GPU ACCELERATION IN PYTHON

Matthew Nicely | Solutions Architect | mnicely@nvidia.com

GTC FALL

AGENDA

Getting Started

Background

Testing Setup

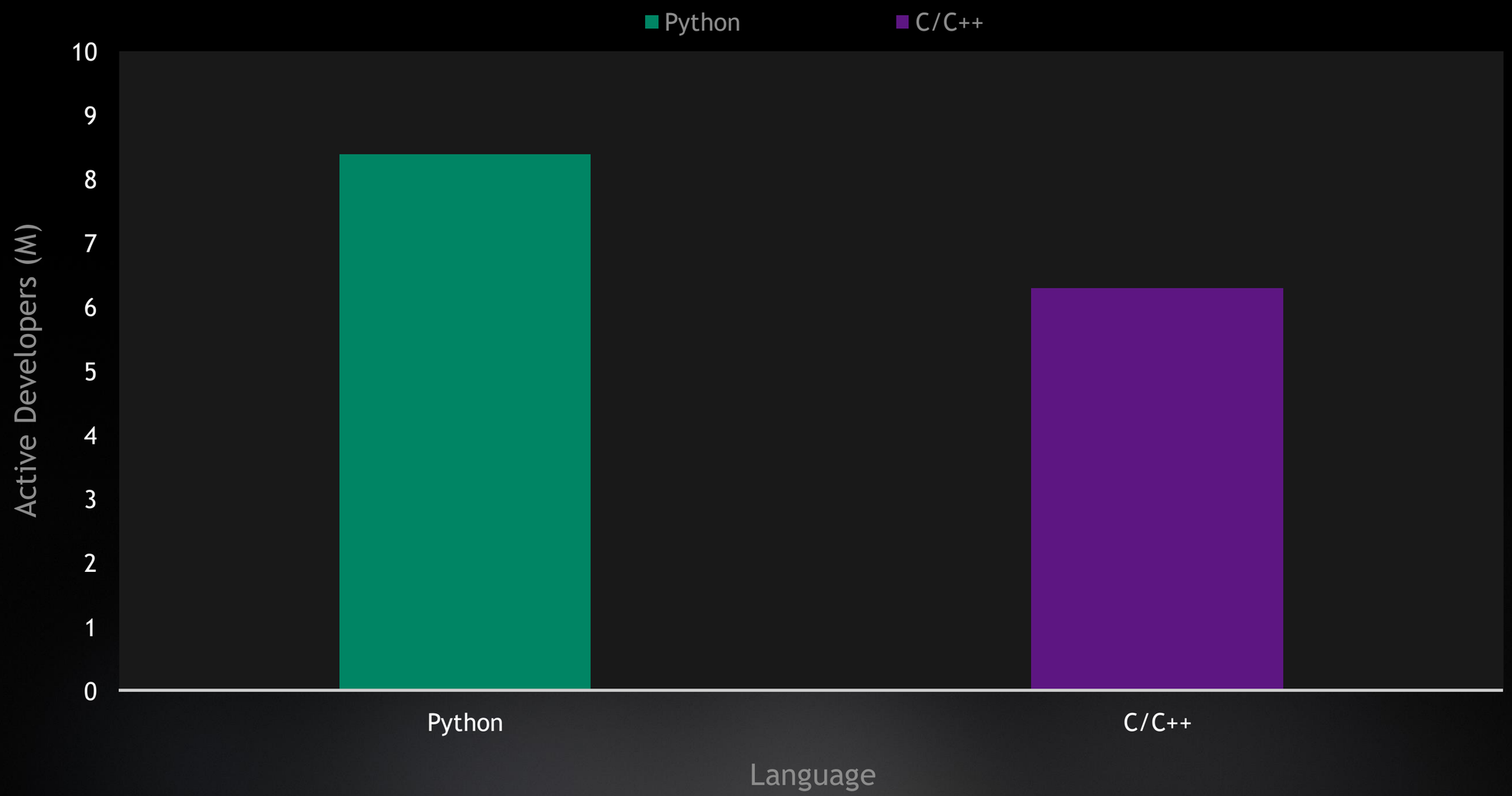
Numba Code

Step through Numba modifications

CuPy Code

Step through CuPy modifications

AVERAGE USERS



Source: https://www.zdnet.com/article/programming-languages-javascript-now-used-by-12-million-developers-but-kotlin-rises-fastest/

WHY ARE WE HERE?

- “Am a Python developer, but really need the performance of CUDA C++.”
- “I have custom arithmetic, i.e. SciPy, that doesn’t exist other GPU accelerated package, i.e. CuPy.”
- “I have custom Numba kernels and I’m nervous about porting code to CuPy’s RawKernel.”
- “Are there any improvements that can be made to my current Numba/CuPy code”?

GETTING STARTED

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
Support multiple dtypes

Source: cuSignal - GPU Accelerating SciPy Signal with Numba and CuPy |SciPy 2020| Adam Thompson
https://www.youtube.com/watch?v=yYlX2bbdXDk&feature=youtu.be

TESTING

- Find and run the code
  - https://github.com/mnicely/gtc_fall
  - conda env create -f gtc_fall.yml
  - bash test_script.sh <iterations>
  - Input size - 2^10; Output size - 2^20
- Performed on a DGX1
  - Tesla V100-SXM2-32GB
  - Intel Xeon CPU E5-2598 v4 @ 2.2GHz
- Setting GPU
  - sudo nvidia-smi -ac 877,1530 -i 0 # Set clocks
  - sudo nvidia-smi -pl 250 -i 0 # Set power levels

PYTHON CODE

SciPy (Lombscargle)

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

    xc = 0.
    xs = 0.
    cc = 0.
    ss = 0.
    cs = 0.

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

        c = cos(freqs[i] * x[j])
        s = sin(freqs[i] * x[j])

        xc += y[j] * c
        xs += y[j] * s
        cc += c * c
        ss += s * s
        cs += c * s

    tau = atan2(2 * cs, cc - ss) / (2 * freqs[i])
    c_tau = cos(freqs[i] * tau)
    s_tau = sin(freqs[i] * tau)
    c_tau2 = c_tau * c_tau
    s_tau2 = s_tau * s_tau
    cs_tau = 2 * c_tau * s_tau

    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)))

return pgram
```

Source: https://github.com/scipy/scipy/blob/master/scipy/signal/_spectral.pyx

PROFILING

NVTX and Nsight Systems

```
from cupy import prof

# Run baseline with scipy.signal.lombscargle
with prof.time_range("scipy_lombscargle", 0):
    cpu_lombscargle = signal.lombscargle(x, y, f)

# Run Numba version
with prof.time_range("numba_lombscargle", 1):
    gpu_lombscargle = lombscargle(d_x, d_y, d_f)

# Copy result to host
gpu_lombscargle = gpu_lombscargle.copy_to_host()

# Compare results
np.testing.assert_allclose(cpu_lombscargle, gpu_lombscargle, 1e-3)

# Run multiple passes to get average
for _ in range(loops):
    with prof.time_range("numba_lombscargle_loop", 2):
        gpu_lombscargle = lombscargle(d_x, d_y, d_f)
```

Example: nsys profile --sample=none --trace=cuda,nvtx --stats=true python3 gtc_fall_numba_v1.py b32 10000


NUMBA CUSTOM KERNEL

NUMBA CODE

Baseline

```
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):

        # Copy data to registers
        freq = freqs[i]

        xc = 0.0
        xs = 0.0
        cc = 0.0
        ss = 0.0
        cs = 0.0
```

New code

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

        c = cos(freq * x[j])
        s = sin(freq * x[j])

        xc += y[j] * c
        xs += y[j] * s
        cc += c * c
        ss += s * s
        cs += c * s

    tau = atan2(2.0 * cs, cc - ss) / (2.0 * freq)
    c_tau = cos(freq * tau)
    s_tau = sin(freq * tau)
    c_tau2 = c_tau * c_tau
    s_tau2 = s_tau * s_tau
    cs_tau = 2.0 * c_tau * s_tau
```

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_numba_v1.py

NUMBA CODE

Version 1

```
def _lombscargle(x, y, freqs, pgram, y_dot):

    if (pgram.dtype == 'float32'):
        numba_type = float32
    elif (pgram.dtype == 'float64'):
        numba_type = float64

    device_id = cp.cuda.Device()
    numSM = device_id.attributes["MultiProcessorCount"]
    threadsperblock = (128, )
    blockspergrid = (numSM * 20,)

    sig = _numba_lombscargle_signature(numba_type)
    kernel = cuda.jit(sig)(_numba_lombscargle)

    kernel[blockspergrid, threadsperblock](x, y, freqs, pgram, y_dot)

    cuda.synchronize()
```

Allow for multiple kernels based on data type

Determine number of blocks for grid-stride looping

Compile Numba kernel

Launch Numba Kernel

Block host until finished

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_numba_v1.py

NUMBA COMPARISON

	Double Precision				Single Precision			
	Registers	JIT (ms)	Kernel (ms)	Speed Up	Registers	JIT (ms)	Kernel (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
Numba (Baseline)	66	413.2	2.7	1419.35	58	445.5	2.1	1744.38
Numba (User Cache)								
Numba (Data Type)								
Numba (Fast Math)								
Numba (Max Registers)								

Baseline Numba kernel, implicit casting on single precision.

NUMBA CODE

Cached Kernel

```
def _lombscargle(x, y, freqs, pgram, y_dot):

    if (pgram.dtype == 'float32'):
        numba_type = float32
    elif (pgram.dtype == 'float64'):
        numba_type = float64

    if (str(numba_type)) in _kernel_cache:
        kernel = _kernel_cache[(str(numba_type))]
    else:
        sig = _numba_lombscargle_signature(numba_type)
        kernel = _kernel_cache[(str(numba_type))] = cuda.jit(sig)(_numba_lombscargle)

    device_id = cp.cuda.Device()
    numSM = device_id.attributes["MultiProcessorCount"]
    threadsperblock = (128, )
    blockspergrid = (numSM * 20,)

    kernel[blockspergrid, threadsperblock](x, y, freqs, pgram, y_dot)
```

Check if compiled kernel exist

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_numba_v2.py

NUMBA COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
Numba (Baseline)	66	413.2	2.7	1419.35	58	445.5	2.1	1744.38
Numba (User Cache)	66	420.1	2.7	1419.16	58	465.3	2.1	1756.59
Numba (Data Type)								
Numba (Fast Math)								
Numba (Max Registers)								

Cached compiled kernel in user defined dictionary, skip Numba logic.

NUMBA CODE

Explicit kernel per type

```
from numba import int32, float32

…

def _numba_lombscargle_32(x, y, freqs, pgram, y_dot):

    dtype = float32

    F = int32(cuda.grid(1))
    strideF = int32(cuda.gridsize(1))

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

    for i in range(F, int32(freqs.shape[0]), strideF):

        # Copy data to registers
        freq = dtype(freqs[i])
```

Specify data type

```
        xc = dtype(0.0)
        xs = dtype(0.0)
        cc = dtype(0.0)
        ss = dtype(0.0)
        cs = dtype(0.0)

        for j in range(int32(x.shape[0])):

            c = dtype(cos(dtype(freq * x[j])))
            s = dtype(sin(dtype(freq * x[j])))

            xc += dtype(y[j] * c)
            xs += dtype(y[j] * s)
            cc += dtype(c * c)
            ss += dtype(s * s)
            cs += dtype(c * s)

    …
```

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_numba_v3.py

NUMBA COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
Numba (Baseline)	66	413.2	2.7	1419.35	58	445.5	2.1	1744.38
Numba (User Cache)	66	420.1	2.7	1419.16	58	465.3	2.1	1756.59
Numba (Data Type)	66	481.2	2.7	1428.01	40	479.9	2.0	1862.57
Numba (Fast Math)								
Numba (Max Registers)								

Add data type casting to kernel to minimize register usage.

NUMBA CODE

Using --use_fast_math

```
def _lombscargle(x, y, freqs, pgram, y_dot):

    if (pgram.dtype == 'float32'):
        numba_type = float32
    elif (pgram.dtype == 'float64'):
        numba_type = float64

    if (str(numba_type)) in _kernel_cache:
        kernel = _kernel_cache[(str(numba_type))]
    else:
        sig = _numba_lombscargle_signature(numba_type)
        if (pgram.dtype == 'float32'):
            kernel = _kernel_cache[(str(numba_type))] = cuda.jit(sig, fastmath=True)(_numba_lombscargle_32)
        elif (pgram.dtype == 'float64'):
            kernel = _kernel_cache[(str(numba_type))] = cuda.jit(sig, fastmath=True)(_numba_lombscargle_64)

    …

    kernel[blockspergrid, threadsperblock](x, y, freqs, pgram, y_dot)
```

Add fast math flag

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_numba_v4.py

NUMBA COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
Numba (Baseline)	66	413.2	2.7	1419.35	58	445.5	2.1	1744.38
Numba (User Cache)	66	420.1	2.7	1419.16	58	465.3	2.1	1756.59
Numba (Data Type)	66	481.2	2.7	1428.01	40	479.9	2.0	1862.57
Numba (Fast Math)	66	495.5	2.7	1428.44	33	478.3	2.0	1888.72
Numba (Max Registers)								

Pass -use_fast_math flag, only effective on single-precision.

NUMBA CODE

Using --max_registers

```
def _lombscargle(x, y, freqs, pgram, y_dot):

    if (pgram.dtype == 'float32'):
        numba_type = float32
    elif (pgram.dtype == 'float64'):
        numba_type = float64

    if (str(numba_type)) in _kernel_cache:
        kernel = _kernel_cache[(str(numba_type))]
    else:
        sig = _numba_lombscargle_signature(numba_type)
        if (pgram.dtype == 'float32'):
            kernel = _kernel_cache[(str(numba_type))] = cuda.jit(sig, fastmath=True, max_registers=32)(_numba_lombscargle_32)
        elif (pgram.dtype == 'float64'):
            kernel = _kernel_cache[(str(numba_type))] = cuda.jit(sig, fastmath=True, max_registers=64)(_numba_lombscargle_64)

    …

    kernel[blockspergrid, threadsperblock](x, y, freqs, pgram, y_dot)
```

Add
max registers

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_numba_v5.py

NUMBA COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
Numba (Baseline)	66	413.2	2.7	1419.35	58	445.5	2.1	1744.38
Numba (User Cache)	66	420.1	2.7	1419.16	58	465.3	2.1	1756.59
Numba (Data Type)	66	481.2	2.7	1428.01	40	479.9	2.0	1862.57
Numba (Fast Math)	66	495.5	2.7	1428.44	33	478.3	2.0	1888.72
Numba (Max Registers)	64	483.4	2.7	1422.03	32	478.3	1.9	1987.20

Add max_registers flag to limit the number of register per thread

CUPY RAW KERNEL

CUPY CODE

Baseline

Stored as string

```
_cupy_lombscargle_src = Template("""
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 ) };

    ${datatype} yD {};
```

```
if ( y_dot[0] == 0 ) {
    yD = 1.0;
} else {
    yD = 2.0 / y_dot[0];
}

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

    ${datatype} freq { freqs[tid] };

    ${datatype} xc {};
    ${datatype} xs {};
    ${datatype} cc {};
    ${datatype} ss {};
    ${datatype} cs {};
    ${datatype} c {};
    ${datatype} s {};
    …
```

Explicitly specify data types

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_cupy_v1.py

CUPY CODE

Baseline

```
def _lombscargle(x, y, freqs, pgram, y_dot):

    if (pgram.dtype == 'float32'):
        c_type = "float"
    elif (pgram.dtype == 'float64'):
        c_type = "double"

    device_id = cp.cuda.Device()
    numSM = device_id.attributes["MultiProcessorCount"]
    threadsperblock = (128, )
    blockspergrid = (numSM * 20,)

    src = _cupy_lombscargle_src.substitute(datatype=c_type)
    module = cp.RawModule(code=src, options=("-std=c++11",))
    kernel = module.get_function("_cupy_lombscargle")

    kernel_args = (x.shape[0], freqs.shape[0], x, y, freqs, pgram, y_dot,)

    kernel(blockspergrid, threadsperblock, kernel_args)

    cp.cuda.runtime.deviceSynchronize()
```

Allow for multiple kernels based on data type

Determine number of blocks for grid-stride looping

Compile CuPy kernel

Launch CuPy Kernel

Block host until finished

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_cupy_v1.py

CUPY COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
CuPy (Baseline)	58	113.4	2.0	1820.48	36	108.7	1.4	2573.03
CuPy (User Cache)								
CuPy (Data Type)								
CuPy (Fast Math)								
CuPy (Fatbin)								
CuPy (Launch Bounds)								

Baseline CuPy kernel, less registers with absence of type promotion.

CUPY CODE

Cached Kernel

```
def _lombscargle(x, y, freqs, pgram, y_dot):

    if (pgram.dtype == 'float32'):
        c_type = "float"
    elif (pgram.dtype == 'float64'):
        c_type = "double"

    if (str(c_type)) in _kernel_cache:
        kernel = _kernel_cache[(str(c_type))]
    else:
        src = _cupy_lombscargle_src.substitute(datatype=c_type)
        module = cp.RawModule(code=src, options=("-std=c++11",))
        kernel = _kernel_cache[(str(c_type))] = module.get_function("_cupy_lombscargle")

    …

    kernel_args = ( x.shape[0], freqs.shape[0], x, y, freqs, pgram, y_dot, )

    kernel(blockspergrid, threadsperblock, kernel_args)
```

Check if compiled kernel exist

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_cupy_v2.py

CUPY COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
CuPy (Baseline)	58	113.4	2.0	1820.48	36	108.7	1.7	2123.03
CuPy (User Cache)	58	109.3	1.6	2456.03	36	117.9	0.9	4171.66
CuPy (Data Type)								
CuPy (Fast Math)								
CuPy (Fatbin)								
CuPy (Launch Bounds)								

Cached compiled kernel in user defined dictionary, skip Numba logic.

CUPY CODE

Explicit kernel per type

```
_cupy_lombscargle_src =  r"""
extern "C" {
__global__ void _cupy_lombscargle_float32(
        const int x_shape,
        const int freqs_shape,
        const float * __restrict__ x,
        const float * __restrict__ y,
        const float * __restrict__ freqs,
        float * __restrict__ pgram,
        const float * __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 ) };

    float yD {};
    if ( y_dot[0] == 0 ) {
        yD = 1.0f;
    } else {
        yD = 2.0f / y_dot[0];
    }

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

        float freq { freqs[tid] };

        float xc {};
        float xs {};
        float cc {};
        float ss {};
        float cs {};
        float c {};
        float s {};
        …
```

Kernel per type

Explicitly specify data types

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_cupy_v3.py

CUPY COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
CuPy (Baseline)	58	113.4	2.0	1820.48	36	108.7	1.7	2123.03
CuPy (User Cache)	58	109.3	1.6	2456.03	36	117.9	0.9	4171.66
CuPy (Data Type)	58	99.5	1.6	2408.45	32	109.2	0.9	4077.69
CuPy (Fast Math)								
CuPy (Fatbin)								
CuPy (Launch Bounds)								

Add data type casting to kernel to minimize register usage.

CUPY CODE

Using --use_fast_math

```
def _lombscargle(x, y, freqs, pgram, y_dot):

    if (str(pgram.dtype)) in _kernel_cache:
        kernel = _kernel_cache[(str(pgram.dtype))]
    else:
        module = cp.RawModule(code=_cupy_lombscargle_src, options=("-std=c++11", "--use_fast_math")
        _kernel_cache[(str(pgram.dtype))] = module.get_function("_cupy_lombscargle_" + str(pgram.dtype))
        kernel = _kernel_cache[(str(pgram.dtype))]

    device_id = cp.cuda.Device()
    numSM = device_id.attributes["MultiProcessorCount"]
    threadsperblock = (128, )
    blockspergrid = (numSM * 20,)

    kernel_args = ( x.shape[0], freqs.shape[0], x, y, freqs, pgram, y_dot, )

    kernel(blockspergrid, threadsperblock, kernel_args)
```

Add fast math flag

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_cupy_v4.py

CUPY COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
CuPy (Baseline)	58	113.4	2.0	1820.48	36	108.7	1.7	2123.03
CuPy (User Cache)	58	109.3	1.6	2456.03	36	117.9	0.9	4171.66
CuPy (Data Type)	58	99.5	1.6	2408.45	32	109.2	0.9	4077.69
CuPy (Fast Math)	58	107.2	1.6	2415.21	32	107.6	0.2	16026.22
CuPy (Fatbin)								
CuPy (Launch Bounds)								

Pass -use_fast_math flag, only effective on single-precision.

CUPY CODE

Loading from fatbin

```
def _lombscargle(x, y, freqs, pgram, y_dot):

    if (str(pgram.dtype)) in _kernel_cache:
        kernel = _kernel_cache[(str(pgram.dtype))]
    else:
        module = cp.RawModule(path="./_lombscargle.fatbin")
        kernel = _kernel_cache[(str(pgram.dtype))] = module.get_function("_cupy_lombscargle_" + str(pgram.dtype))

    device_id = cp.cuda.Device()
    numSM = device_id.attributes["MultiProcessorCount"]
    threadsperblock = (128, )
    blockspergrid = (numSM * 20,)

    kernel_args = ( x.shape[0], freqs.shape[0], x, y, freqs, pgram, y_dot, )

    kernel(blockspergrid, threadsperblock, kernel_args)
```

Load precompile kernels from fatbin

Source: https://github.com/mnicely/gtc_fall/blob/master/gtc_fall_cupy_v5.py

CUPY CODE

Loading from fatbin

```
nvcc --fatbin -std=c++11 --use_fast_math \
    --generate-code arch=compute_35,code=sm_35 \
    --generate-code arch=compute_35,code=sm_37 \
    --generate-code arch=compute_50,code=sm_50 \
    --generate-code arch=compute_50,code=sm_52 \
    --generate-code arch=compute_53,code=sm_53 \
    --generate-code arch=compute_60,code=sm_60 \
    --generate-code arch=compute_61,code=sm_61 \
    --generate-code arch=compute_62,code=sm_62 \
    --generate-code arch=compute_70,code=sm_70 \
    --generate-code arch=compute_72,code=sm_72 \
    --generate-code arch=compute_75,code=[sm_75,compute_75] \
    _lombscargle.cu -odir .
```

Compile SASS for all architectures

Compile PTX for only 7.5

Source: https://github.com/mnicely/gtc_fall/blob/master/_lombscargle.cu

CUPY COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
CuPy (Baseline)	58	113.4	2.0	1820.48	36	108.7	1.7	2123.03
CuPy (User Cache)	58	109.3	1.6	2456.03	36	117.9	0.9	4171.66
CuPy (Data Type)	58	99.5	1.6	2408.45	32	109.2	0.9	4077.69
CuPy (Fast Math)	58	107.2	1.6	2415.21	32	107.6	0.2	16026.22
CuPy (Fatbin)	48	4.8	1.5	2533.80	32	6.6	0.2	15964.36
CuPy (Launch Bounds)								

Load precompiled kernels from fatbin (nvcc), removing compile time on first pass.

CUPY CODE

Using __launch_bounds__()

```
template<typename T>
__device__ void _cupy_lombscargle_double( const int x_shape,
                                          const int freqs_shape,
                                          const T *__restrict__ x,
                                          const T *__restrict__ y,
                                          const T *__restrict__ freqs,
                                          T *__restrict__ pgram,
                                          const T *__restrict__ y_dot ) {
                                  …
}

extern "C" __global__ void __launch_bounds__( 128 ) _cupy_lombscargle_float64( const int x_shape,
                                                                               const int freqs_shape,
                                                                               const double *__restrict__ x,
                                                                               const double *__restrict__ y,
                                                                               const double *__restrict__ freqs,
                                                                               double *__restrict__ pgram,
                                                                               const double *__restrict__ y_dot ) {
    _cupy_lombscargle_double<double>( x_shape, freqs_shape, x, y, freqs, pgram, y_dot );
}
```

Template wrapper

Same as threads per block

Source: https://github.com/mnicely/gtc_fall/blob/master/_lombscargle_lb.cu

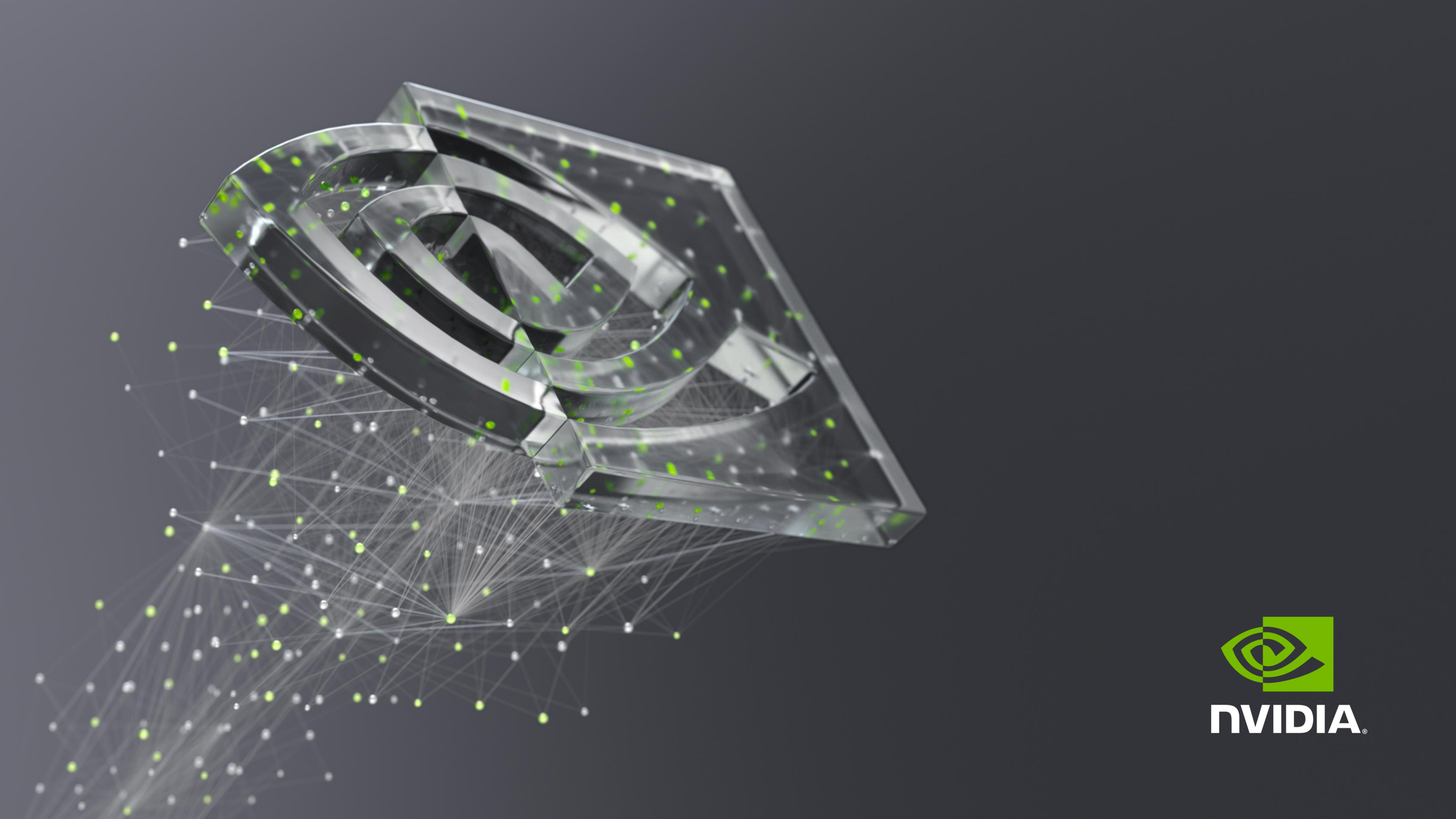
CUPY COMPARISON

	Double Precision				Single Precision			
	Registers	First Pass (ms)	Average (ms)	Speed Up	Registers	First Pass (ms)	Average (ms)	Speed Up
SciPy	-	3,803.6	3,809.6	1.00	-	3,691.1	3,679.5	1.00
CuPy (Baseline)	58	113.4	2.0	1820.48	36	108.7	1.4	2573.03
CuPy (User Cache)	58	109.3	1.6	2456.03	36	117.9	0.9	4187.66
CuPy (Data Type)	58	99.5	1.6	2408.45	32	109.2	0.9	4105.69
CuPy (Fast Math)	58	107.2	1.6	2415.21	32	107.6	0.2	17335.22
CuPy (Fatbin)	48	4.8	1.5	2533.80	32	3.5	0.2	17611.36
CuPy (Launch Bounds)	**48**	**4.8**	**1.5**	**2469.85**	**31**	**3.0**	**0.2**	**17278.29**

Add launch_bounds to kernels, allowing further compiler optimizations.

FINAL THOUGHTS

- User level caches can reduce kernel launch times.
- Kernel performance of Numba is very similar to CuPy for double(-complex) precisions.
- Fast math can be an attractive feature but be mindful of precision loss.
- Precompiled kernels from fatbin can reduce first pass execution times.
- Numba to CuPy requires explicit data types and variable management.
- Mileage of optimizations varies based on kernel.
- Checkout https://courses.nvidia.com/courses/course-v1:DLI+C-AC-02+V1/about.
- Checkout https://github.com/rapidsai/cusignal for more optimization techniques.


NVIDIA