim=1] freqs):

    for i in range(freqs.shape[0]):

        # Code not shown

        for j in range(x.shape[0]):

            # Code not shown

        pgram[i] = 0.5 * (((c_tau * xc + s_tau * xs)**2 /
            (c_tau2 * cc + cs_tau * cs + s_tau2 * ss)) +
            ((c_tau * xs - s_tau * xc)**2 /
            (c_tau2 * ss - cs_tau * cs + s_tau2 * cc)))
```

cuSignal

```
_cupy_lombscargle_src = Template(
  """
$header
extern "C" {
  __global__ void _cupy_lombscargle(const int x_shape,
        const int freqs_shape,
        const ${datatype} * __restrict__ x,
        const ${datatype} * __restrict__ y,
        const ${datatype} * __restrict__ freqs,
        ${datatype} * __restrict__ pgram,
        const ${datatype} * __restrict__ y_dot
        ) {
    const int tx {
        static_cast<int>( blockIdx.x * blockDim.x + threadIdx.x ) };
    const int stride { static_cast<int>( blockDim.x * gridDim.x ) };

    for ( int tid = tx; tid < freqs_shape; tid += stride ) {

        // Code not shown

        for ( int j = 0; j < x_shape; j++ ) {

         // Code not shown

       pgram[tid] = 0.5 * (((c_tau * xc + s_tau * xs)**2 /
            (c_tau2 * cc + cs_tau * cs + s_tau2 * ss)) +
            ((c_tau * xs - s_tau * xc)**2 /
            (c_tau2 * ss - cs_tau * cs + s_tau2 * cc)))
```

NVIDIA V100, 1e4 float64 samples, **386x speedup**

MATLAB -> PYTHON (CPU)

Resample and Magnitude

MATLAB (CPU) – 27.23 s

```
% Setup Parameters
fs = 10e6;
num_sig = 2000;
N = 2^15;
up = 5;
down = 3;

% Generate Random Data of size (num_sig x N)
sig_ensemble = rand([num_sig, N]) + 1j*rand([num_sig,
   N]);

tic;
% Resample each row
resample_sig_ensemble = resample(sig_ensemble.', up,
   down).';

% Find magnitude
mag_sig_ensemble =
   abs(fftshift(fft(resample_sig_ensemble,[],2)));
toc;
```

Python (CPU) – 16.143 s

```
import numpy as np
from time import time
from scipy import signal

# Setup parameters
fs = 10e6
num_sig = 2000
N = 2**15
up = 5
down = 3

# Generate Random Data of size (num_sig x N)
sig_ensemble = np.random.rand(num_sig, N) +
  1j*np.random.rand(num_sig, N)

start = time()

# Resample each row
resample_sig_ensemble = signal.resample_poly(
  sig_ensemble, up, down, window=('kaiser', 0.5))

# Find magnitude
mag_sig_ensemble = np.abs(np.fft.fftshift(np.fft.fft(
   resample_sig_ensemble)))

stop = time()

print('Elapsed time: ', stop-start)
```

PYTHON (CPU) -> CUSIGNAL (GPU)

Resample and Magnitude – ~220x Speedup

Python (CPU) – 16.143 s

```
import numpy as np
from time import time
from scipy import signal

# Setup parameters
fs = 10e6
num_sig = 2000
N = 2**15
up = 5
down = 3

# Generate Random Data of size (num_sig x N)
sig_ensemble = np.random.rand(num_sig, N) +
   1j*np.random.rand(num_sig, N)

start = time()

# Resample each row
resample_sig_ensemble = signal.resample_poly(
  sig_ensemble, up, down, window=('kaiser', 0.5))

# Find magnitude
mag_sig_ensemble = np.abs(np.fft.fftshift(np.fft.fft(
  resample_sig_ensemble)))

stop = time()

print('Elapsed time: ', stop-start)
```

cuSignal (A100, GPU) – 73 ms

```
import cupy as cp
from time import time
import cusignal

# Setup parameters
fs = 10e6
num_sig = 2000
N = 2**15
up = 5
down = 3

# Generate Random Data of size (num_sig x N)
sig_ensemble = cp.random.rand(num_sig, N) +
  1j*cp.random.rand(num_sig, N)

start = time()

# Resample each row
resample_sig_ensemble = cusignal.resample_poly(
  sig_ensemble, up, down, window=('kaiser', 0.5))

# Find magnitude
mag_sig_ensemble = cp.abs(cp.fft.fftshift(cp.fft.fft(
  resample_sig_ensemble)))

stop = time()

print('Elapsed time: ', stop-start)
```

ZERO-COPY CONNECTION TO PYTORCH

```python
import cusignal
import cupy as cp
import torch

num_samps = int(1e8)

# Create random array on GPU
sig = cp.random.randn(num_samps, dtype=cp.float64) + 1j*cp.random.randn(num_samps, dtype=cp.float64)

print('GPU Pointer: ', sig.__cuda_array_interface__['data'])

# Move to PyTorch
torch_sig = torch.as_tensor(sig, device='cuda')

print('Torch Pointer: ', torch_sig.__cuda_array_interface__['data'])

>>> GPU Pointer:  (47442242306048, False)
>>> Torch Pointer: (47442242306048, False)
```

Signal Pre-Processing -> DL Training and Inferencing Example in cuSignal Notebooks

ENABLING ONLINE SIGNAL PROCESSING

```
import cusignal
import numpy as np

num_samps = int(1e8)

# Create shared memory between CPU and GPU; similar to np.empty
sig = cusignal.get_shared_mem(num_samps, dtype=np.float64)

print('GPU Pointer: ', sig.__cuda_array_interface__['data'])
print('CPU Pointer: ', sig.__array_interface__['data’])

>>> GPU Pointer:  (47394527903744, False)
>>> CPU Pointer:  (47394527903744, False)
```

cusignal.get_shared_mem wraps Numba’s mapped_array functionality within the CUDA module and enables a direct memory access between CPU and GPU, reducing I/O overhead

Deepwave Digital exhibits real-time signal collection and polyphase resampling at 62.5 MSps with cuSignal and their AIR-T Software Defined Radio platform that includes an NVIDIA Jetson TX2. They also use cuSignal to run a power-detector in real time

… AND FAST I/O OF SIGNAL FILES

GPU-Accelerated SigMF and Binary Read/Write

Using Northeastern University's Oracle SigMF recordings found here, each ~320MB in size (interleaved I/Q data is read on GPU, header is read on CPU)

Reader:

Baseline, CPU (np.fromfile): 128ms

Baseline, GPU (cp.fromfile): 188ms

cuSignal, Pinned/Mapped: 82.2ms

Writer:

Baseline, CPU (np.tofile): 224ms

Baseline, GPU (cp.tofile): 338ms

cuSignal, Pinned/Mapped: 253ms

GET STARTED

Join the cuSignal Community!

79,082 Anaconda Downloads (2020-2021)

478 GitHub Stars

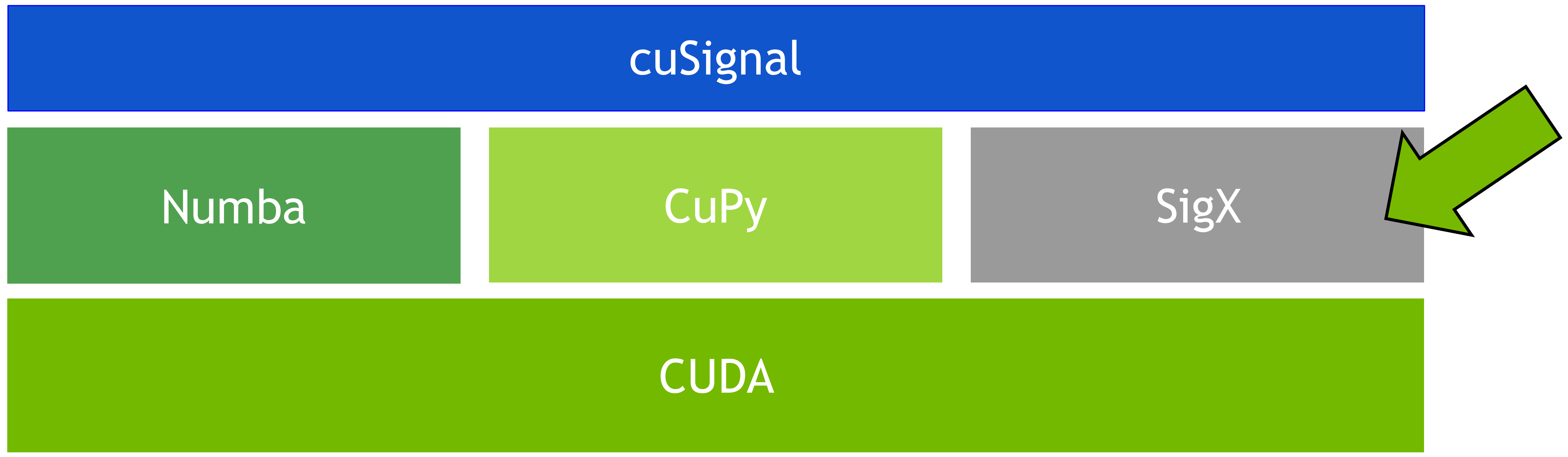
27 Contributors

https://github.com/rapidsai/cusignal

```
conda install -c rapidsai cusignal
```

SIGX: A LOOK AHEAD

Some Programs Cannot Deploy Python-Based Code to Production (Or Need More Speed!)



SigX is a header-only host/device signal processing primitives library for CUDA-compatible cards and modern C++ compilers aimed at providing maximum GPU performance with a straightforward C++ based API

SigX stands alone, but our intention is to expose Python bindings and integrate SigX into the cuSignal API, giving developers the option to develop/deploy in C++ or Python

SIGX – C++ TEMPLATE LIBRARY FOR SIGNAL PROCESSING

Design Overview

Features

👍 Ease of Use:

Straightforward programming model with familiar interfaces (MATLAB/Python-like)

Wraps existing libraries like cuFFT, CUTLASS, and cuRAND

Easily customizable

🔥 High Performance:

Prioritization of efficiently handling streaming data

Separates allocation and processing

Key Concepts

SigX leverages CUDA Managed Memory, freeing the developer from worrying about data locality

Compute operations are performed on Tensor Views: lightweight descriptors of data either on host or device. Supports up to 4-D

Zero data movement view manipulations (clone, slice, permute) that can be chained together

Supports many transforms: FFT, convolution, filtering, GEMM, pointwise operators, and contraction

SIGX EXAMPLE

FFT Based Resampler – No Windowing

Python

```python
import numpy as np
from numpy import fft as fft

N = min(num_samp, num_samp_resamp)
nyq = N // 2 + 1

# Real to complex FFT, time to freq domain
fft_sig = fft.rfft(sig)

# Slice
slice_sig = fft_sig[0:nyq]

# Complex to real IFFT
resamp_sig = fft.irfft(slice_sig, num_samp_resamp)
```

cuSignal

```python
import cusignal

resamp_sig = cusignal.resample(sig, num_samp)
```

SigX

```cpp
uint32_t N = std::min(num_samp, num_samp_resamp);
uint32_t nyq = N / 2 + 1;

sigxTensorData_t<float, 1> sig({num_samp});
sigxTensorData_t<complex, 1> fft_sig({num_samp/2+1});
sigxTensorData_t<float, 1> r_sig({num_samp_resamp});

auto sigView = sig.View();
auto sigViewComplex = fft_sig.View();
auto resampView = r_sig.View();

// Real to Complex FFT, time to freq domain
auto fftPlan = sigxFFTPlan1D_t(sigViewComplex, sigView);
fftPlan.Forward(sigViewComplex, sigView, stream);

// Slice to half spectrum based on num_samp_resamp
auto sliceView = sigViewComplex.Slice({0}, {nyq});

// Complex to Real IFFT, back to time domain
auto resampFFTPlan = sigxFFTPlan1D_t(resampView, sliceView);
resampFFTPlan.Inverse(resampView, sliceView, stream);
```

SIGX PERFORMANCE – A100 GPU

FFT Based Resampler – 1e8 -> 1e5 float32

Python

```
import numpy as np
from numpy import fft as fft

N = min(num_samp, num_samp_resamp)
nyq = N // 2 + 1

# Rea
fft_s

# Slice
slice_sig = fft_sig[0:nyq]

# Complex to real IFFT
resamp_sig = fft.irfft(slice_sig, num_samp_resamp)
```

1500 ms

cuSignal

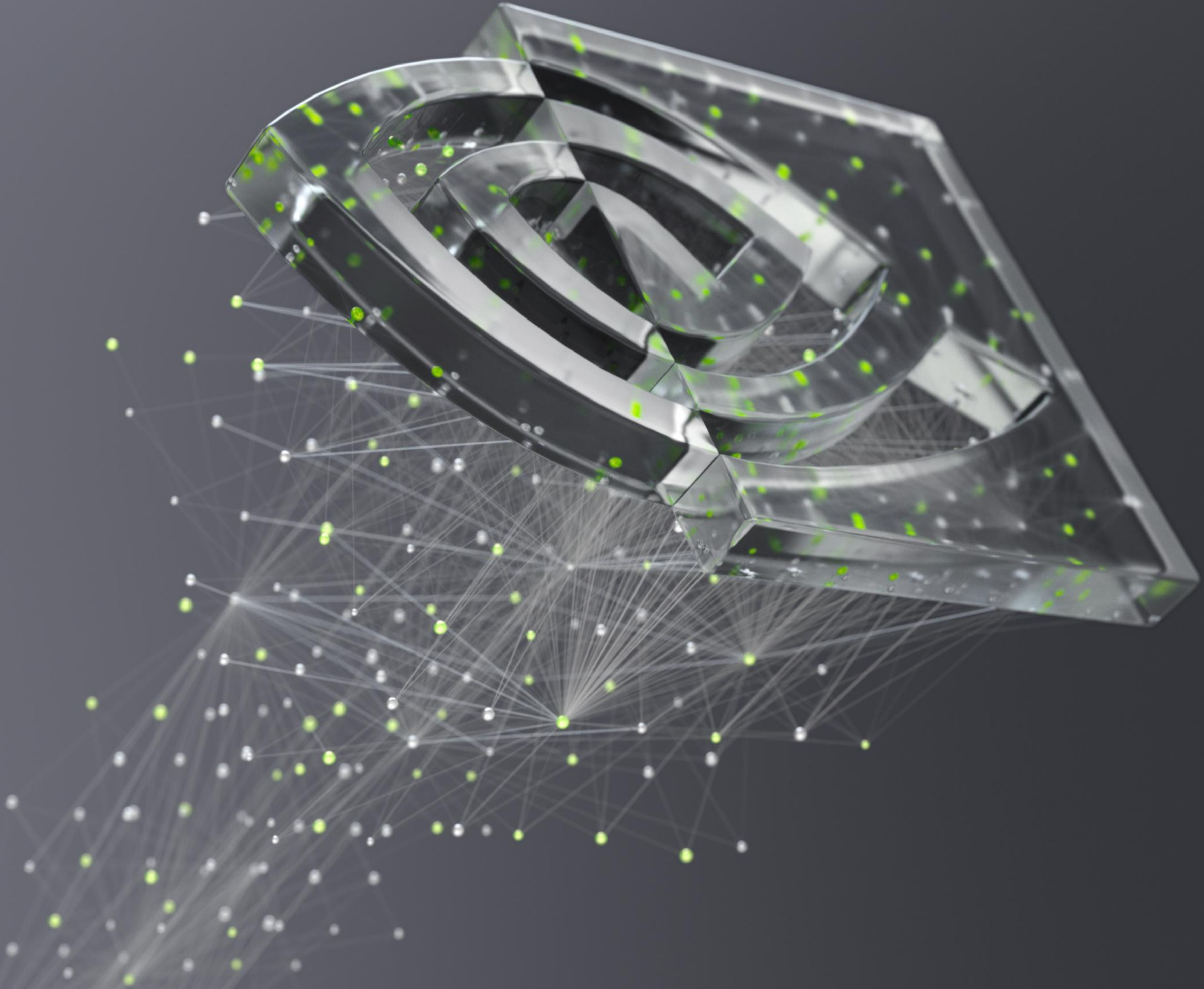
```
import cusignal

resam
```

5.6 ms

SigX

```
uint32_t N = std::min(num_samp, num_samp_resamp);
uint32_t nyq = N / 2 + 1;

sigxTensorData_t<float, 1> sig({num_samp});
sigxTensorData_t<complex, 1> fft_sig({num_samp/2+1});
sigxTensorData_t<float, 1> r_sig({num_samp_resamp});

auto sigView = sig.View();
auto sigViewComplex = fft_sig.View();
auto resamp

// Real to
auto fftPlan = sigxFFTPlan1D_t(sigViewComplex, sigView);
fftPlan.Forward(sigViewComplex, sigView, stream);

// Slice to half spectrum based on num_samp_resamp
auto sliceView = sigViewComplex.Slice({0}, {nyq});

// Complex to Real IFFT, back to time domain
auto resampFFTPlan = sigxFFTPlan1D_t(resampView, sliceView);
resampFFTPlan.Inverse(resampView, sliceView, stream);
```

3.8 ms



NVIDIA®