WRITING CUDA KERNELS IN PYTHON WITH NUMBA

CUDA COMMUNITY MEETUP, 15 FEBRUARY 2022

GRAHAM MARKALL – SOFTWARE ENGINEER, RAPIDS

BRIEF SELF-INTRODUCTION

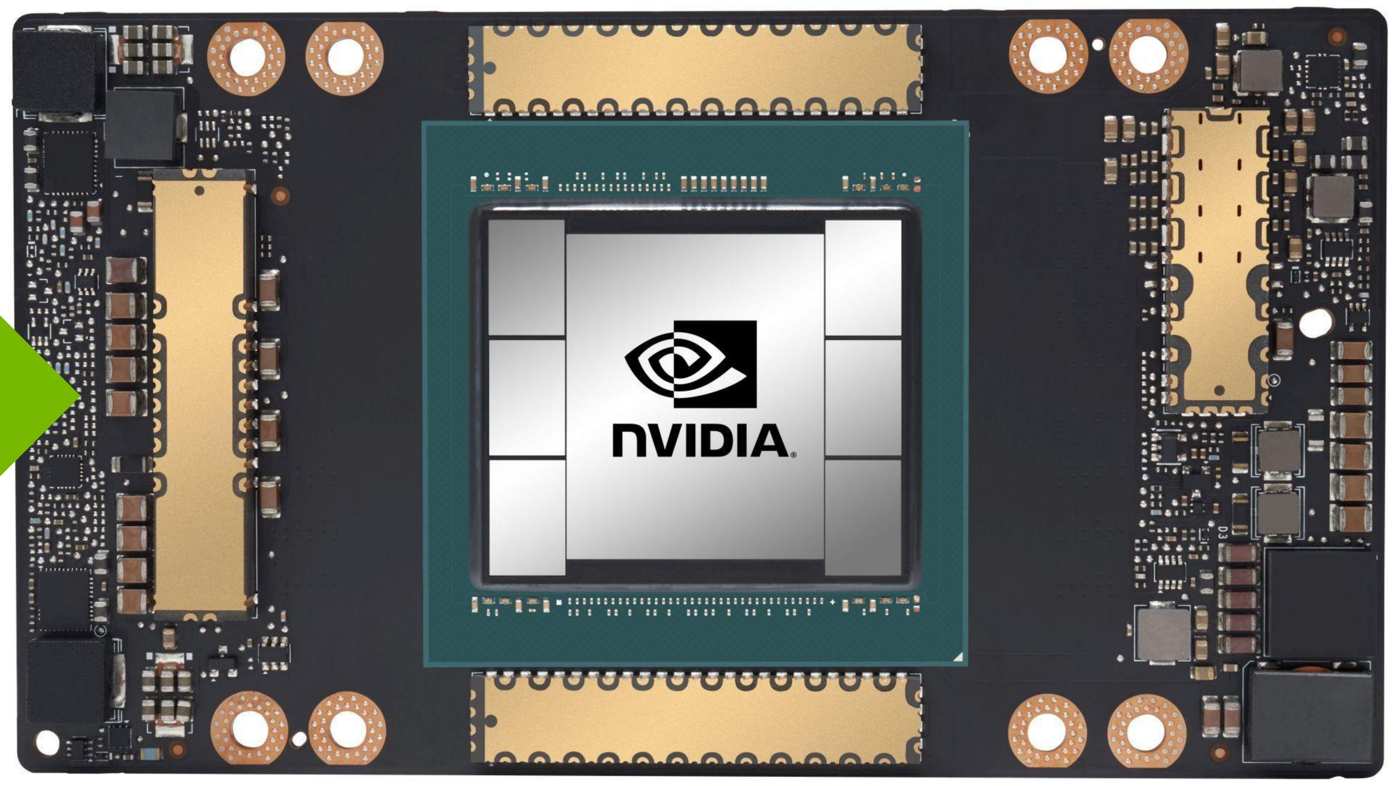
- Software Engineer in RAPIDS
- Numba CUDA target maintainer
- Supporting cuDF / RAPIDS use cases
- Joined NVIDIA Dec 2019

- Background in compilers / numerical methods / HPC:
  - GCC, Binutils, GDB, LLVM, …
  - PDEs, Finite elements, sparse linear solvers,
  - Domain-specific languages for HPC

Numba

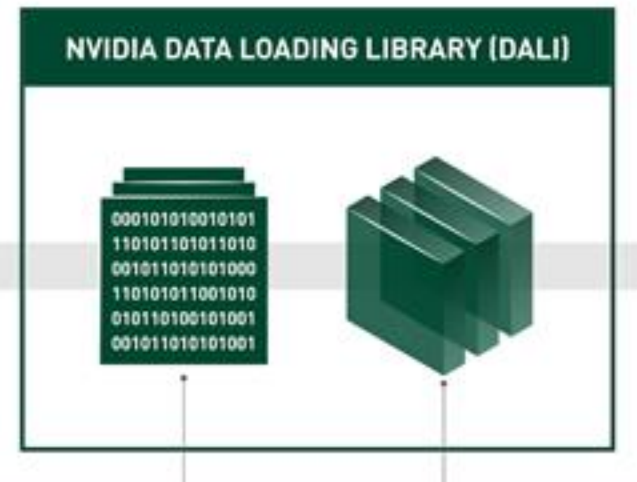
- A compiler: Python code to CUDA GPUs, focused on NumPy / arrays

```
@cuda.jit
def increment_a_2D_array(an_array):
    x, y = cuda.grid(2)
    if x < an_array.shape[0] and y < an_array.shape[1]:
        an_array[x, y] += 1
```



- Glue that brings together all parts of the CUDA Python ecosystem

RAPIDS   Numba   CuPy   PyTorch



+ more…

- Aim: Make writing CUDA code in Python as natural a choice as C / C++

TALK ROADMAP

- Overview of Numba and CUDA Python
- Notable examples
- Tool support
- Extra material:
  - Code generation Utilities
  - The CUDA Python ecosystem

WHAT IS NUMBA?

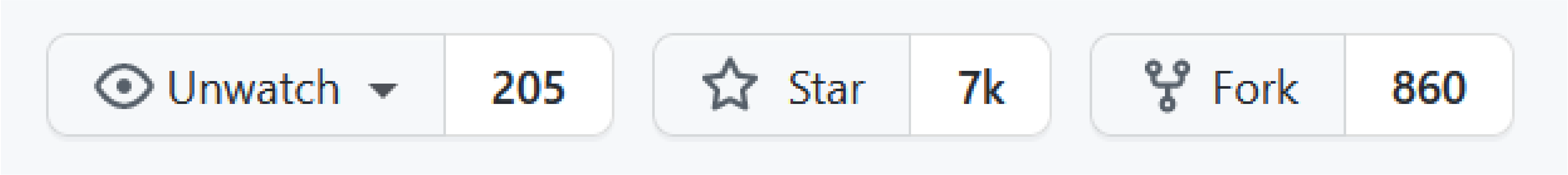
- A ***Just-in-time*** (JIT) compiler for Python functions.
- ***Opt-in***: Numba only compiles the functions you specify
- Focused on ***array-oriented*** and ***numerical*** code
  - Trade-off: subset of Python for better performance
- Alternative to native code, e.g. C / Fortran / Cython / CUDA C/C++
- Targets:
  - CPUs: x86, PPC, ARMv7 / v8
  - GPUs: CUDA, ~~AMD ROC~~

WHY NUMBA AND PYTHON?

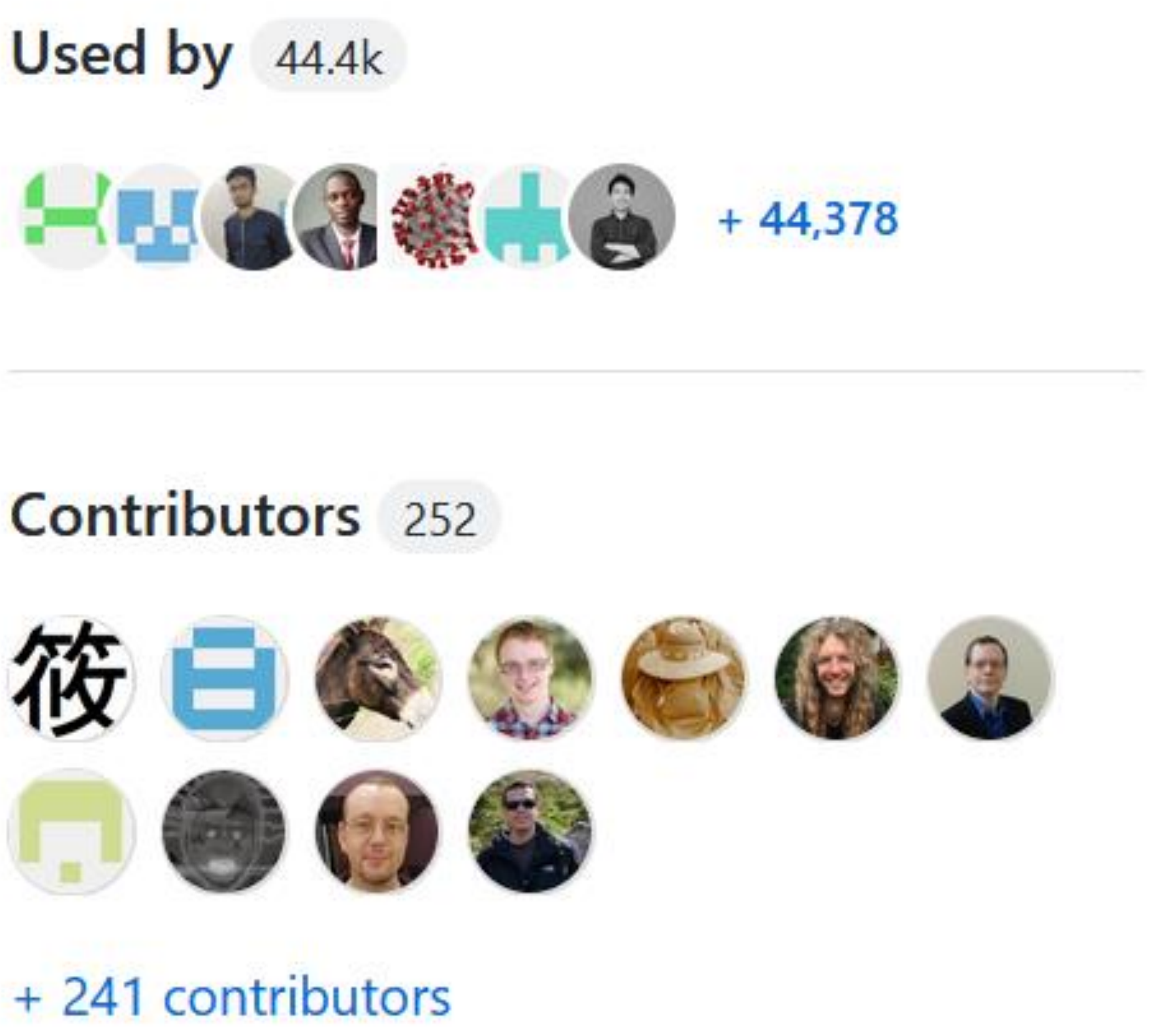
- PyData ecosystem strength:
  - Libraries: NumPy, Pandas, scikit-learn, etc.
  - GPU: RAPIDS, PyTorch, CuPy, TensorFlow, JAX, etc.
- Comfort Zone: keeping all code as Python code
  - Allows focus on algorithmic development
  - Minimise development time
  - Maintain interoperability

NUMBA USERS AND DEVELOPERS

- PyPI: 250,000 / Conda 16,000 downloads per day



- Libraries:
  - RAPIDS (DS / AI / ML)
  - STUMPY (Time series)
  - Datashader (viz)
  - Legate Pandas (DS)
  - Comcast Rapid IP Checker
  - More: https://github.com/gmarkall/numba-cuda-users



- Core maintainers:
  - 3 Anaconda, 1 NVIDIA

RUNNING PYTHON CODE ON CUDA GPUS

```python
def vector_add(r, x, y):
    for i in range(len(x)):
        r[i] = x[i] + y[i]
```

```python
from numba import cuda

@cuda.jit
def vector_add(r, x, y):
    start = cuda.grid(1)
    step = cuda.gridsize(1)
    stop = len(r)

    for i in range(start, stop, step):
        r[i] = x[i] + y[i]
```

```python
vector_add[grid_dim, block_dim](r, x, y)
```

(CUDA) PYTHON VS. (CUDA) C/C++

- Not an exhaustive comparison! just some high-level observations:

Feature	C/C++	Python
Typing	Mostly static, strong	Weaker, “duck”
Memory	Programmer-managed	Garbage-collected
Compilation	Ahead-of-time	Runtime to bytecode, interpreted
Usage mode	Write, compile, then run	Write then run / interactive

CUDA PYTHON VS. CUDA C++

```cpp
__device__ int sumReduction(thread_group g, int *x, int val)
{
    // rank of this thread in the group
    int lane = g.thread_rank();

    // for each iteration of this loop, the number of threads active in the
    // reduction, i, is halved, and each active thread (with index [lane])
    // performs a single summation of it's own value with that
    // of a "partner" (with index [lane+i]).
    for (int i = g.size()/2; i > 0; i /= 2)
        {

            // store value for this thread in temporary array
            x[lane] = val;

            // synchronize all threads in group
            g.sync();


            if(lane<i)
                // active threads perform summation of their value with
                // their partner's value
                val += x[lane + i];

            // synchronize all threads in group
            g.sync();

        }

    // master thread in group returns result, and others return -1.
    if (g.thread_rank()==0)
        return val;
    else
        return -1;
}
```

```python
@cuda.jit(device=True)
def sumReduction(g, x, val):
    """
    Calculates the sum of val across the group g. The workspace array, x,
    must be large enough to contain `g.size` integers.
    """
    # Rank of this thread in the group
    lane = g.thread_rank

    # For each iteration of this loop, the number of threads active in the
    # reduction, i, is halved, and each active thread (with index [lane])
    # performs a single summation of it's own value with that
    # of a "partner" (with index [lane+i]).
    i = g.size // 2
    while i > 0:
        # Store value for this thread in temporary array
        x[lane] = val

        # Synchronize all threads in group
        g.sync()

        if lane < i:
            # Active threads perform summation of their value with
            # their partner's value
            val += x[lane + i]

        # synchronize all threads in group
        g.sync()

        i //= 2

    # Master thread in group returns result, and others return -1.
    if g.thread_rank == 0:
        return val
    else:
        return -1
```

SUPPORTED CUDA FEATURES

- Memory:
  - On-device memory, pinned memory, unified memory, managed memory
  - Shared memory, local memory
- Intrinsics:
  - The usual indices – thread ID, block ID, etc.
  - Atomics
  - Cooperative groups: grid groups and sync
  - Sychronization: thread fences, sync threads, warp ballot and sync functions
  - Integer: popcount, bit reverse, count leading zeroes, find first set
  - Other: nanosleep
- Multi-device / multi-context:
  - Multiple contexts, multiple devices, legacy IPC
- Concurrency:
  - Streams: Create/use multiple streams, legacy default stream, Per-Thread Default Stream
  - Asynchronous callbacks on streams
  - Events: Create and record, sync, wait
- Profiler APIs:
  - Profile start and stop
- Compilation:
  - Compilation to PTX for external use

Request additional features:
https://github.com/numba/numba/issues
Feature request link

SUPPORTED PYTHON SYNTAX

- Inside functions decorated with `@cuda.jit`:
  - assignment, indexing, arithmetic
  - `if` / `else` / `for` / `while` / `break` / `continue`
  - raising exceptions
  - `assert`, when passing `debug=True`
  - calling other compiled functions (CPU or CUDA `jit`)

- Documentation on supported language constructs

UNSUPPORTED PYTHON SYNTAX

- Also inside functions decorated with @cuda.jit:
  - `try` / `except` / `finally`
  - `with`
  - (`list`, `set`, `dict`) comprehensions
  - Generators
- Classes cannot be decorated with `@cuda.jit`

SUPPORTED PYTHON FEATURES

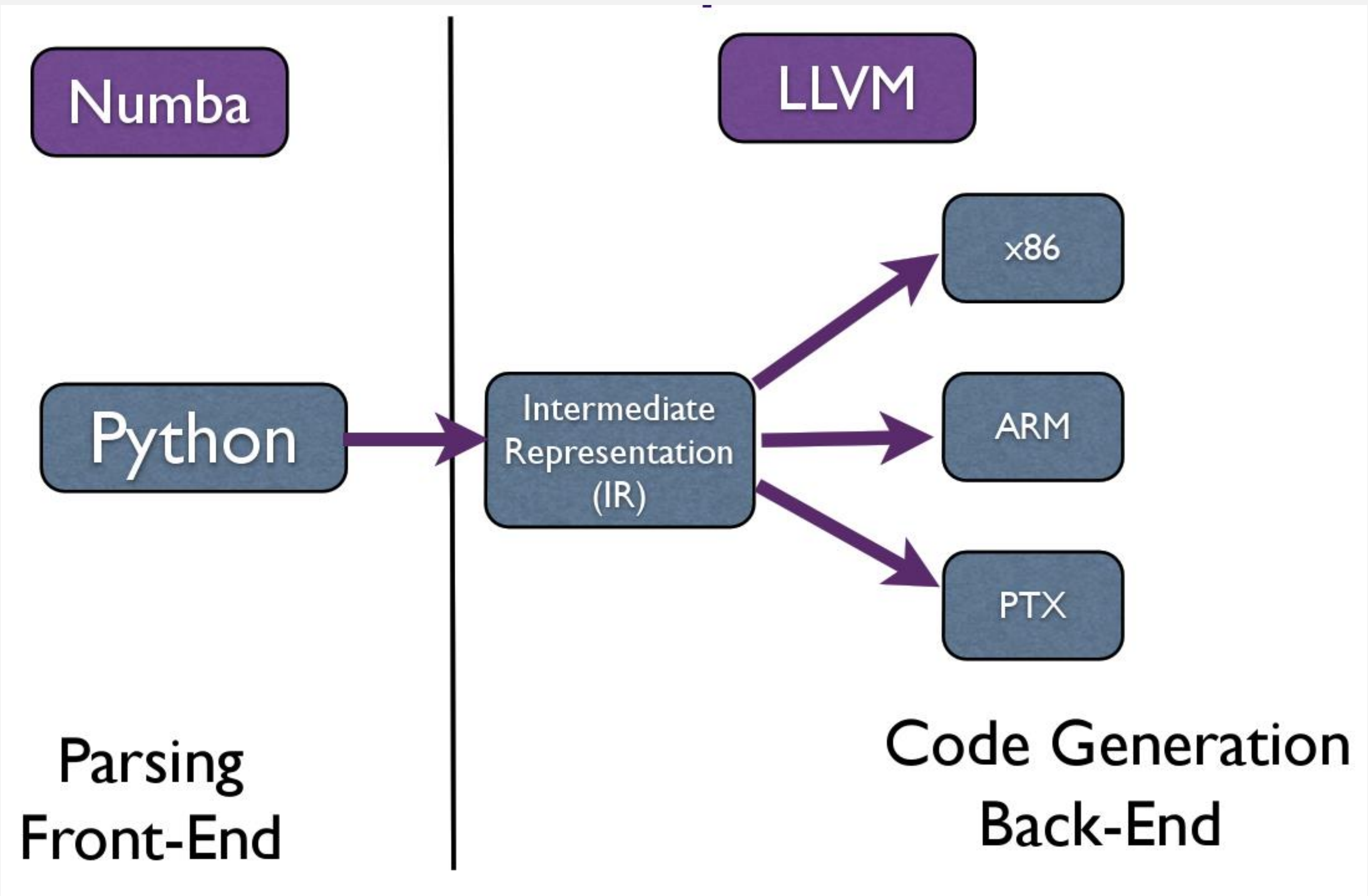
- Types:
  - `int`, `bool`, `float`, `complex`
  - `tuple`, `None`
  - Documentation on supported types

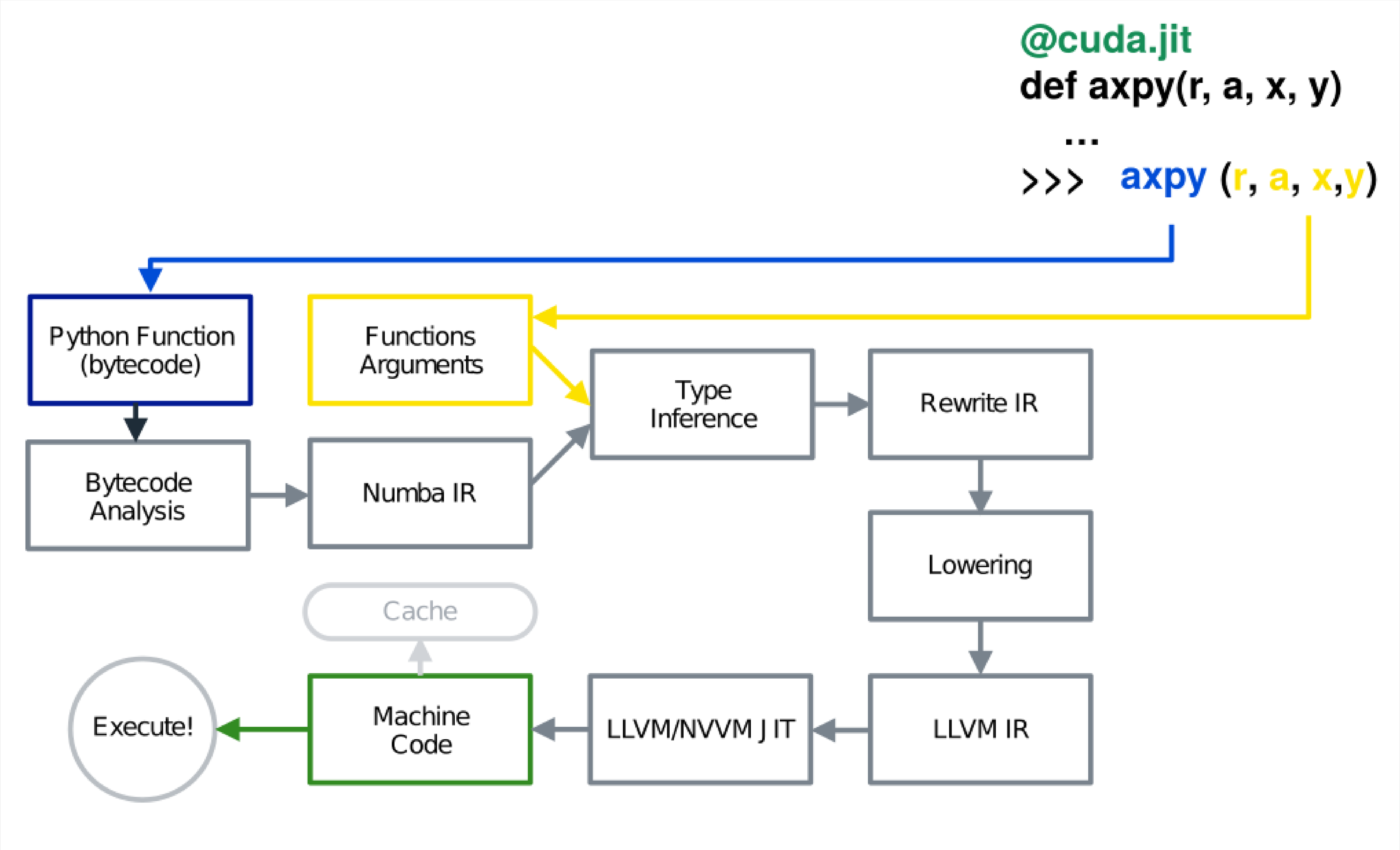
- Built-in functions:
  - `abs`, `enumerate`, `len`, `min`, `max`, `print`, `range`, `round`, `zip`
  - Documentation on supported builtins

SUPPORTED PYTHON MODULES

- Standard library:
  - `cmath`, `math`, `operator`
  - [Comprehensive list in documentation]
- NumPy:
  - Arrays: scalar and structured type
    - except when containing Python objects
  - Array attributes: `shape`, `strides`, etc.
  - indexing, slicing
  - Scalar types and values (including `datetime` types)
  - Scalar ufuncs (e.g. `np.sin`)

NUMBA ARCHITECTURE / INTERNALS



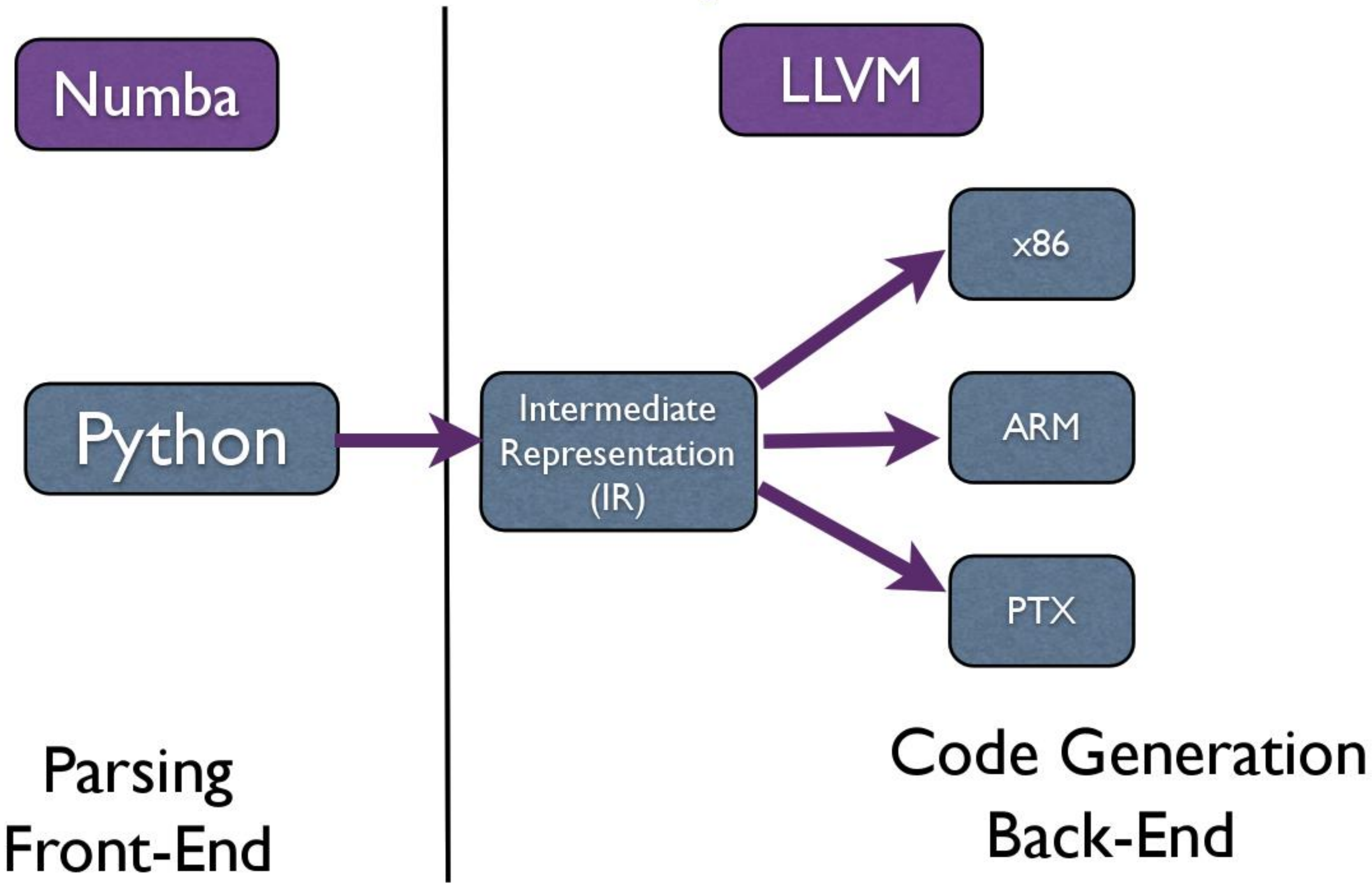


COMPONENTS OF THE CUDA TARGET

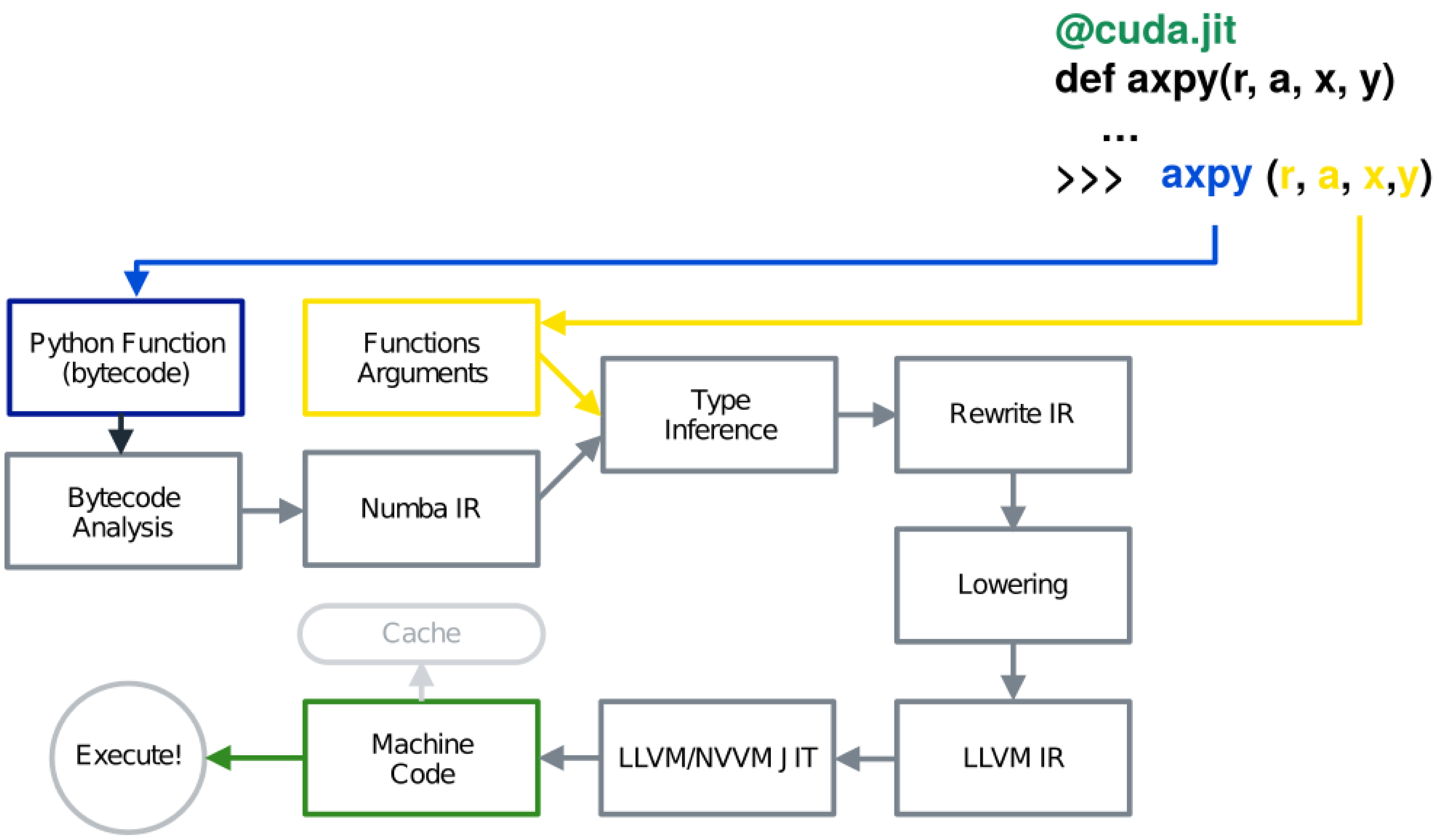
- A Python-to-PTX compiler that uses NVVM
  - (@cuda.jit)
- A Python wrapper to the driver API
  - cuInit, cuCtxCreate, and many more…
  - Transparent for most use cases
- A NumPy-like array library for CUDA GPUs
  - Device Arrays
  - cf. CuPy

ARCHITECTURE OVERVIEW

LLVM: A compiler infrastructure



COMPILATION PIPELINE



DISPATCH PROCESS

Calling a `@jit` function:

1. Lookup types of arguments
2. Do any compiled versions match the types of these arguments?
   a) Yes: retrieve the compiled code from the cache
   b) No: compile a new specialisation
3. Marshal arguments to native values
4. Call the native code function
5. Marshal the native return value to a Python value

CUDA DISPATCH PROCESS

Extra work for calling a `@cuda.jit` function:

1. Compilation: use NVVM for LLVM IR -> PTX
2. Linking: create a module (cubin) with driver API
3. Loading: load module with driver API
4. Data transfer: move data in host memory to GPU
5. Kernel launch: more marshalling, call through driver API

TYPE INFERENCE

- No typing in Python source
- Numba propagates type information:
  - Starts with kernel arguments
  - Follows data flow
  - For functions: uses a mapping of input types -> output types

```python
# a:= float32, b:= float64
@cuda.jit
def f(a, b):
    # c:= float64
    c = a + b
    # return := float64
    return c
```

DEVICE ARRAY LIBRARY

NUMPY

- "NumPy is the fundamental package for scientific computing with Python."
  - N-dimensional array objects
  - Many functions operating on array objects
  - Interfaces for third-party libraries to implement
- Every Python numerical library built on NumPy
  - Or speaks its interfaces

```
import numpy as np

x = np.arange(10) # Array of integers from 0 to 9
x = x * 2         # Elementwise multiplication
np.cos(x)         # One of many functions for
                  # operating on arrays
```

UNSUPPORTED NUMPY FUNCTIONS (CUDA TARGET)

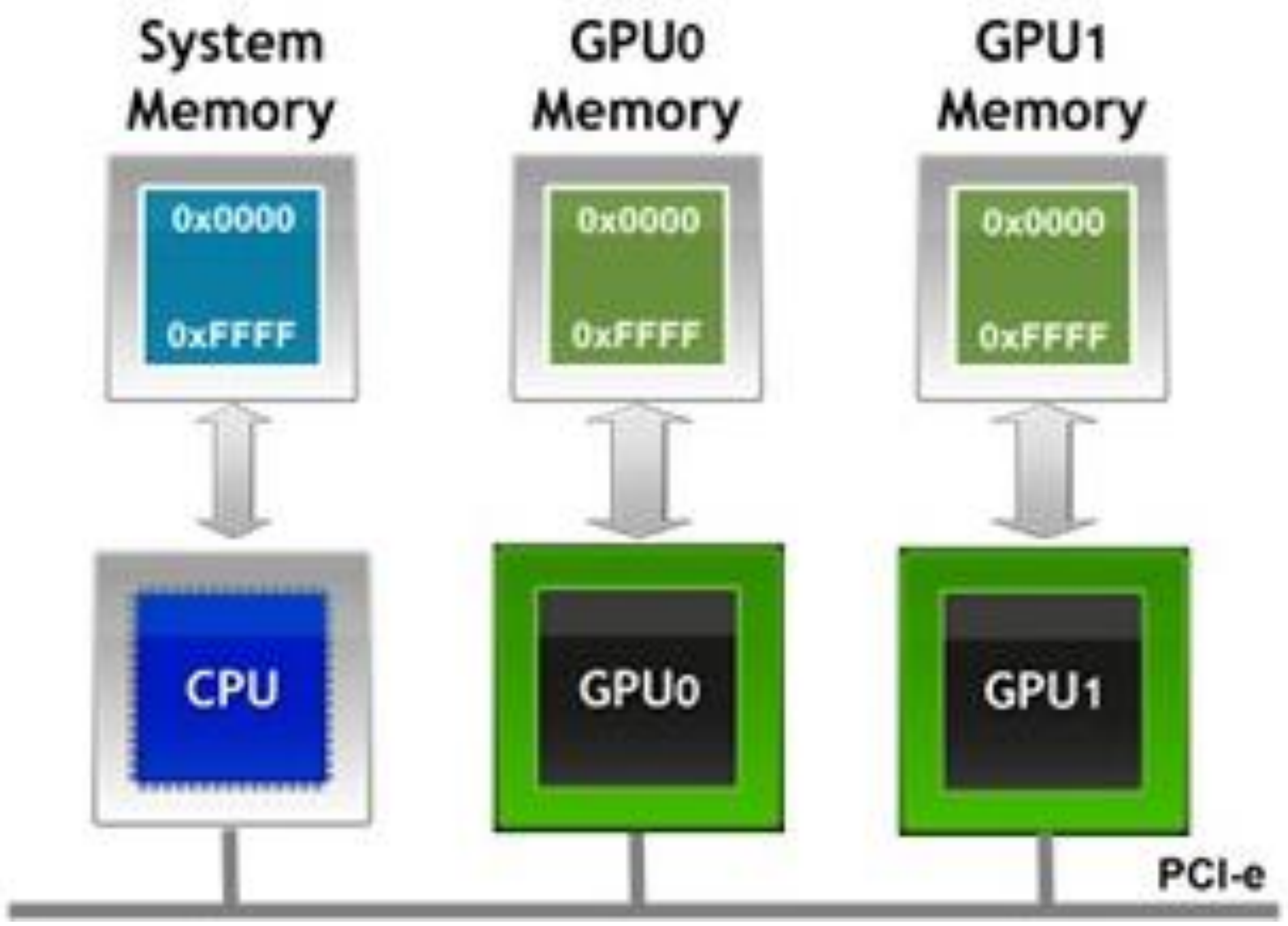
- Array creation
- NumPy Functions returning a new array
  - Most functions that accept an array, e.g. `np.cos(array)`
- Array methods (e.g. `x.mean()`)
- Aiming to for greater coverage in 0.56 (RC by May/June 2022)

MEMORY MANAGEMENT

```
import numpy as np
from numba import cuda

@cuda.jit
def add(r, x, y):
    i = cuda.grid(1)
    if i < len(r):
        r[i] = x[i] + y[i]

# Create arrays on host
x = np.arange(10)
y = x * 2
r = np.zeros_like(x)

# Transfers to and from host memory implied
add[config](r, x, y)
```

MANUAL MEMORY MANAGEMENT

```
d_x = cuda.to_device(x)
d_y = cuda.to_device(y)
d_r = cuda.device_array_like(x)

# No transfer implied
add[config](d_r, d_x, d_y)

# Bring data back when needed
r = d_r.copy_to_host()
```



- Creating device arrays:

```
# 1024 x 2048 matrix of float32
arr = cuda.device_array((1024, 2048), dtype=np.float32)
```

FREEING MEMORY

- No direct `cudaFree` / `cuMemFree` equivalent
- Python is garbage-collected
- When array object GC'd, Numba finalizer releases memory (eventually)

```
# Remove reference from current namespace:
del d_r
```

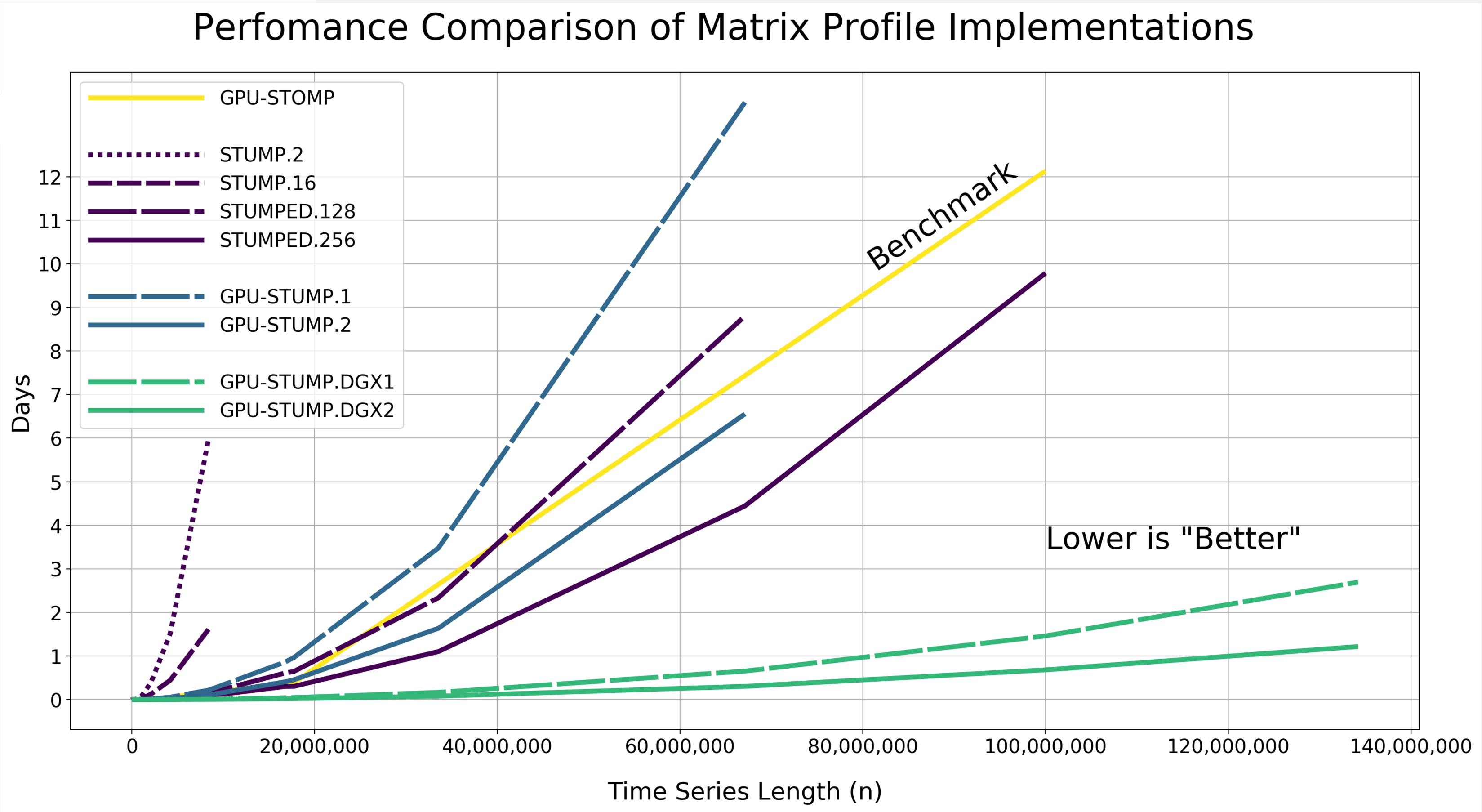
- [Deallocation Behaviour]
- [Numba Memory Management documentation]

NOTABLE EXAMPLES

- Applications using Numba
- In lieu of benchmark presentation

[30]:

```python
def f(row):
    x = row['a']
    y = row['b']
    if x + y > 3:
        return cudf.NA
    else:
        return x + y

df = cudf.DataFrame({
    'a': [1, 2, 3],
    'b': [2, 1, 1]
})
df
```



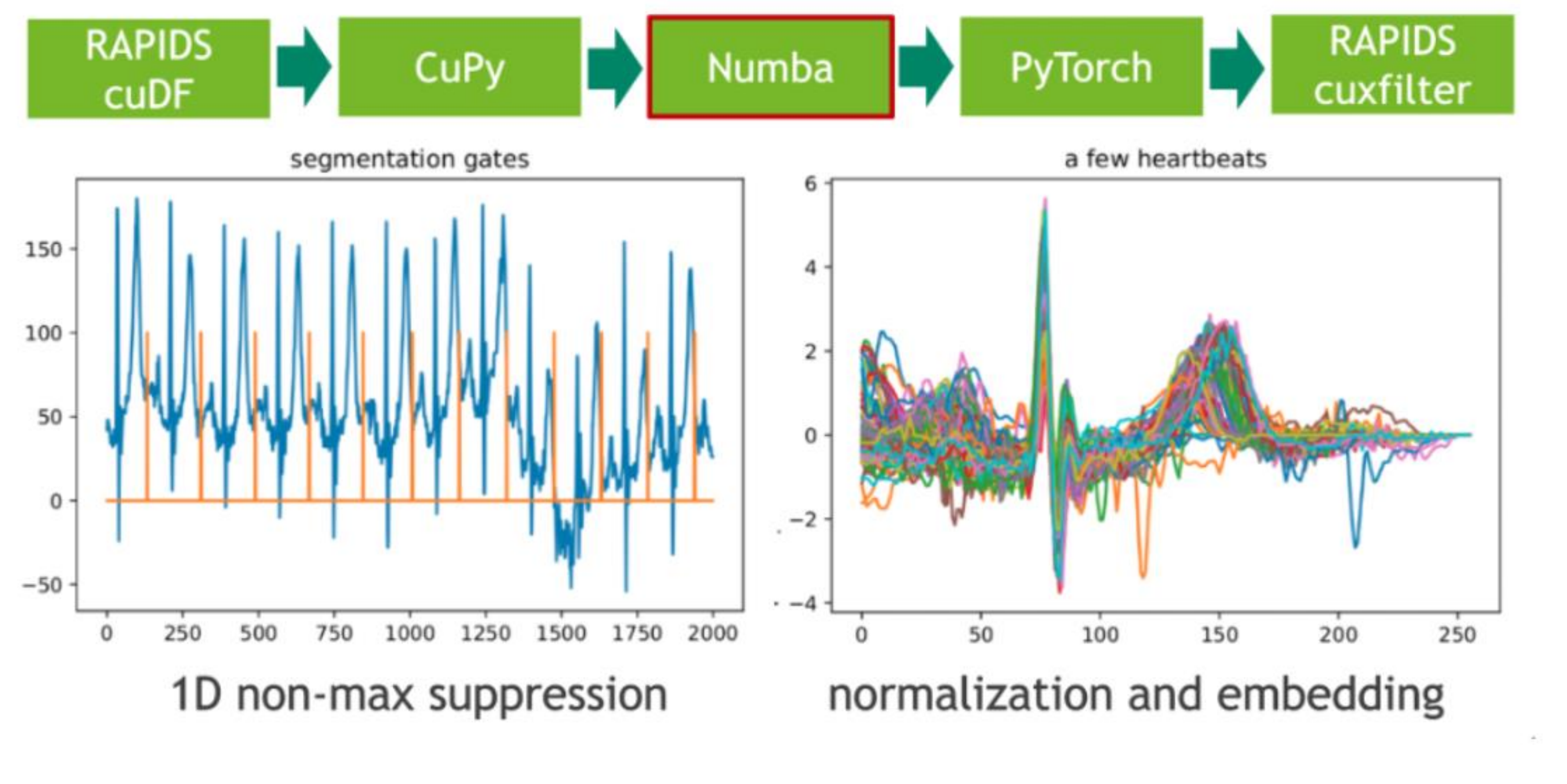
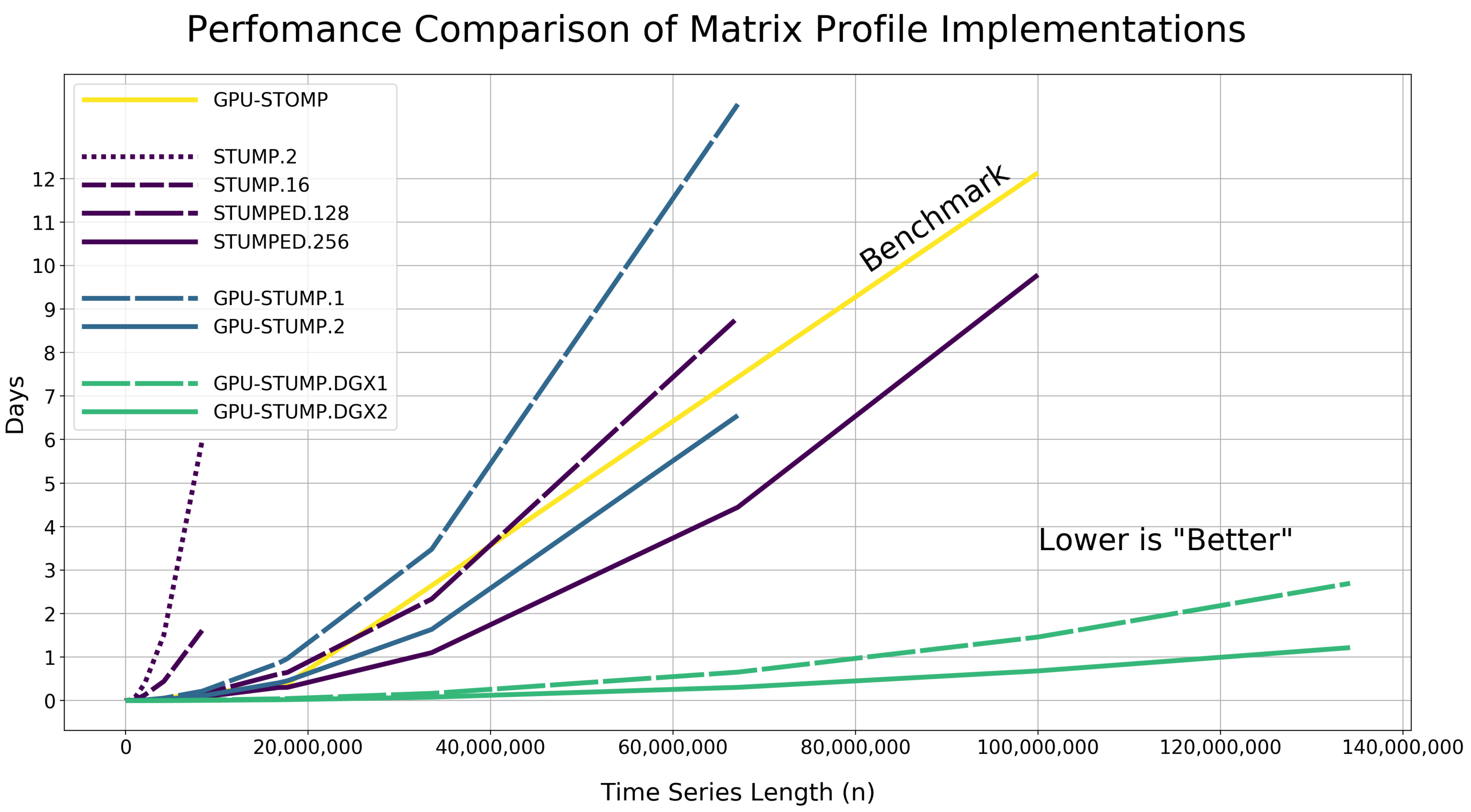


Figure 3: 1D non-maximum suppression and embedding of heartbeats using Numba JIT.

STUMPY

https://github.com/TDAmeritrade/stumpy

- "STUMPY is a powerful and scalable library that efficiently computes something called the matrix profile, which can be used for a variety of time series data mining tasks"



MACHINE LEARNING FRAMEWORKS INTEROPERABILITY

Christian Hundt and Miguel Martinez, NVIDIA Developer Blog, September 2021

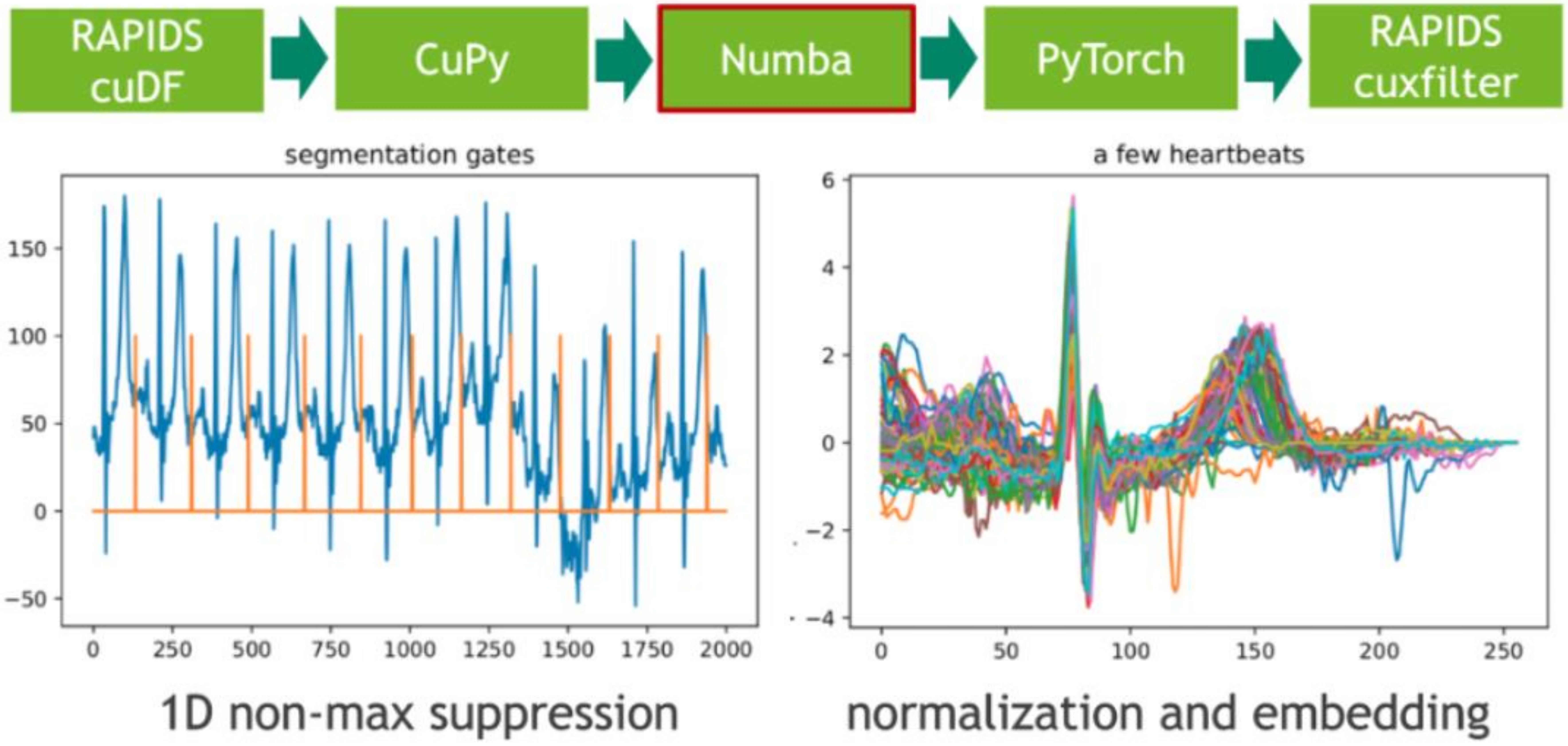


Figure 3: 1D non-maximum suppression and embedding of heartbeats using Numba JIT.

Image from: [Machine Learning Frameworks Interoperability, Part 3: Zero-Copy in Action using an E2E Pipeline]()

UDF EXAMPLE: RAPIDS UDFS

- cuDF: API of Pandas, but users require expression of custom functions from Python
- UDFs: Row-at-time, window-at-a-time, group-at-a-time, custom

[30]:
```python
def f(row):
    x = row['a']
    y = row['b']
    if x + y > 3:
        return cudf.NA
    else:
        return x + y

df = cudf.DataFrame({
    'a': [1, 2, 3],
    'b': [2, 1, 1]
})
df
```

[30]:

	a	b
0	1	2
1	2	1
2	3	1

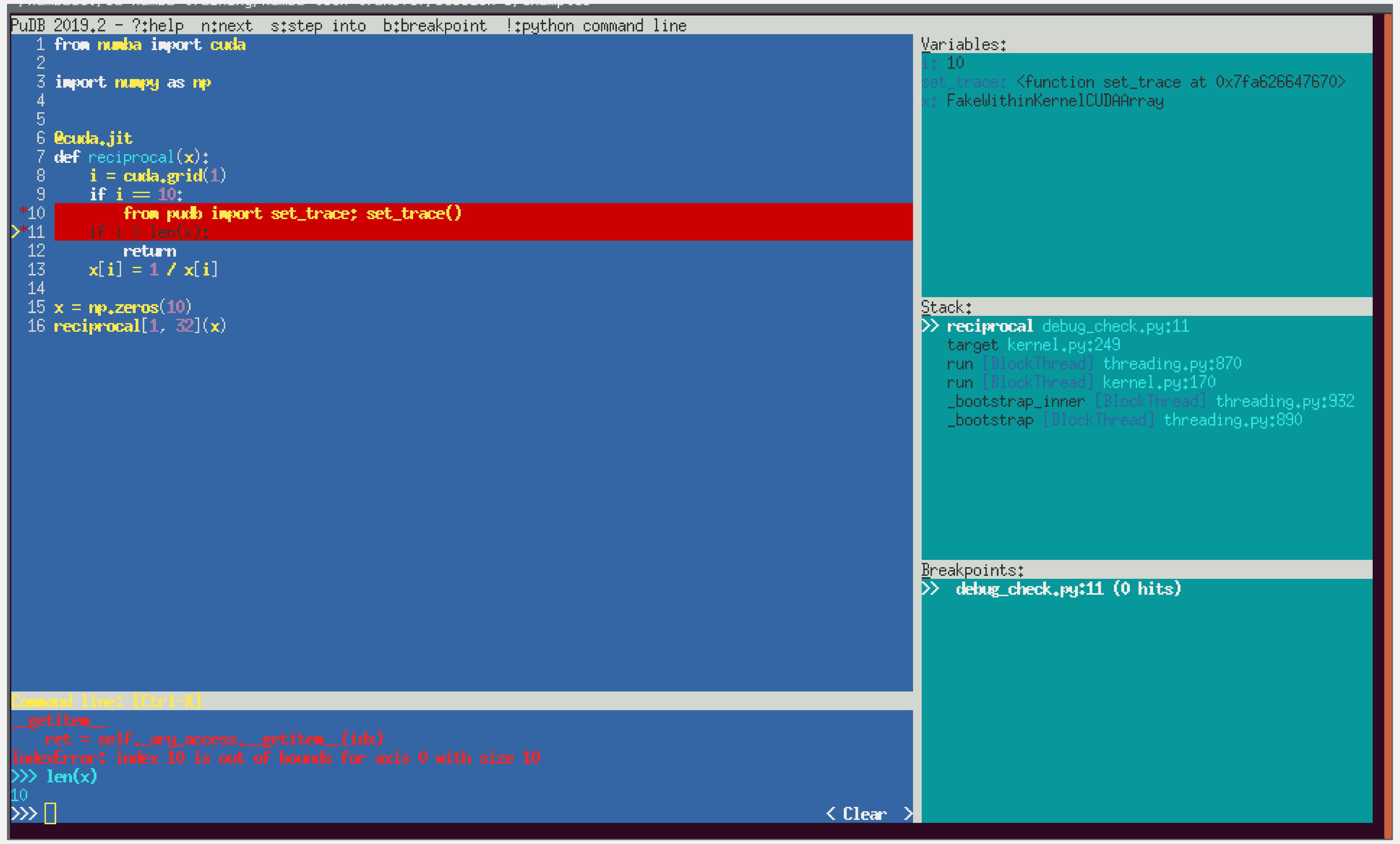
[31]:
```python
df.apply(f, axis=1)
```

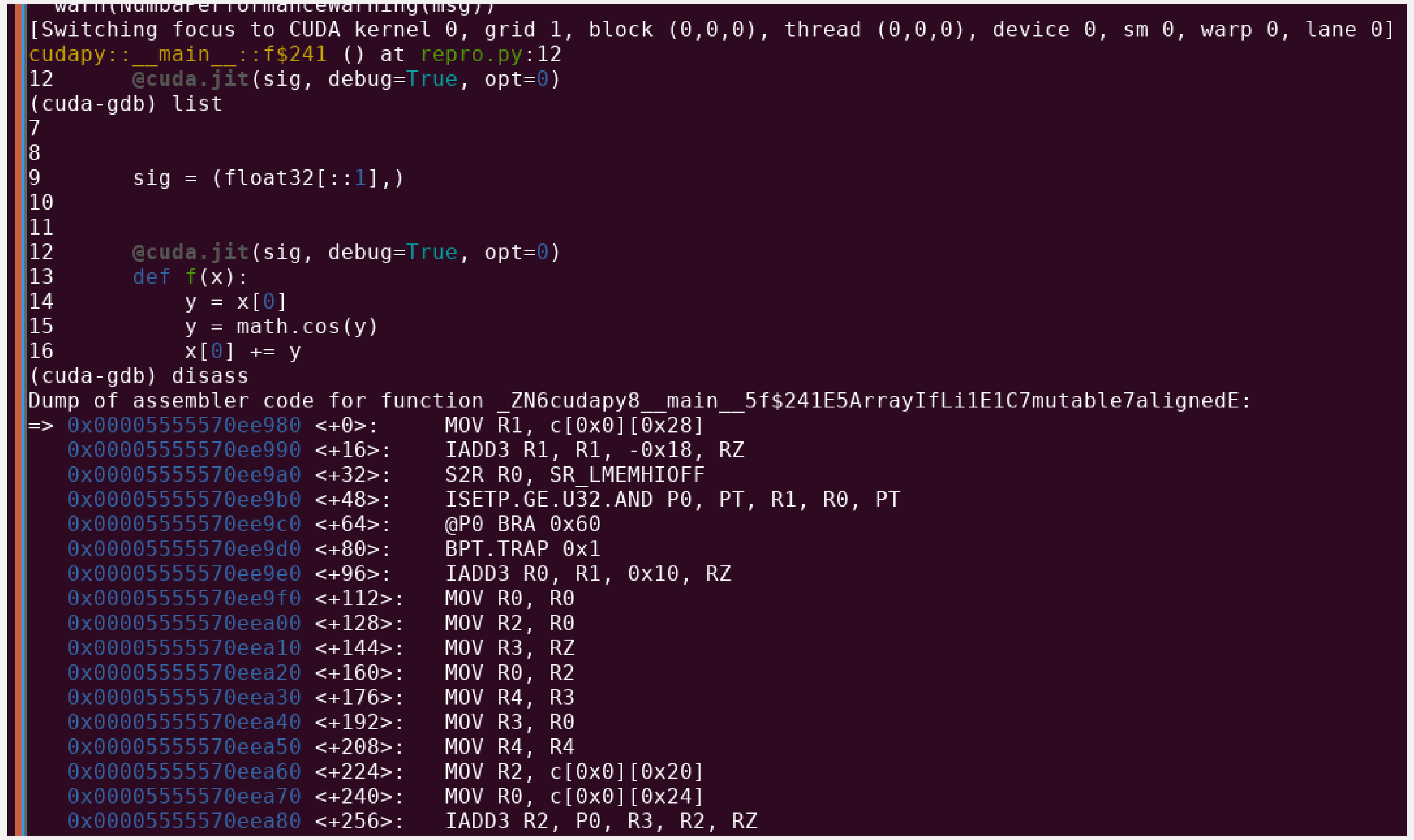
[31]:
```
0       3
1       3
2    <NA>
dtype: int64
```

Example from: RAPIDS Guide to UDFs

TOOL SUPPORT

- Compute Sanitizer
- CUDA-GDB
- CUDA Simulator
- Nsight Compute





GENERATING DEBUG INFO

- Pass `debug=True` to the `@cuda.jit` decorator
  - Adds checks for raised exceptions
  - Adds line number information

- Helps interpret `compute-sanitizer` output

OFF-BY-ONE ERROR EXAMPLE

```
@cuda.jit(debug=True)
def reciprocal(x):
    i = cuda.grid(1)
    # Off-by-one - accesses beyond end of array
    if i > x.shape[0]:
        return
    x[i] = 1 / x[i]

x = np.zeros(10)
reciprocal[1, 32](x)
```

- Without `debug=True`: overwrites memory, no exception
- With `debug=True`: exception raised

COMPUTE-SANITIZER

```python
@cuda.jit(debug=True)
def add_1(x):
    i = cuda.grid(1)
    # Off-by-one - accesses beyond end of array
    if i > x.shape[0]:
        return
    x[i] += 1

x = np.zeros(10)
add_1[1, 32](x)
```

```
Invalid __global__ read of size 8
    at 0x00000360 in .../examples/debug_memcheck.py:11:
        cudapy::__main__::add_1$241(
            Array<double, int=1, C, mutable, aligned>)
    by thread (10,0,0) in block (0,0,0)
    Address 0x7f4f75800050 is out of bounds
    Device Frame: .../examples/debug_memcheck.py:11:
        cudapy::__main__::add_1$241(
            Array<double, int=1, C, mutable, aligned>)
            (cudapy::__main__::add_1$241(
                Array<double, int=1, C, mutable, aligned>)
            : 0x360)
```

DIVISION BY ZERO EXCEPTION

```
Traceback (most recent call last):
  File "blackscholes_cuda.py", line 126, in <module>
    main(*sys.argv[1:])
  File "blackscholes_cuda.py", line 104, in main
    black_scholes_cuda[griddim, blockdim, stream](
  File ".../numba/cuda/compiler.py", line 817, in __call__
    cfg(*args)
  File ".../numba/cuda/compiler.py", line 576, in __call__
    self._kernel_call(args=args,
  File ".../numba/cuda/compiler.py", line 688, in _kernel_call
    raise exccls(*exc_args)

ZeroDivisionError: tid=[0, 0, 256] ctaid=[0, 0, 3906]:
                   division by zero
```

CUDA-GDB

- Functionality available:
  - cuda-memcheck: set cuda memcheck on to break on memcheck error.
  - Set breakpoints on kernels
  - Step by instruction
  - View corresponding source
- Limitation:
  - Backtraces don’t work through host code
  - Stepping by source line limited

DEBUGGING - CUDA-GDB

```
  warn(NumbaPerformanceWarning(msg))
[Switching focus to CUDA kernel 0, grid 1, block (0,0,0), thread (0,0,0), device 0, sm 0, warp 0, lane 0]
cudapy::__main__::f$241 () at repro.py:12
12      @cuda.jit(sig, debug=True, opt=0)
(cuda-gdb) list
7
8
9       sig = (float32[::1],)
10
11
12      @cuda.jit(sig, debug=True, opt=0)
13      def f(x):
14          y = x[0]
15          y = math.cos(y)
16          x[0] += y
(cuda-gdb) disass
Dump of assembler code for function _ZN6cudapy8__main__5f$241E5ArrayIfLi1E1C7mutable7alignedE:
=> 0x00005555570ee980 <+0>:     MOV R1, c[0x0][0x28]
   0x00005555570ee990 <+16>:    IADD3 R1, R1, -0x18, RZ
   0x00005555570ee9a0 <+32>:    S2R R0, SR_LMEMHIOFF
   0x00005555570ee9b0 <+48>:    ISETP.GE.U32.AND P0, PT, R1, R0, PT
   0x00005555570ee9c0 <+64>:    @P0 BRA 0x60
   0x00005555570ee9d0 <+80>:    BPT.TRAP 0x1
   0x00005555570ee9e0 <+96>:    IADD3 R0, R1, 0x10, RZ
   0x00005555570ee9f0 <+112>:   MOV R0, R0
   0x00005555570eea00 <+128>:   MOV R2, R0
   0x00005555570eea10 <+144>:   MOV R3, RZ
   0x00005555570eea20 <+160>:   MOV R0, R2
   0x00005555570eea30 <+176>:   MOV R4, R3
   0x00005555570eea40 <+192>:   MOV R3, R0
   0x00005555570eea50 <+208>:   MOV R4, R4
   0x00005555570eea60 <+224>:   MOV R2, c[0x0][0x20]
   0x00005555570eea70 <+240>:   MOV R0, c[0x0][0x24]
   0x00005555570eea80 <+256>:   IADD3 R2, P0, R3, R2, RZ
```

EXAMPLE CUDA-GDB SESSION

```
$ cuda-gdb --args python debug_memcheck.py
(cuda-gdb) set cuda memcheck on
(cuda-gdb) run
Starting program: .../envs/numba/bin/python debug_memcheck.py
Illegal access to address (@global)0x7fffb7800050 detected.
Thread 1 "python" received signal CUDA_EXCEPTION_1,
   Lane Illegal Address.
[Switching focus to CUDA block (0,0,0), thread (10,0,0), ...]
0x00005555571a47e0 in cudapy::__main__::add_1$241(Array<...>) ()

(cuda-gdb) bt
#0  0x00005555571a47e0 in cudapy::__main__::add_1$241(Array<...>) ()
#1  0x00005555571a47e0 in cudapy::__main__::add_1$241(Array<...>)
                              <<<(1,1,1),(32,1,1)>>> ()
```

EXAMPLE CUDA-GDB SESSION (2)

```
(cuda-gdb) disas
Dump of assembler code for function _ZN6cudapy8__main__9add_...:
=> 0x00005555574c6980 <+0>:  MOV R1, c[0x0][0x28]
   0x00005555574c6990 <+16>: MOV R2, 0x180
   0x00005555574c69a0 <+32>: LDC.64 R2, c[0x0][R2]
   ...

(cuda-gdb) stepi
   0x00005555574c6990 in cudapy::__main__::add_1$241(Array<...>)
                              <<<(1,1,1),(32,1,1)>>> ()

(cuda-gdb) stepi
   0x00005555574c69a0 in cudapy::__main__::add_1$241(Array<...>)
                              <<<(1,1,1),(32,1,1)>>> ()

(cuda-gdb) disas
Dump of assembler code for function _ZN6cudapy8__main__9add_...:
   0x00005555574c6980 <+0>:  MOV R1, c[0x0][0x28]
   0x00005555574c6990 <+16>: MOV R2, 0x180
=> 0x00005555574c69a0 <+32>: LDC.64 R2, c[0x0][R2]
```

DEBUGGING - CUDA SIMULATOR

```
PuDB 2019.2 - ?:help  n:next  s:step into  b:breakpoint  !:python command line
  1 from numba import cuda
  2
  3 import numpy as np
  4
  5
  6 @cuda.jit
  7 def reciprocal(x):
  8     i = cuda.grid(1)
  9     if i == 10:
 *10         from pudb import set_trace; set_trace()
>*11     if i > len(x):
 12         return
 13     x[i] = 1 / x[i]
 14
 15 x = np.zeros(10)
 16 reciprocal[1, 32](x)
```

```
Variables:
i: 10
set_trace: <function set_trace at 0x7fa626647670>
x: FakeWithinKernelCUDAArray
```

```
Stack:
>> reciprocal debug_check.py:11
   target kernel.py:249
   run [BlockThread] threading.py:870
   run [BlockThread] kernel.py:170
   _bootstrap_inner [BlockThread] threading.py:932
   _bootstrap [BlockThread] threading.py:890
```

```
Breakpoints:
>>  debug_check.py:11 (0 hits)
```

```
Command line: [Ctrl-X]
__getitem__
    ret = self._ary_access.__getitem__(idx)
IndexError: index 10 is out of bounds for axis 0 with size 10
>>> len(x)
10
>>>
                                                    < Clear >
```

CUDA SIMULATOR

- Features:
  - Emulates CUDA execution model in Python
  - Regular Python exceptions occur (e.g. OOB access)
  - Break in kernels with Python debugger
  - Step through kernels, view all variables, print out, etc.
- Limitations:
  - Slow!
  - Only one GPU simulated
  - Threaded access / kernel calls not supported
  - Most driver API unimplemented

USING THE CUDA SIMULATOR

- Getting started:
  - Set `NUMBA_ENABLE_CUDASIM=1` in your environment
  - May need to run with small data set
  - Or, strip down to minimal reproducer

Then:
  - Run and see if exceptions occur, and/or
  - Use e.g. `from pudb import set_trace; set_trace()`
- Starting under debugger doesn't work well:
  - OS threads implement CUDA threads - confuses debugger

CUDA SIMULATOR DEBUG EXAMPLE

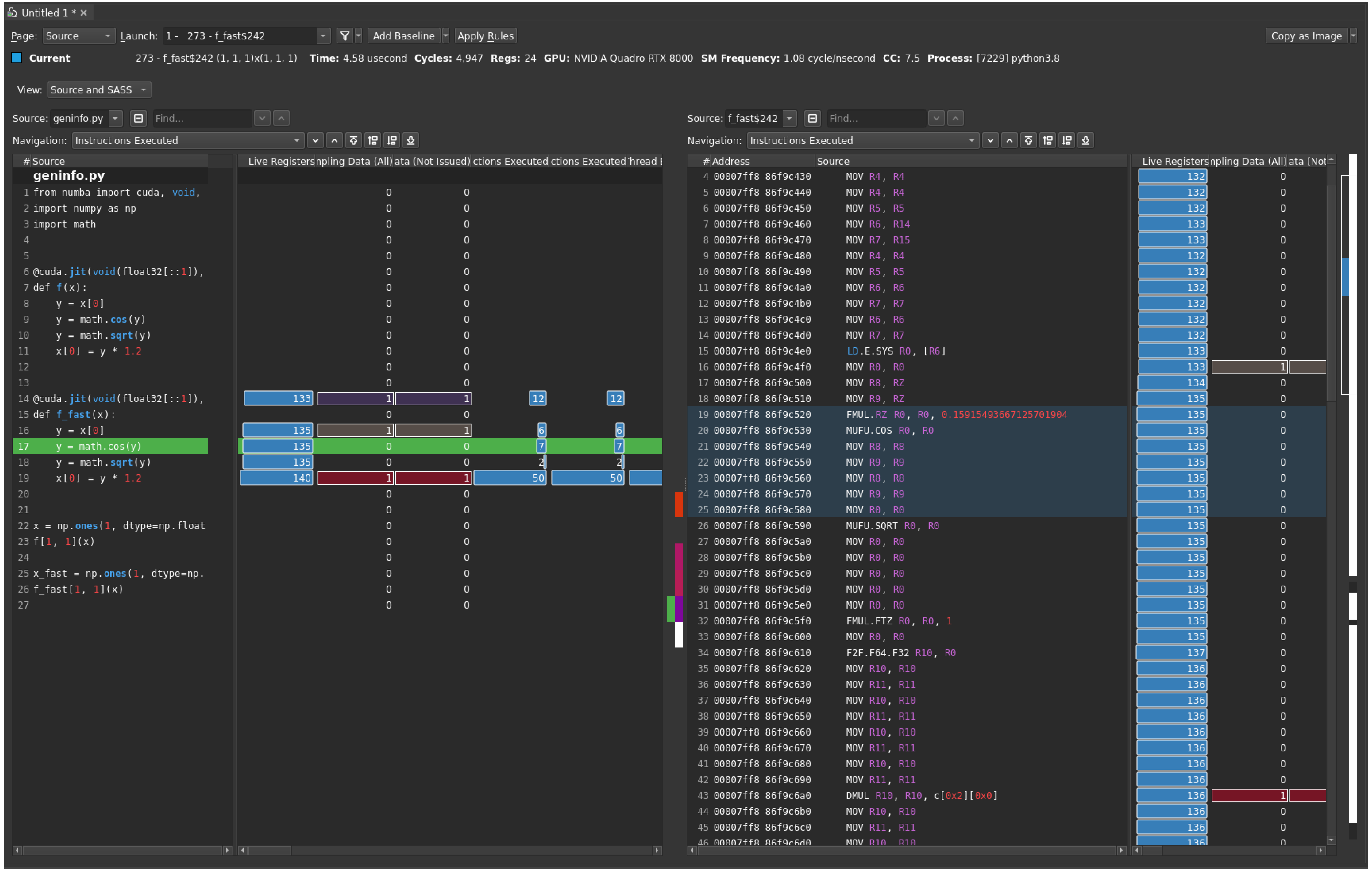
- Run with simulator:

```
$ NUMBA_ENABLE_CUDASIM=1 python debug_check.py
...
  File "debug_check.py", line 11, in reciprocal
    x[i] = 1 / x[i]
...
IndexError: tid=[10, 0, 0] ctaid=[0, 0, 0]:
    index 10 is out of bounds for axis 0 with size 10
```

- Add debug break to kernel:

```
if i == 10:
    from pudb import set_trace; set_trace()
```

NSIGHT COMPUTE



SUMMARY / GETTING STARTED WITH NUMBA

- Numba is a JIT compiler focused on compiling type-specialised versions of numerically-focused code.
  - CUDA Python enables writing CUDA kernels
  - Integrates with CUDA tooling, and other CUDA Python libraries

- Trying it out:
  - `conda install numba cudatoolkit`
  - `pip install numba`
  - Google Colab: https://colab.research.google.com/
    - “**Runtime**”, “**Change Runtime Type**”, set “**Hardware Accelerator**” to “**GPU**”
  - Notebook / repo / slides for this talk: https://github.com/gmarkall/cuda-community-talk

- Full NVIDIA CUDA Numba tutorial: https://github.com/numba/nvidia-cuda-tutorial
- GTC 2022 Talk: Enabling Python User-Defined Functions in Accelerated Applications with Numba:
  - https://www.nvidia.com/gtc/

EXTRA MATERIAL

CODING UTILITIES

- Random Number Generator
- Reduction Generator
- Forall generator

RANDOM GENERATOR USAGE

```
@cuda.jit
def compute_pi(rng_states, iterations, out):
    """Find the maximum value in values and store in result[0]""“
    tid = cuda.grid(1)

    # Compute pi by drawing random (x, y) points and finding what
    # fraction lie inside a unit circle
    inside = 0
    for i in range(iterations):
        x = xoroshiro128p_uniform_float32(rng_states, tid)
        y = xoroshiro128p_uniform_float32(rng_states, tid)
        if x**2 + y**2 <= 1.0:
            inside += 1

    out[tid] = 4.0 * inside / iterations

state_size = nblocks * nthreads
rng_states = create_xoroshiro128p_states(state_size, seed=1)
compute_pi[nblocks, nthreads](rng_states, 10000, out)
```

REDUCTION GENERATOR USAGE

```
@cuda.reduce
def sum_reduce(a, b):
    return a + b

A = (np.arange(1234, dtype=np.float64)) + 1

# Transfers result to host
sum_reduce(A)

# Keep result on device
r = cuda.device_array(1, dtype=np.float64)
sum_reduce(A, res=r)

# Use device array, keep result on device
A_d = cuda.to_device(A)
sum_reduce(A_d, res=r)
```

FORALL USAGE

(Example borrowed from cuDF)

```
@cuda.jit
def gpu_round(in_col, out_col, decimal):
    i = cuda.grid(1)
    f = 10 ** decimal

    if i < in_col.size:
        ret = in_col[i] * f
        ret = rint(ret)
        tmp = ret / f
        out_col[i] = tmp

# in_data and out_data are device arrays
nelems = len(in_data)
# Round all elements to 3 d.p.
gpu_round.forall(nelems)(in_data, out_data, 3)
```

VECTORIZE

- UFuncs: operate element-by-element on arrays
- Supports broadcasting, reduction, accumulation, etc.

```
@vectorize
def rel_diff(x, y):
    return 2 * (x - y) / (x + y)
```

Call:

```
a = np.arange(1000, dtype = float32)
b = a * 2 + 1
rel_diff(a, b)
```

CUDA VECTORIZE

- Add target='cuda' to generate CUDA implementation
- Note: CUDA target needs type specification

```
@vectorize([float32(float32, float32)], target='cuda')
def rel_diff(x, y):
    return 2 * (x - y) / (x + y)
```

Call (no change):

```
a = np.arange(1000, dtype = float32)
b = a * 2 + 1
rel_diff(a, b)
```

CUDA PYTHON ECOSYSTEM

- CUDA Array Interface
- CUDA Python bindings

CUDA ARRAY INTERFACE

- Zero-copy interface for array data between CUDA Python libraries
  - (c.f. NumPy Array Interface `arr.__array_interface__`)

```
In [2]: x = cuda.device_array((100,2))

In [3]: x.__cuda_array_interface__
Out[3]:
{'shape': (100, 2),
 'strides': None,
 'data': (139773052190720, False),
 'typestr': '<f8',
 'stream': None,
 'version': 3}
```

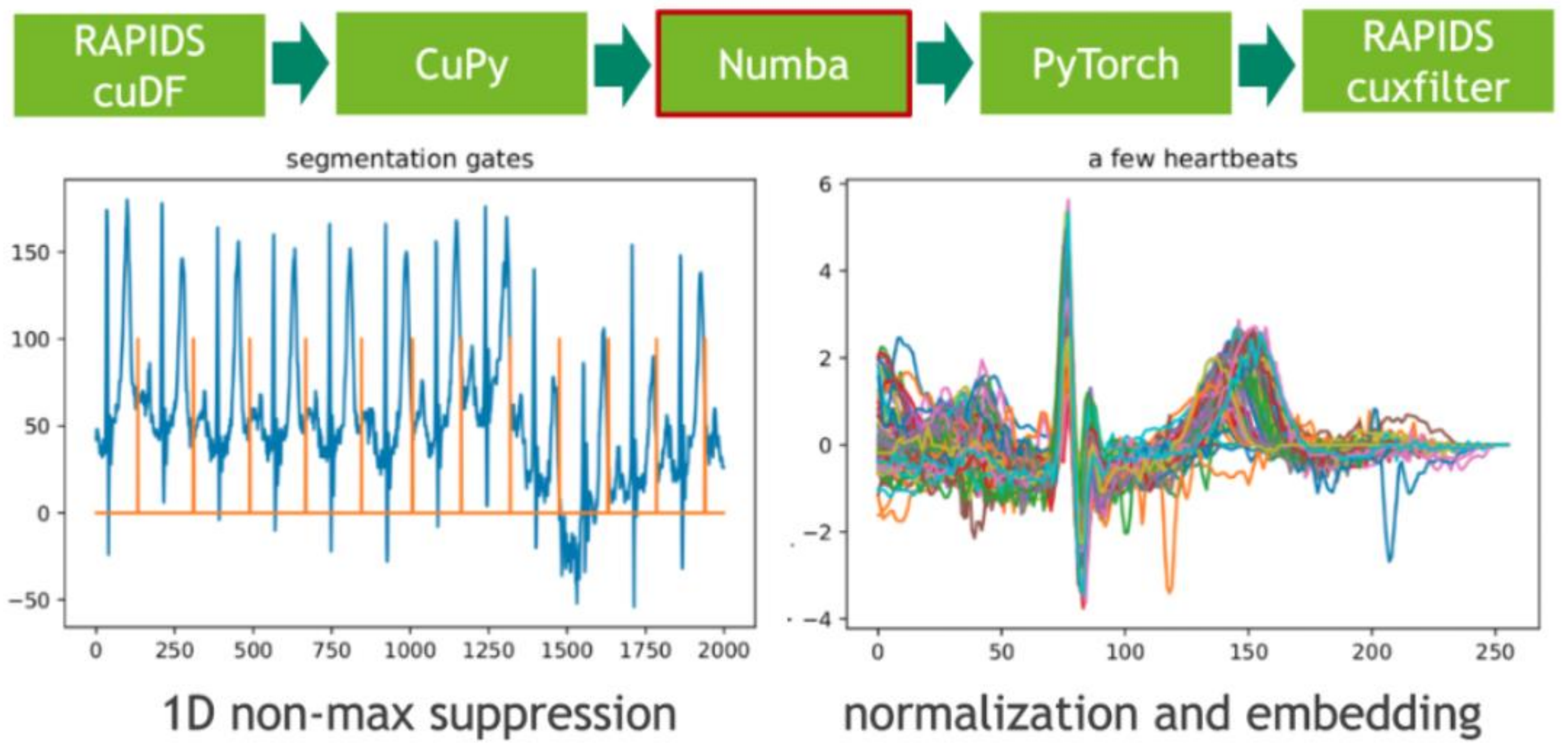


Figure 3: 1D non-maximum suppression and embedding of heartbeats using Numba JIT.

Image from: [Machine Learning Frameworks Interoperability, Part 3: Zero-Copy in Action using an E2E Pipeline]

CUDA ARRAY INTERFACE

- A standard for different libraries to exchange / use each others' on-device data.
- Objects implement `__cuda_array_interface__`. Returns a dict:
    - pointer, shape, strides, etc.
- Implemented by:
    - Numba, CuPy, PyTorch, PyArrow, mpi4py, ArrayViews, JAX
    - RAPIDS: cuDF, cuML, cuSignal, RMM

EXAMPLE - NUMBA ON CUPY DATA

Calling a Numba Kernel on a CuPy array:

```
import cupy
from numba import cuda

@cuda.jit
def add(x, y, out):
    start = cuda.grid(1)
    stride = cuda.gridsize(1)
    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]

a = cupy.arange(10)
b = a * 2
out = cupy.zeros_like(a)

add[1, 32](a, b, out)
```

EXAMPLE - CUPY ON NUMBA DATA

Zero-copy conversion using CUDA Array Interface:

```
import numpy
import numba
import cupy

# type: numpy.ndarray
x = numpy.arange(10)

# type: numba.cuda.cudadrv.devicearray.DeviceNDArray
x_numba = numba.cuda.to_device(x)

# type: cupy.ndarray - a view of x_numba's dat
x_cupy = cupy.asarray(x_numba)
```

LIBRARY MEMORY MANAGEMENT

Problem: Combining libraries oversubscribes GPU memory

- Numba: Deferred Deallocation
- RAPIDS: Rapids Memory Manager device pool
- CuPy: Device and Pinned Memory Pools
- PyTorch: Caching memory allocator

USING ONE STRATEGY

- Solution: Configure all libraries to use one implementation
- Example:

```
import cupy
import rmm

cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)

# Allocated using RMM
a = cupy.arange(5)
```

USING RMM WITH NUMBA

```
from numba import cuda
import rmm

cuda.set_memory_manager(rmm.RMMNumbaManager)

# Allocated using RMM
a = cuda.device_array(5)
```

- CuPy + Numba using RMM pool:

```
from numba import cuda
import rmm
import cupy

cupy.cuda.set_allocator(rmm.rmm_cupy_allocator)
cuda.set_memory_manager(rmm.RMMNumbaManager)
```

USING NUMBA AS A UDF COMPILER

- User Defined Functions (UDFs)
  - Critical path to performance and productivity

- Numba Interface:
  - Typing
  - Lowering
  - `compile_to_ptx()` function
  - Or, launch with `@cuda.jit`

UDF EXAMPLE 2: PYOPTIX

- PyOptiX: bindings for OptiX host API: https://github.com/keithroe/pyoptix

```
extern "C"
__global__ void __raygen__hello()
{
    uint3 launch_index = optixGetLaunchIndex();
    RayGenData* rtData = (RayGenData*)optixGetSbtDataPointer();
    params.image[launch_index.y * params.image_width + launch_index.x] =
        make_uchar4(
                max( 0.0f, min( 255.0f, rtData->r*255.0f ) ),
                max( 0.0f, min( 255.0f, rtData->g*255.0f ) ),
                max( 0.0f, min( 255.0f, rtData->b*255.0f ) ),
                255
                );
}
```

```
def __raygen__hello():
    launch_index = optix.GetLaunchIndex()
    rtData = RayGenDataStruct(optix.GetSbtDataPointer())

    f0 = types.float32(0.0)
    f255 = types.float32(255.0)

    idx = launch_index.y * params.image_width + launch_index.x

    params.image[idx] = make_uchar4(
            max(f0, min(f255, rtData.r * types.float32(launch_index.x))),
            max(f0, min(f255, rtData.g * types.float32(launch_index.y))),
            max(f0, min(f255, rtData.b * f255)),
            255
    )
```


NVIDIA®