NVIDIA

Accelerated Python with Numba

Graham Markall <gmarkall@nvidia.com> | Excalibur-SysGenX Seminar, 8 May 2024

Aims & Outline

- Audience:
  - Some new users
  - Some experienced users
- An overview of Numba
- Specific topics:
  - CUDA support
  - Dealing with “external data”
  - Understanding Numba’s generated code
  - How to approach NumPy functions
- Extensibility and embedding
- Time for questions / deep dives
  - Happy to chat anytime: gmarkall@nvidia.com

Omissions

- Much in-depth discussion of performance
- Comparative analysis to the other Python compilers in the ecosystem. There are many:
  - CuPy JIT, NVIDIA Warp, Jax, PyTorch Thunder / Lightning, Taichi, LPython, PyPy, etc.
  - https://lpython.org lists over 30!
- Many other items!...

Brief self-introduction

- Software Engineer in RAPIDS at NVIDIA since 2019
- Numba maintainer 2014-16 (Anaconda), 2019- (NVIDIA)
- Mostly supporting cuDF / RAPIDS / NVIDIA library use cases

- Background in compilers / numerical methods / HPC:
  - PhD in SPO group at Imperial with Paul Kelly and David Ham
    - Worked on PyOP2 and Firedrake
    - PDEs, Finite elements, sparse linear solvers,
    - Domain-specific languages for HPC
  - Since then:
    - GCC, Binutils, GDB, LLVM, …

Numba overview

- What is Numba?
- Who uses it?
- How does it work?

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 x86, PPC, ARMv8, CUDA
  - Could target other LLVM-supported CPUs

Numba Users

- Feb ‘22 stats:
  - PyPI: 250,000 / Conda 16,000 downloads per day
  - 📦 48,000 dependent Github repositories
  - 📦 2,000 PyPI packages list Numba dependency
  - ⭐ 7,300 Github stars
  - 🌳 879 Github forks
  - 👁️ 205 Github watchers

Jim Pivarski’s Numba usage stats (Feb 2024):
https://github.com/jpivarski-talks/2024-02-13-numba-usage-stats
Numba dependents on PyPI (Oct 2022):
https://github.com/gmarkall/numba-dependents/blob/main/dependents.txt

- Why use Numba?
  - Comfort Zone: keeping all code as Python code
  - Allows focus on algorithmic development
  - Minimise development time
  - Maintain interoperability

Basic Example

In [87]:
```
@jit
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
```

Python support overview

Supported in functions decorated with `@jit`:

- assignment, indexing, arithmetic
- `if` / `else` / `for` / `while` / `break` / `continue`
- raising exceptions
- `assert`
- calling other compiled functions
- Generators (partial)

Unsupported:

- `try` / `except` / `finally`
- `with`
- (list, set, dict) comprehensions

Basic types:

- `int`, `bool`, `float`, `complex`
- `tuple`, `None`

Built-in functions:

- `abs`, `enumerate`, `len`, `min`, `max`, `print`, `range`, `round`, `zip`

[Documentation on supported Python language features, types, and 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)
  - Random number generation
  - Calculations / reductions / linear algebra / …
  - Many ufuncs (e.g. `np.sin()`)
  - Supported NumPy features documentation

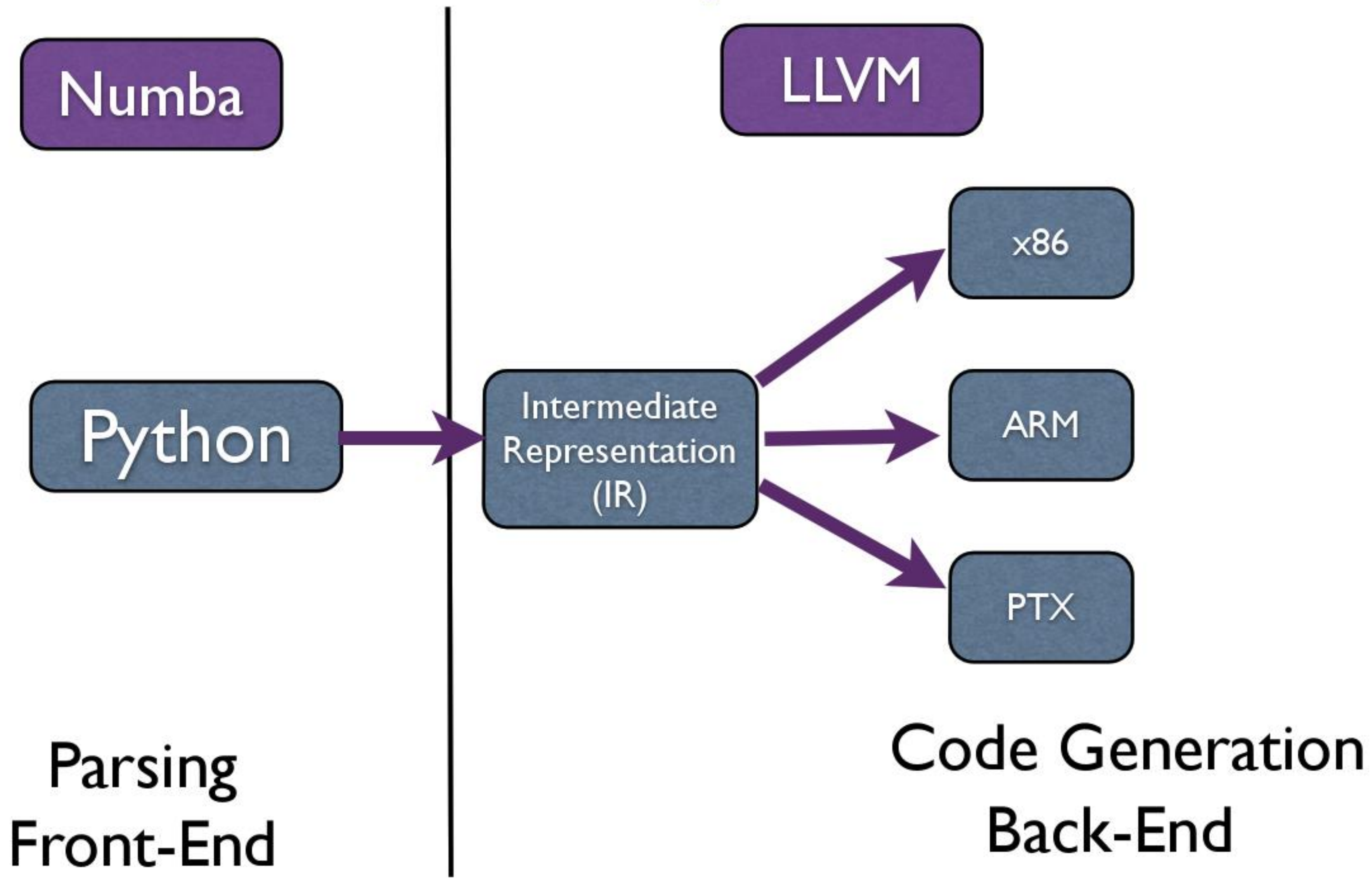
Dispatch process

What happens when you call a @jit function?

1. **Type inspection:** Lookup types of arguments
2. **Caching:** Do any compiled versions match the types of these arguments?
   - ***Yes:*** retrieve the compiled code from the cache
   - ***No:*** compile a new specialisation
3. **Unboxing:** Marshal arguments to native values
4. **Dispatch:** Call the native code function
5. **Boxing:** Marshal the native return value to a Python value

Numba compilation process

- Very high-level overview:



- *Switch to "Life of a Numba Kernel" notebook…*

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

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

CUDA Support

- CUDA Functionality notebook
- Tool support
- Users

CUDA Python with Numba

- ***"CUDA Python"*** introduced in 2012 via NumbaPro (closed-source)
  - Developed by Continuum Analytics, now Anaconda
  - Commercially licensed, but free academic licenses were available
  - Open-sourced as part of Numba in 2014
- Components:
  - The Python JIT compiler
  - Ctypes bindings and Python driver API interface
  - A minimal NumPy-like array library for CUDA
  - The CUDA Array Interface
    - Interop with CuPy, PyCUDA, JAX, etc …

Numba CUDA Users

- In NVIDIA:
  - NeMo https://developer.nvidia.com/nvidia-nemo
    - Used to train MegaTRON on Selene
  - RAPIDS https://rapids.ai/
  - Merlin Recommender systems https://developer.nvidia.com/nvidia-merlin
  - DALI Data Loading Library: https://developer.nvidia.com/dali
  - Triton Inference Server: https://developer.nvidia.com/nvidia-triton-inference-server
    - Model analyzer https://github.com/triton-inference-server/model_analyzer

- Outside NVIDIA:
  - STUMPY Time series analysis: https://github.com/TDAmeritrade/stumpy/ - time series analysis
  - Datashader for rendering large data: https://datashader.org
  - Holoviews for analysis and visualization: https://holoviews.org/
  - Others: https://github.com/gmarkall/numba-cuda-users/

Supported CUDA Kernel Features

- **Basics:**
  - Thread and block indices
  - Shared, local, and const memory
  - Atomic operations (Add, CAS, Inc, etc.)
- **Data types:**
  - Standard scalars and vectors (float3, etc.)
- **Cooperative groups:**
  - Cooperative launch
  - Grid groups and grid sync only
- **Synchronization:**
  - Thread fences
  - Warp intrinsics (syncwarp, shfl_sync, etc.)
- **Integer intrinsics:**
  - Popc, brev, clz, ffs
- **FP16:**
  - Data type and basic arithmetic
- **Others:**
  - nanosleep

```python
@intrinsic
def syncthreads(typingctx):
    '''
    Synchronize all threads in the same thread block.  This function implements
    the same pattern as barriers in traditional multi-threaded programming: this
    function waits until all threads in the block call it, at which point it
    returns control to all its callers.
    '''
    sig = signature(types.none)

    def codegen(context, builder, sig, args):
        fname = 'llvm.nvvm.barrier0'
        lmod = builder.module
        fnty = ir.FunctionType(ir.VoidType(), ())
        sync = cgutils.get_or_insert_function(lmod, fnty, fname)
        builder.call(sync, ())
        return context.get_dummy_value()

    return sig, codegen
```

Supported CUDA host-side features

- Kernel launch
- Streams
- Events
- Synchronization
- Memory allocation:
  - Device memory
  - Page-locked host memory
  - Pinning
  - Managed memory
- Multi-device management
- Legacy IPC API

```python
@require_context
def pinned_array(shape, dtype=np.float_, strides=None, order='C'):
    """pinned_array(shape, dtype=np.float_, strides=None, order='C')

    Allocate an :class:`ndarray <numpy.ndarray>` with a buffer that is pinned
    (pagelocked).  Similar to :func:`np.empty() <numpy.empty>`.
    """
    shape, strides, dtype = prepare_shape_strides_dtype(shape, strides, dtype,
                                                         order)
    bytesize = driver.memory_size_from_info(shape, strides,
                                            dtype.itemsize)
    buffer = current_context().memhostalloc(bytesize)
    return np.ndarray(shape=shape, strides=strides, dtype=dtype, order=order,
                      buffer=buffer)
```

CUDA Example

Grid-strided vector add

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

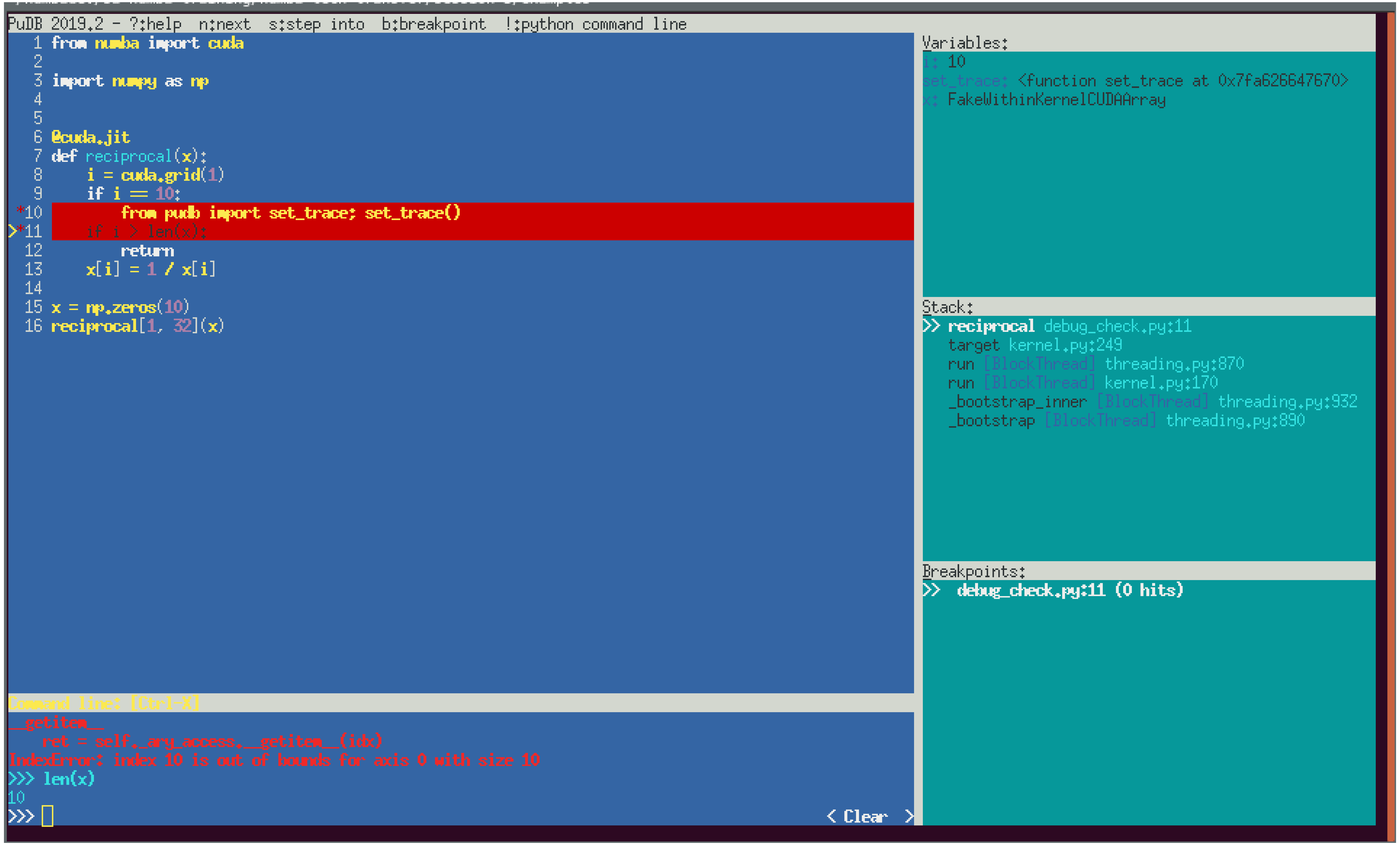
```
from numba import cuda

@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]
```

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

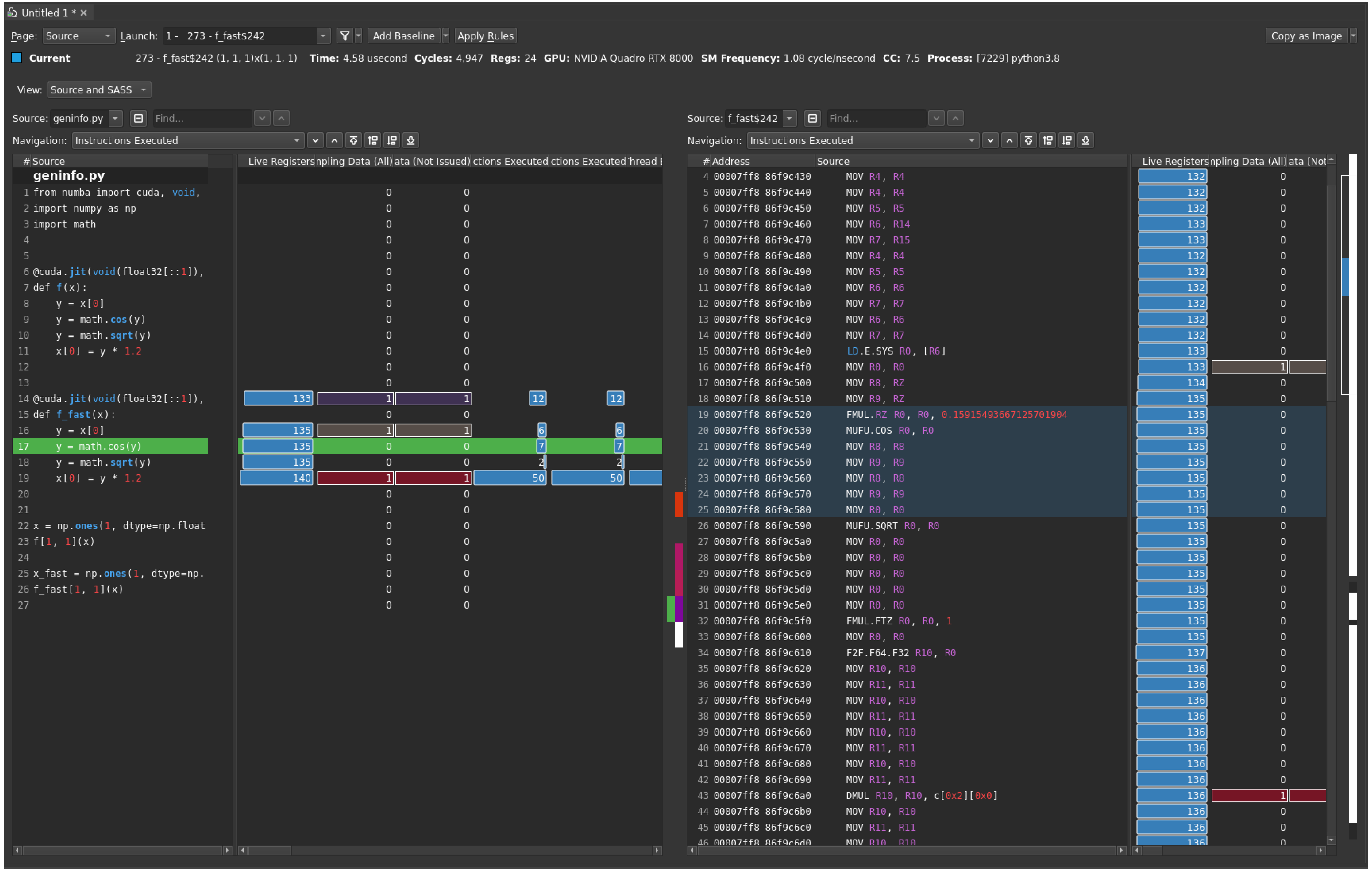
Debugging – cuda-gdb

```
  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
```

Debugging – cuda simulator



Profiling Python kernels - Nsight compute



Detecting memory errors - 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)
```

Development tips

- Inspecting compiled code
- Capturing external data
- Approach to NumPy functions

Extensions

- Extension APIs
- Embedding as a User-Defined Function (UDF) compiler

Use as a User-defined function (UDF) compiler

`numba.cuda.compile_ptx`(*pyfunc, args, debug=False, lineinfo=False, device=False, fastmath=False, cc=None, opt=True*)

Compile a Python function to PTX for a given set of argument types.

Parameters:

- **pyfunc** – The Python function to compile.
- **args** – A tuple of argument types to compile for.
- **debug** (*bool*) – Whether to include debug info in the generated PTX.
- **lineinfo** (*bool*) – Whether to include a line mapping from the generated PTX to the source code. Usually this is used with optimized code (since debug mode would automatically include this), so we want debug info in the LLVM but only the line mapping in the final PTX.
- **device** (*bool*) – Whether to compile a device function. Defaults to `False`, to compile global kernel functions.
- **fastmath** (*bool*) – Whether to enable fast math flags (ftz=1, prec_sqrt=0, prec_div=, and fma=1)
- **cc** (*tuple*) – Compute capability to compile for, as a tuple `(MAJOR, MINOR)`. Defaults to `(5, 3)`.
- **opt** (*bool*) – Enable optimizations. Defaults to `True`.

Returns: (ptx, resty): The PTX code and inferred return type

Return type: tuple

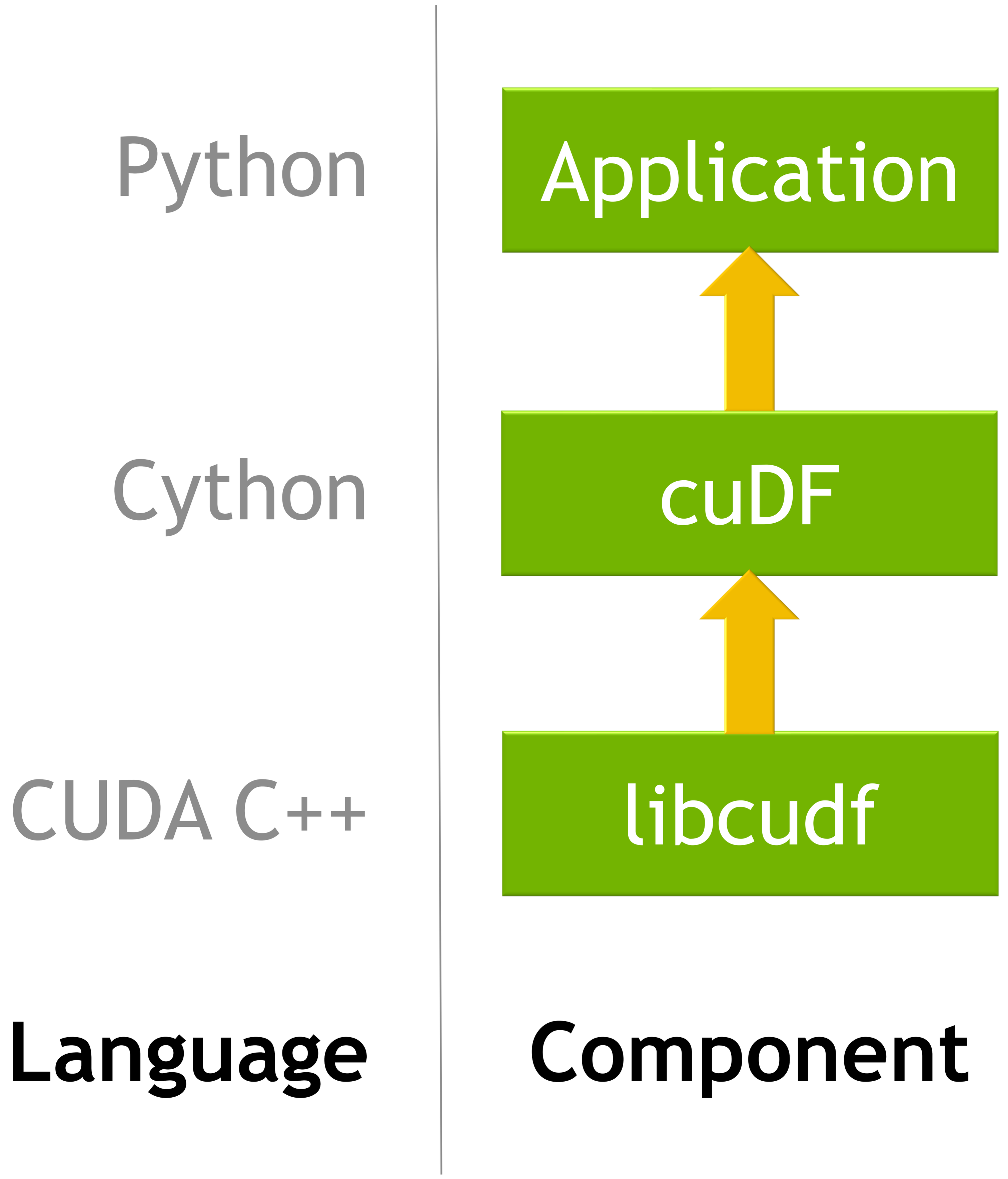
UDF compilation example - cuDF

RAPIDS

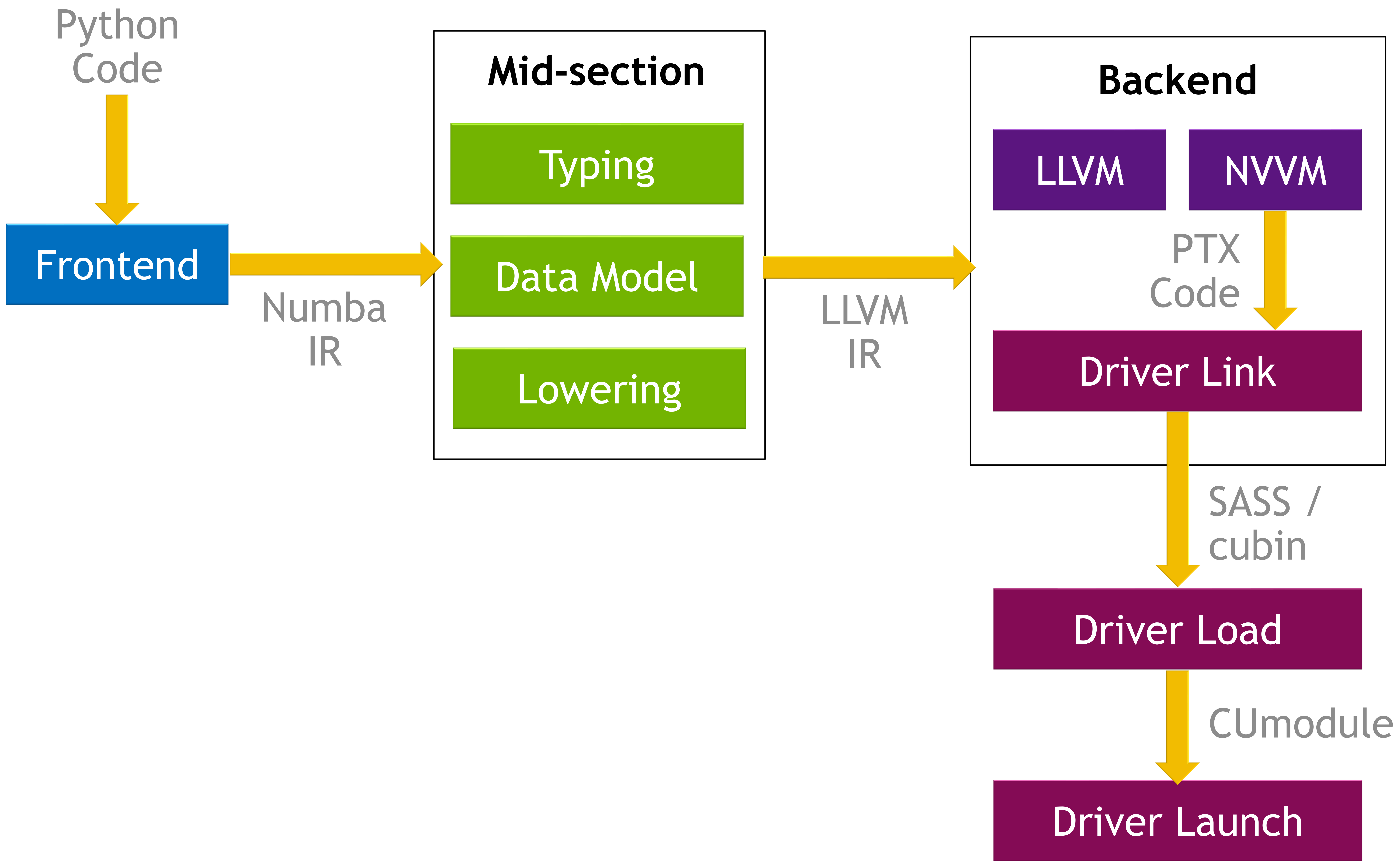
- Pandas for GPUs, or:

"cuDF is a Python GPU DataFrame library (built on the Apache Arrow columnar memory format) for loading, joining, aggregating, filtering, and otherwise manipulating tabular data using a DataFrame style API."

```python
# Defining a series:
s = cudf.Series([1, 2, 3, None, 4])

# Gives (2.5, 1.6666666666666666)
s.mean(), s.var()

def add_ten(num):
    return num + 10

# Compiles add_ten() for CUDA GPU and runs it
# Gives (11, 12, 13, <NA>, 14)
s.applymap(add_ten)
```



Numba Pipeline



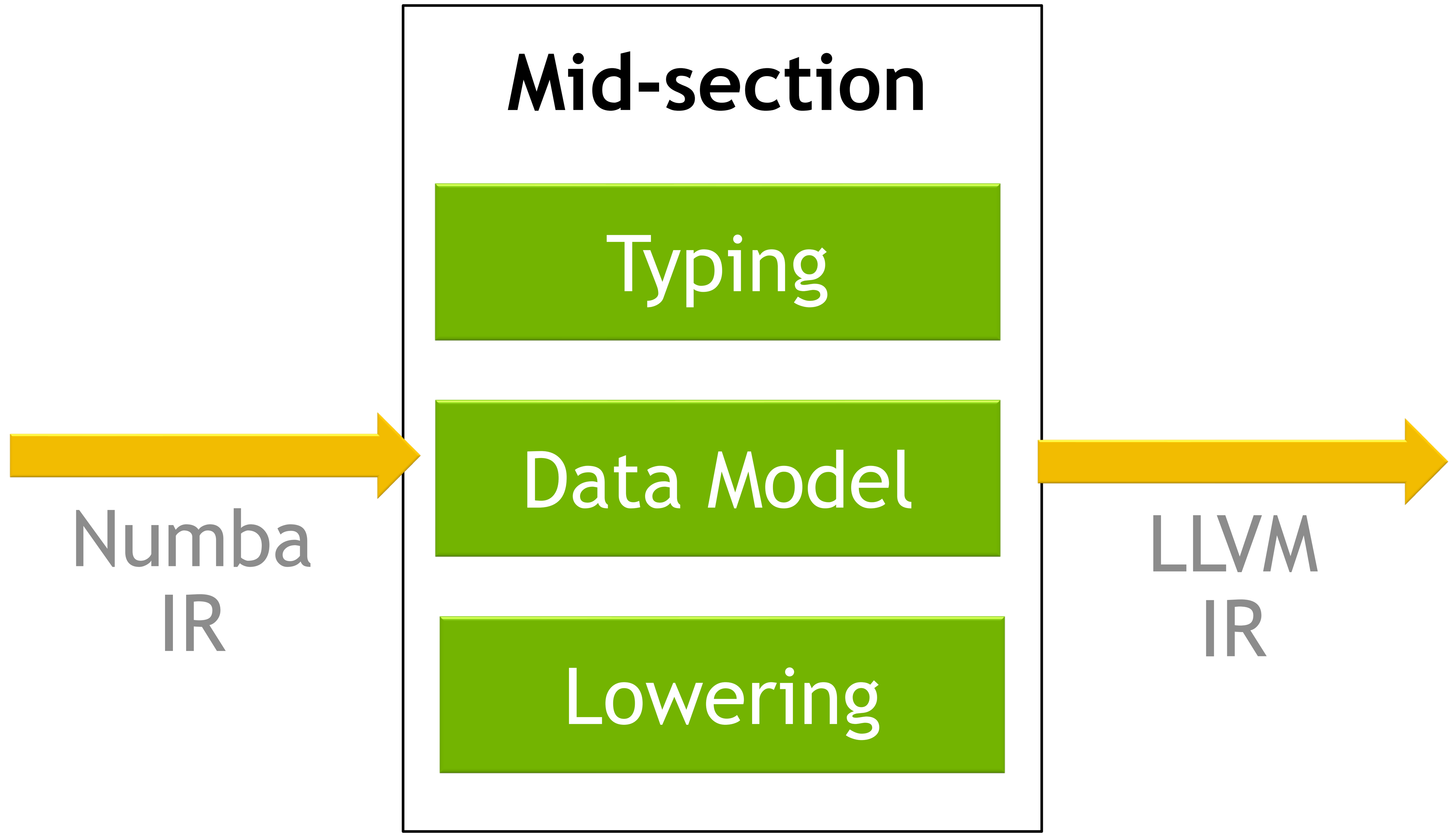
Extension API

- Typing (adds type info):
  - Teach Numba to recognise new types,
  - and functions operating on those types

- Data model (Maps Numba -> LLVM types):
  - Add mappings for new types / data structures

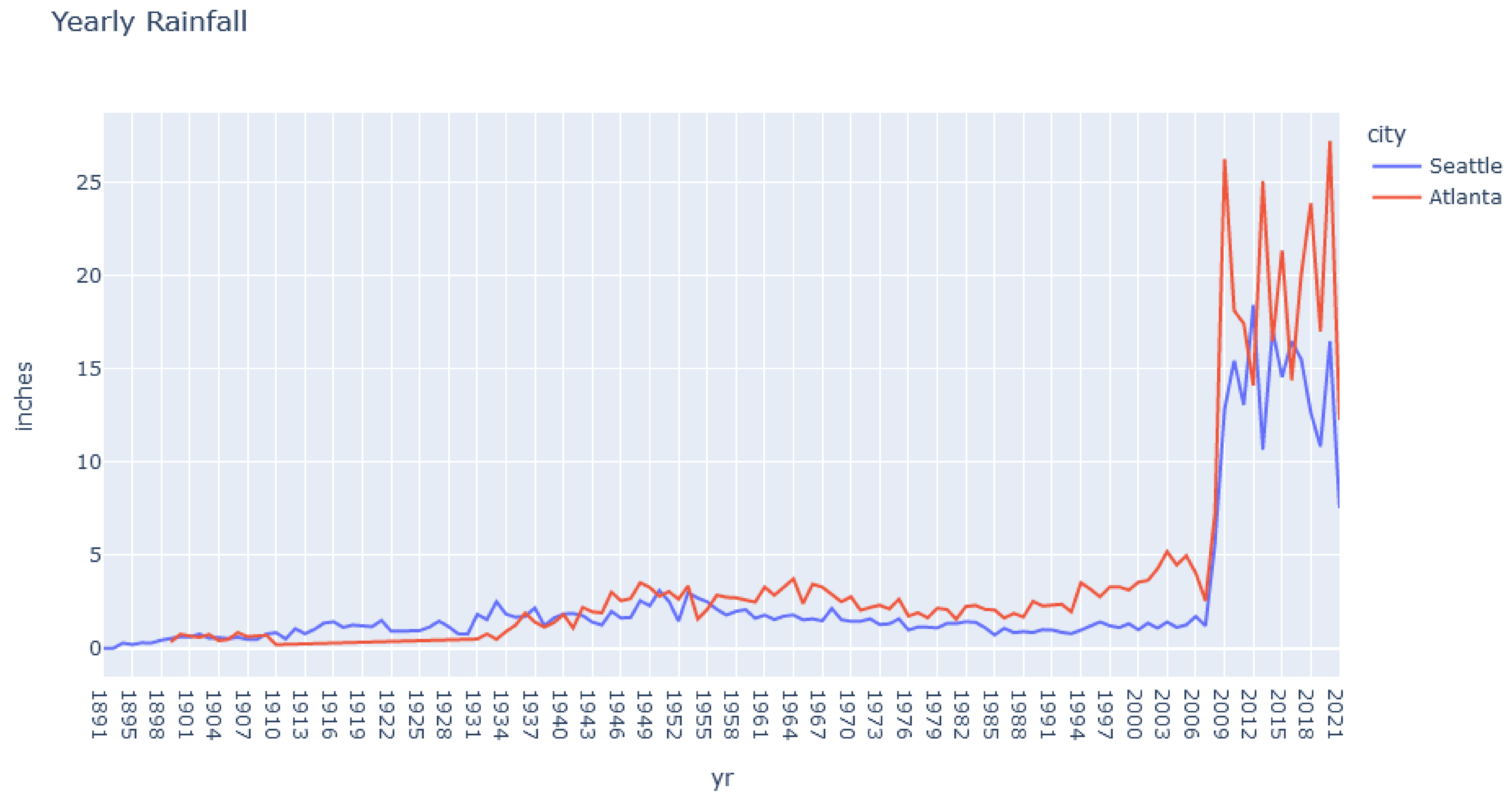
- Lowering (Numba IR -> LLVM IR):
  - Add implementations of operations on new data structures

- Used extensively by cuDF to enable a rich set of features in UDFs on cuDF data types / series
- See also GTC 22: [Enabling Python User-defined functions in accelerated applications]()



Example 1 – UDF performance in cudf and dask-sql

- Example UDF from [“The Weather Notebook”, Dask-SQL for Data Exploration & Analysis]
- First presented in *“Accelerating Data Science: State of RAPIDS”* -John Zedlewski, Ben Zaitlen, Randy Gelhausen, GTC Fall 2021
- Query execution time **0.83s** on ~3M rows on 8-node dask cluster on DGX-1
- CPU execution time for comparison: **1.7s**




```python
def haversine_dist(row, target_latitude, target_longitude):
    x_1 = row["lat1"]
    y_1 = row["lon1"]
    x_2 = target_latitude
    y_2 = target_longitude

    x_1 = math.pi / 180 * x_1
    y_1 = math.pi / 180 * y_1
    x_2 = math.pi / 180 * x_2
    y_2 = math.pi / 180 * y_2

    dlon = y_2 - y_1
    dlat = x_2 - x_1
    a = (
        math.sin(dlat / 2) ** 2
        + math.cos(x_1) * math.cos(x_2) * math.sin(dlon / 2) ** 2
    )

    c = 2 * math.asin(math.sqrt(a))
    r = 6371  # Radius of earth in kilometers

    return c * r
```

Interoperability with CUDA C / C++

- Call CUDA C / C++ device functions from CUDA Python kernels:
  - CUDA Python kernel compiled to PTX with Numba
  - CUDA C / C++ compiled to PTX with NVRTC
  - PTX for both linked together with driver API

- Benefit:
  - Numba CUDA Python users can leverage existing CUDA C / C++ device code

C / C++ Interop – cuRAND Device-side API example

```python
curand_init = cuda.declare_device('_numba_curand_init', curand_init_sig)
curand = cuda.declare_device('_numba_curand',
                             types.uint32(curand_state_pointer, types.uint64))
```

```c
extern "C"
__device__ unsigned int _numba_curand(
    int* numba_return_value,
    curandState *states,
    unsigned long long index)
{
  *numba_return_value = curand(&states[index]);

  return 0;
}
```

C / C++ Interop – cuRAND Device-side API example

```
@cuda.jit(link=['shim.cu'], extensions=[curand_state_arg_handler])
def count_low_bits_native(states, sample_count, results):
    i = cuda.grid(1)
    count = 0

    # Copy state to local memory
    # XXX: TBC

    # Generate pseudo-random numbers
    for sample in range(sample_count):
        x = curand(states, i)

        # Check if low bit set
        if(x & 1):
            count += 1

    # Copy state back to global memory
    # XXX: TBC

    # Store results
    results[i] += count
```

Wrap up

- Summary
- Conclusions
- References

Summary / conclusions

- Numba is a JIT compiler focused on compiling type-specialised versions of numerically-focused code.
  - Keep all code as Python code, but approach C/C++-like performance
  - CUDA Python enables writing CUDA kernels
  - Integrates with CUDA tooling, and other CUDA Python libraries

- Trying it out:
  - `conda install numba cuda-nvcc-impl cuda-nvrtc`
  - `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/excalibur-sysgenx-numba-talk

References / resources

Repo for this talk:

- Numba documentation: https://numba.readthedocs.io/en/stable/
  - Extending Numba with the high- and low-level APIs: https://numba.readthedocs.io/en/stable/extending/index.html
  - Low-level extension API: https://numba.readthedocs.io/en/stable/extending/low-level.html
  - Notes on Numba's architecture: https://numba.readthedocs.io/en/stable/developer/repomap.html
- The Life of a Numba Kernel: https://github.com/gmarkall/life-of-a-numba-kernel/
  - Blog post / Jupyter Notebook
- NVIDIA Numba CUDA tutorial:
  - Github repository: https://github.com/numba/nvidia-cuda-tutorial
  - All slides: https://raw.githubusercontent.com/numba/nvidia-cuda-tutorial/main/numba-for-cuda-programmers-complete.pdf
- Talk on embedding Numba as a UDF compiler (GTC 2022) – lots of detail on Numba internals:
  - Recording / slides: https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41056/
  - https://github.com/gmarkall/numba-accelerated-udfs
- Other example extensions: https://github.com/gmarkall/extending-numba-cuda
  - Jupyter Notebook / Quaternion Example / Interval Example
- Application use cases:
  - cuDF Extension code: https://github.com/rapidsai/cudf/tree/branch-22.04/python/cudf/cudf/core/udf
  - PyOptiX Extension code: https://github.com/gmarkall/PyOptiX/tree/gtc2022
- Contact:
  - Numba real-time chat: https://gitter.im/numba/numba
  - Numba Discourse forums: https://numba.discourse.group/
  - Email: gmarkall@nvidia.com

NVIDIA

Numba development – collaboration / process

- Collaboration:
  - Generally through pull requests / issues / dev meetings / forums
  - CUDA Pull requests usually reviewed by someone outside NVIDIA

- Larger changes in Numba Enhancement Proposals (NBEPs)
  - Example: NBEP 7: https://numba.readthedocs.io/en/latest/proposals/external-memory-management.html

NBEP 7: CUDA External Memory Management Plugins

Author:	Graham Markall, NVIDIA
Contributors:	Thomson Comer, Peter Entschev, Leo Fang, John Kirkham, Keith Kraus
Date:	March 2020
Status:	Final

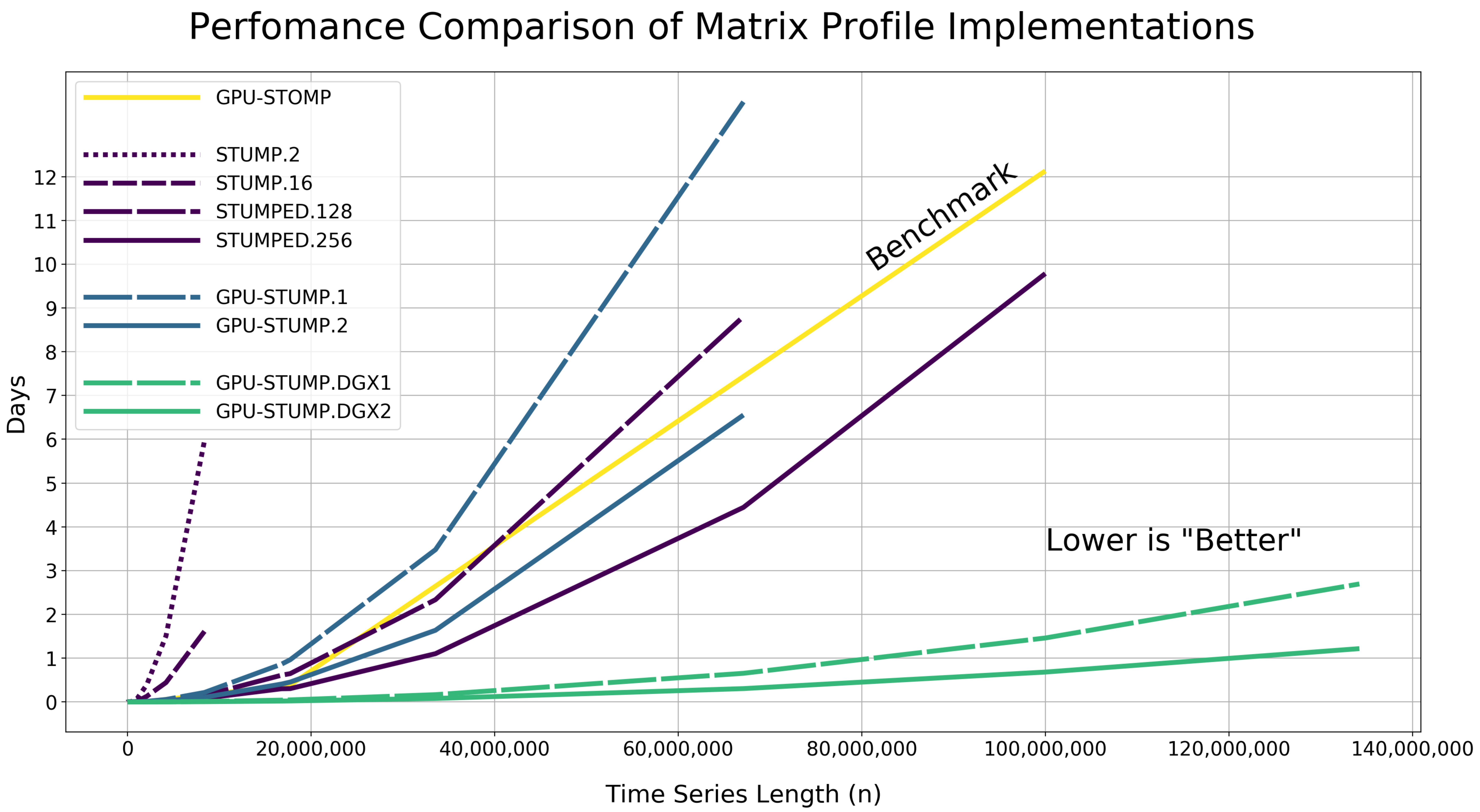
Background and goals

The CUDA Array Interface enables sharing of data between different Python libraries that access CUDA devices. However, each library manages its own memory distinctly from the others. For example:

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, such as:*
  - *pattern/motif (approximately repeated subsequences within a longer time series) discovery*
  - *anomaly/novelty (discord) discovery*
  - *shapelet discovery*
  - *semantic segmentation*
  - *streaming (on-line) data*
  - *fast approximate matrix profiles*
  - *time series chains)*
  - *snippets for summarizing long time series*
  - *pan matrix profiles for selecting the best subsequence window size(s)*
  - *[and more …]()"*

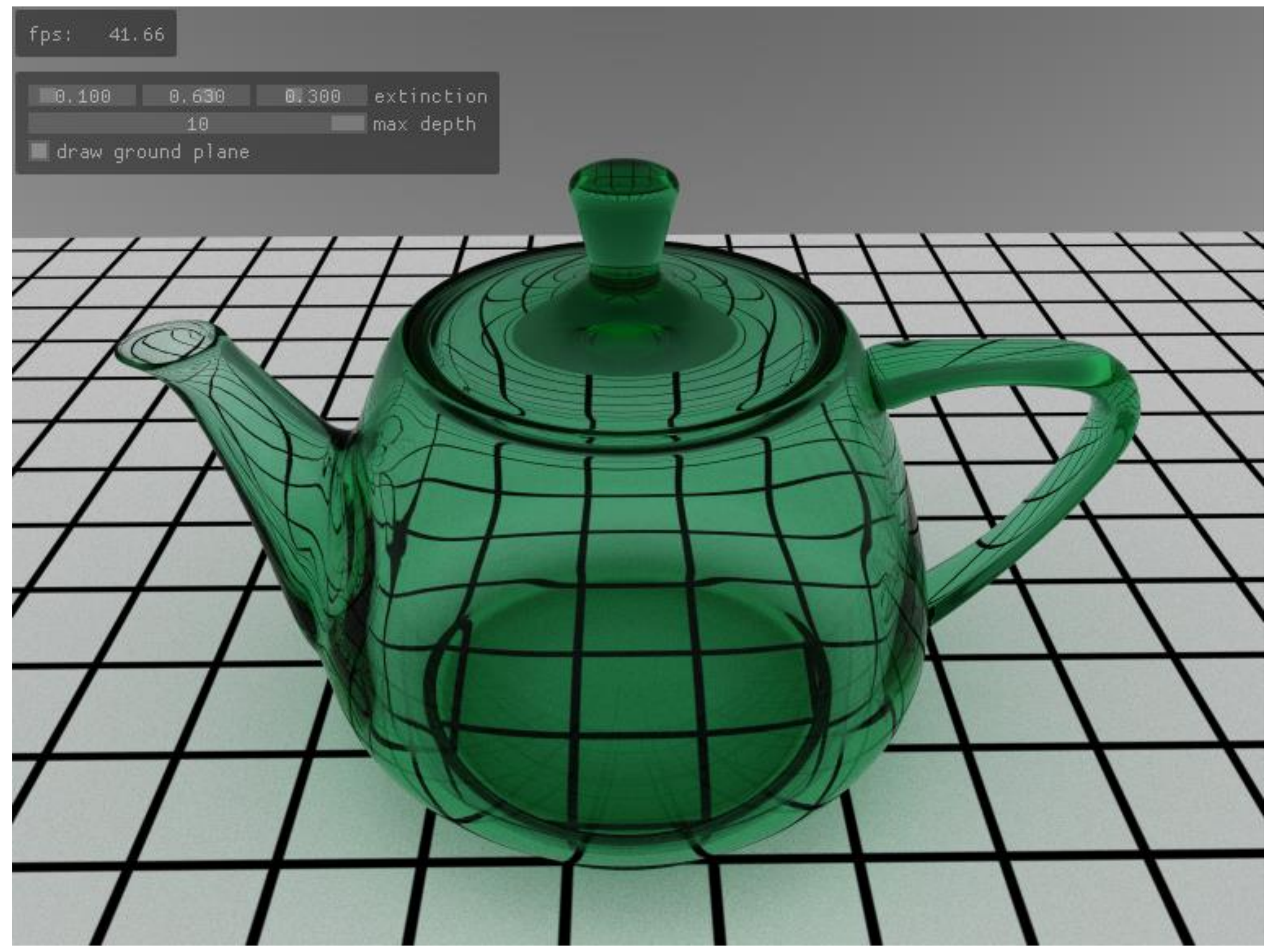


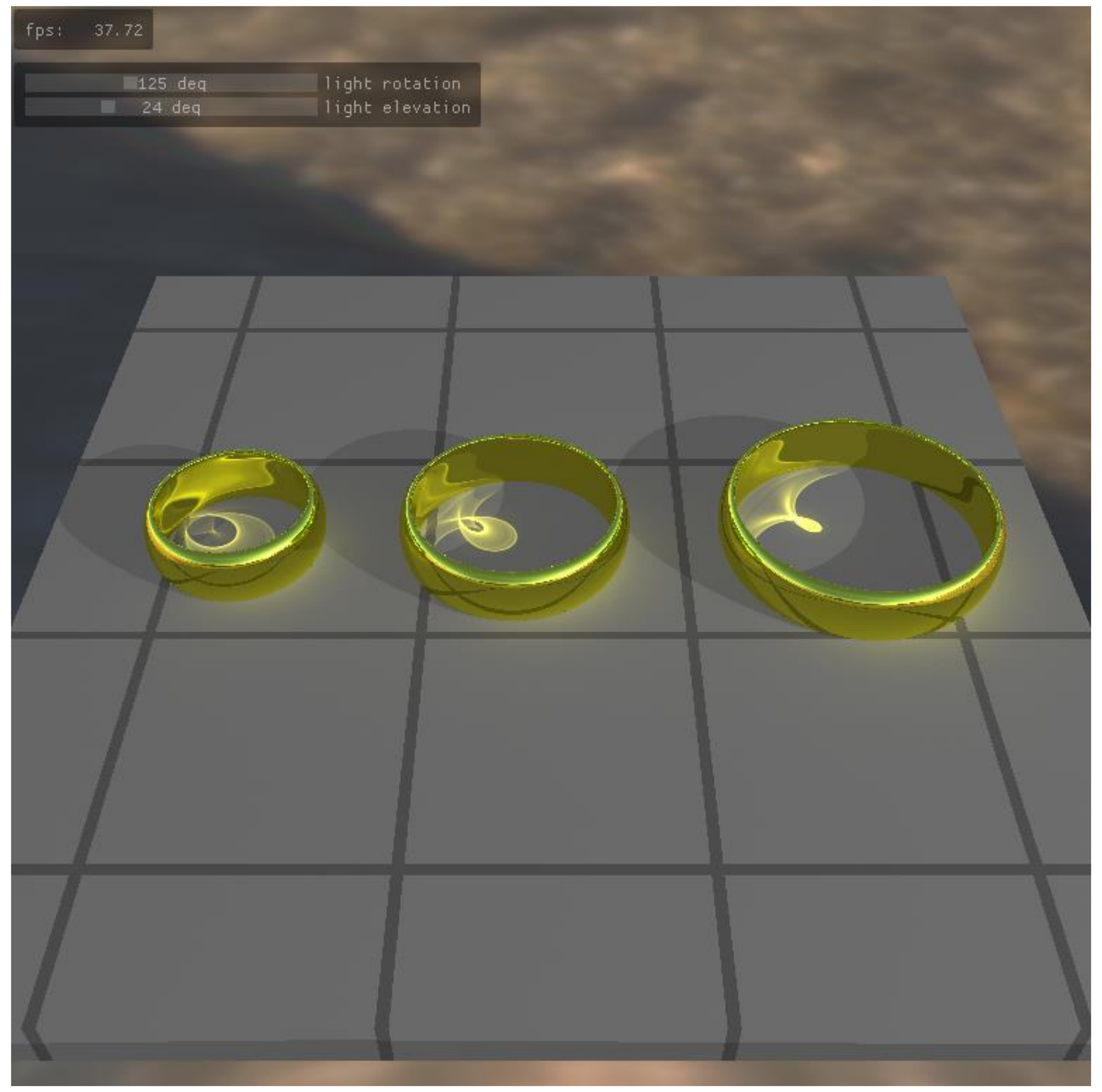
Example 2 - PyOptiX

- **OptiX:** optimal performance GPU-accelerated ray tracing
- **PyOptiX:** Python bindings for host-side functions, CUDA C/C++ kernels
- **Numba + PyOptiX:** write on-GPU raytracing kernels in Python

NVIDIA OPTIX™

Images rendered by OptiX sample applications





Example 2 – PyOptiX CUDA C/C++ kernel

```
static __forceinline__ __device__ void computeRay(uint3 idx, uint3 dim, float3& origin, float3& direction)
{
    const float3 U = params.cam_u;
    const float3 V = params.cam_v;
    const float3 W = params.cam_w;
    const float2 d = 2.0f * make_float2(
            static_cast<float>( idx.x ) / static_cast<float>( dim.x ),
            static_cast<float>( idx.y ) / static_cast<float>( dim.y )
            ) - 1.0f;

    origin    = params.cam_eye;
    direction = normalize( d.x * U + d.y * V + W );
}

extern "C" __global__ void __raygen__rg()
{
    // Lookup our location within the launch grid
    const uint3 idx = optixGetLaunchIndex();
    const uint3 dim = optixGetLaunchDimensions();

    // Map our launch idx to a screen location and create a ray from the camera
    // location through the screen
    float3 ray_origin, ray_direction;
    computeRay( make_uint3( idx.x, idx.y, 0 ), dim, ray_origin, ray_direction );
    // …
```

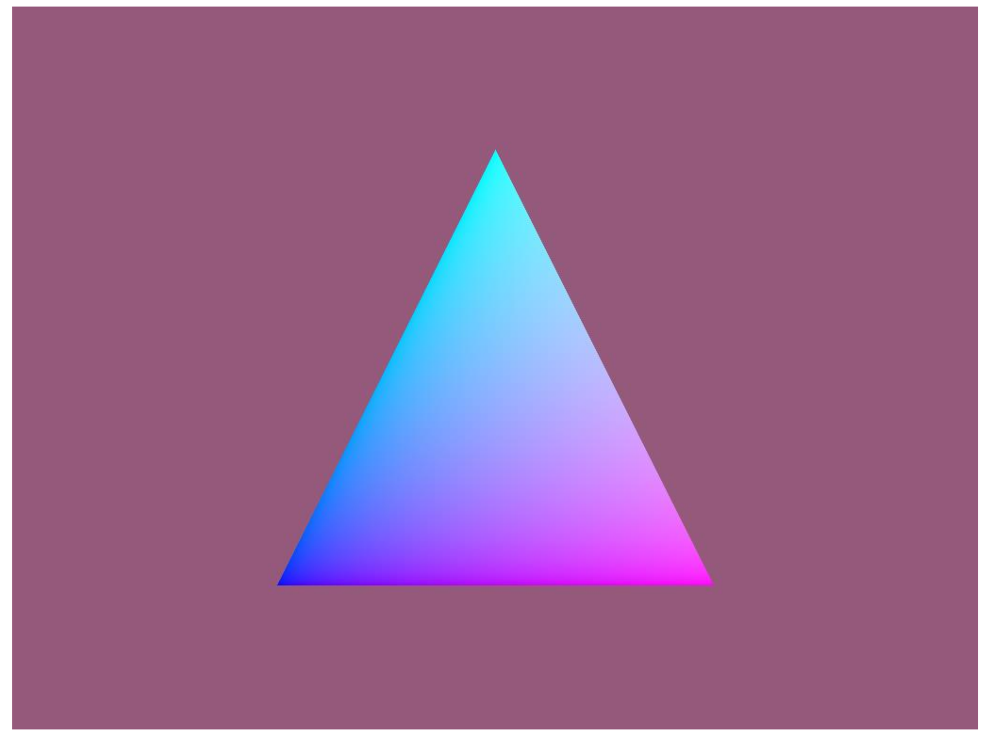
Example 2 – PyOptiX Python kernel with Numba

```
@cuda.jit(device=True, fast_math=True)
def computeRay(idx, dim):
    U = params.cam_u
    V = params.cam_v
    W = params.cam_w
    # Normalizing coordinates to [-1.0, 1.0]
    d = float32(2.0) * make_float2(
        float32(idx.x) / float32(dim.x), float32(idx.y) / float32(dim.y)
    ) - float32(1.0)

    origin = params.cam_eye
    direction = normalize(d.x * U + d.y * V + W)
    return origin, direction

def __raygen__rg():
    # Lookup our location within the launch grid
    idx = optix.GetLaunchIndex()
    dim = optix.GetLaunchDimensions()

    # Map our launch idx to a screen location and create a ray from the camera
    # location through the screen
    ray_origin, ray_direction = computeRay(make_uint3(idx.x, idx.y, 0), dim)

    # …
```

Example 2 – PyOptiX Raygen kernel performance

- Kernel execution time measured with Nsight Compute:

Language	Kernel execution time (cycles)	% of baseline
C++	94,172	100.0
Python	106,776	113.3



Triangle rendered by example kernel

- Further optimization: force inlining, fastmath flags
  - Target: Numba 0.56 (June / July)

CUDA Array Interface

- Zero-copy interface for array data between CUDA Python libraries

```
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}
```

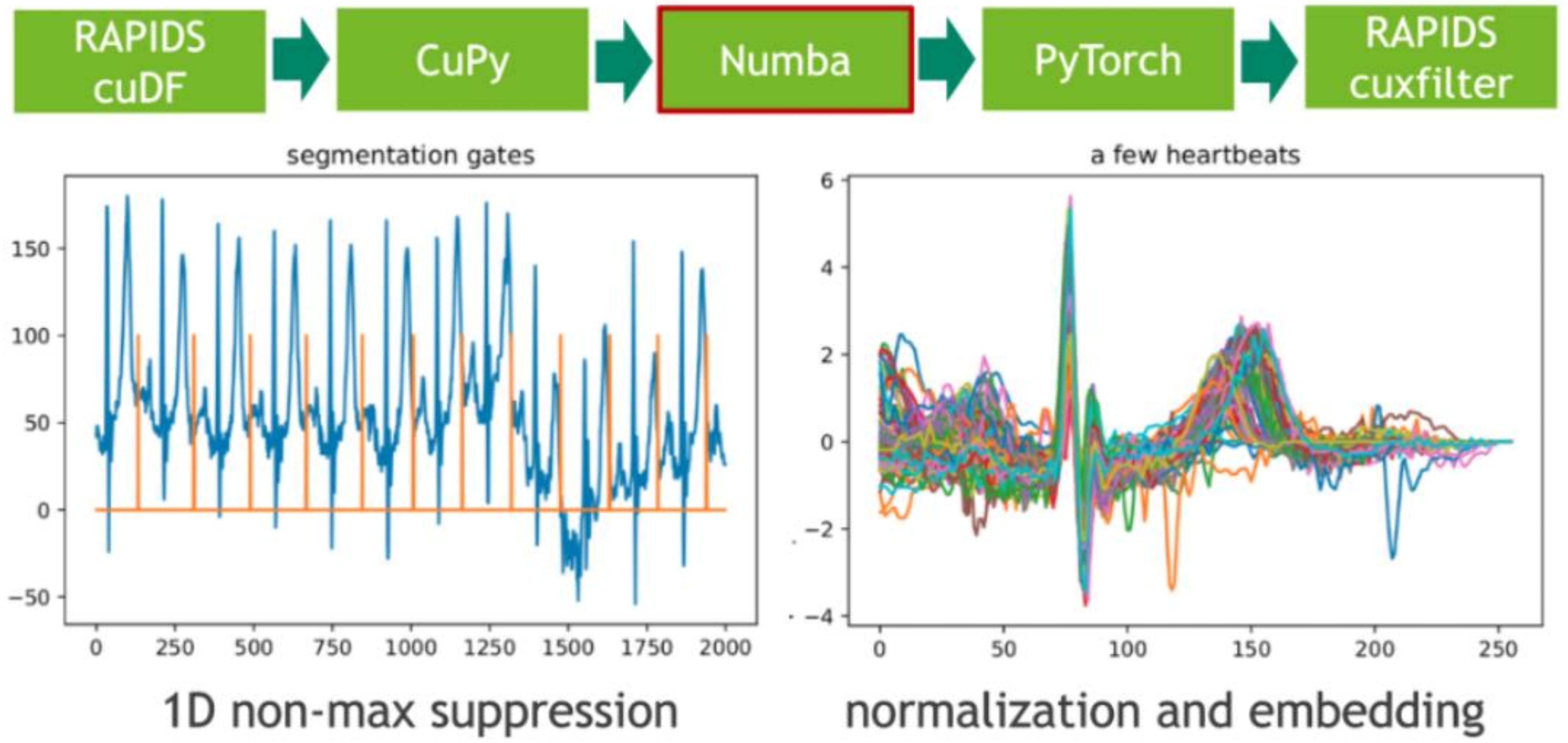


Figure 3: 1D non-maximum suppression and embedding of heartbeats using Numba JIT.

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