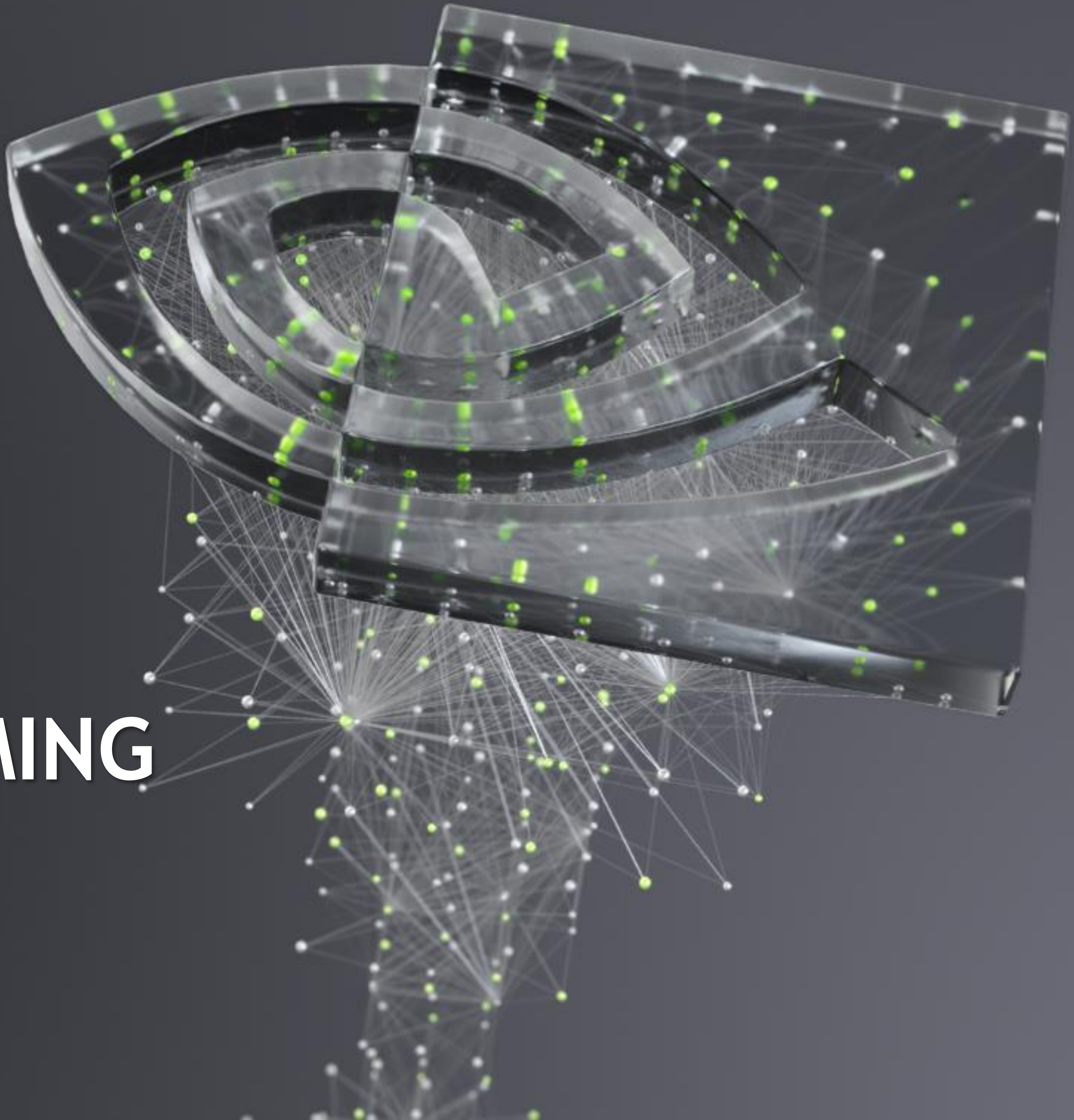
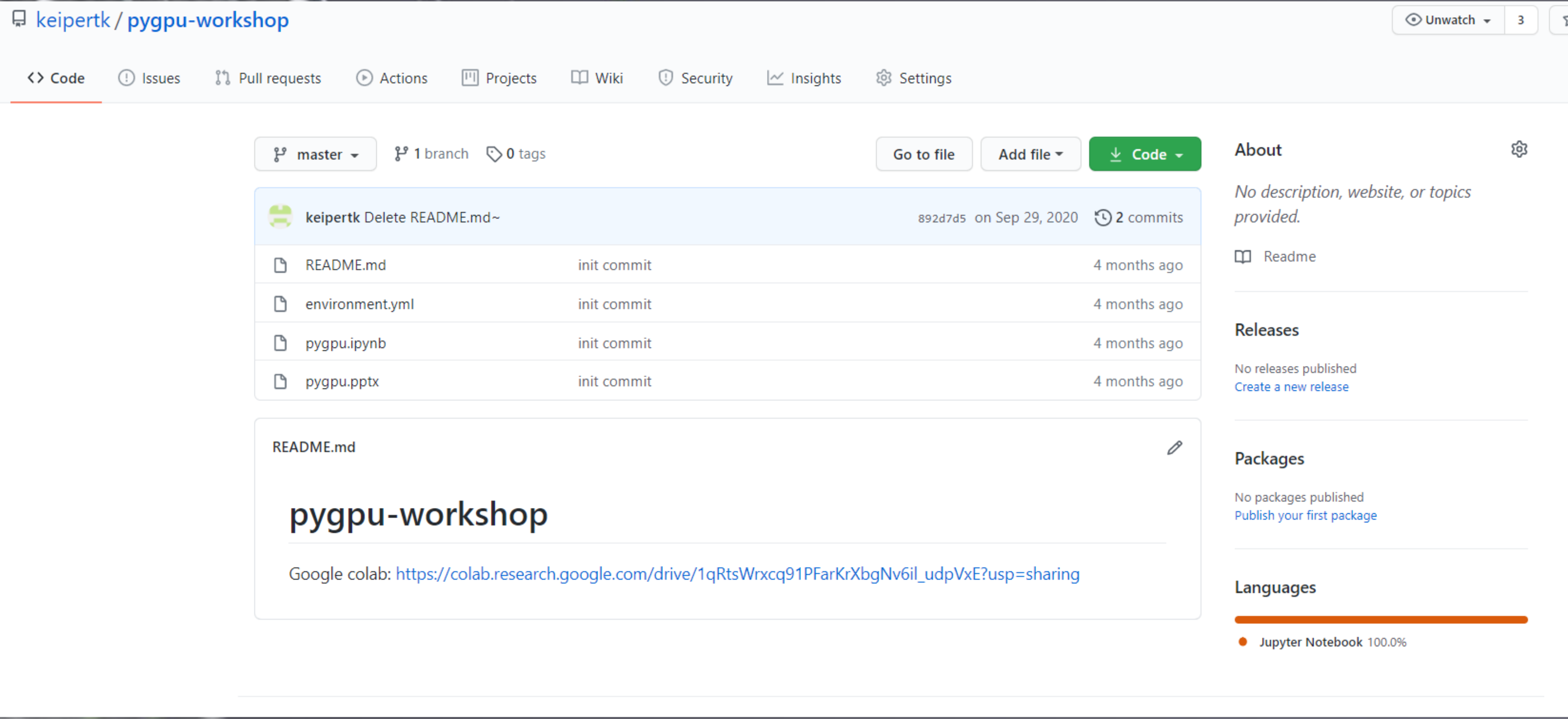


PYTHON PROGRAMMING FOR GPUS

Kristopher Keipert

kkeipert@nvidia.com

Slides and Notebook: https://github.com/keipertk/pygpu-workshop



keipertk Delete README.md~ 892d7d5 on Sep 29, 2020 2 commits

README.md	init commit	4 months ago
environment.yml	init commit	4 months ago
pygpu.ipynb	init commit	4 months ago
pygpu.pptx	init commit	4 months ago

README.md

pygpu-workshop

Google colab: https://colab.research.google.com/drive/1qRtsWrxcq91PFarKrXbgNv6il_udpVxE?usp=sharing

About

No description, website, or topics provided.

Readme

Releases

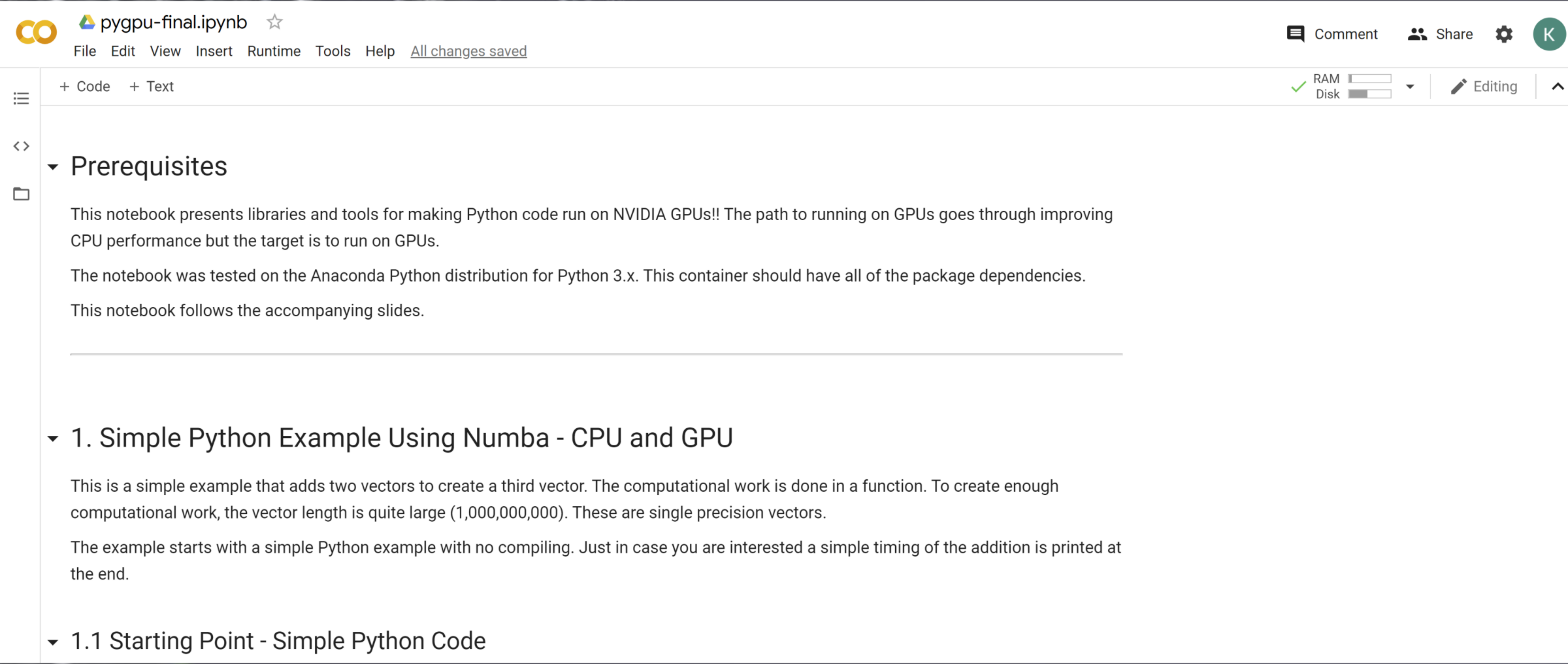
No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

● Jupyter Notebook 100.0%



Prerequisites

This notebook presents libraries and tools for making Python code run on NVIDIA GPUs!! The path to running on GPUs goes through improving CPU performance but the target is to run on GPUs.

The notebook was tested on the Anaconda Python distribution for Python 3.x. This container should have all of the package dependencies.

This notebook follows the accompanying slides.

---

1. Simple Python Example Using Numba - CPU and GPU

This is a simple example that adds two vectors to create a third vector. The computational work is done in a function. To create enough computational work, the vector length is quite large (1,000,000,000). These are single precision vectors.

The example starts with a simple Python example with no compiling. Just in case you are interested a simple timing of the addition is printed at the end.

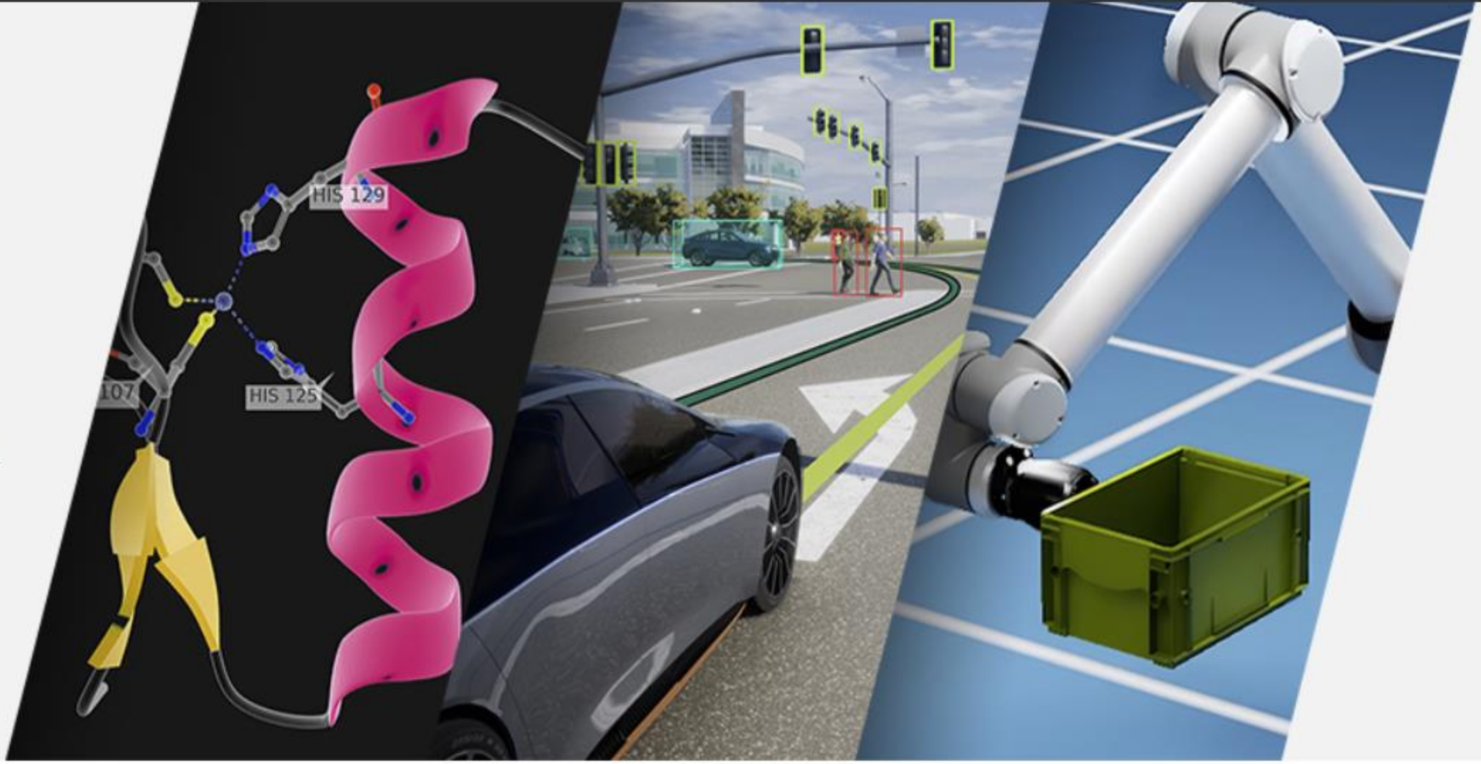
1.1 Starting Point - Simple Python Code

GTC CONFERENCE & TRAININGS MARCH 21 - 24, 2022 NOTIFY ME

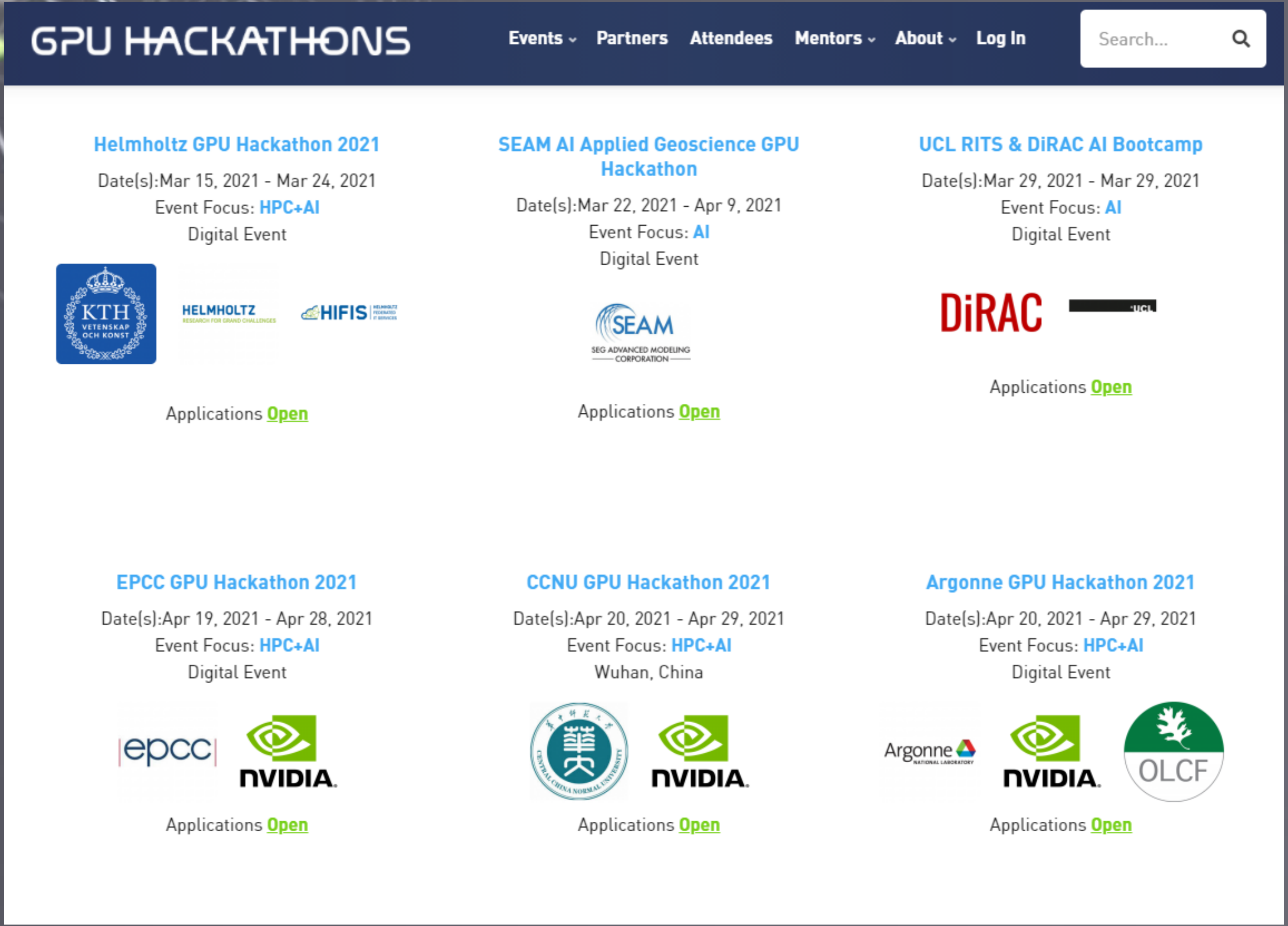
Keynote Session Catalog Session Content Hands-on Training Sponsors More

CALL FOR SUBMISSIONS

Showcase your brilliant work at GTC 2022.



NVIDIA GTC is returning on March 21 - 24, 2022 with a ton of exciting new content and speakers. This event will bring together a broad community of developers, researchers, engineers, and innovators with the common goal of sharing their achievements while discovering new technologies and tools that drive change around the globe.



Helmholtz GPU Hackathon 2021
Date(s):Mar 15, 2021 - Mar 24, 2021
Event Focus: HPC+AI
Digital Event
Applications Open

SEAM AI Applied Geoscience GPU Hackathon
Date(s):Mar 22, 2021 - Apr 9, 2021
Event Focus: AI
Digital Event
Applications Open

UCL RITS & DiRAC AI Bootcamp
Date(s):Mar 29, 2021 - Mar 29, 2021
Event Focus: AI
Digital Event
Applications Open

EPCC GPU Hackathon 2021
Date(s):Apr 19, 2021 - Apr 28, 2021
Event Focus: HPC+AI
Digital Event
Applications Open

CCNU GPU Hackathon 2021
Date(s):Apr 20, 2021 - Apr 29, 2021
Event Focus: HPC+AI
Wuhan, China
Applications Open

Argonne GPU Hackathon 2021
Date(s):Apr 20, 2021 - Apr 29, 2021
Event Focus: HPC+AI
Digital Event
Applications Open

gpuhackathons.org

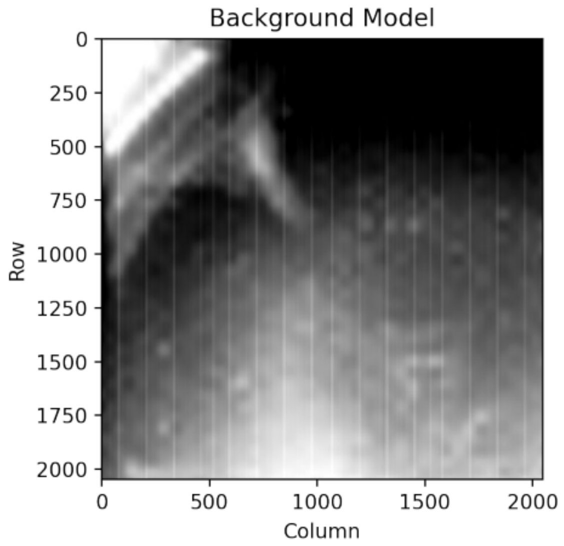
SSDL

tess-backdrop/scatterbrain

tess-backdrop, Code S NASA Ames Research Center

Application Background

- We have an app "*tess-backdrop*" which builds a simple linear model for the scattered light background in TESS images.
- This model can be built for small patches very cheaply, after we first find and fix the weights.
- We wanted to run the first weight fit on the supercomputer, to process the images more quickly. Started with ~1 hour to process a single CCD/Sector of TESS data.



Hackathon Objectives and Approach

- We developed a "mini-app" which has only the core functionality of tess-backdrop; **scatterbrain**
- Python code
- Original code was fully numpy/CPU, we have updated the mini-app to allow either CPU or GPU computing.

Technical Accomplishments and Impact

- We were able to achieve a >40x speed up over CPU, which greatly improves what we will be able to achieve with our main tool.
- We have learned cupy and MPI to obtain these speed ups and will be implementing the changes in our full application after the hackathon.
- We've learned how to profile our new GPU code with nsight

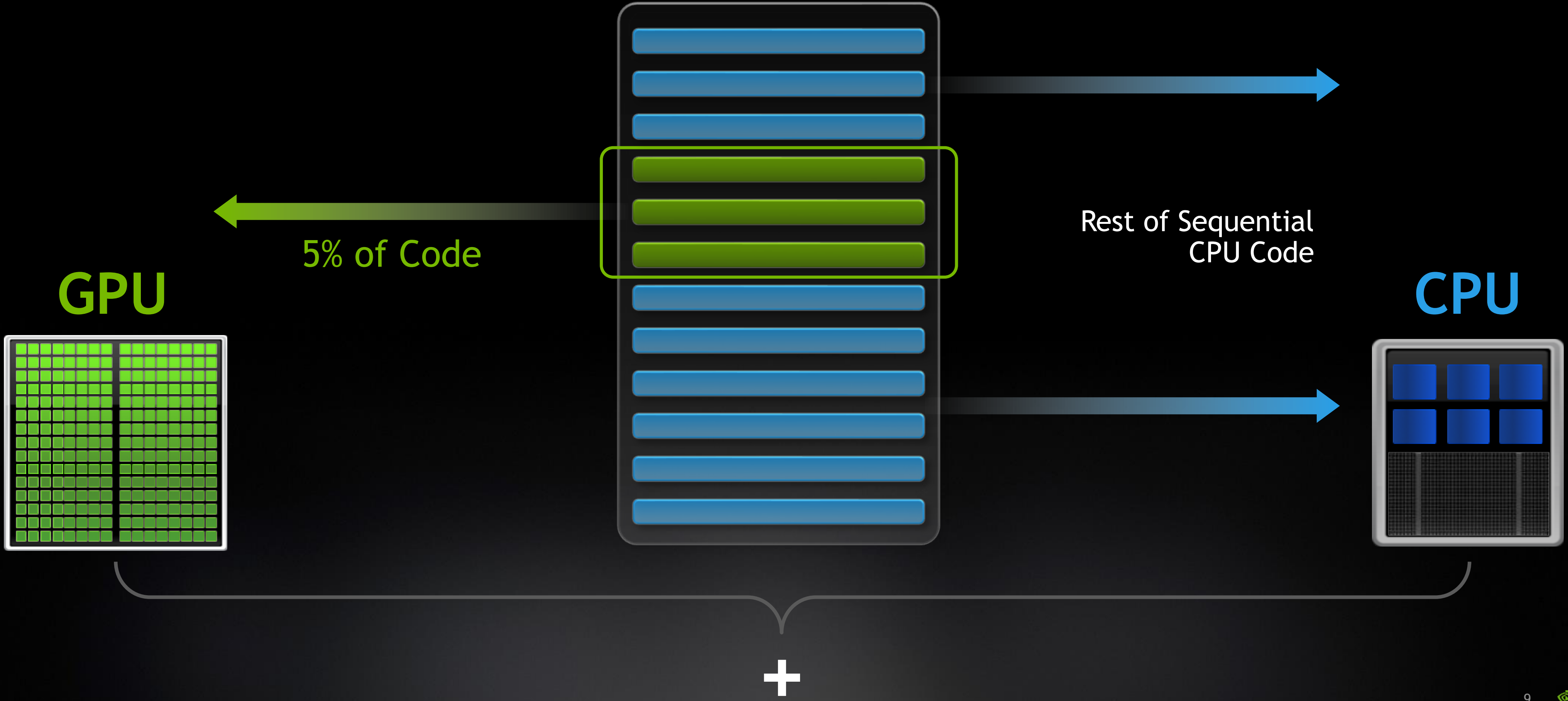
GPU Programming questions?
kkeipert@nvidia.com

Python GPU Programming
CUDA C/C++/Fortran
GPU Porting Strategy
Performance Profiling and Developer Tools

AGENDA

- What is a GPU?
- Why Python?
- Python Coding for GPUs
  - Numba (JIT compiler)
    - CPU and GPU Decorators
    - Compiling ahead of time
    - Example of serial->GPU porting process
    - GPU functions calling GPU functions
  - CuPy – GPU Enabled Numpy
    - Moving data to/from GPUs
    - Dask integration (if time permits)

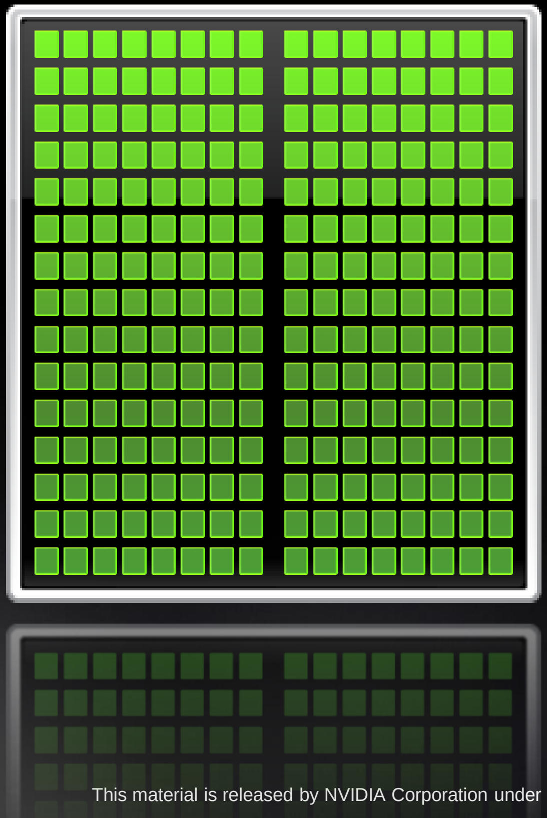
HOW GPU ACCELERATION WORKS



ACCELERATED COMPUTING

CPU
Optimized for
Serial Tasks

GPU Accelerator
Optimized for
Parallel Tasks



CPU IS A LATENCY REDUCING ARCHITECTURE

CPU
Optimized for
Serial Tasks

CPU Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

CPU Weaknesses

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt

GPU IS ALL ABOUT HIDING LATENCY

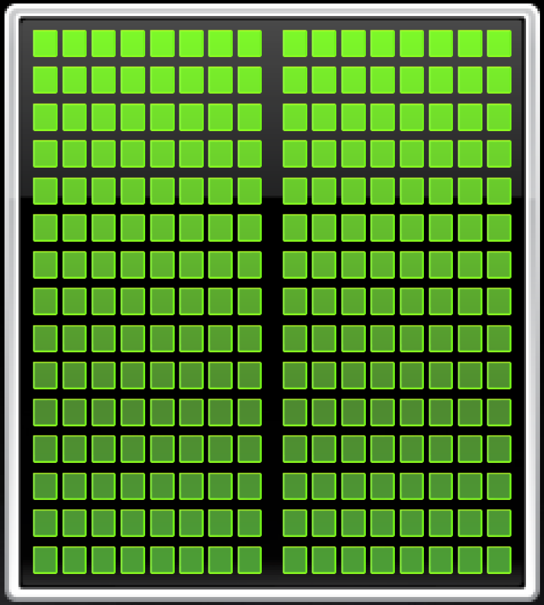
GPU Strengths

- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

GPU Weaknesses

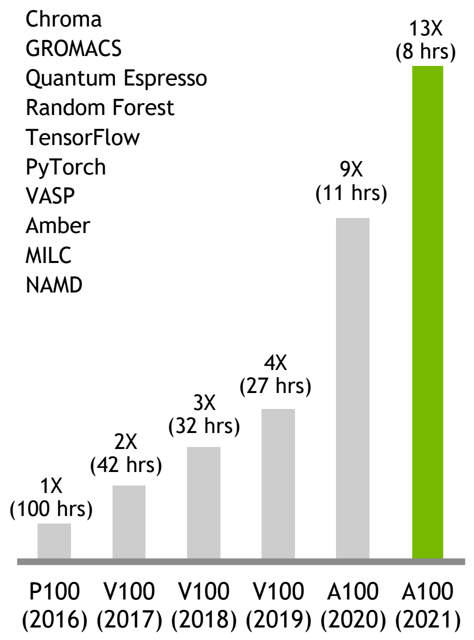
- Relatively low memory capacity
- Low per-thread performance

GPU Accelerator
Optimized for
Parallel Tasks

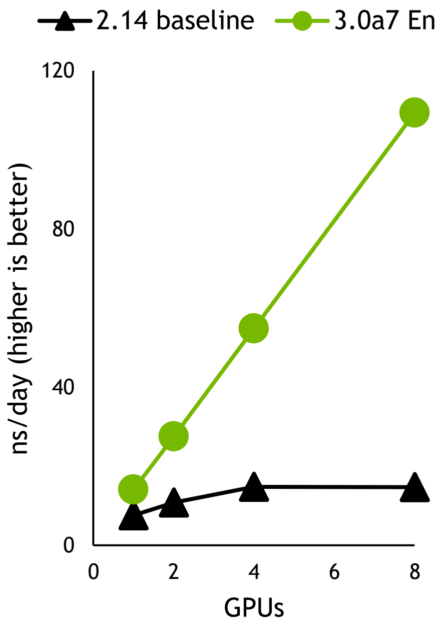


GIVING SCIENTISTS A TIME MACHINE

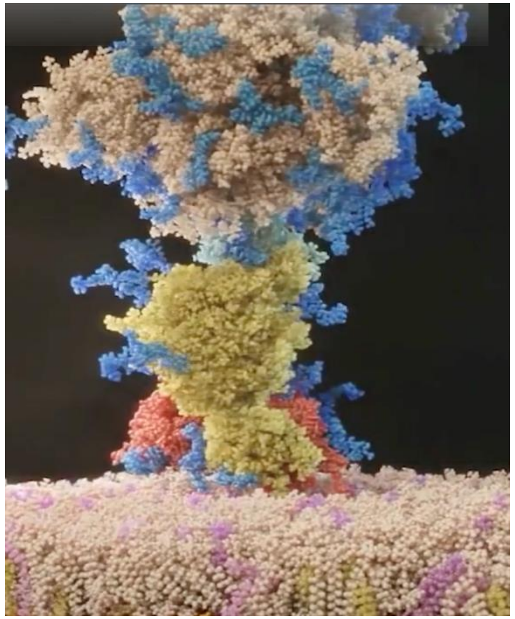
CONTINUOUS IMPROVEMENT
13x in 5 years



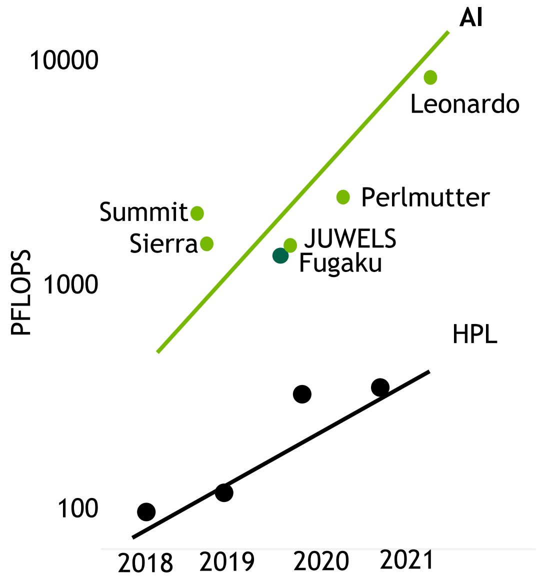
15X MULTI-GPU ACCELERATION
NAMD version 3.0



LARGEST AI + MD SIMULATION
DeepDriveMD + NAMD



ERA OF EXASCALE AI IS NOW
HPL vs. AI performance



PYTHON CODING FOR GPUS

WHY PYTHON?

- Very popular in many fields
  - Scientific community is quickly adopting Python, especially for data analytics
  - #1 language for Deep Learning

- Features:
  - High-level, interactive, versatile (Jupyter Notebooks)
  - Easy to prototype
  - Can expose bindings for lower-level languages (PyBind11, cppyy)
  - Automatic Memory Management + Garbage collection (reclaim memory from deleted variables)
  - Dynamic typing (Runtime type checks)
  - Wide range of data types and structures (intrinsic Complex and Boolean data types)
  - **<u>Many</u> libraries (modules) available**
  - Robust package and environment management (conda, pip)

WHY PYTHON?

And GPUs?

Easy to use

- You don’t have to learn CUDA! (CUDA-less GPU programming)
  - Unless you really want to ☺

Performance

- High-level scripting languages are in many ways' polar-opposite to GPUs
  - GPUs are highly parallel, designed for maximum throughput, and they offer a tremendous advance in performance
  - Scripting languages such as Python favor ease of use over computational speed and do not generally emphasize parallelism
- Numba and CuPy

TL;DR - OVERALL PYTHON CODING RECOMMENDATIONS

- Make sure you have a relevant test suite
  - Unit tests helpful for detecting where the problem is when extending for GPU
  - End-to-end tests also helpful - need to be aware of numerical sensitivities
  - Benchmarks: good to measure wall time before / during / after your porting efforts
- First of all, profile the code and make sure you understand where you need better performance.
- Python Profiling Tools:
  - Snakeviz
  - cProfile + gprof2dot
  - Line_profiler

TL;DR - OVERALL PYTHON-GPU CODING RECOMMENDATIONS

- These apply to any language coding for GPU, but are especially important for Python
- Avoid data movement to/from GPU and CPU
  - Do as much as you can on the GPU
- “What happens on the GPU, stays on the GPU”
- Look for loops and arrays to achieve performance improvements
  - GEMMs!
  - Don’t port string handling or integer computations to the GPU
- Avoid divergence and complex logic branching (GPU is SIMT)
- When “porting” to GPU, intermediate code can be slower than CPUs
  - Don’t be surprised, don’t give up – profile!
  - Giving the GPU enough work to amortize data movement?

"Locality is efficiency,
Efficiency is power,
Power is performance,
Performance is king"
-Bill Dally

NUMBA

DIRECTIVES

Compiled Languages

- Language “Directives” tell compilers about the code and how to build for target architecture (descriptive)
- Let compilers and runtime do the work for us
- They appear as “comments” so if the compiler doesn’t understand the pragma, it is ignored
- OpenMP, OpenACC for Fortran and C/C++

```
#pragma acc parallel
{
    #pragma acc loop
    for(int i= 0; j < N; i++) {
        a[i] = 0;
    }
}

==============================================

@jit
def Add(a, b):
    return a + b
```

DIRECTIVES AND THE JIT

JIT Compiler

- Language “Directives” tell compilers about the code and how to build it for a target architecture (descriptive)
- Python is designed to be an interpreted language
    - No compilation or static typing
    - “Interactive”
- JIT = “Just in Time” Compiler
    - Compiles or creates object code “on the fly”
    - Combines interactivity and compilation
- Allows for computationally intensive sections of Python code to be run on GPU (or multi-core CPU)

NUMBA

Introduction

- Numba is a just-in-time (JIT), type-specializing, function compiler for accelerating numerically-focused Python
  - It has a numerical focus
  - Type-specializing
    - Numba generates a specialized implementation for the specific data types you are using or specifying
      - Python functions are designed to operate on generic data types
    - Typically, you only will call a function with a small number of argument types (simple functions)
      - Allows Numba to generate fast implementation for each set of types
- Not every Python function can be compiled (subset of Python and NumPy)
  - ~~Async features, class definitions, set/dict/generater comprehensions, generator delegation~~
- Look for functions with high arithmetic intensity (e.g. chunky loops)

NUMBA

Decorators

- Typically enabled by applying a *decorator* to a Python function
  - By definition, a decorator is a function that takes another function and extends the behavior of the latter function *without* explicitly modifying it
  - Functions that transform Python functions
- Decorators used in this workshop:
  - @jit
  - @cuda.jit
  - @vectorize

NUMBA

More information

- Numba runs inside the standard Python interpreter
  - Can target the CPU, either single-core or multiple-cores, or GPU
- Uses LLVM to compile Python functions (comes with Numba)
- The first time the function is called, the compiler creates a machine code implementation for inputs
  - Saves the original python implementation as .py_func
  - Can test compiled function against original Python function
- Subsequent calls to function use machine code (much faster)
  - You can create compiled code prior to running
- Data Types (dtypes):
  - bool_, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, float64, numpy.complex64, numpy.complex128

Example 1:
Simple Python Example for CPU

BASE PYTHON

1.1

```
import numpy as np
from time import perf_counter

def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 10000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter()
C = Add(A, B)
stop_time = perf_counter()

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- Simple vector addition
- Contains one function that does virtually all of the computation
- This is our starting point

@JIT EXAMPLE (SINGLE CORE)

1.2.1

```
import numpy as np
from time import perf_counter
from numba import jit

@jit
def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 1000000000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
C = Add(A, B)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

Function Decorator

Function to be compiled

- Compile for CPU (use defaults)
- @jit is the Numba "decorator"
    - Lets the compiler decide how to proceed
- Uses default params for @jit
- Try running the same cell multiple times – does the wall clock change?
    - Why or why not?

WHAT JUST HAPPENED?

- Compiled a Python function (code and Python interface) just before it was <u>first</u> used
  - Used default @jit parameters (we'll find out what some of those are)
- Executed the function along with the Python code
- Since this is executed on the CPU, no data movement was required
- Sounds simple but a great deal of work was done (thank you Numba!)

WHAT'S NEXT?

- The @jit decorator has several parameters that can be very useful
- @jit(cache=True)
    - File-based cache of compiled function, stored in $NUMBA_CACHE_DIR
- @jit(parallel=True)
    - Automatic parallelization (plus optimizations) in the function known to have parallel semantics
- @jit(nopython=True)
    - By default, if Numba can't compile function, it leaves the code in Python
    - This option tells Numba ***not*** to do this. Instead, Numba gives an error and stops.

@JIT CPU EXAMPLE WITH PARALLEL

1.2.2

```
import numpy as np
from time import perf_counter
from numba import jit

@jit(nopython=True, parallel=True)
def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 1000000000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
C = Add(A, B)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- By default, if Numba cannot compile the Python code, it will leave the Python in place
  - It doesn't get compiled
- You can turn off this behavior with the option, "nopython=True"
  - If it can't compile, Numba will throw an error.
- "parallel" allows the compiler to parallelize for all CPU cores

@JIT CPU EXAMPLE WITH CACHE=TRUE

1.2.3

```
import numpy as np
from time import perf_counter
from numba import jit

@jit(nopython=True, cache=True)
def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 1000000000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
C = Add(A, B)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- If you want to save the compiled code in a cache, you can use the option, “cache=True”

@JIT CPU EXAMPLE WITH PARALLEL AND CACHE

1.2.4

```
import numpy as np
from time import perf_counter
from numba import jit

@jit(nopython=True, cache=True, parallel=True)
def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 1000000000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
C = Add(A, B)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- I recommend using this option set, adding parallel=True after you're happy with the performance on a single core

WHAT ABOUT GPUS?…

- Note that we can’t use @jit with GPUs
- You have to use either @cuda.jit or @vectorize
    - If you like, try replacing @jit with @cuda.jit and trying executing the cell
- For @cuda.jit, we need to write code that addresses list (arrays) element-by-element
    - Effectively means we'll be using loops
- While you experiment with @cuda.jit, let's continue with @vectorize

NUMBA

Universal Functions

- NumPy has the concept of universal functions ("ufuncs")
    - Functions that can take Numpy arrays of varying dimensions (ndarray) and operate on them element-by-element (loop)
    - Perfect for processing on GPUs!
- Numba can create compiled ufuncs
    - This is not easy to do by hand
    - Numba makes it easy for us
        - Write ufuncs to operate on scalars
        - Numba generates the surrounding loop/kernel

UNIVERSAL FUNCTIONS

UFUNCS

- Numba @jit lets us write NumPy ufuncs in pure Python!
- ufunc = universal function
- Vectorized wrapper for a function that takes a specific number of inputs with a specific number of outputs
- NumPy can use ufunc arrays of varying dimensions, or scalars, and operate on them element-by-element
- Numba can compile a pure Python function into a ufunc that operates over NumPy arrays as fast as traditional ufuncs written in C
- Perfect for GPUs!! **Same operation on each element in an array**

@VECTORIZE DECORATOR

- Writing NumPy ufuncs isn’t easy (need to write C code)
- Numba makes it easier to write ufuncs
  - Can compile a pure Python function into a ufunc
  - Operates over NumPy arrays
- The @vectorize decorator allows Python functions taking scalar inputs to be used as NumPy ufuncs (i.e. write with scalar inputs/operations, run with Numpy arrays)
- Great option for GPUs!
- You write the function as if it were operating on scalars
  - Numba will generate surrounding loop (or kernel)

@VECTORIZE DECORATOR

- Function has to be written as element-by-element
  - The same operation on each element
- Only 1 return variable allowed
  - Return a scalar result value
  - C
    - `double f(double a, double b);`
  - Numba
    - `@vectorize([float64(float64, float64)])`

@VECTORIZE DECORATOR

Type Signatures

- There are two options with the @vectorize decorator
  - Eager, or decoration-time, compilation.
  - Lazy, or call-time, compilation.
- Decoration-time compilation uses a type signature with the decorator (**NumPy** Ufunc)
  - `@vectorize(['float32(float32, float32)’])`
  - You can specify more than one data type if you use more than one data type in your code
- Call-time compilation (**Numba** Dynamic UFunc)
  - No data type signature(s)
  - Numba creates compiled code for data type when it is first called

Best Practice:

<u>If you require precise support for various type signatures, specify them in the vectorize decorator</u>

@VECTORIZE - NO TYPE SIGNATURE

1.2.5

```
import numpy as np
from time import perf_counter
from numba import vectorize

@vectorize
def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 1000000000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
C = Add(A, B)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- @vectorize - Python functions taking scalar input arguments to be used as NumPy ufuncs
- No type signature (call time compilation)
- You are only allowed one return value. Has to be written like a scalar
- NumPy ufuncs resulting from @vectorize automatically inherit features such as reduction, accumulation or broadcasting. (e.g. f.accumulate())
- This example uses the default targets (single CPU)

DETAILS OF PROCESS

```
@vectorize
def Add(a, b):
  return a + b
```

- Decorator
- Numba does not compile the decorated function when it is encountered
  - It creates compiled code only when the function “Add” is used

@VECTORIZE - TYPE SIGNATURE

1.2.6

```
import numpy as np
from time import perf_counter
from numba import vectorize

@vectorize(['float32(float32, float32)'])
def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 1000000000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
C = Add(A, B)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- Define a “type signature”
- “Decorator-time compilation”

DETAILS OF PROCESS

```
@vectorize(['float32(float32, float32)'])
def Add(a, b):
  return a + b
```

- Decorator with type signature
  - Numba compiled the function “Add” when the decorator is encountered
- “vectorize” allows Python functions taking scalar input arguments to be used as NumPy ufuncs
  - Not easy to write a ufunc (need to write C code)
  - Numba can compile a pure Python function into a ufunc that operates over NumPy arrays as fast as traditional ufuncs written in C

@VECTORIZE CPU TARGET EXAMPLE

1.2.7

```
import numpy as np
from time import perf_counter
from numba import vectorize

@vectorize(['float32(float32, float32)'], target='cpu')
def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 1000000000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
C = Add(A, B)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- How to we specify “targets” for the compilation?
- What targets do we have?
  - “cpu”
  - “parallel”
  - “cuda”
- Specifically target the “cpu”
  - Single thread CPU

@VECTORIZE PARALLEL TARGET EXAMPLE

1.2.8

```
import numpy as np
from time import perf_counter
from numba import vectorize

@vectorize(['float32(float32, float32)'], target='parallel')
def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 1000000000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
C = Add(A, B)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- Target “parallel” (multi-core CPU)

FUNCTION USES LOOPS?

STICK WITH @JIT

@JIT CPU TARGET EXAMPLE

1.2.9

```
import numpy as np
from time import perf_counter
from numba import jit

@jit
def Add(a, b, c):
    max_length = len(a)
    for i in range(0, max_length):
        c[i] = a[i] + b[i]
    # end for
# end def

# Initialize arrays
N = 100000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
Add(A, B, C)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - 
start_time) )
print('')
```

- What if we use loops in our compiled function?
- @jit is more general than @vectorize, and can work on many more types of calulcations
- Why not stick with @jit?
    - Numpy ufuncs inherit reduction, accumulation, broadcasting

CPU SUMMARY

- Compilation can give us some pretty great performance improvements
- Numba can target a single core or parallelize across all cores
    - Using all cores may not produce faster code, but not because of Numba
- The @jit decorator is a good place to start using Numba
    - Limited to CPUs only ☹

CPU SUMMARY

- @vectorize is a decorator for handling arrays but writing Python code as if it were operating on scalars
  - Data is returned like C code
  - A type signature is optional (pay attention to the Numba compilers – this can explain performance differences)
  - You can write a type signature for each input data type you expect to use
- Numba can compile Python+CUDA code (**you don’t have to learn CUDA**!)
  - Bring on GPUs!!!

Access GPUs with:

```
@vectorize(target='cuda')
@cuda.jit
```

GPU INTRODUCTION

- Numba can compile for NVIDIA GPUs!
  - Can use the @vectorize decorator
  - Just use target='cuda'
- Numba cannot handle all Python functions on the GPU
- Can make writing functions for the GPU easier
- You can dive deep and make the function code look more like CUDA
  - Need to understand more about CUDA programming
  - https://numba.pydata.org/numba-doc/dev/cuda/kernels.html

```
@cuda.jit
def my_kernel(io_array):
    # Thread id in a 1D block
    tx = cuda.threadIdx.x
    # Block id in a 1D grid
    ty = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    bw = cuda.blockDim.x
    # Compute flattened index inside the array
    pos = tx + ty * bw
    if pos < io_array.size:  # Check array boundaries
        io_array[pos] *= 2 # do the computation
```

ALLOWED NUMBA FUNCTIONS

(WARNING: It's not everything)

- Allowed statements/functions:
  - `if/elif/else`
  - `while` and `for` loops
  - Basic math operators
  - Selected functions from the math and cmath modules
  - Tuples
  - int, float, complex, bool, None, tuple
- http://numba.pydata.org/numba-doc/latest/cuda/cudapysupported.html

@VECTORIZE NV GPU TARGET EXAMPLE

1.3.1

```
import numpy as np
from time import perf_counter
from numba import vectorize

@vectorize(['float32(float32, float32)'], target='cuda')
def Add(a, b):
    return a + b
# end def

# Initialize arrays
N = 100000000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on CPU
start_time = perf_counter( )
C = Add(A, B)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- Reminder of “targets” for vectorize
  - Target=‘cpu’ Single-threaded CPU
  - Target=‘parallel’ Multi-core CPU
  - Target=‘cuda’ NVIDIA GPU
- This example targets the NVIDIA GPU
- Simplest way to compile functions that run on the GPU
  - Everything stays as Numpy arrays or scalars
  - Python code stays the same (make sure you only have one return variable)

WHAT HAPPENED?

- Compiled a CUDA kernel when first used
- Allocated GPU memory for the input(s) and the output (one return variable is allowed)
    - Also any intermediate variables
- Copied the input data to the GPU
- Executed the CUDA kernel with the correct kernel dimensions given the input sizes
- Copied the results back from the GPU to the CPU
- Returned the result as a NumPy object on the host

@CUDA.JIT

- You might be tempted to use this decorator right away – be careful
    - You must explicitly write loops
        - Everything passed in/out of the function is a NumPy array – even scalars
        - Can still define local scalars in the function (e.g. loop counters)
- A bit more work than @vectorize
- Can simply use `@cuda.jit` or write CUDA kernels in Python
- For today, we’ll just use `@cuda.jit`

@CUDA.JIT EXAMPLE

1.3.2

```
import numpy as np
from time import perf_counter
from numba import cuda

@cuda.jit
def Add(a, b, c):
    max_length = len(a)
    for i in range(0, max_length):
        c[i] = a[i] + b[i]
    # end for
# end def

# Initialize arrays
N = 100000
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays on GPU
start_time = perf_counter( )
Add(A, B, C)
stop_time = perf_counter( )

print(C)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time -
start_time) )
print('')
```

- Gets you closer to “CUDA” coding
  - More flexibility
  - Theoretically better performance
- Some code changes are required
  - Everything needs to be a Numpy variable
  - Scalars are ndarrays of size 1
  - “Return” variables are part of function call (like C code)
- Data is passed through the function (as often seen in C, Fortran codes)

@CUDA.JIT

- @cuda.jit takes you closer to CUDA coding
  - More flexibility
  - Theoretically more performance

- Can control kernel launch (grid, blocks, threads)
  - numba.cuda.threadIdx, numba.cuda.blockDim ...
- Thread placement
- Memory Management
  - Three types of GPU memory:
    - Global device memory (the large, relatively slow off-chip memory that’s connected to the GPU itself)
    - On-chip shared memory
    - Local memory
- Can define atomic operations

@CUDA.JIT CODE CHANGES

- To use the `@cuda.jit` decorator, a few code changes need to be made
  - All loops must be explicitly written (no vectorization)
  - All variables _passed_ to/from compiled function must be created by Numpy – even scalars
    - Scalars are Numpy arrays of length 1
  - Data is passed through call statement (like C code)
- Do not create data in the compiled function
  - Create arrays on host, copy them to function, modify them, copy back to host

NUMBA SUMMARY TO DATE

- For CPUs:
  - @jit for non-element-by-element functions
    - Can target single CPU or multi-core
    - Python fallback
  - @vectorize for element-by-element operations
    - Don’t have to write loops! Element-by-element
    - Can target single CPU or multi-core
- For GPUs:
  - @cuda.jit for non element-by-element operations
    - Takes more work
    - Explicitly have to move data and write loops
    - Everything is a NumPy array
  - @vectorize for element-by-element operations
    - Same as CPU but with different target (target=cuda)
    - Have to write a type signature for each data type

Compiling Ahead of Time

COMPILING AHEAD OF TIME

Benefits

- Numba is usually thought of as Just-in-Time (JIT)
  - Impacts runtime due to compilation
- How do we do Ahead-of-Time compiling (AOT)?
  - AOT compilation produces a compiled extension module which does not depend on Numba
    - You can distribute the module on machines which don’t have Numba installed (does require NumPy)
  - There is no compilation overhead at runtime (but see the @jit cache option)
  - No overhead of importing Numba

COMPILING AHEAD OF TIME

Limitations

- AOT compilation only allows for regular functions, **not ufuncs**
- You must specify function signatures explicitly
- Each exported function can have only one signature (but you can export several different signatures under different names)
- AOT compilation produces generic code for your CPU's architectural family (for example "x86-64"), while JIT compilation produces code optimized for your CPU model

COMPILING AHEAD OF TIME

Process

- http://numba.pydata.org/numba-doc/dev/user/pycc.html
- Need to install build system, etc.
  - Not covered in this workshop

Example of Porting Process

Serial -> Functions -> CPU -> GPU

PYTHON PORTING LIFECYCLE EXAMPLE

- Example of taking serial Python code, porting to Python code with functions, to using Numba on CPUs, to using Numba on GPUs
  - Moved from plain Python code to `@jit` on CPUs, to `@cuda.jit` on GPU
    - `@cuda.jit` on GPU requires code changes
- This is a very, very simplified version of the initialization portion of a Molecular Dynamics (MD) mini-app
  - It only uses loops – not details of computations in loops
  - Computations are contrived (i.e. not real)
- Original application is GPL:
  - https://people.sc.fsu.edu/~jburkardt/py_src/md/md.html

EXAMPLE – SERIAL PYTHON

2

```
import numpy as np
from time import perf_counter

# main loop
start_time = perf_counter( )
d_num = 5000
p_num = 5000

pos = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
accel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
for j in range(0, p_num):
    for i in range(0, d_num):
        pos[i,j] = 6.5
    # end for i
    for i in range(0, d_num):
        accel[i,j] = 4.2*pos[i,j]
    # end for
# end for j

stop_time = perf_counter( )

print(pos)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

- Serial Python code
- Two nested loops (you can easily combine them, but for this sample code, we won’t)
- Classic starting point for initial code development
  - Write the code serially to get the correct answers
  - Then worry about performance

EXAMPLE – PYTHON WITH FUNCTIONS

2.1

```
import numpy as np
from time import perf_counter

def init(p_num, d_num):
    pos = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    accel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    for j in range(0, p_num):
        for i in range(0, d_num):
            pos[i,j] = 6.5
        # end for i
        for i in range(0, d_num):
            accel[i,j] = 4.2*pos[i,j]
        # end for i
    # end for j
    return pos, accel
# end def


# main
d_num = 5000
p_num = 5000

start_time = perf_counter( )
pos, accel = init(p_num, d_num)
stop_time = perf_counter( )

print(pos)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

- Move arithmetic intense code to function(s)
- Isolates code - simpler main
- Multiple arrays returned to main

EXAMPLE – @JIT – TARGET SINGLE CPU CORE

2.2

```
import numpy as np
from time import perf_counter
from numba import jit

@jit
def init(p_num, d_num):
    pos = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    accel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    for j in range(0, p_num):
        for i in range(0, d_num):
            pos[i,j] = 6.5
        # end for i
        for i in range(0, d_num):
            accel[i,j] = 4.2*pos[i,j]
        # end for i
    # end for j
    return pos, accel
# end def


# main
d_num = 5000
p_num = 5000

start_time = perf_counter( )
pos, accel = init(p_num, d_num)
stop_time = perf_counter( )

print(pos)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

- @jit compiles for the CPU by default
- Just put the desired decorator before the function. No other changes

EXAMPLE – @JIT – TARGET MULTI-CORE

2.3

```
import numpy as np
from time import clock
from numba import jit

@jit(nopython=True, parallel=True)
def init(p_num, d_num):
    pos = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    accel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    for j in range(0, p_num):
        for i in range(0, d_num):
            pos[i,j] = 6.5
        # end for i
        for i in range(0, d_num):
            accel[i,j] = 4.2*pos[i,j]
        # end for i
    # end for j
    return pos, accel
# end def


# main
d_num = 5000
p_num = 5000

start_time = perf_counter( )
pos, accel = init(p_num, d_num)
stop_time = perf_counter( )

print(pos)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

- @jit compiles for multiple cores (all cores)
- Just put the desired decorator before the function. No other changes

USING @VECTORIZE

- The decorator `@vectorize` is used for simple functions
    - Input and a single return
- Need type signature(s) for each data type
- Write in scalar notation
- Must be element-by-element
    - Check by writing in scalar
- For this particular code, we take the one function and break it into two functions to get one return per function
    - Each operates on an array

EXAMPLE – @VECTORIZE SINGLE CORE

2.4

```
import numpy as np
from time import perf_counter
from numba import vectorize

@vectorize(['float32(float32)'])
def set_pos(pos):
    return 6.5
# end def

@vectorize(['float32(float32, float32)'])
def set_accel(pos, accel):
    return 4.2*pos
# end def


# main
d_num = 5000
p_num = 5000
pos = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
accel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )

start_time = perf_counter( )
pos = set_pos(pos)
accel = set_accel(pos,accel)
stop_time = perf_counter( )

print(pos)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

- Split larger function into two functions
  - Each function has a single return
- Set the input definition for each function
- Use default target

EXAMPLE – @VECTORIZE PARALLEL

2.5

```
import numpy as np
from time import perf_counter
from numba import vectorize

@vectorize(['float32(float32)'], target='parallel')
def set_pos(pos):
    return 6.5
# end def

@vectorize(['float32(float32, float32)'], target='parallel')
def set_accel(pos, accel):
    return 4.2*pos
# end def


# main
d_num = 5000
p_num = 5000
pos = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
accel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )

start_time = perf_counter( )
pos = set_pos(pos)
accel = set_accel(pos,accel)
stop_time = perf_counter( )

print(pos)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

- Specify target, “parallel”

EXAMPLE – @VECTORIZE CUDA

2.6

```
import numpy as np
from time import perf_counter
from numba import vectorize

@vectorize(['float32(float32)'], target='cuda')
def set_pos(pos):
    return 6.5
# end def

@vectorize(['float32(float32, float32)'], target='cuda')
def set_accel(pos, accel):
    return 4.2*pos
# end def


# main
d_num = 5000
p_num = 5000
pos = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
accel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )

start_time = perf_counter( )
pos = set_pos(pos)
accel = set_accel(pos,accel)
stop_time = perf_counter( )

print(pos)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

- Switch target to NVIDIA GPU!

EXAMPLE – @CUDA.JIT

2.7

```
import numpy as np
from time import perf_counter
from numba import cuda

@cuda.jit
def init(p_num, d_num, pos, accel):
    for j in range(0, p_num[0]):
        for i in range(0, d_num[0]):
            pos[i,j] = 6.5
        # end for i
        for i in range(0, d_num[0]):
            accel[i,j] = 4.2*pos[i,j]
        # end for i
    # end for j
# end def


# main loop
d_num = np.zeros(1, dtype=int)
p_num = np.zeros(1, dtype=int)
d_num[0] = 5000
p_num[0] = 5000
pos = np.zeros( shape=(d_num[0], p_num[0]), dtype=np.float32 )
accel = np.zeros( shape=(d_num[0], p_num[0]), dtype=np.float32 )

start_time = perf_counter( )
d_p_num = cuda.to_device(p_num)
d_d_num = cuda.to_device(d_num)
d_pos = cuda.to_device(pos)
d_accel = cuda.to_device(accel)

init(p_num, d_num, pos, accel)

stop_time = perf_counter( )

print(pos)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

- @cuda.jit compiles for NVIDIA GPU by default
- Function cannot “return” data
  - Have to move data to/from GPU explicitly
  - Like C code
- Allows us to write a single function
- Scalars are Numpy array of size 1
  - Have to refer to scalar as the [0] element of the array

Functions Calling Functions

ROUTINES CALLING ROUTINES

- Let’s go a little further in our use of Numba and compile multiple functions that call each other
  - Better than putting everything in one giant function (can help compiler)
- Start with CPU (single pool of memory, don’t have to move data Host<->Device)
- Move to GPU
- The code is a slight modification of the previous code `@vectorize` CPU code
  - Add new function
  - Two functions now (additional array)

PLAIN PYTHON

Routines calling Routines

3

```python
import numpy as np
from time import perf_counter
from numba import jit

def init_pos_accel(p_num, d_num):
    pos = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    accel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    for j in range(0, p_num):
        for i in range(0, d_num):
            pos[i,j] = 6.5
        # end for i
        for i in range(0, d_num):
            accel[i,j] = 4.2*pos[i,j]
        # end for i
     # end for j

    vel = init_vel(p_num, d_num, pos, accel)

    return pos, accel, vel
# end init

def init_vel(p_num, d_num, pos, accel):
    vel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    for j in range(0, p_num):
        for i in range(0, d_num):
            vel[i,j] = pos[i,i] + accel[i,j] * 0.1*pos[i,j]
        # end for i
    # end for j

    return vel
# end def
```

```python
# main
d_num = 5000
p_num = 5000

start_time = perf_counter( )
pos, accel, vel = init_pos_accel(p_num, d_num)
stop_time = perf_counter( )

print(pos)
print(vel)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time
- start_time) )
print('')
```

- One routine calls another
  - `init_pos_accel()` calls `init_vel()`

CPU COMPILE BOTH ROUTINES

Routines calling Routines

3.1

```
import numpy as np
from time import perf_counter
from numba import jit

@jit
def init_pos_accel(p_num, d_num):
    pos = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    accel = np.zeros( shape=(d_num, p_num), dtype=np.float32
)
    for j in range(0, p_num):
        for i in range(0, d_num):
            pos[i,j] = 6.5
        # end for i
        for i in range(0, d_num):
            accel[i,j] = 4.2*pos[i,j]
        # end for i
    # end for j

    vel = init_vel(p_num, d_num, pos, accel)

    return pos, accel, vel
# end def

@jit
def init_vel(p_num, d_num, pos, accel):
    vel = np.zeros( shape=(d_num, p_num), dtype=np.float32 )
    for j in range(0, p_num):
        for i in range(0, d_num):
            vel[i,j] = pos[i,i] + accel[i,j] * 0.1*pos[i,j]
        # end for i
    # end for j

    return vel
# end def
```

```
# main
d_num = 5000
p_num = 5000

start_time = perf_counter( )
pos, accel, vel = init_pos_accel(p_num, d_num)
stop_time = perf_counter( )

print(pos)
print(accel)
print(vel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

- Much faster performance
- For a CPU, you need to specifically compile routines for which you want better performance
- Try targeting multi-core CPU (3.2)

@CUDA.JIT – MULTIPLE ROUTINES

Device routines

3.3

```
import numpy as np
from time import perf_counter
from numba import cuda

@cuda.jit
def init_all(p_num, d_num, pos, accel, vel):
    for j in range(0, p_num[0]):
        for i in range(0, d_num[0]):
            pos[i,j] = 6.5
        # end for i
        for i in range(0, d_num[0]):
            accel[i,j] = 4.2*pos[i,j]
        # end for i
    # end for j

    init_vel(p_num, d_num, pos, accel, vel)
# end def

@cuda.jit(device=True)
def init_vel(p_num, d_num, pos, accel, vel):
    for j in range(0, p_num[0]):
        for i in range(0, d_num[0]):
            vel[i,j] = pos[i,i] + accel[i,j] * 0.1*pos[i,j]
        # end for i
    # end for j
# end def
```

```
# main loop
d_num = np.zeros(1, dtype=int)
p_num = np.zeros(1, dtype=int)
d_num[0] = 5000
p_num[0] = 5000
pos = np.zeros( shape=(d_num[0], p_num[0]), dtype=np.float32 )
accel = np.zeros( shape=(d_num[0], p_num[0]), dtype=np.float32 )
vel = np.zeros( shape=(d_num[0], p_num[0]), dtype=np.float32 )

start_time = perf_counter( )
d_p_num = cuda.to_device(p_num)
d_d_num = cuda.to_device(d_num)
d_pos = cuda.to_device(pos)
d_accel = cuda.to_device(accel)
d_vel = cuda.to_device(vel)

init_all(p_num, d_num, pos, accel, vel)

stop_time = perf_counter( )

print(pos)
print(vel)
print(accel)
print('')
print('    Elapsed wall clock time = %g seconds.' % (stop_time - start_time) )
print('')
```

Device=True :: This GPU function will be called by another GPU function!

NUMBA SUMMARY

- Numba is great for improving performance (of arithmetically intensive code)
  - Even for a single core, you can get lots of speedup. You still get to write in Python!!!
  - You can even target multiple CPU cores to maybe get more speed
  - You can target NVIDIA GPUs very easily
    - You can do it simply using `@vectorize`
    - You can take advantage of CUDA features, but it is still mostly Python!
    - `@cuda.jit` is very powerful but you can use it with simple Python code
- Numba interoperates with other Python-GPU tools!

BEST PRACTICES

- There may be times when your Numba compiled function will not improve performance
    - Ask yourself – why?
- Example:
    - Don’t try to put an already compiled function into a Numba compiled function
    - Example – NumPy “convolve” function. It is already compiled for CPUs.
    - What do we do for GPUs?
        - Write your own convolution function ☺
        - Or find a library that already has this function available

TL;DR - OVERALL PYTHON-GPU CODING RECOMMENDATIONS

- Move computationally intensive sections of code to functions
- Use @jit on CPUs (Numba)
    - Start with single core then move to multi-core
- Switch to @vectorize (Numba)
- Port to GPUs (@vectorize and @cuda.jit)
    - May need to optimize data movement
- Investigate custom kernels when you need more performance (or have more time)
    - Good topic to bring up this afternoon at office hours!

CUPY

GENERALITIES

Scientific/Engineering Applications

- Good: Python has a huge number of libraries
- Bad: Python has a huge number of libraries
- Many scientific/engineering/data science codes are built using:
    - NumPy
    - SciPy
    - Scikit-learn
    - Pandas
    - Many others…
- A great deal of focus has been on Numpy

NUMPY

- NumPy is probably the most popular library/tool outside of the core Python
  - One of the first “big” projects for Python
  - Many years of development
  - Numeric focus
- Highly tuned (for CPUs)
- What about GPUs?

CUPY

https://github.com/cupy/cupy

- NumPy-like API accelerated with CUDA
  - Implements a subset of NumPy interface, but it’s very close to being complete
  - Adding some SciPy routines!
- Originally used by Chainer (DL framework popular in Japan)
- Calling sequence is **like** NumPy

```
>> import numpy as np
>> import cupy as cp
>> x_cpu = np.array([1, 2, 3])
>> l2_cpu = np.linalg.norm(x_cpu)
>> x_gpu = cp.array([1 ,2 ,3])
>> l2_gpu = cp.linalg.norm(x_gpu)
```

- Very Pythonic and very easy to use!

CUPY

Installation Notes

- “`pip install cupy`”
  - Last time I checked it was a really old version
- “`conda install cupy`”
  - On Windows, a bit on the older side (as of this writing – 6 months)
  - Linux has more up to date versions

CUPY FEATURES - 1

- NumPy uses a class: `numpy.ndarray`
- Cupy has a class: `cupy.ndarray` (almost identical – extra information)
- Features:
  - NumPy-like indexing
  - Most of Advanced indexing
  - Data Types (dtypes):
    - bool_, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, float64, complex64, complex128
  - Most of the array creation and manipulation routines
  - All operators with broadcasting
  - All universal functions for elementwise operations (add(), subtrace(), exp(), log(), sin()…)

CUPY FEATURES - 2

- Linear algebra functions:
    - dot, inner, outer, matmul, tensordot, kron, etc.
    - Linalg:
        - cholesky, QR, SVD, eigh, eigvalsh, deter, norm, matrix_rank, trace, solve, tensorsolve, inv, pinv
    - Transpose, other matrix functions (e.g trig, comparisons, logical), saves to file, random functions, distributions
    - Multi-dimensional matrices – indexing routines
    - Matrix arithmetic
- Reduction along axes (sum, max, argmax, etc.)
- FFT (forward and inverse)
- Uses NV libraries (cuFFT, cuBLAS, etc.)

CUPY FEATURES - 3

- Mathematical Functions:
  - Many standard functions: cos, tan, sin, floor, ceil, sum, prod, log, multiple, maximum
- Random functions:
  - rand, random, ranf, sample, random.seed, shuffle
- Sort, Search, Count:
  - sort, argmax, partition, nonzero, where
- Stats:
  - mean, var, std, bincount, amin, amax, sum, histograms (binsort)
- Sparse Matrix class:
  - Similar functions to full matrix

CUPY FEATURES - 4

- NumPy - CuPy CUDA Code Support
    - Move data to/from GPU, device management, memory management, debugging, streams, events, hooks for profiling, testing modules
- Can do custom kernels
    - User-defined elementwise CUDA kernels
    - User-defined reduction CUDA kernels
- Documentation is excellent

CUPY NEW-ISH FEATURES

- Started to write SciPy functions
  - `cupyx`
- Examples:
  - `cupyx.scipy.fft.X()` - FFT
  - `cupyx.scipy.fftpack.X()` - Legacy FFT
  - `cupyx.scipy.linalg.lu_factor()`
  - `cupyx.scipy.linalg.lu_solve()`
  - `cupyx.scipy.linalg.solve.triangular()`

CUPY SIMPLE EXAMPLE

SVD

4.1

```
import cupy as cp

A = cp.random.uniform(low=-1., high=1., size=(64, 64)).astype(cp.float32)

u, s, v = cp.linalg.svd(A)
```

u, s, v are still on GPU

CUPY SIMPLE EXAMPLE

SVD – 2 (copy data to/from CPU)

4.2

```
import cupy as cp
import numpy as np

A_cpu = np.random.uniform(low=-1., high=1., size=(64, 64)).astype(np.float32)
A_gpu = cp.asarray(A_cpu)

u_gpu, s_gpu, v_gpu = cp.linalg.svd(A_gpu)
print "type(u_gpu) = ",type(u_gpu)

u_cpu = cp.asnumpy(u_gpu)
print "type(u_cpu) = ",type(u_cpu)
```

Copy A_cpu to GPU.
Becomes CuPy object

Copy u_gpu to Host.
Becomes Numpy object

```
[laytonjb@home4 CUPY]$ python svd2.py
type(u_gpu) =  <type 'cupy.core.core.ndarray'>
type(u_cpu) =  <type 'numpy.ndarray'
```

CUPY SIMPLE EXAMPLE

Matrix Multiplication on GPU

```
import math
import cupy as cp

A = cp.random.uniform(low=-1., high=1., size=(64,64)).astype(cp.float32)
B = cp.random.uniform(low=-1., high=1., size=(64,64)).astype(cp.float32)

C = cp.matmul(A,B)
```

MATRIX MULTIPLICATION

4.3
- The example in the first cell of the Jupyter notebook creates the two input arrays on either the CPU or the GPU (uses cupy method, `asnumpy()`)
  - Then the multiplication is performed (and timed)
  - Try varying the “size” variable to see where the GPU is faster than the CPU.

4.4
- Second cell copies the data from the CPU to the GPU using a method as `array()`
  - The timings include the time to transfer the data from the CPU to the GPU (and back)
  - Try varying the size of the problem to where the CPU and GPU times are about the same
  - How does this compare to the previous cell where data was created on the GPU?

OBSERVATIONS

- Notice you must explicitly copy data from host to device or from device to host
  - Remember that this can limit performance
  - **“What is computed on the GPU, stays on the GPU”**
  - Can create data on GPU
- After computation, data remains on device
  - You can copy it back to host
  - Or you can perform further computations using those results
- Almost anything that is coded in NumPy, can be coded in CuPy

ADVANTAGES OF CUPY

Matrix Manipulation

- Leaving data on GPU allows easier coding for equations such as,

$$Q = XA^T + AX + DWD^T$$

$$C_1 = (I - B_p B_p^+) Q_p (I - B_p B_p^+)$$

- Allows equations to be broken into pieces for easier coding
    - Can copy back intermediate results as needed

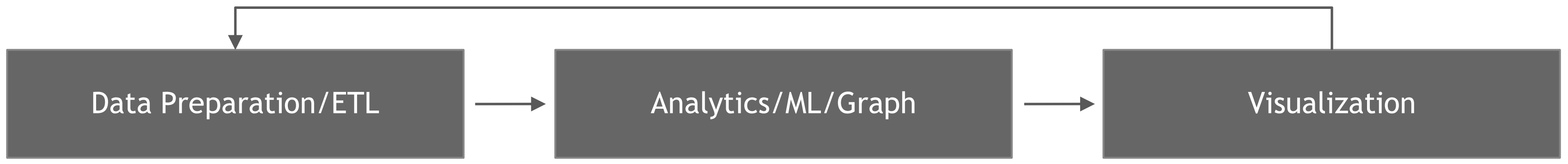
TL;DR - OVERALL PYTHON-GPU CODING RECOMMENDATIONS

- Move computationally intensive sections of code to functions
- Use @jit on CPUs (Numba)
    - Start with single core then move to multi-core
- Switch to @vectorize (Numba)
- Port to GPUs (@vectorize and @cuda.jit)
    - May need to optimize data movement
- **Use CuPy for common computations**
- **Consider custom kernels when you need more performance (or have more time)**

MORE GPU PYTHON TOOLS WORTH CHECKING OUT

What is RAPIDS?

End-to-End GPU Accelerated Data Science



cuDF

- ➢ GPU-accelerated ETL functions
- ➢ Tracks pandas and other common PyData APIs
- ➢ Dask + UCX integration for scaling

cuML

- ➢ GPU-native ML library, including XGBoost, FIL, HPO, and more

cuGraph

- ➢ GPU graph analytics, including TSP, PageRank, and more

cuxfilter

- ➢ GPU-accelerated cross-filtering

pyViz integration

- ➢ Plotly Dash, Bokeh, Datashader, HoloViews, hvPlot

Domain-Specific Libraries

CLX + Morpheus
Cyber log processing + anomaly detection

cuSignal
Signals processing

cuSpatial
Spatial analytics

cuStreamz
Streaming analytics

cuCIM
Computer vision & image processing primitives

node-RAPIDS
Bindings for node.js

...and more!

USING RAPIDS

- Only runs on Linux
- Available via pip and conda
- Use RAPIDS for heavy computational components
  - Modeling algorithms such as
  - Then use Pandas for visualization
- Can do:

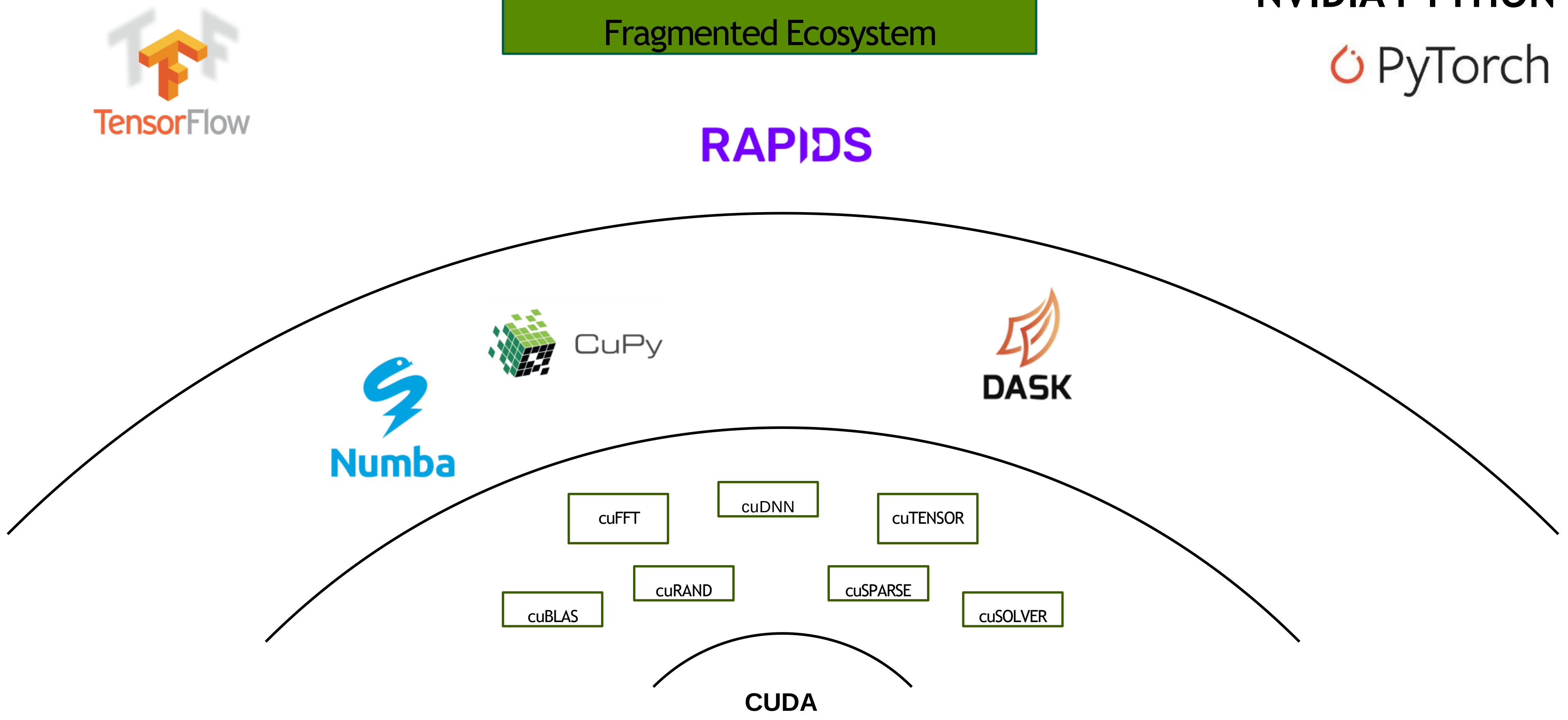
```
import cudf as pd
```

```
import cudf as pd
import numpy as np
from time import time

import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline

wine_set = pd.read_csv("data/winequality-red.csv", sep=';')

wine_set.head(n=5)
wine_set.tail(n=5)
```

NVIDIA PYTHON

Where we are with Python today

Fragmented Ecosystem



CUDA PYTHON

Native Access to Platform Specialization

EA Release – Out in August

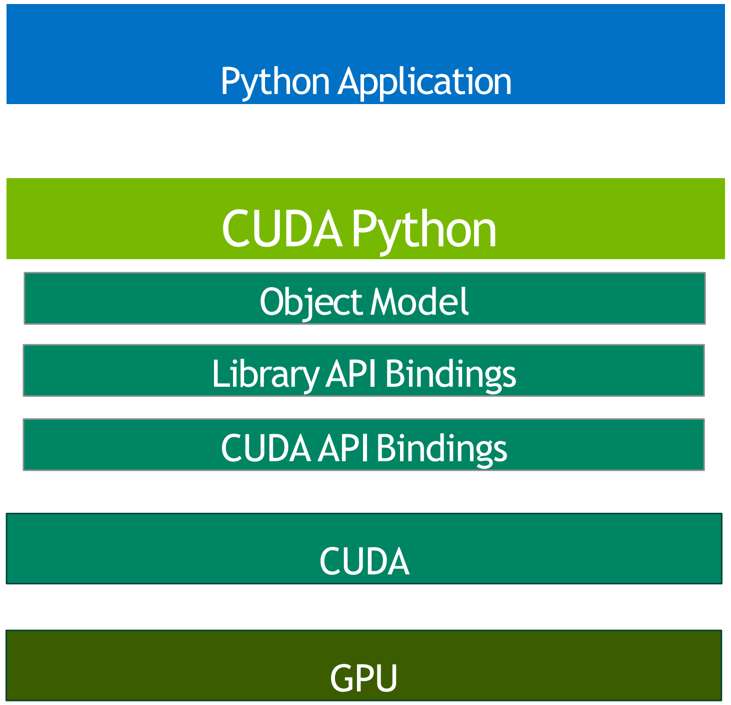
- Python Bindings on CUDA 11.4 Driver/Runtime API
- GitHub and Documentation Site

GA Release – Out now with CUDA 11.5

- GitHub source code
- Packaging: PIP and Conda
- Additional CUDA Functions

Future Releases

- Python bindings for Libraries
- Introduce Python Object Model

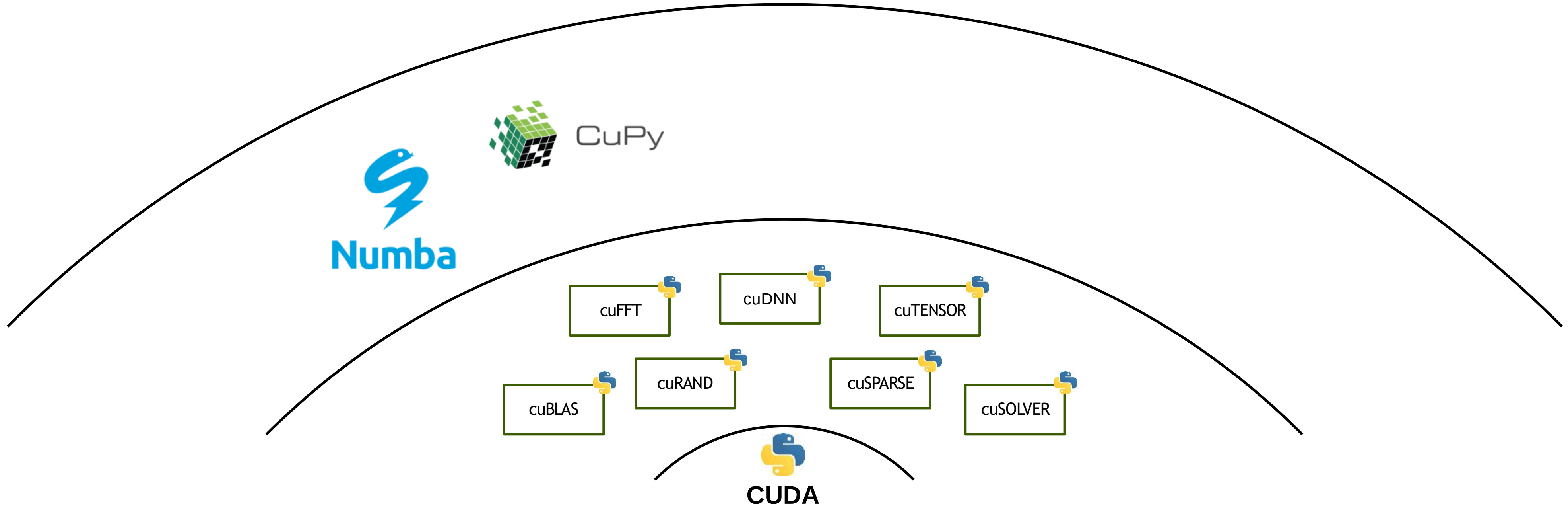


GitHub: https://github.com/NVIDIA/cuda-python
Docs: https://nvidia.github.io/cuda-python/

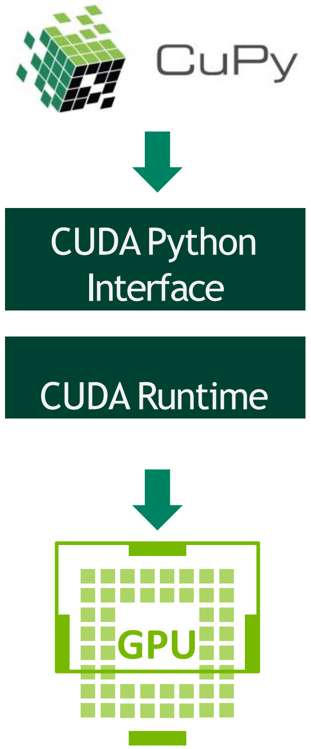
Email: cuda-python-dev@nvidia.com

CUDA Python Adoption in the Ecosystem

NVIDIA PYTHON



AVAILABLE NOW IN CUPY



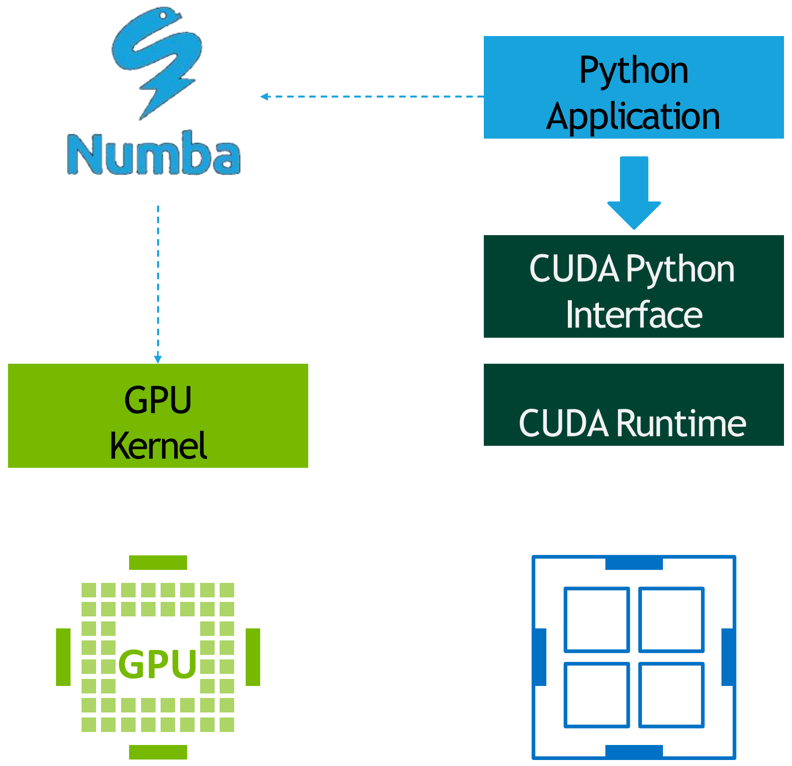
PR on GitHub

https://github.com/cupy/cupy/pull/563

Build from source with flag

```
export CUPY_USE_CUDA_PYTHON=1
```

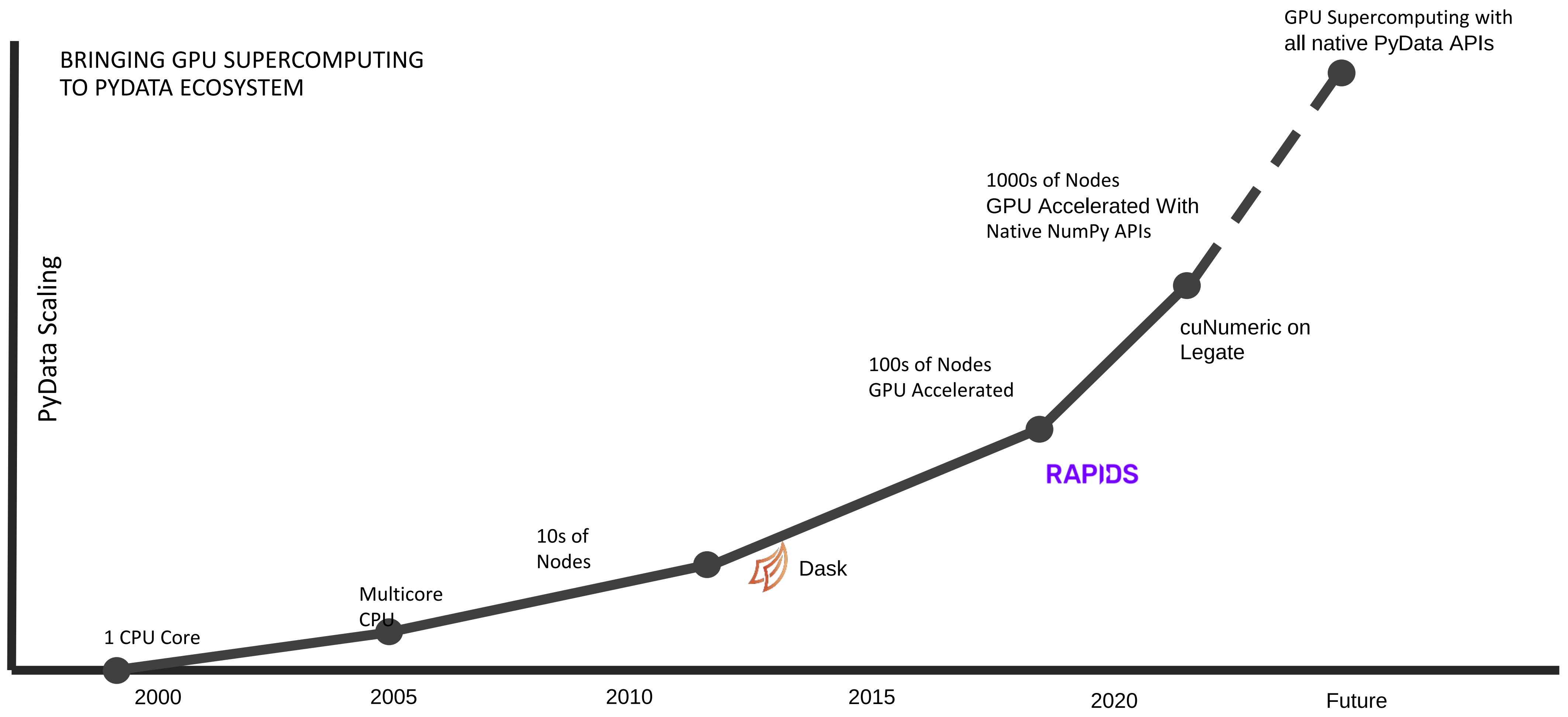
AVAILABLE SOON IN NUMBA



PR on GitHub

https://github.com/numba/numba/pull/7461

BRINGING GPU SUPERCOMPUTING TO PYDATA ECOSYSTEM



```
import numpy as np

a = np.random.randn(16).reshape(4,
4)
b = a + a.T
b
```

```
import dask.array as da
import numpy as np

a = da.from_array(
    np.random.randn(160_000).reshape(400,
400),
    chunks=(100, 100))
b = a + a.T
b.compute()
```

```
import dask.array as da
import cupy as cp

a = da.from_array(
    cp.random.randn(160_000).reshape(400, 400),
    chunks=(100, 100),
    asarray=False)
b = a + a.T
b.compute()
```

```
import cunumeric as np

a = np.random.randn(160_000).reshape(400,
400)
b = a + a.T
b
```

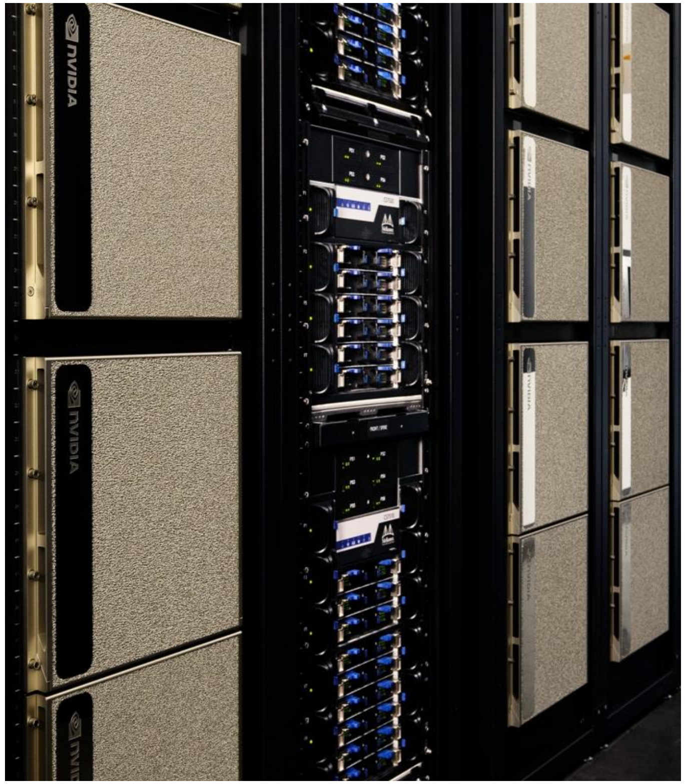
PYTHON ECOSYSTEM GOALS

Have Your Cake and Eat It Too

Productivity

```
def cg_solve(A, b, conv_iters):
    x = np.zeros_like(b)
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    converged = False
    max_iters = b.shape[0]

    for i in range(max_iters):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)

        if i % conv_iters == 0 and \
            np.sqrt(rsnew) < 1e-10:
            converged = i
            break

        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
```

Performance



PRODUCTIVITY

Sequential and Composable Code

```
def cg_solve(A, b, conv_iters):
    x = np.zeros_like(b)
    r = b - A.dot(x)
    p = r
    rsold = r.dot(r)
    converged = False
    max_iters = b.shape[0]

    for i in range(max_iters):
        Ap = A.dot(p)
        alpha = rsold / (p.dot(Ap))
        x = x + alpha * p
        r = r - alpha * Ap
        rsnew = r.dot(r)

        if i % conv_iters == 0 and \
            np.sqrt(rsnew) < 1e-10:
            converged = i
            break

        beta = rsnew / rsold
        p = r + beta * p
        rsold = rsnew
```

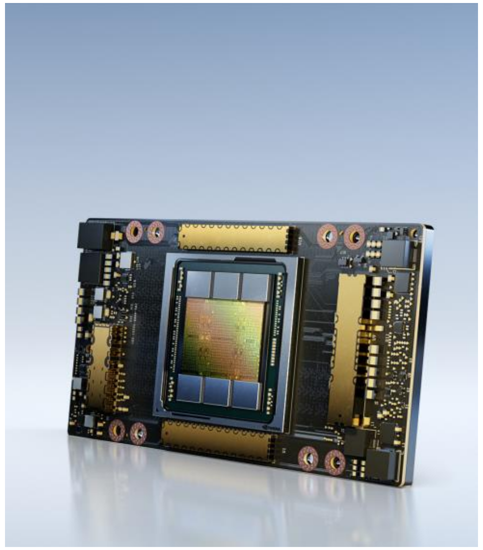
- Sequential semantics - no visible parallelism or synchronization
- Name-based global data – no partitioning
- Composable – can combine with other libraries and datatypes

PERFORMANCE

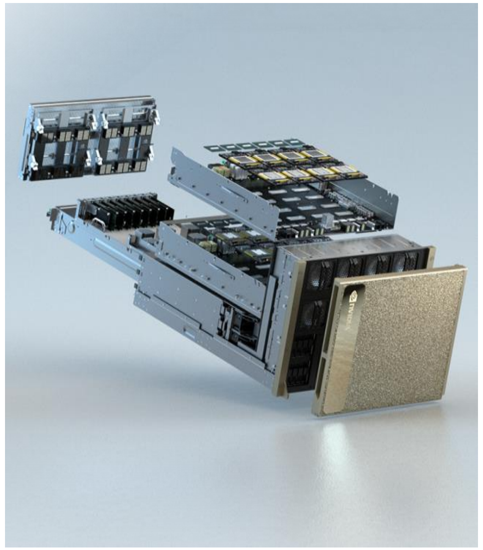
Transparent Acceleration

- Transparently run at any scale needed to address computational challenges at hand
- Automatically leverage all the available hardware

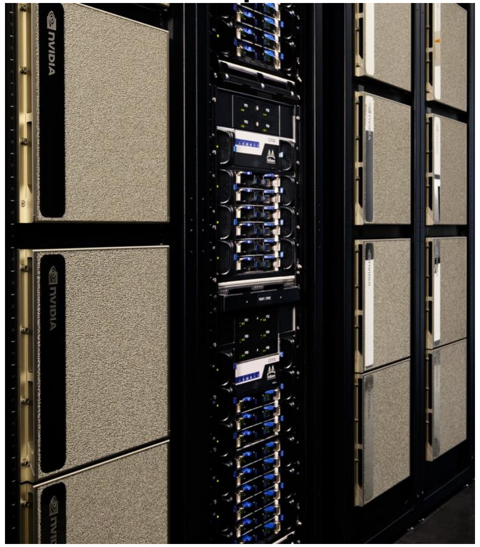
GPU



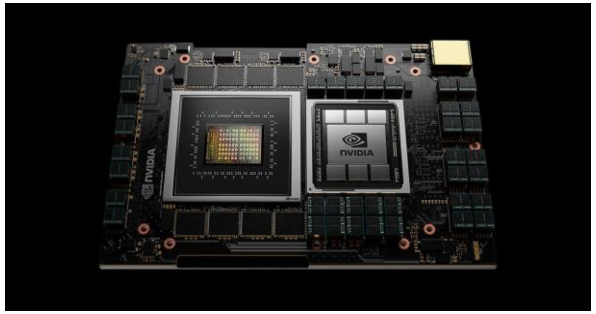
DGX-2



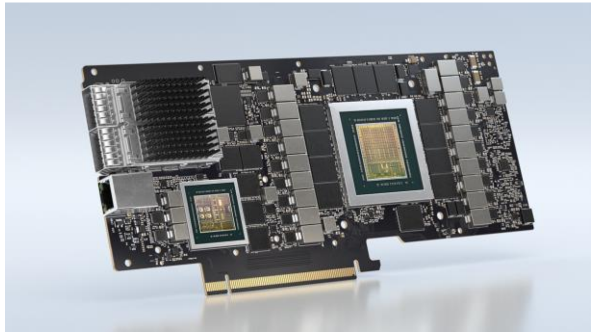
DGX SuperPod



Grace CPU



DPU



CUNUMERIC

Automatic NumPy Acceleration and Scalability

cuNumeric

CuNumeric transparently accelerates and scales existing Numpy workloads

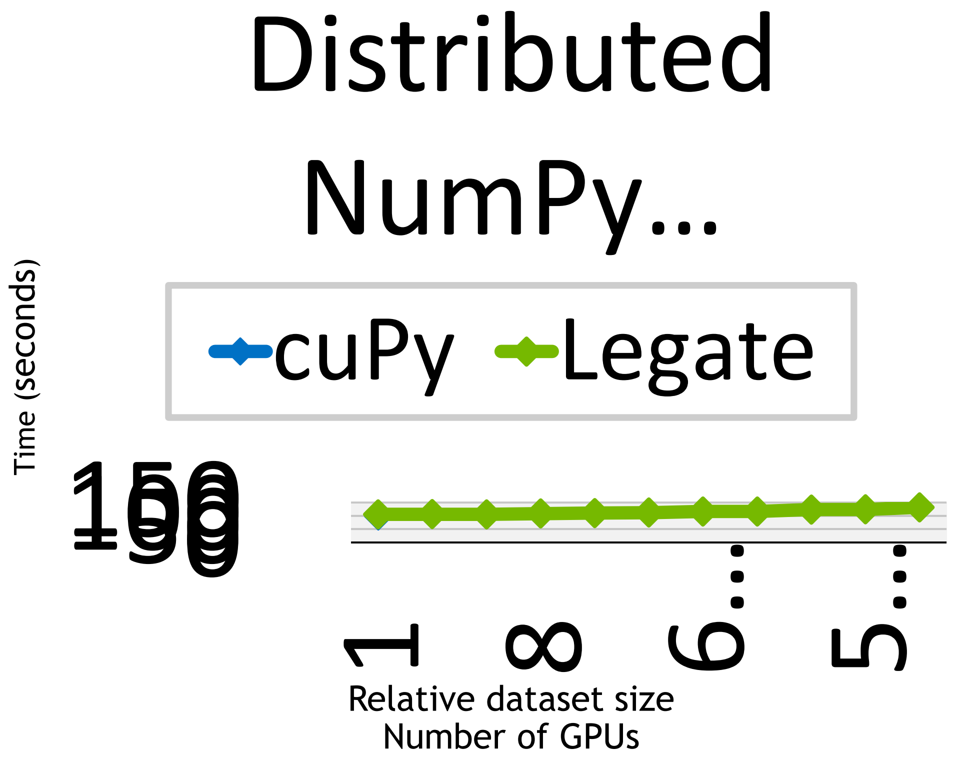
Program from the edge to the supercomputer in Python by changing 1 import line

Pass data between Legate libraries without worrying about distribution or synchronization requirements

Alpha release available at github.com/nv-legate

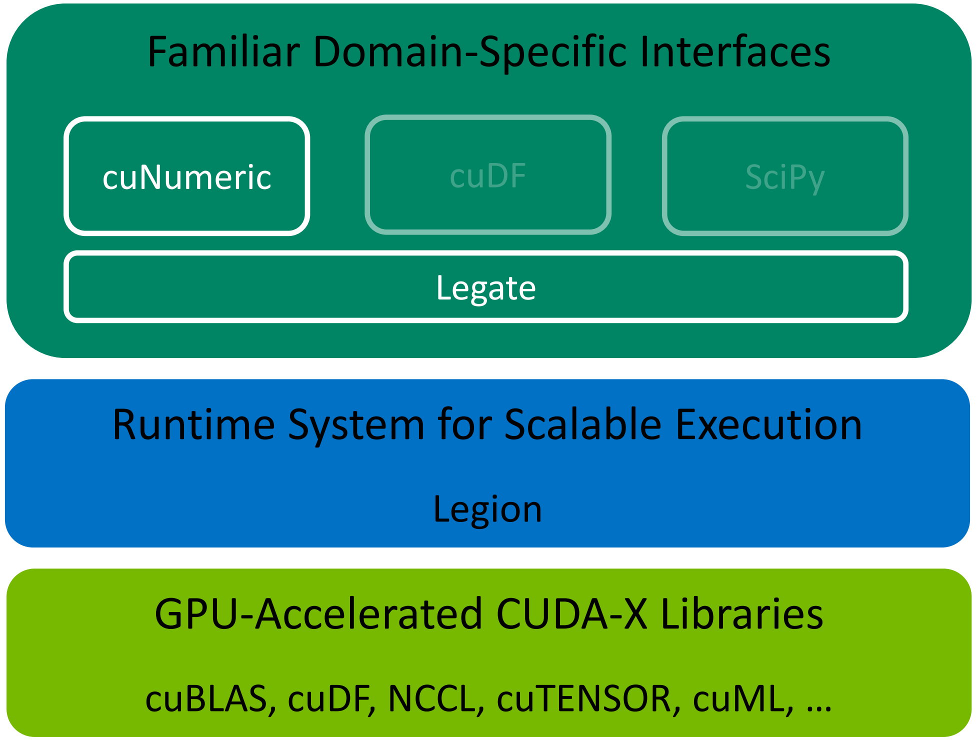
```
for _ in range(iter):
    un = u.copy()

    vn = v.copy()
    b = build_up_b(rho, dt, dx, dy, u, v)
    p = pressure_poisson_periodic(b, nit, p, dx, dy)

…
```

Extracted from “CFD Python” course at https://github.com/barbagroup/CFDPython
Barba, Lorena A., and Forsyth, Gilbert F. (2018). CFD Python: the 12 steps to Navier-Stokes equations. *Journal of Open Source Education*, **1**(9), 21, https://doi.org/10.21105/jose.00021



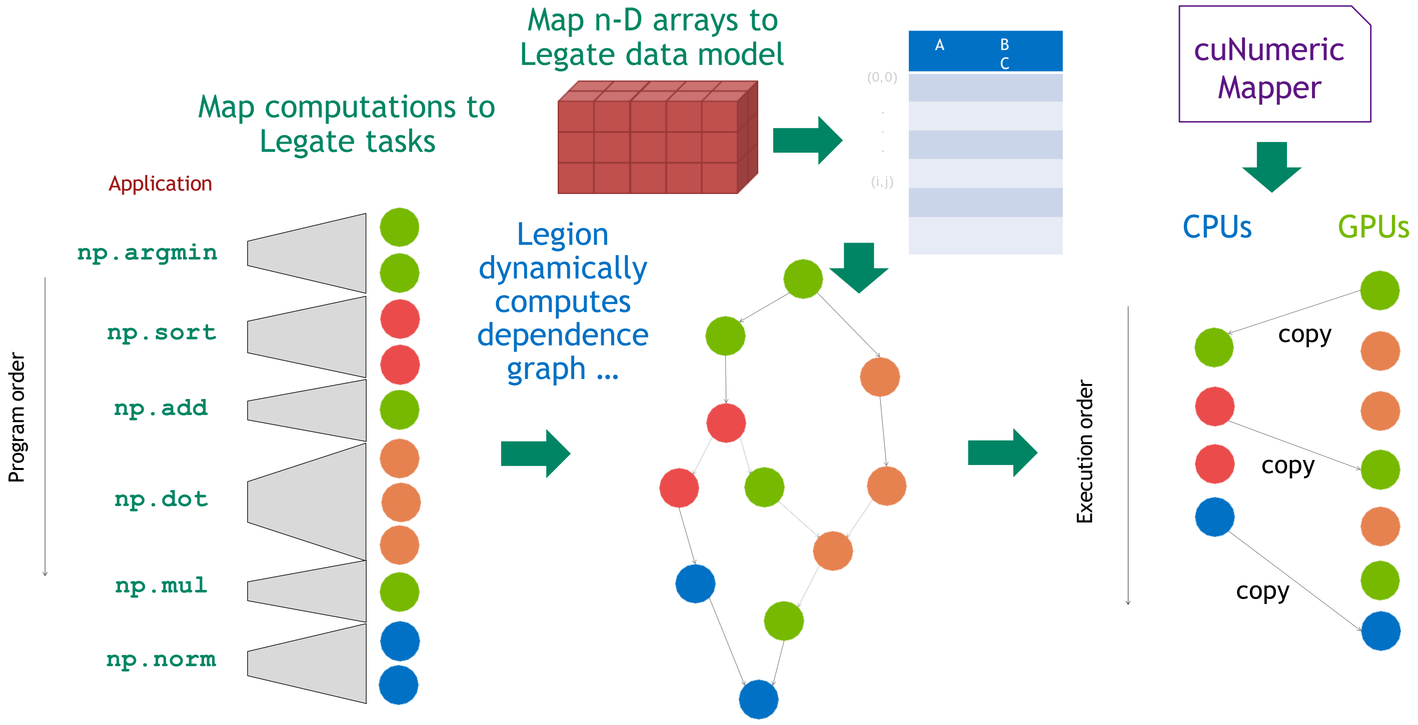
LEGATE ECOSYSTEM ARCHITECTURE

Scalable implementations of popular domain-specific APIs


Familiar Domain-Specific Interfaces
cuNumeric
cuDF
SciPy
Legate
Runtime System for Scalable Execution
Legion
GPU-Accelerated CUDA-X Libraries
cuBLAS, cuDF, NCCL, cuTENSOR, cuML, …

CUNUMERIC ARCHITECTURE

Leveraging the Strengths of Legion through Legate



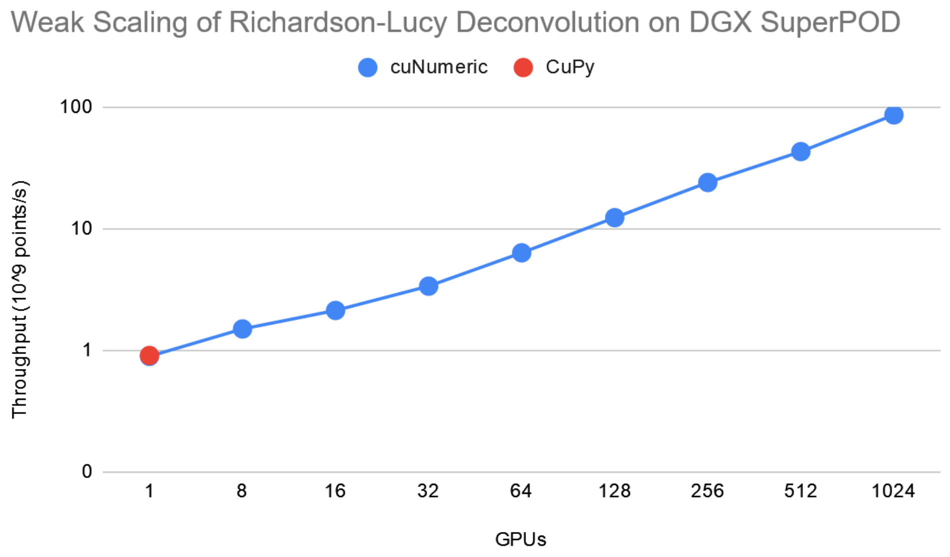
GTC21 DEEP-DIVE TALK

"LEGATE: SCALING THE PYTHON ECOSYSTEM [A31168]"

PERFORMANCE RESULTS

Microscopy Demo with Richardson-Lucy Deconvolution

```
def richardson_lucy(image, psf, num_iter=50,
                    clip=True, filter_epsilon=None):
  float_type = _supported_float_type(image.dtype)
  image = image.astype(float_type, copy=False)
  psf = psf.astype(float_type, copy=False)
  im_deconv = np.full(image.shape, 0.5, dtype=float_type)
  psf_mirror = np.flip(psf)

  for _ in range(num_iter):
    conv = convolve(im_deconv, psf, mode='same’)
      if filter_epsilon:
         with np.errstate(invalid='ignore’):
            relative_blur = np.where(conv < filter_epsilon, 0,
                                     image / conv)
      else:
         relative_blur = image / conv
      im_deconv *= convolve(relative_blur, psf_mirror,
                            mode='same’)

   if clip:
      im_deconv[im_deconv > 1] = 1
      im_deconv[im_deconv < -1] = -1

  return im_deconv
```



CUNUMERIC API COVERAGE

Module Name	Module Path	NumPy	CuPy	cuNumeric
Top Level	np.*	401	229	86
NDArray	np.ndarray	56	47	32
Linear Algebra	np.linalg	20	16	1
FFT	np.fft	18	18	0
Random Sampling	np.random	51	49	5

https://github.com/nv-legate/cunumeric

GPU Programming questions?
kkeipert@nvidia.com

Python GPU Programming
CUDA C/C++/Fortran
GPU Porting Strategy
Performance Profiling and Developer Tools