# GPU Computing With Apache Spark And Python

Siu Kwan Lam

Continuum Analytics

Spark summit

SPARK SUMMIT 2016

DATA SCIENCE AND ENGINEERING AT SCALE

ANACONDA®

- I'm going to use Anaconda throughout this presentation.

- Anaconda is a free Mac/Win/Linux Python distribution:
  - Based on conda, an open source package manager
  - Installs both *Python* and *non-Python* dependencies
  - Easiest way to get the software I will talk about today

- https://www.continuum.io/downloads

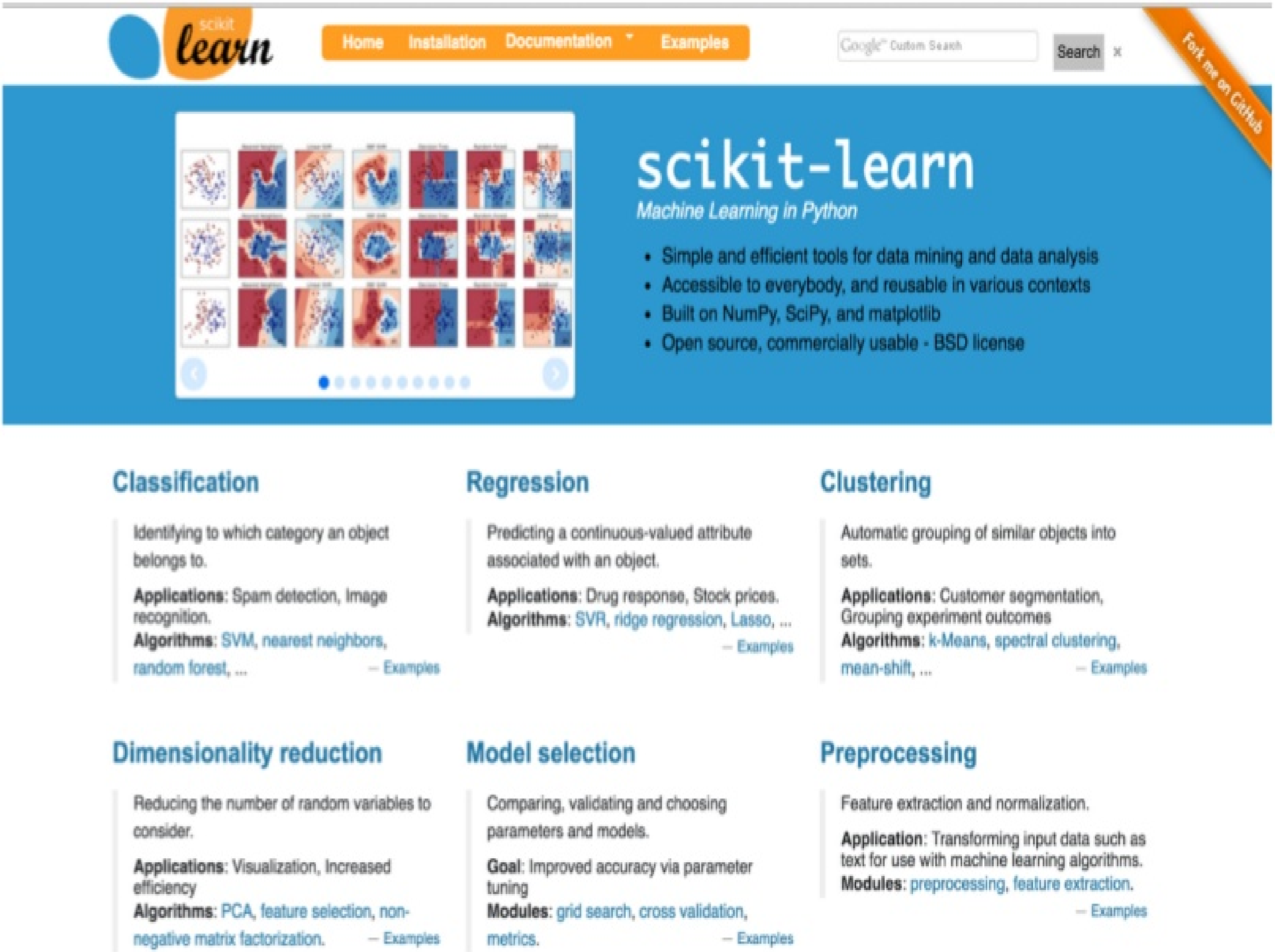
# Overview

- Why Python?
- Using GPU in PySpark
  - An example: Image registration
  - Accelerate: Drop-in GPU-accelerated functions
  - Numba: JIT Custom GPU-accelerated functions
- Tips & Tricks

# WHY PYTHON?

# Why is Python so popular?

- Straightforward, productive language for *system administrators, programmers, scientists, analysts and hobbyists*
- Great community:
  - Lots of tutorial and reference materials
  - Vast ecosystem of useful libraries
  - Easy to interface with other languages


scikit learn
Home
Installation
Documentation
Examples
Google Custom Search
Search
Fork me on GitHub
scikit-learn
Machine Learning in Python
Simple and efficient tools for data mining and data analysis
Accessible to everybody, and reusable in various contexts
Built on NumPy, SciPy, and matplotlib
Open source, commercially usable - BSD license
Classification
Identifying to which category an object belongs to.
Applications: Spam detection, Image recognition.
Algorithms: SVM, nearest neighbors, random forest, ... — Examples
Regression
Predicting a continuous-valued attribute associated with an object.
Applications: Drug response, Stock prices.
Algorithms: SVR, ridge regression, Lasso, ... — Examples
Clustering
Automatic grouping of similar objects into sets.
Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, ... — Examples
Dimensionality reduction
Reducing the number of random variables to consider.
Applications: Visualization, Increased efficiency
Algorithms: PCA, feature selection, non-negative matrix factorization. — Examples
Model selection
Comparing, validating and choosing parameters and models.
Goal: Improved accuracy via parameter tuning
Modules: grid search, cross validation, metrics. — Examples
Preprocessing
Feature extraction and normalization.
Application: Transforming input data such as text for use with machine learning algorithms.
Modules: preprocessing, feature extraction. — Examples

# But… Python is slow!

- *Pure, interpreted Python is slow.*
- Python excels at interfacing with other languages used in HPC:
    - C: ctypes, CFFI, Cython
    - C++: Cython, Boost.Python
    - FORTRAN: f2py
- Secret: Most scientific Python packages put the speed critical sections of their algorithms in a compiled language.

# Is there another way?

- Switching languages for speed in your projects can be a little clunky
- Generating compiled functions for the wide range of data types can be tedious
- How can we use cutting edge hardware, like GPUs?

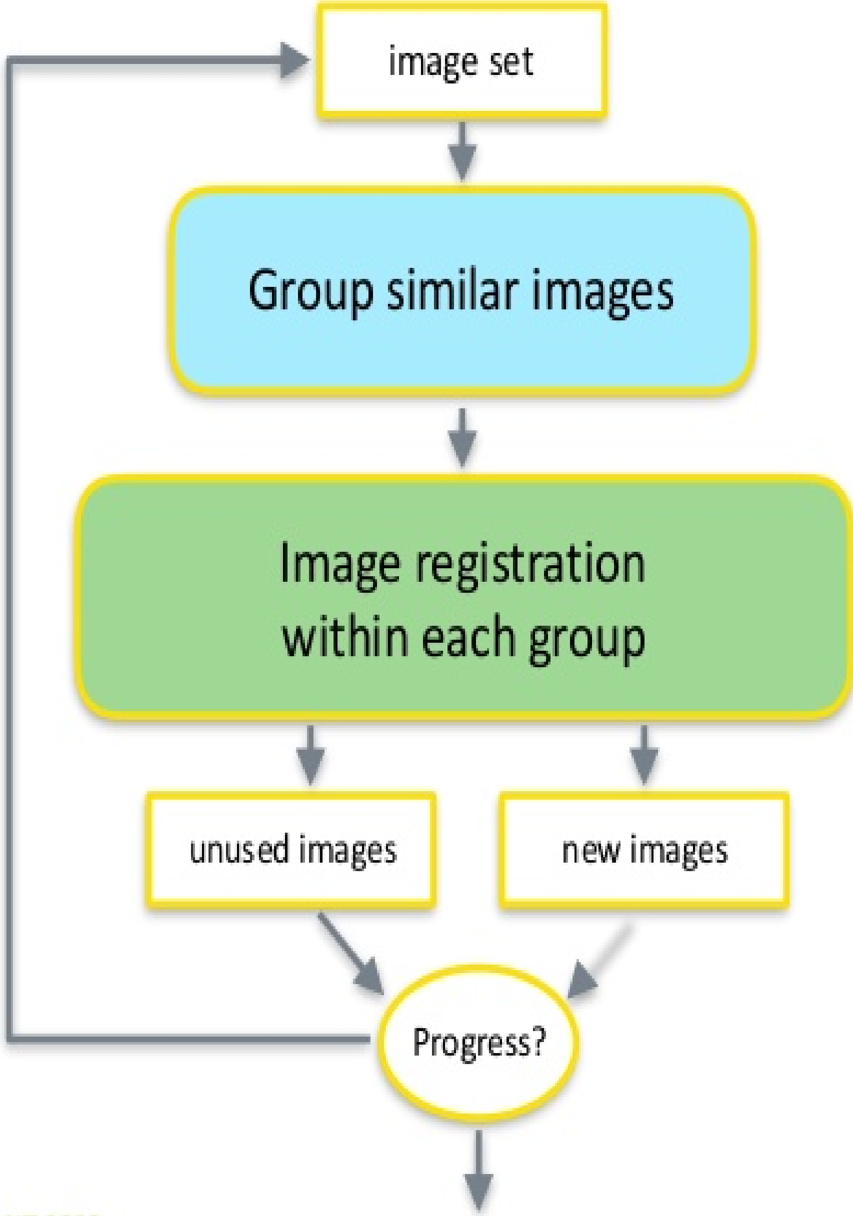
An example for using GPU in PySpark

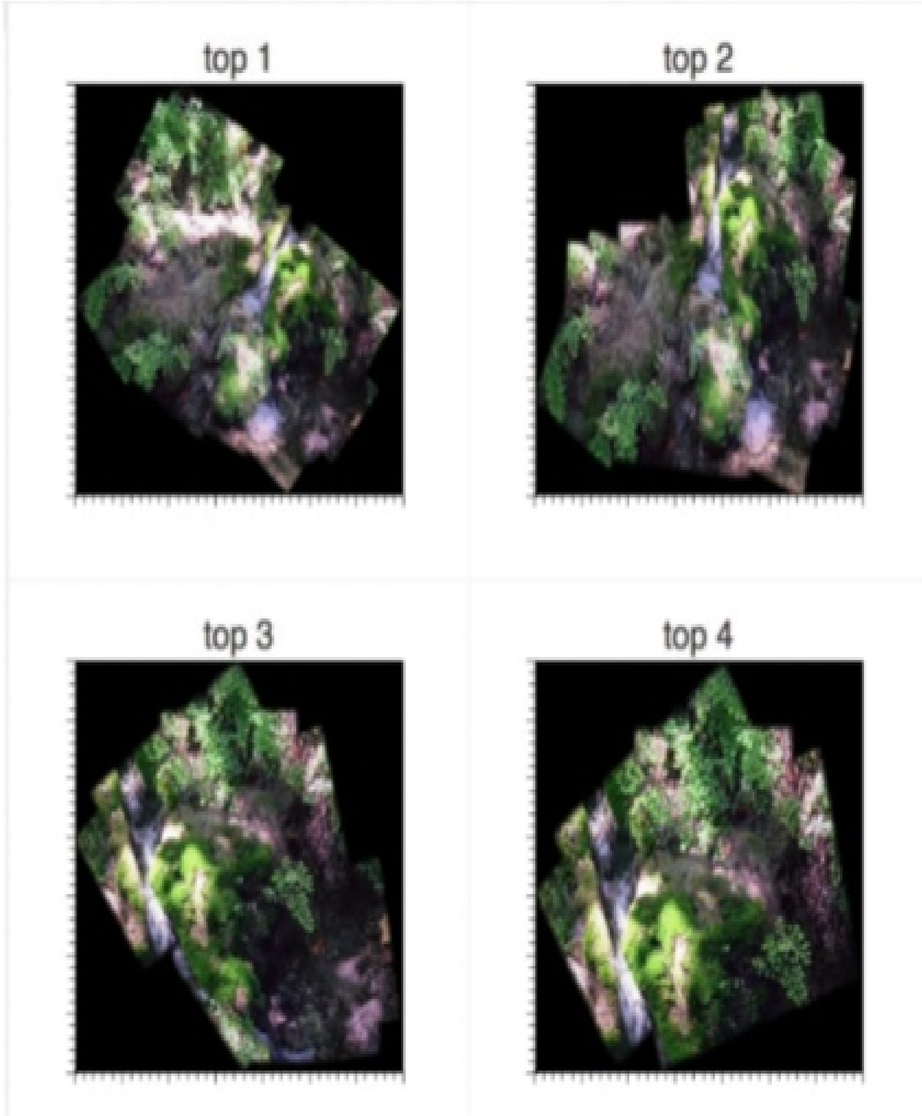
# IMAGE REGISTRATION

# Image Registration

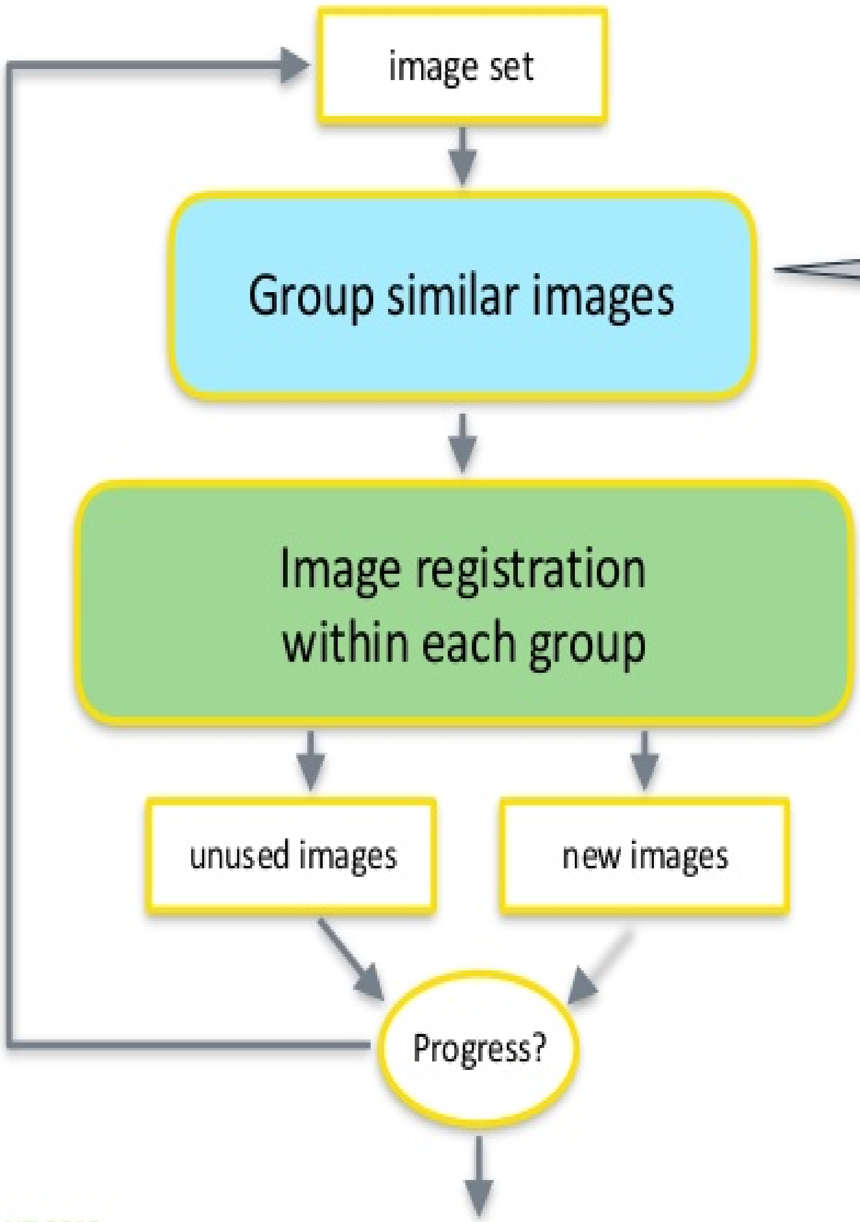
- An experiment to demonstrate GPU usage
- The problem:
  - stitch image fragments
  - fragments are randomly orientated, translated and scaled.
- phase-correlation for image registration
  - FFT heavy

# Basic Algorithm



# Basic Algorithm




- KMeans Clustering
- Histogram as metric

- FFT 2D heavy
- Expensive
- Potential speedup on GPU

# Setup

- conda can create a local environment for Spark for you:

```
conda create -n spark -c anaconda-cluster python=3.5 spark \
                                    accelerate ipython-notebook

source activate spark

IPYTHON_OPTS="notebook" ./bin/pyspark  # starts jupyter notebook
```

# Performance Bottleneck

- Most of the time spent in 2D FFT

```
def cross_power_spectrum(im0, im1):
    f0 = numpy.fft.fft2(im0)
    f1 = numpy.fft.fft2(im1)
    eps = 1e-15
    cps = (f0 * f1.conjugate()) / (abs(f0) * abs(f1) + eps)
    return abs(numpy.fft.ifft2(cps))
```

# ACCELERATE
# DROP-IN GPU-ACCELERATED FUNCTIONS

# Accelerate

- Commercial licensed
- Hardware optimized numerical functions
- SIMD optimized via MKL
- GPU accelerated via CUDA

# CUDA Library Bindings: cuFFT

In [2]:
```
from accelerate.cuda import fft

arr = np.random.random(10**6).astype(np.float32)
out = np.zeros_like(arr, dtype=np.complex64)
fft.fft(arr, out)
```

Out[2]:
```
array([  5.00258000e+05 +0.j        ,  -1.29911041e+01-79.63054657j,
        -2.77468071e+01+74.94405365j, ...,   1.35268259e+00 +1.04822063j,
         1.32095528e+00 +1.1744678j ,   8.91982377e-01 +1.14550018j], dtype=complex64)
```

In [3]:
```
%%timeit

res1 = np.fft.fft(arr)
```

MKL accelerated FFT

```
100 loops, best of 3: 16.6 ms per loop
```

In [4]:
```
%%timeit

res2 = fft.fft(arr, out)
```

**>2x speedup** incl. host<->device round trip on GeForce GT 650M

```
100 loops, best of 3: 7.33 ms per loop
```

# CPS with GPU drop-in

- Replace *numpy* FFT with *accelerate* version

```
from accelerate.cuda import fft as cufft

def cross_power_spectrum(im0, im1):
    f0 = im0.astype(numpy.complex64)
    f1 = im1.astype(numpy.complex64)
    cufft.fft_inplace(f0)
    cufft.fft_inplace(f1)
    eps = 1e-15
    cps = (f0 * f1.conjugate()) / (abs(f0) * abs(f1) + eps)
    cufft.ifft_inplace(cps)
    return abs(cps)
```

# CPS with GPU drop-in

- Replace *numpy* FFT with *accelerate* version

```python
from accelerate.cuda import fft as cufft

def cross_power_spectrum(im0, im1):
    f0 = im0.astype(numpy.complex64)
    f1 = im1.astype(numpy.complex64)
    cufft.fft_inplace(f0)
    cufft.fft_inplace(f1)
    eps = 1e-15
    cps = (f0 * f1.conjugate()) / (abs(f0) * abs(f1) + eps)
    cufft.ifft_inplace(cps)
    return abs(cps)
```

CPU-GPU transfer

CPU-GPU transfer

# NUMBA
# JIT CUSTOM GPU-ACCELERATED FUNCTIONS

# Numba

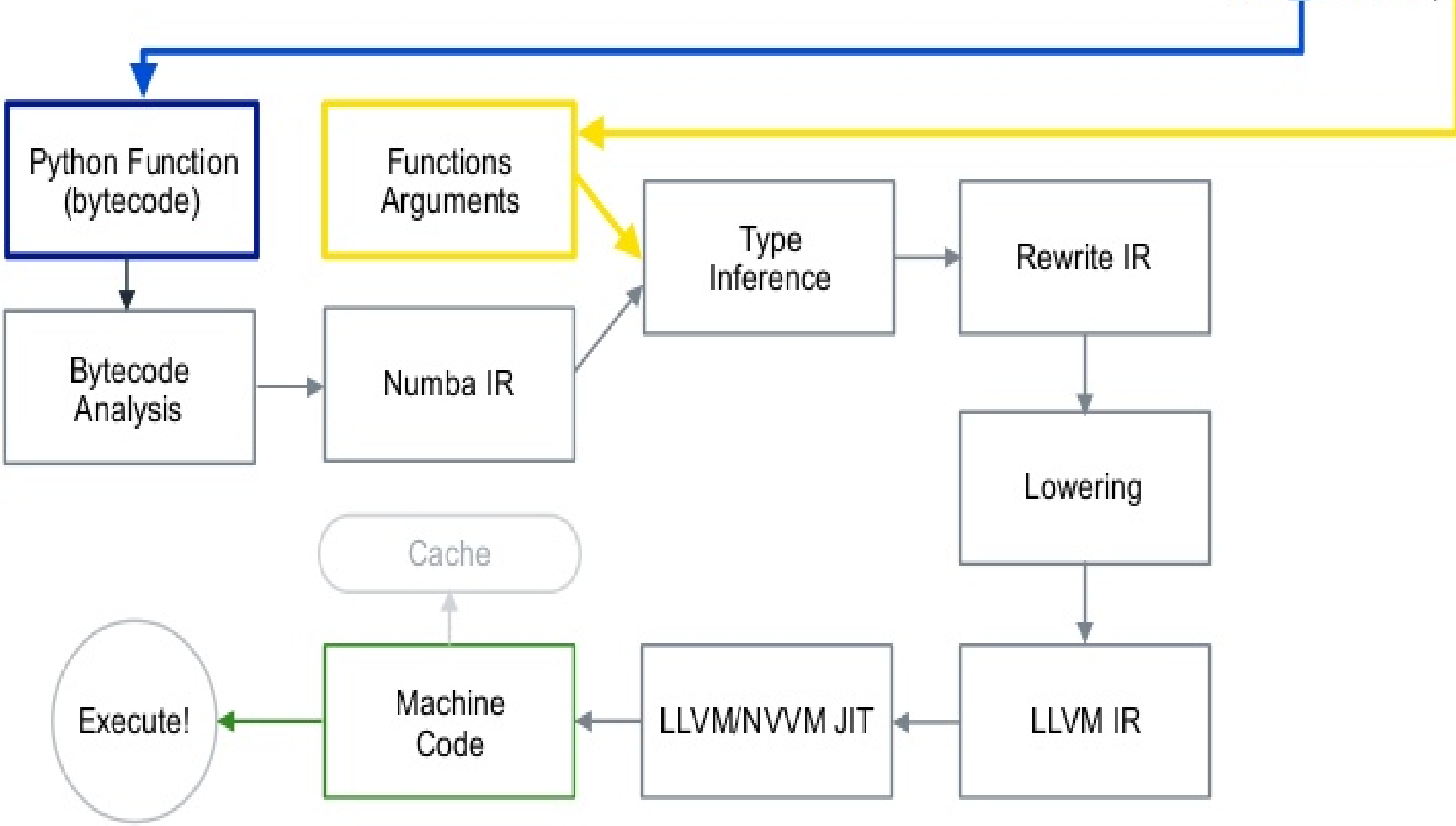
- Opensource licensed
- A Python JIT as a CPython library
- Array/numerical subset
- Targets CPU and GPU

# Supported Platforms

| OS | HW | SW |
|---|---|---|
| • Windows (7 and later) | • 32 and 64-bit x86 CPUs | • Python 2 and 3 |
| • OS X (10.7 and later) | • **CUDA-capable NVIDIA GPUs** | • NumPy 1.7 through 1.11 |
| • Linux (~RHEL 5 and later) | • HSA-capable AMD GPUs | |
| | • Experimental support for ARMv7 (Raspberry Pi 2) | |

# How Does Numba Work?

```
@jit
def do_math(a, b):
    ...
>>> do_math(x, y)
```



# Ufuncs—Map operation for ND arrays

```
In [1]: import numpy as np
        import math
        from numba import vectorize


        @vectorize(["float32(float32, float32)",
                    "float64(float64, float64)"], target='cpu')
        def cpu_some_trig(x, y):
            return math.cos(x) + math.sin(y)


        @vectorize(["float32(float32, float32)",
                    "float64(float64, float64)"], target='cuda')
        def cuda_some_trig(x, y):
            return math.cos(x) + math.sin(y)
```

# Ufuncs—Map operation for ND arrays

```
In [1]: import numpy as np
        import math
        from numba import vectorize


        @vectorize(["float32(float32, float32)",
                    "float64(float64, float64)"], target='cpu')
        def cpu_some_trig(x, y):
            return math.cos(x) + math.sin(y)


        @vectorize(["float32(float32, float32)",
                    "float64(float64, float64)"], target='cuda')
        def cuda_some_trig(x, y):
            return math.cos(x) + math.sin(y)
```

Decorator for creating ufunc

List of supported type signatures

Code generation target

# GPU Ufuncs Performance

In [2]:
```
nelem = 10 ** 6
xs = np.random.random(nelem).astype(np.float32)
ys = np.random.random(nelem).astype(np.float32)
```

In [3]:
```
%%timeit
res1 = cpu_some_trig(xs, ys)
```

```
100 loops, best of 3: 18.8 ms per loop
```

In [4]:
```
%%timeit
res2 = cuda_some_trig(xs, ys)
```

```
100 loops, best of 3: 4.19 ms per loop
```

**4x speedup** incl. host<->device round trip on GeForce GT 650M

# Numba in Spark

- Compiles to IR on client
    - Or not if type information is not available yet
- Send IR to workers
- Finalize to machine code on workers

# CPS with cuFFT + GPU ufuncs

```python
@vectorize(['complex64(float32)',
            'complex64(float64)',
            'complex64(uint8)'], target='cuda')
def as_complex64(x):
    return np.complex64(x)
```

```python
@vectorize(['complex64(complex64, complex64)'], target='cuda')
def elemwise_mult_conjugate(f0, f1):
    eps = 1e-15
    return (f0 * f1.conjugate()) / (abs(f0) * abs(f1) + eps)
```

```python
@vectorize(['float32(complex64)'], target='cuda')
def complex_abs(x):
    return abs(x)
```

cuFFT

```python
def cross_power_spectrum(im0, im1):
    f0 = as_complex64(cuda.to_device(im0))
    f1 = as_complex64(cuda.to_device(im1))
    cufft.fft_inplace(f0)
    cufft.fft_inplace(f1)
    d_cps = elemwise_mult_conjugate(f0, f1)
    cufft.ifft_inplace(d_cps)
    cps = complex_abs(d_cps).copy_to_host()
    return cps
```

explicit memory transfer

# TIPS & TRICKS

# Operate in Batches

- GPUs have many-cores
- Best to do many similar task at once
- GPU kernel launch has overhead
- prefer `mapPartitions`, `mapValues` over `map`

# Under-utilization of GPU

- PySpark spawns 1 Python process per core
- Only 1 CUDA process per GPU at a time
- **Under-utilize** the GPU easily
- GPU context-switching between processes

# Under-utilization of GPU (Fix)

- `nvidia-cuda-mps-control`
- Originally for MPI
- Allow multiple process per GPU
- Reduce per-process overhead
- Increase GPU utilization
  - 10-15% speedup in our experiment

# Summary

- Anaconda:
  - creates Spark environment for experimentation
  - manages Python packages for use in Spark
- Accelerate:
  - Pre-built GPU functions within PySpark
- Numba:
  - JIT custom GPU functions within PySpark

# THANK YOU.

email: slam@continuum.io

ANACONDA
by Continuum Analytics®

SPARK SUMMIT 2016
DATA SCIENCE AND ENGINEERING AT SCALE

# Extras

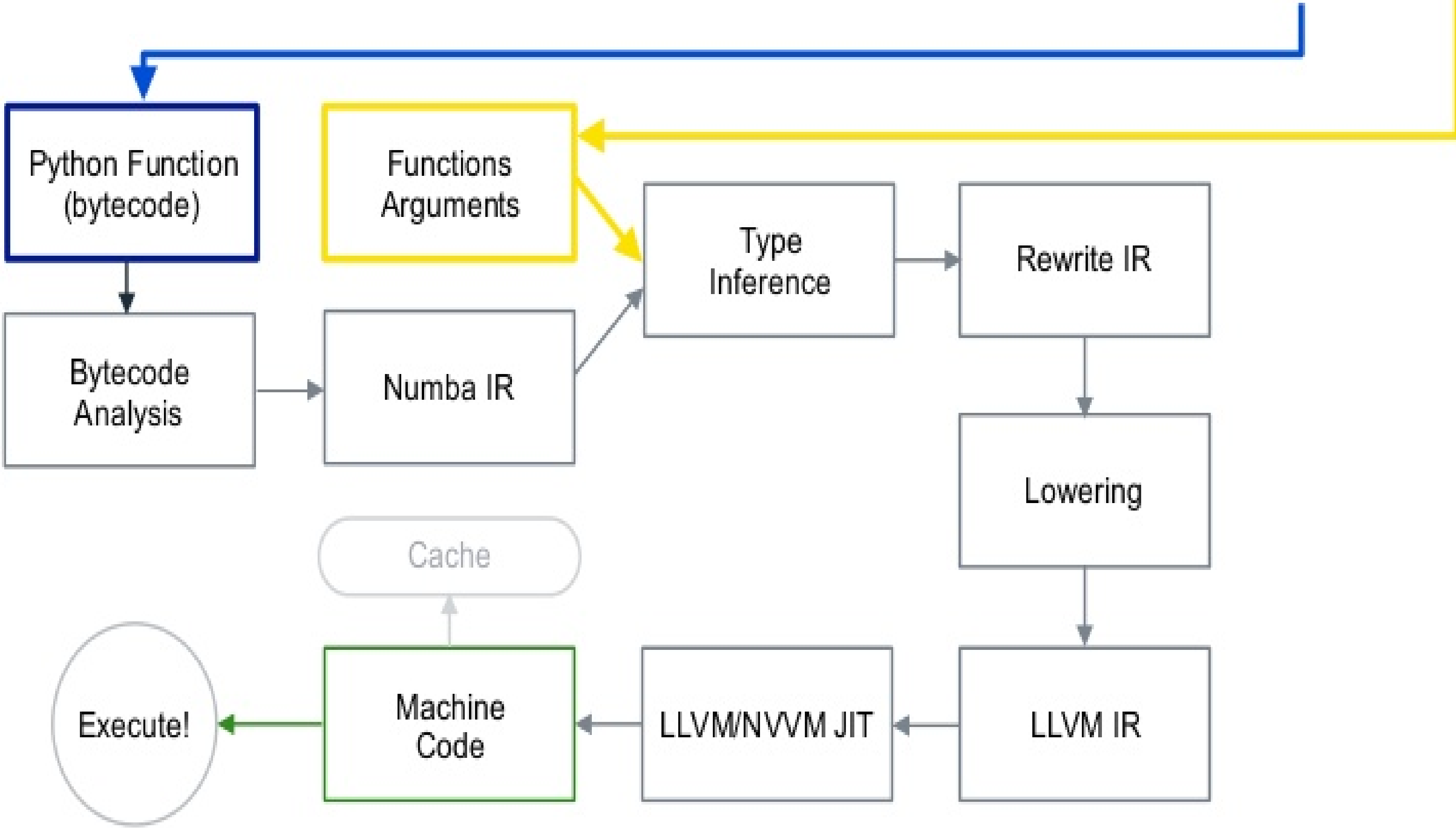
# NUMBA: A PYTHON JIT COMPILER

# Compiling Python

- Numba is a type-specializing compiler for Python functions
- Can translate Python syntax into machine code if all type information can be deduced when the function is called.
- Code generation done with:
  - LLVM (for CPU)
  - NVVM (for CUDA GPUs).

# How Does Numba Work?

```python
@jit
def do_math(a, b):
    …
>>> do_math(x, y)
```



# Numba on the CPU

```
In [87]: @jit(nopython=True)
         def nan_compact(x):
             out = np.empty_like(x)
             out_index = 0
             for element in x:
                 if not np.isnan(element):
                     out[out_index] = element
                     out_index += 1
             return out[:out_index]
```

```
In [88]: a = np.random.uniform(size=10000)
         a[a < 0.2] = np.nan
         np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])
```

```
In [89]: %timeit a[~np.isnan(a)]
         %timeit nan_compact(a)

         10000 loops, best of 3: 52 µs per loop
         100000 loops, best of 3: 19.6 µs per loop
```

# Numba on the CPU

*Numba decorator (nopython=True not required)*

*Array Allocation*

*Looping over ndarray x as an iterator*

*Using numpy math functions*

*Returning a slice of the array*

In [87]:
```
@jit(nopython=True)
def nan_compact(x):
    out = np.empty_like(x)
    out_index = 0
    for element in x:
        if not np.isnan(element):
            out[out_index] = element
            out_index += 1
    return out[:out_index]
```

In [88]:
```
a = np.random.uniform(size=10000)
a[a < 0.2] = np.nan
np.testing.assert_equal(nan_compact(a), a[~np.isnan(a)])
```

In [89]:
```
%timeit a[~np.isnan(a)]
%timeit nan_compact(a)
```

```
10000 loops, best of 3: 52 µs per loop
100000 loops, best of 3: 19.6 µs per loop
```

2.7x speedup!

# CUDA Kernels in Python

```
In [2]: @numba.cuda.jit
        def zero_suppression_gpu(x, threshold, out):
            i = numba.cuda.grid(1)

            while i < x.size:
                element = x[i]
                if abs(element) > threshold:
                    out[i] = element
                else:
                    out[i] = 0

                i += numba.cuda.gridsize(1)
```

# CUDA Kernels in Python

Decorator will infer type signature when you call it

```
In [2]: @numba.cuda.jit
        def zero_suppression_gpu(x, threshold, out):
            i = numba.cuda.grid(1)

            while i < x.size:
                element = x[i]
                if abs(element) > threshold:
                    out[i] = element
                else:
                    out[i] = 0

                i += numba.cuda.gridsize(1)
```

Helper function to compute `blockIdx.x * blockDim.x + threadIdx.x`

NumPy arrays have expected attributes and indexing

Helper function to compute `blockDim.x * gridDim.x`

# Calling the Kernel from Python

```
In [3]: # Create some sample data and an output array
        x = np.random.randint(-4096, 4096, size=100000).astype(np.int16)
        out = np.empty_like(x)

        # Pick configuration and launch
        threadsperblock = 256
        blockspergrid = (x.size + (threadsperblock - 1)) // threadsperblock
        zero_suppression_gpu[threadsperblock, blockspergrid](x, 50, out)

        print(out)

        [ 2447 -3900     0 ...,  3323 -1089  1995]
```

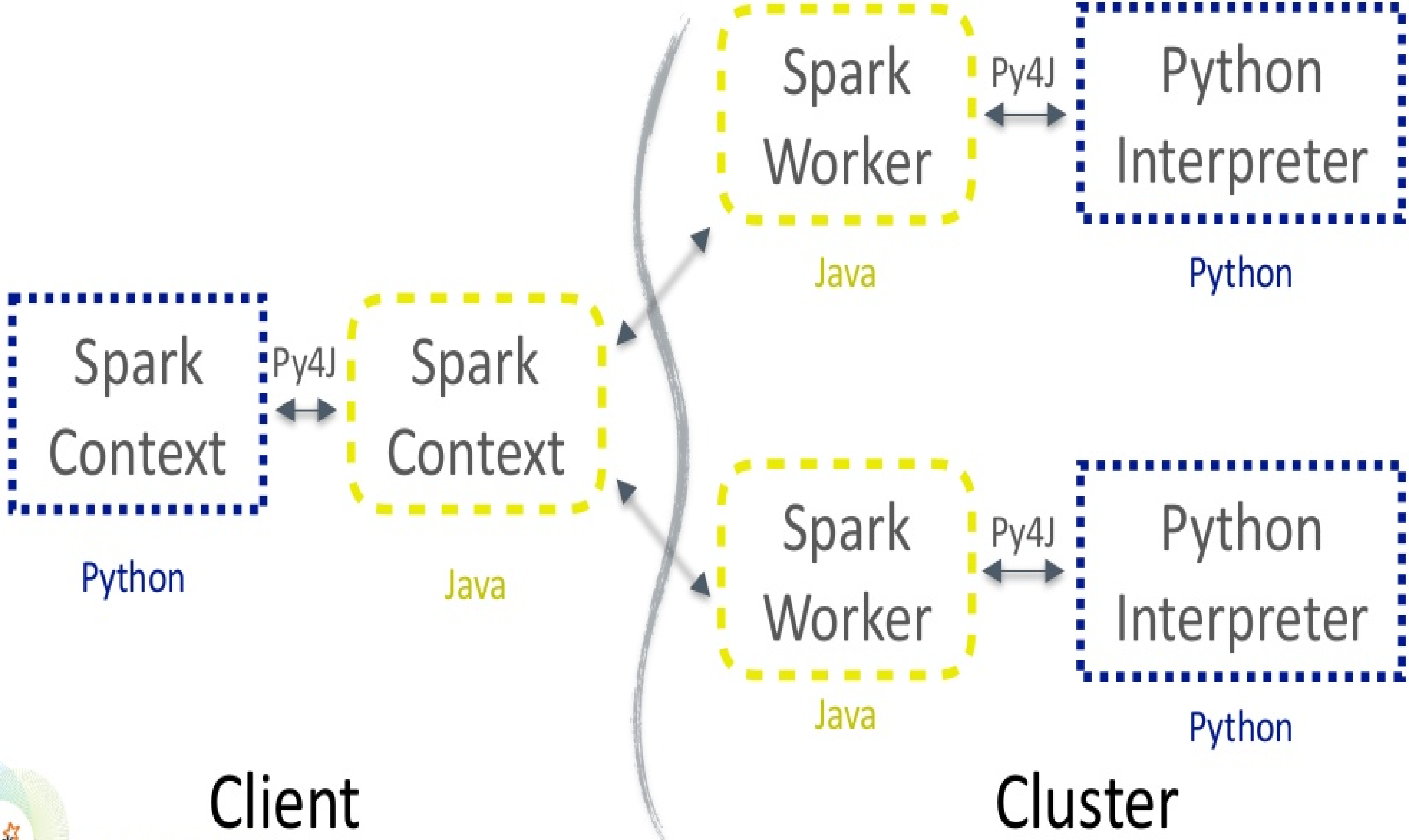
Works just like CUDA C, except we handle allocating and copying data to/from the host if needed

# Handling Device Memory Directly

```
In [6]: %timeit zero_suppression_gpu[threadsperblock, blockspergrid](x, 50, out)

The slowest run took 7.42 times longer than the fastest. This could mean
that an intermediate result is being cached.
1000 loops, best of 3: 927 µs per loop

In [7]: gpu_x = numba.cuda.to_device(x)
        gpu_out = numba.cuda.to_device(out)

        %timeit zero_suppression_gpu[threadsperblock, blockspergrid](gpu_x, 50, gpu_out)

1000 loops, best of 3: 198 µs per loop
```

Memory allocation matters in small tasks.

# NUMBA IN SPARK


Spark Context
Python
Py4J
Spark Context
Java
Spark Worker
Java
Py4J
Python Interpreter
Python
Spark Worker
Java
Py4J
Python Interpreter
Python
Client
Cluster

# Using Numba with Spark

In [37]:
```
random_arrays = [np.random.randint(-4096, 4096, size=10000).astype(np.int16)
                 for i in range(1000)]
chunks = sc.parallelize(random_arrays)
```

In [38]:
```
@numba.jit(nopython=True)
def zero_suppression(x, threshold):
    result = np.empty_like(x)
    for i in range(x.shape[0]):
        if np.abs(x[i]) > threshold:
            result[i] = x[i]
        else:
            result[i] = 0
    return result
```

In [39]:
```
%timeit np.where(np.abs(random_arrays[0]) > 25, random_arrays[0], 0)
%timeit zero_suppression(random_arrays[0], 25)
```

```
The slowest run took 6.77 times longer than the fastest. This could mean
that an intermediate result is being cached.
10000 loops, best of 3: 41.6 µs per loop
The slowest run took 5202.15 times longer than the fastest. This could me
an that an intermediate result is being cached.
10000 loops, best of 3: 20.9 µs per loop
```

In [40]:
```
chunks.map(lambda x: zero_suppression(x, 25)).first()
```

Out[40]:
```
array([  -43, -3824, -3618, ...,   349, -3929, -4018], dtype=int16)
```

# Using CUDA Python with Spark

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

@cuda.jit("(float32[:], float32[:])")
def foo(inp, out):
    i = cuda.grid(1)
    if i < out.size:
        out[i] = inp[i] ** 2
```

Define CUDA kernel
Compilation happens here

In [2]:
```
def gpu_work(xs):
    inp = np.asarray(list(xs), dtype=np.float32)
    out = np.zeros_like(inp)
    block_size = 32 * 4
    grid_size = (inp.size + block_size - 1) // block_size
    foo[grid_size, block_size](inp, out)
    return out
```

Wrap CUDA kernel launching logic

In [3]:
```
rdd = sc.parallelize(list(range(100)))
rdd.getNumPartitions()
```

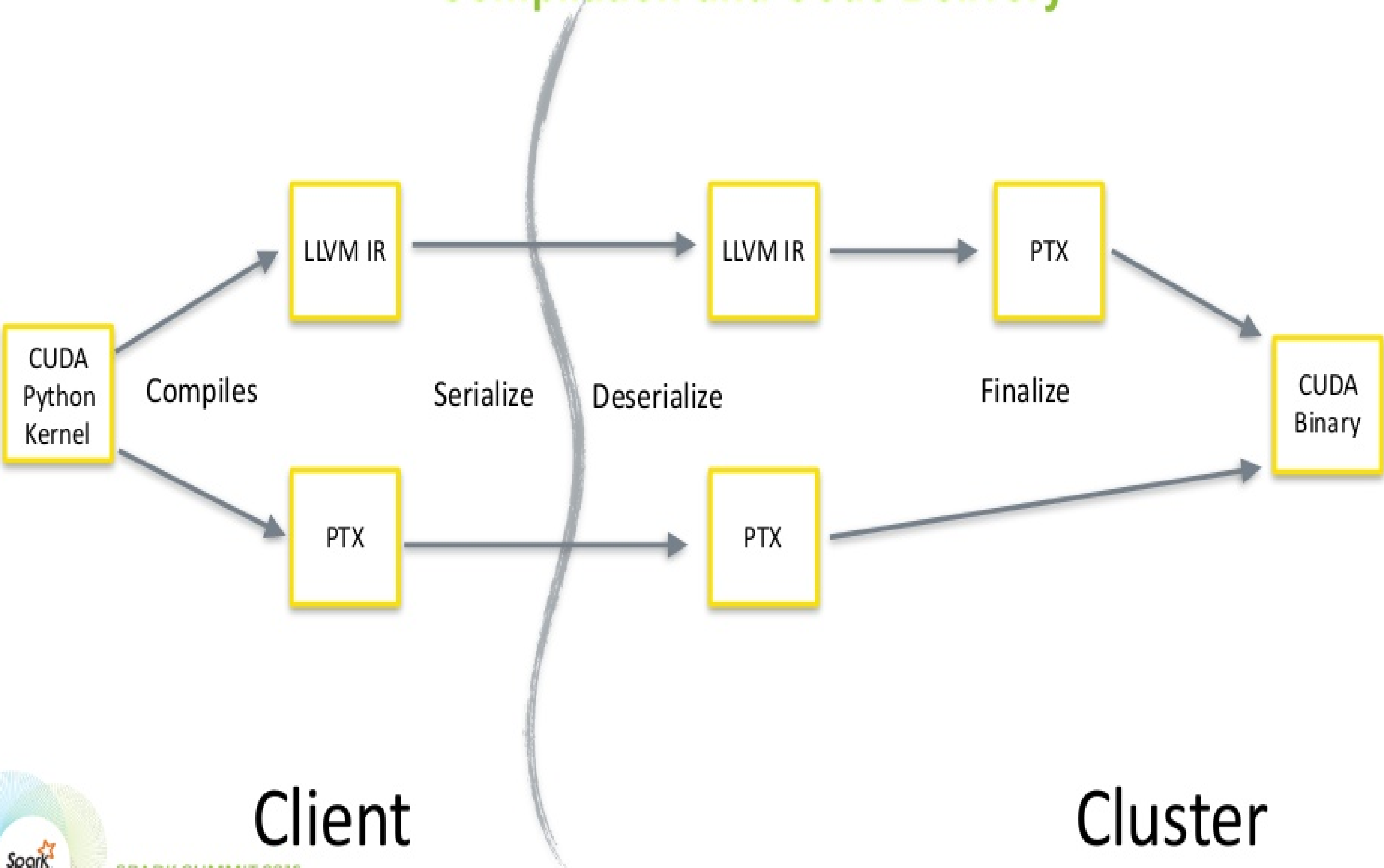
Creates Spark RDD (8 partitions)

Out[3]: 8

In [4]:
```
rdd = rdd.mapPartitions(gpu_work)
print(rdd.collect())
```

Apply gpu_work on each partition

## Compilation and Code Delivery



# Compilation and Code Delivery