# ENABLING PYTHON USER DEFINED FUNCTIONS (UDFS) IN ACCELERATED APPLICATIONS WITH NUMBA

GRAHAM MARKALL, BRANDON MILLER, MICHAEL YH WANG – NVIDIA RAPIDS

# SPEAKER INTRODUCTION - GRAHAM MARKALL

- Software Engineer in RAPIDS at NVIDIA:
  - Numba CUDA target maintainer
  - Supporting cuDF / RAPIDS use cases

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

- Aim: Make powerful software tools
  accessible to many people

# PROBLEM STATEMENT - THE "IMPEDANCE MISMATCH"

- **Applications:**
  - Many CUDA-accelerated applications written in C++
- **Users / developers:**
  - Many application users / developers prefer to work in Python
- **How do we:**
  - Enable Python users to extend accelerated applications…
  - … whilst retaining *developer productivity*…
  - … and *application performance*?
- **Solution:**
  - Use Numba to compile Python user code for CUDA
  - *User-Defined Functions (UDFs)*

# TALK ROADMAP

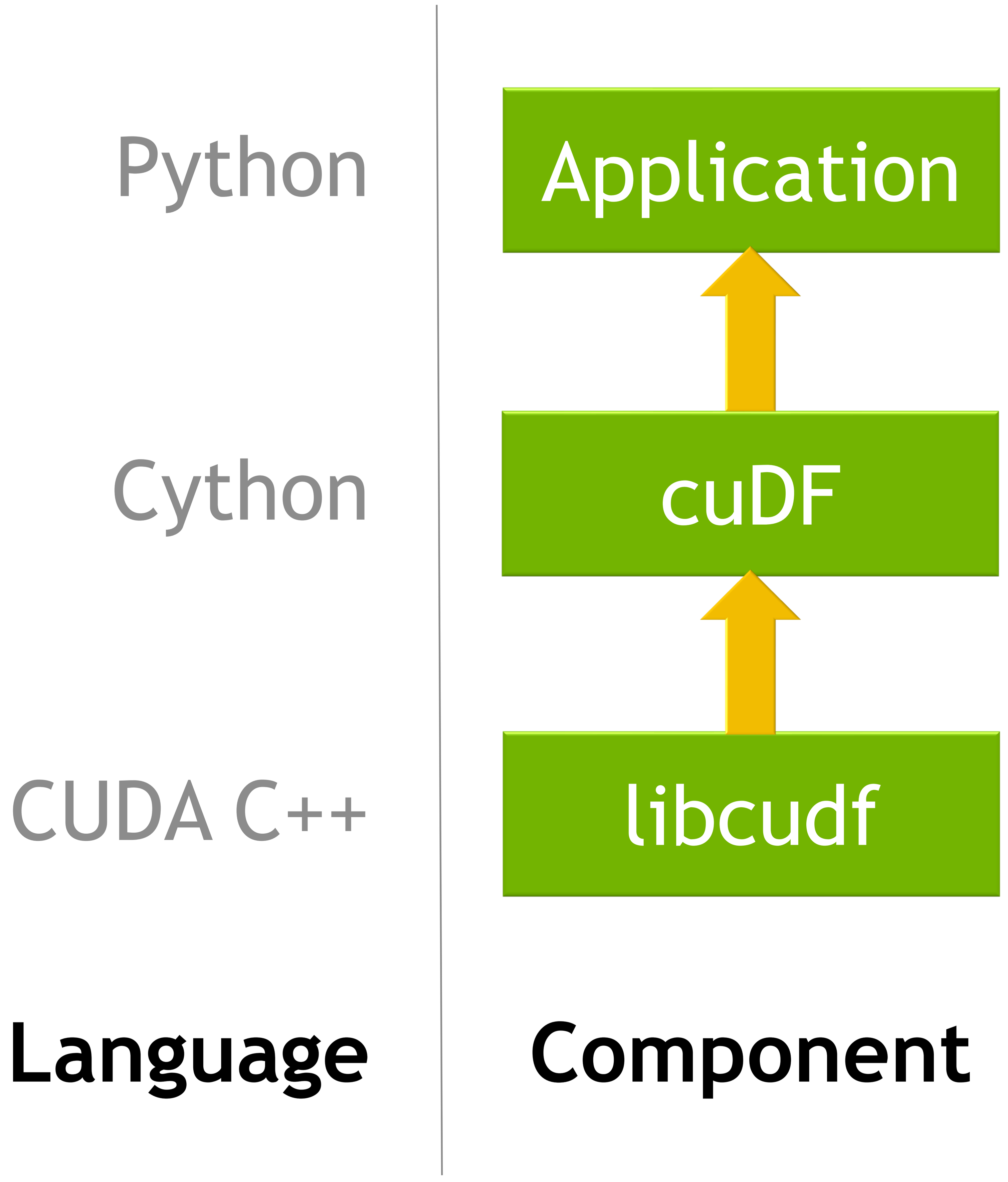
- **Examples:**
  - cuDF
  - PyOptiX
  - Filigree
- **Numba:**
  - Overview
  - Pipeline & extensions
- **Worked example:**
  - Implementation
  - Outcomes (development ease & performance)

# EXAMPLE 1 - CUDF

- Pandas for GPUs, or:

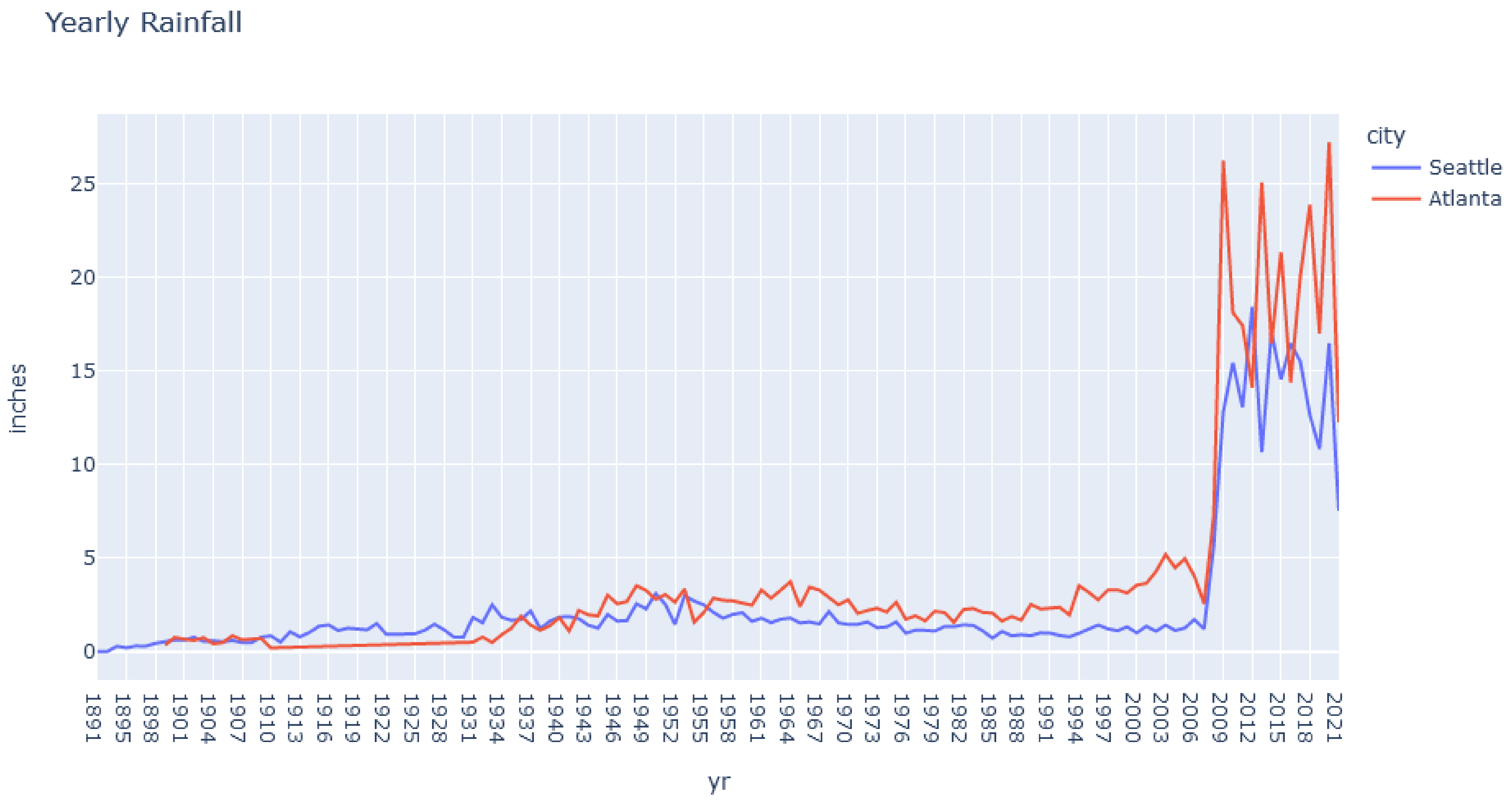
"cuDF is a Python GPU DataFrame library (built on the Apache Arrow columnar memory format) for loading, joining, aggregating, filtering, and otherwise manipulating tabular data using a DataFrame style API."

```
# 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)
```



# EXAMPLE 1 - UDF PERFORMANCE IN CUDF AND DASK-SQL

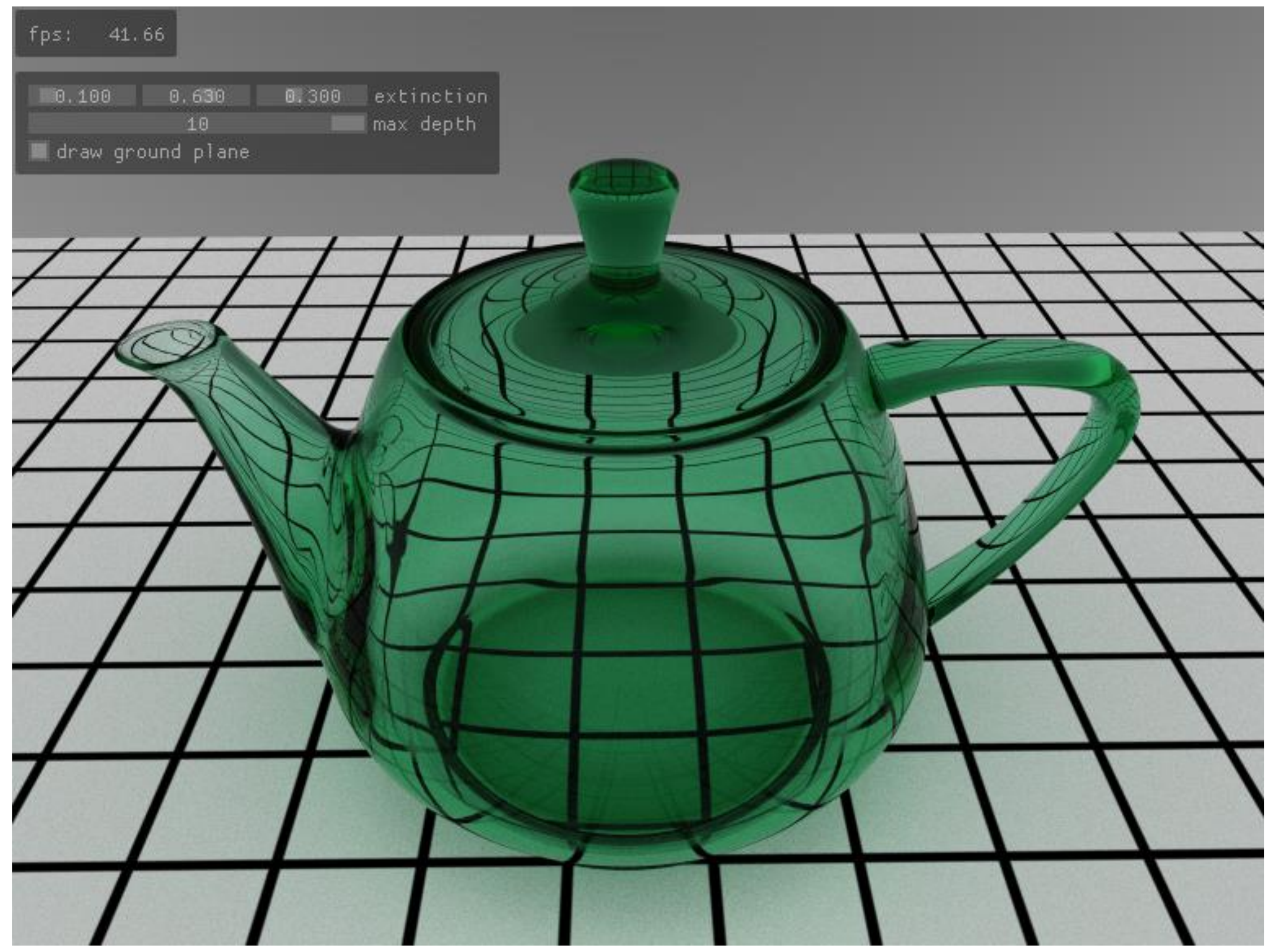
- Example UDF from "The Weather Notebook", Dask-SQL for Data Exploration & Analysis
- First presented in *"Accelerating Data Science: State of RAPIDS"* -John Zedlewski, Ben Zaitlen, Randy Gelhausen, GTC Fall 2021
- Query execution time **0.83s** on ~3M rows on 8-node dask cluster on DGX-1
- CPU execution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

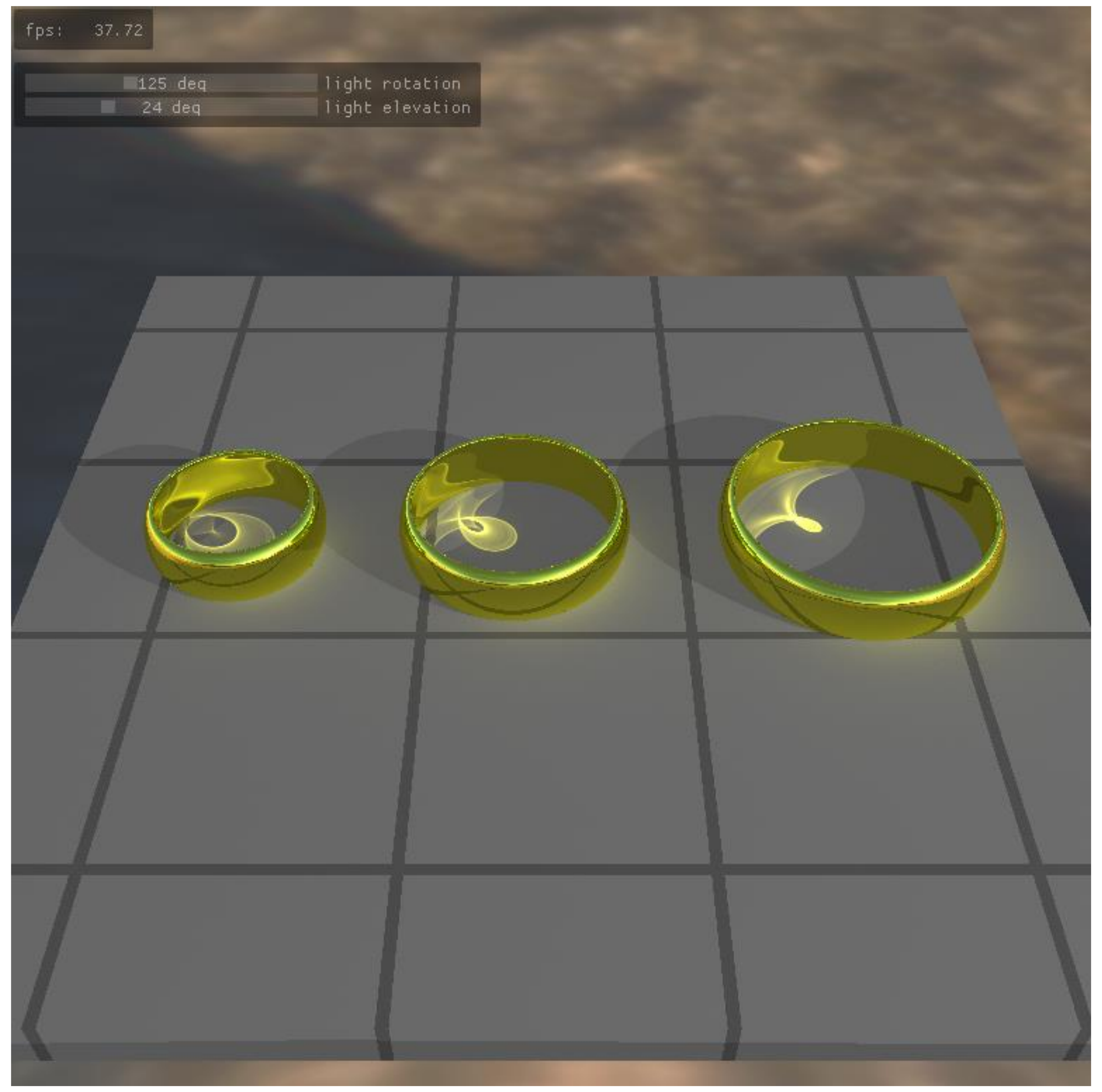



```python
def haversine_dist(row, target_latitude, target_longitude):
    x_1 = row["lat1"]
    y_1 = row["lon1"]
    x_2 = target_latitude
    y_2 = target_longitude

    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
```

# EXAMPLE 2 - PYOPTIX

*Images rendered by OptiX sample applications*





- **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

# 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 );
    // …
```

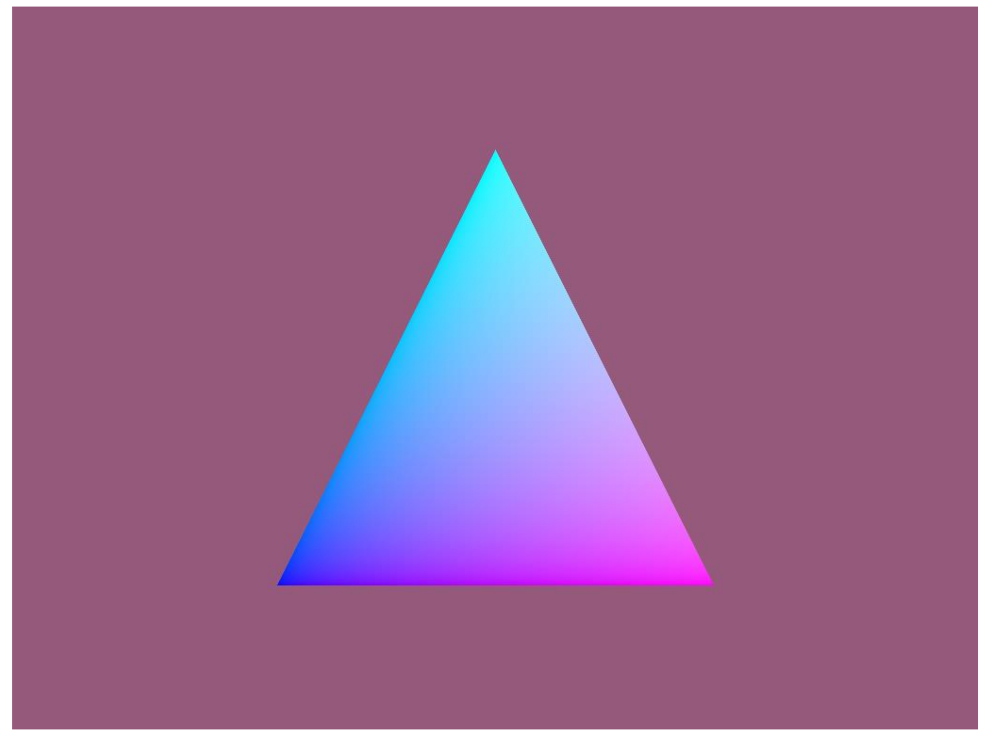
# EXAMPLE 2 - PYOPTIX PYTHON KERNEL WITH NUMBA

```
@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)

# DEMO APPLICATION - FILIGREE

- **Filigree:** an image processing library
  - Written in CUDA C++
  - Uses ImageMagick for file I/O and data structures
  - Python Bindings + API

```
__global__ void to_greyscale_kernel(
    Magick::PixelPacket *pixels,
    size_t width, size_t height) {
  size_t x = blockIdx.x * blockDim.x + threadIdx.x;
  size_t y = blockIdx.y * blockDim.y + threadIdx.y;

  if ((x < width) && (y < height)) {
    Magick::PixelPacket *pixel = &pixels[y * width + x];
    float weighted = (0.299f * pixel->red +
                      0.587f * pixel->green +
                      0.114 * pixel->blue);
    pixel->red = (Magick::Quantum)weighted;
    pixel->green = (Magick::Quantum)weighted;
    pixel->blue = (Magick::Quantum)weighted;
  }
}
```

- Full source, completed example:
  - [https://github.com/gmarkall/numba-accelerated-udfs](https://github.com/gmarkall/numba-accelerated-udfs)

# PYTHON "HOST" API

```
from filigree import Image
image = Image(path)
image.to_pillow()
```

```
image.apply_weighted_greyscale()
```

```
image.to_pillow()
```

# FILIGREE COMPONENTS

# AIM: SUPPORT THIS USE CASE

- User-defined function, written in Python, applied from Python
- Compiled to run on GPU and operate directly on Filigree data
- No user-visible C / C++ / CUDA / low-level API

```
image = Image(path)

center_x = image.width / 2
center_y = image.height // 2
radius = max(center_x, center_y)

def highlight_center(pixel, x, y):
    distance_from_center = math.sqrt(
        abs(x - center_x)**2 + abs(y - center_y)**2)

    # Weight pixels according to distance from centre
    w = 1 - (distance_from_center / radius)

    # Apply weight to each RGB component
    return pixel.r * w, pixel.g * w, pixel.b * w, pixel.a

image.apply_located_pixel_udf(highlight_center)
image.to_pillow()
```

# TECHNICAL FOUNDATION

Numba – a Python JIT Compiler

# A BRIEF INTRODUCTION TO NUMBA

## 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
  - GPU: CUDA

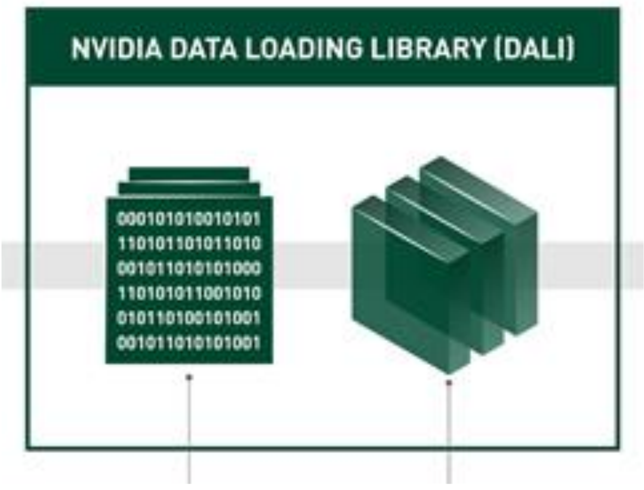
**CUDA Python Ecosystem**

RAPIDS

Numba

CuPy

PyTorch

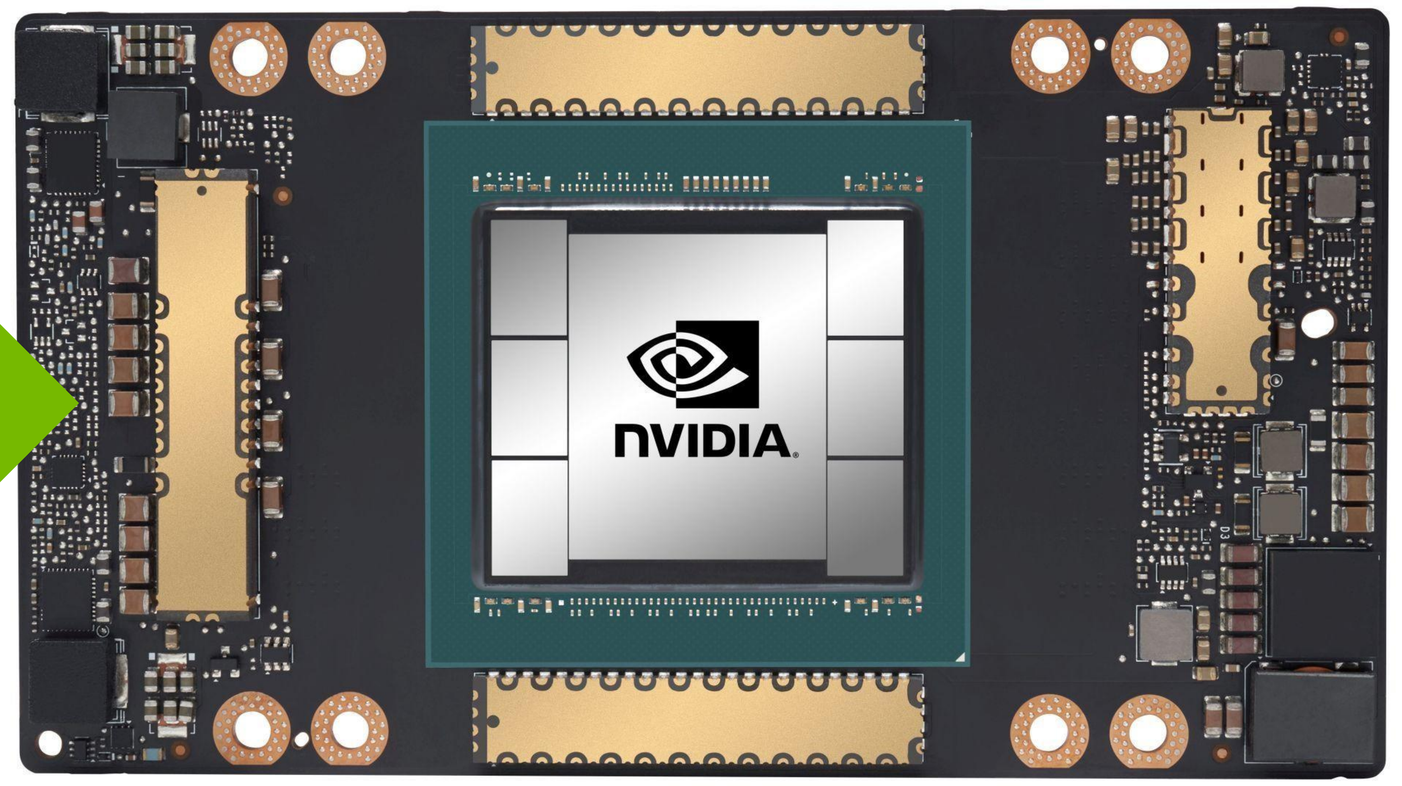



+ more… See:
[CUDA Array Interface documentation]

## CUDA Target:

- Compiler
- Driver bindings
- Device array library

```python
@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
```



# “STANDALONE” NUMBA USAGE

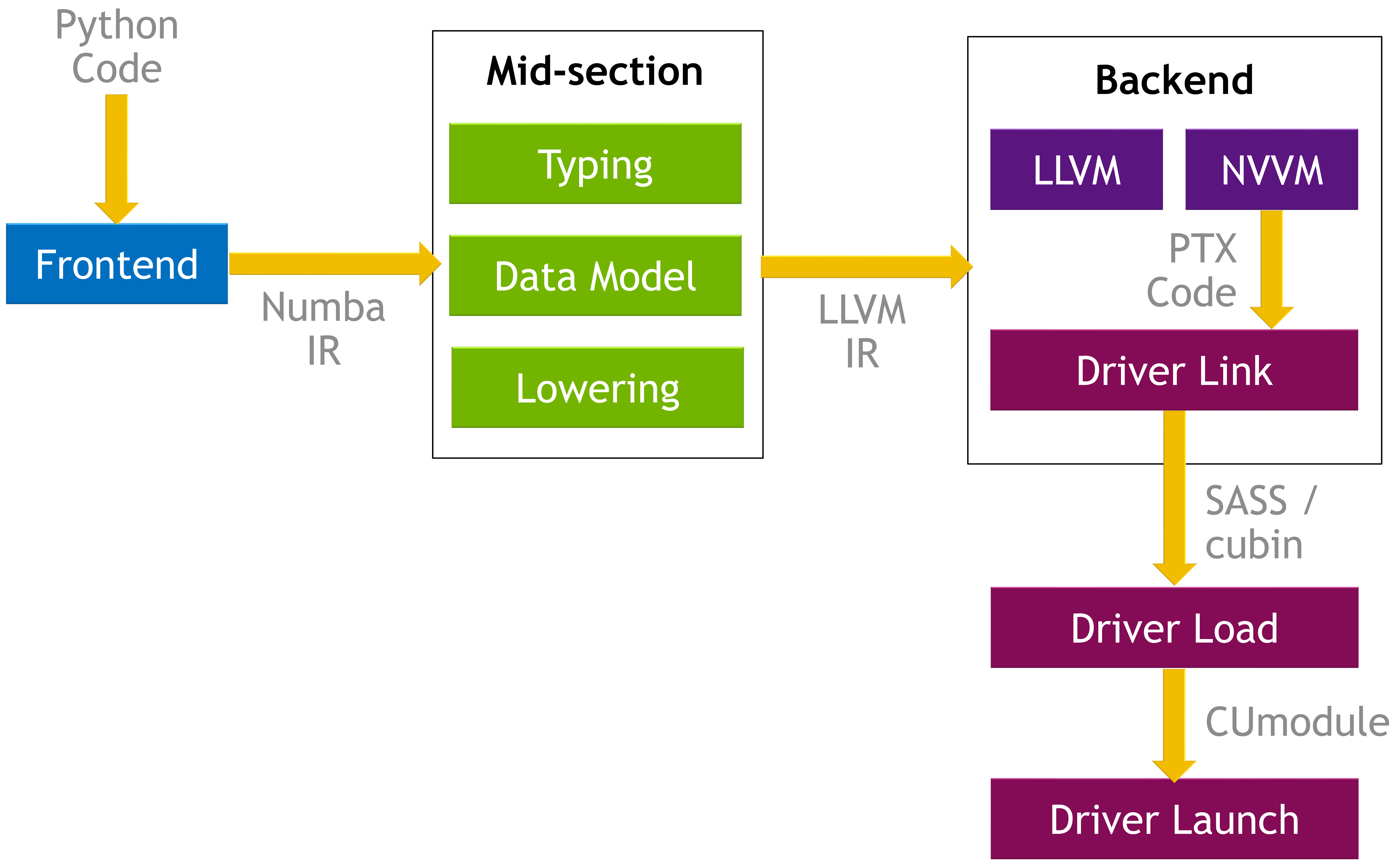
```
from numba import cuda

# Define a kernel that is compiled for 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]

# Allocate some arrays on the device and copy data
N = 2 ** 10
x = cuda.to_device(np.arange(N))
y = cuda.to_device(np.arange(N) * 2)
r = cuda.device_array_like(x)

# Configure and launch kernel
block_dim = 256
grid_dim = (len(x) // block_dim) + 1
vector_add[grid_dim, block_dim](r, x, y)

# Copy result back from the device
result = r.copy_to_host()
```

- Numba + CUDA Tutorial: https://github.com/numba/nvidia-cuda-tutorial
  - **Session 1:** An introduction to Numba and CUDA Python
  - **Session 2:** Typing
  - **Session 3:** Porting strategies, performance, interoperability, debugging
  - **Session 4:** Extending Numba
  - **Session 5:** Memory Management

# WHO IS USING NUMBA?

- PyPI: 250,000 / Conda 16,000+ downloads per day
- Github:
  - 48,000 dependent repositories
  - 7,300 stars
  - 879 forks
  - 205 watchers
- Random sample of applications using the CUDA target:
  - Poliastro (astrodynamics)
  - FBPIC (CUDA-accelerated plasma physics)
  - UMAP (manifold learning)
  - RAPIDS (data science)
  - Talks on more applications in the Numba documentation
  - and https://github.com/gmarkall/numba-cuda-users

# NUMBA PIPELINE



# LLVM

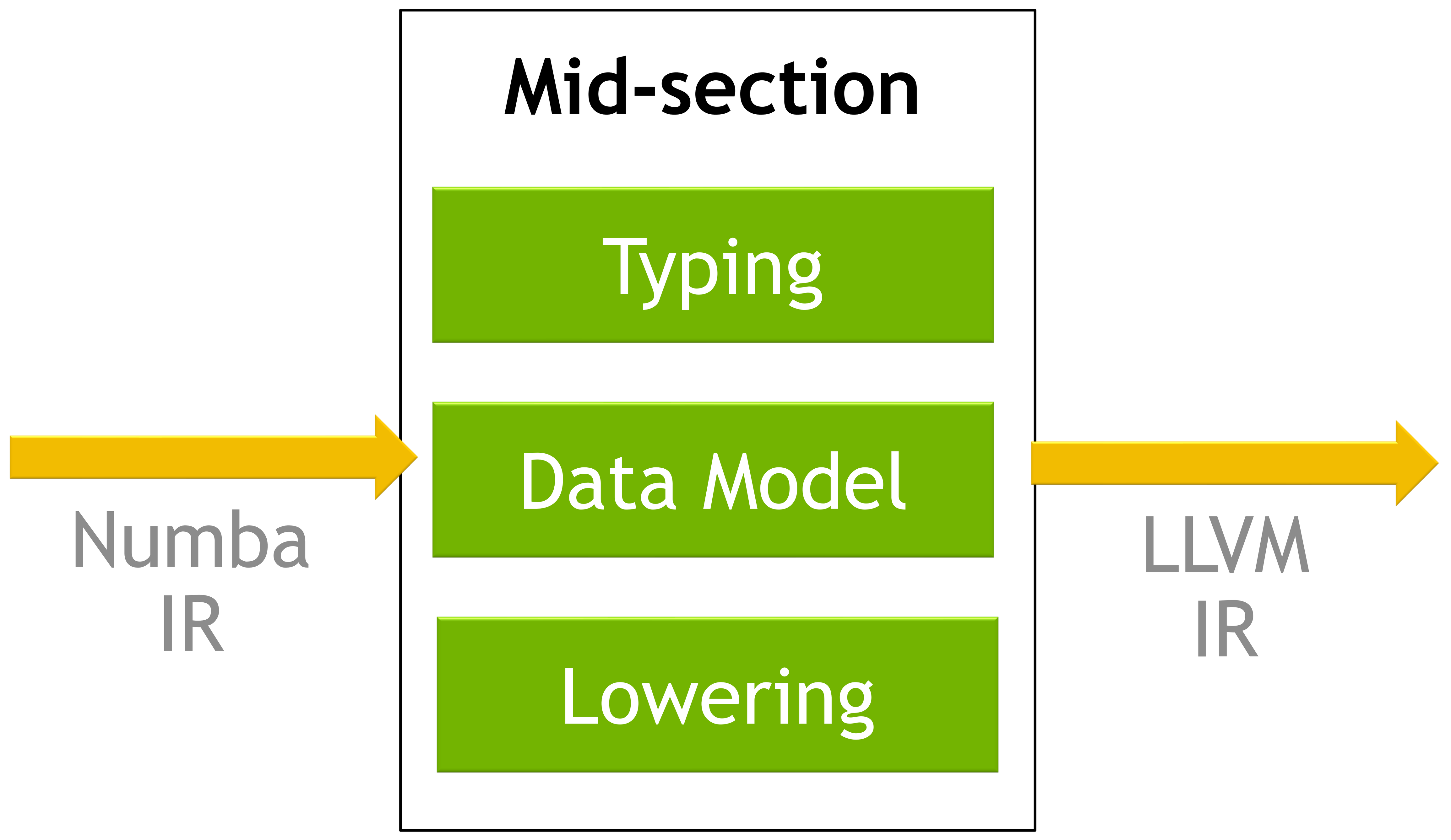
- *"The LLVM Project is a collection of modular and reusable compiler and toolchain technologies."* - [https://llvm.org](https://llvm.org)

Python, C++, Rust, Fortran, etc… → **Language Frontend** → LLVM IR → **LLVM** → PTX, ARM, x86, RISC-V, etc.

*The LLVM Wyvern logo*

# UDF-RELATED COMPONENTS

- Typing (adds type info):
  - Teach Numba to recognise our library’s types,
  - and functions operating on those types
- Data model (Maps Numba -> LLVM types):
  - Add mappings for our types / data structures
- Lowering (Numba IR -> LLVM IR):
  - Add implementations of operations on our library data structures




NVIDIA

# TYPING

- ***Type Inference:*** types determined by propagating type information
  - **Function Typing:** *"For these argument types, what type is returned?"*
  - **Data flow:**
    - *"Where did the inputs come from?"*
    - *"Where is the output / return going?"*

- **"No regrets":** typing does not backtrack

- Simple example:

```
def f(a, b):              # a:= float32, b:= int32
    c = np.log(b)         # c:= float64
    d = a + c             # d = float64
    return d              # return := float64
```

# TYPE UNIFICATION (1)

```
def select(a, b, c):  # a := float32, b := float32, c := bool
    if c:
        ret = a       # ret := float32
    else:
        ret = b       # ret := float32
    return ret        # return := {float32, float32}
                      #            => float32
```

# TYPE UNIFICATION (2)

```
def select(a, b, c):  # a := float32, b := float64, c := bool
    if c:
        ret = a       # ret := float32
    else:
        ret = b       # ret := float64
    return ret        # return := {float32, float64}
                      #           => float64
```

# TYPE UNIFICATION (3)

```
def select(a, b, c):  # a := tuple(int32, int32), b := float64, c := bool
    if c:
        ret = a       # ret := tuple(int32, int32)
    else:
        ret = b       # ret := float64
    return ret        # return := {tuple(int32, int32), float64}
                      #             => XXX!!!

numba.core.errors.TypingError: Failed in nopython mode pipeline (step: nopython frontend)
Can't unify return type from the following types: UniTuple(int32 x 2), float64
```

Unification rules:

- Internal to Numba
- Can be extended

# HOW TO ADD TYPING - CONCRETE TYPING

- **Concrete:** a table of ***specific*** cases

- Examples from within Numba:

```
class Cuda_popc(ConcreteTemplate):
    key = cuda.popc
    cases = [
        signature(types.int8, types.int8),
        signature(types.int16, types.int16),
        signature(types.int32, types.int32),
        signature(types.int64, types.int64),
        signature(types.uint8, types.uint8),
        signature(types.uint16, types.uint16),
        signature(types.uint32, types.uint32),
        signature(types.uint64, types.uint64),
    ]
```

```
class Math_pow(ConcreteTemplate):
    key = math.pow
    cases = [
        signature(types.float64, types.float64, types.int64),
        signature(types.float64, types.float64, types.uint64),
        signature(types.float32, types.float32, types.float32),
        signature(types.float64, types.float64, types.float64),
    ]
```

# HOW TO ADD TYPING - GENERIC / ABSTRACT TYPING

- Write a function that accepts argument types and computes return type:

```
# Simplified typing pseudo code:
def cuda_grid_typer(ndim):
    require_integer_literal(ndim)
    value = ndim.literal_value

    if value == 1:
        return int32
    elif value == 2:
        return tuple(int32, int32)
    elif value == 3:
        return tuple(int32, int32, int32)
    else:
        raise Error(“Grid can only be 1D, 2D, or 3D”)
```

# DATA MODELS

- Connect the frontend to the backend
  - Mapping Numba types -> LLVM Types
- LLVM Type System:
  - Void (`void`)
  - Scalars
    - Integers: `i32`, `i64`, …
    - Floating point: `float`, `double`, …
  - Aggregates
    - Arrays: `[40 x i32]`
    - Srtructs: `{i32, i32, i64}`
  - Pointers (`void*`, `i32*`, `[40 x i32]*`, `float***`, …)
- LLVM IR Reference manual: https://llvm.org/docs/LangRef.html#type-system

# DATA MODEL EXAMPLES

| Python Value | Numba Type | LLVM Type used in Numba |
|---|---|---|
| 3 (int) | int64 | i64 |
| 3.14 (float) | float64 | double |
| (1, 2, 3) | UniTuple(int64, 3) | [3 x i64] |
| (1, 2.5) | Tuple(int64, float64) | {i64, double} |
| np.array([1, 2],<br>dtype=np.int32) | array(int64, 1d, C) | {i8*, i8*, i64, i64,<br>i32*,<br>[1 x i64], [1 x i64]} |
| “Hello” | unicode_type | {i8*, i64, i32, i32,<br>i64, i8*, i8*} |

NVIDIA

# TUPLE REPRESENTATIONS

| Layer | Representation |
|---|---|
| **Python** | `(1, 2.5)` |
| **Numba** | `Tuple(int64, float64)` |
| **C / C++** | `struct {`<br>`    uint64_t v1;`<br>`    double   v2;`<br>`};` |
| **LLVM** | `{`<br>`    i64,      ; v1`<br>`    double    ; v2`<br>`}` |

# ARRAY REPRESENTATIONS

| Layer | Representation |
|---|---|
| **Python** | `np.array([1, 2], dtype=np.int32)` |
| **Numba** | `array(int64, 1d, C)` |
| **C / C++** | `struct {`<br>`    void      *meminfo;`<br>`    PyObject *parent;`<br>`    npy_intp nitems;`<br>`    npy_intp itemsize;`<br>`    void *data;`<br>`   npy_intp shape_and_strides[];`<br>`};` |
| **LLVM** | `{i8*, i8*,                 ; meminfo, parent`<br>` i64, i64,                  ; nitems, itemsize`<br>` i32*,                      ; data`<br>` [1 x i64], [1 x i64]}      ; shape, strides` |

# BUILT-IN MODELS

- Data models can be re-used, subclassed by extensions
- A few examples built-in to Numba:

| Model | Purpose | Examples |
| --- | --- | --- |
| Primitive model | Primitive types | Integer values, floating point values, pointers |
| Opaque model | Objects passed as pointers | String literals, functions, exceptions, |
| Struct model | Objects represented as structures “under the hood” | Arrays, Unicode strings, Tuples |
| UniTuple model | Tuples of homogeneous type | UniTuple |
| Composite model | Objects with structure visible to the user | Records |

- Use case in worked example

# LOWERING

- Provide implementations of all supported operations
- Data flow handled by Numba:
  - Arguments passed in
  - Arguments returned
- 1 lowering per function and set of types
  - (Note: one lowering can handle multiple sets)
- Lowering function accepts:
  - **Context:** used for type-related operations
  - **Builder:** an llvmlite builder for building LLVM IR
  - **Signature** of the function – args and return type
  - **Arguments** to the function, LLVM IR values which will have been constructed earlier by other lowering functions
- Lowering function returns:
  - **LLVM IR** implementing the operation

```
@lower_builtin(<FUNCTION>, *argtypes)
def my_lowering_function(context, builder, signature, args):
    # Using context and builder
    # Generate llvm_ir implementing FUNCTION
    # for the provided signature
    # Operating on the given args
    return llvm_ir
```

# LOWERING EXAMPLE - LEFT SHIFT ON INTS

- Compiling this function:

```
# x and y are int32 type
def f(x, y):
    return x << y
```

- Lowering for << on ints in Numba:

```
@lower_builtin(operator.lshift, Integer, Integer)
def int_shl_impl(context, builder, sig, args):
    [valty, amtty] = sig.args # int64, int64
    [val, amt] = args         # arg.x, arg.y
    val = context.cast(builder, val, valty, sig.return_type)
    amt = context.cast(builder, amt, amtty, sig.return_type)
    return builder.shl(val, amt)
```

- Generated LLVM IR:

(simplified)

```
define i32 @"f"(…, i32 %"arg.x", i32 %"arg.y")
{

  %".6" = sext i32 %"arg.y" to i64
  %".7" = sext i32 %"arg.x" to i64
  %".8" = shl i64 %".7", %".6"
}
```

# BUILDING LLVM IR

- Llvmlite IR builder functions:
  - Arithmetic / logical:  add, sub, mul, sdiv, udiv, and, or, xor, not, shl, lshr, ashr, cttz, ctlz, …
  - Comparisons: Integer compare, floating point compare
  - Branches and control flow: branch, if/then
  - Function call / return: call, invoke, ret
  - Aggregate operations: insert and extract values from structs and arrays
  - Memory operations: load, store, atomics, …
  - Inline assembly: PTX, for CUDA

- (Many not listed)
- Full list at: https://llvmlite.readthedocs.io/en/latest/user-guide/ir/ir-builder.html

# WORKED EXAMPLE

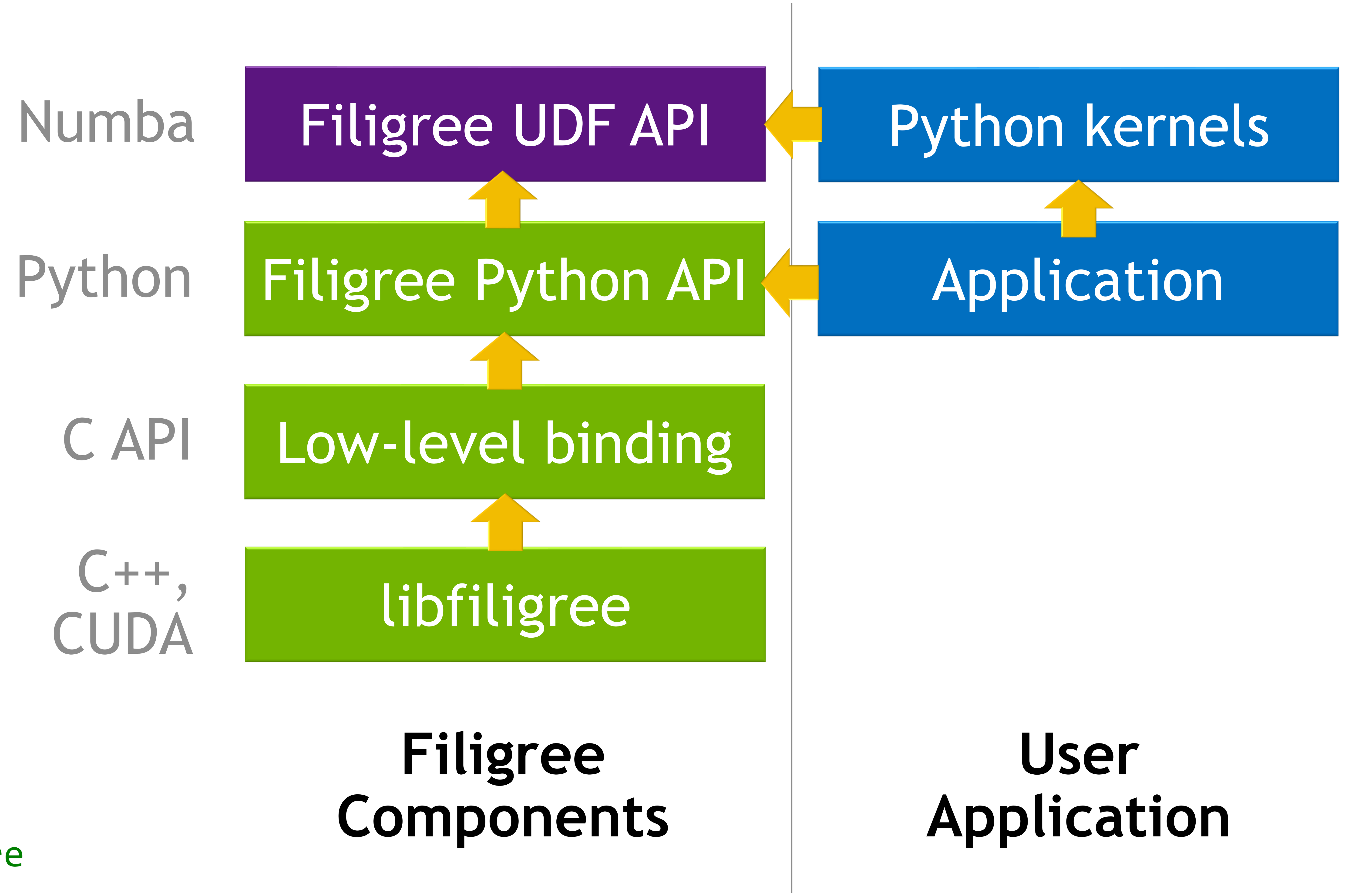
Implementing a UDF compiler for Filigree

- Typing
- Data model
- Lowering
- Integration with Python API
- Full source, completed example, notes:
  - https://github.com/gmarkall/numba-accelerated-udfs

# THE FILIGREE UDF COMPILER

- C++ headers / data structures / library + Python API
- Python API example that we're aiming to support:

```python
image = Image(path)

center_x = image.width / 2
center_y = image.height // 2
radius = max(center_x, center_y)

def highlight_center(pixel, x, y):
    distance_from_center = math.sqrt(
        abs(x - center_x)**2 + abs(y - center_y)**2)

    # Weight pixels according to distance from centre
    w = 1 - (distance_from_center / radius)

    # Apply weight to each RGB component
    return pixel.r * w, pixel.g * w, pixel.b * w, pixel.a

image.apply_located_pixel_udf(highlight_center)
image.to_pillow()
```



# TYPING: FILIGREE PIXEL STRUCTURES

```cpp
// C++ Types (from ImageMagick):

typedef unsigned short Quantum;

struct PixelPacket {
  Quantum blue;
  Quantum green;
  Quantum red;
  Quantum opacity;
};
```

```python
# Python PixelPacket
class PixelPacket:
    r: int
    g: int
    b: int
    a: int
```

```python
# Numba typing

quantum = types.uint16

class PixelPacket(types.Type):
    def __init__(self):
        super().__init__(name="PixelPacket")

@cuda_registry.register_attr
class PixelPacketAttrs(AttributeTemplate):
    key = pixel_packet

    def resolve_r(self, pp):
        return quantum

    def resolve_g(self, pp):
        return quantum

    def resolve_b(self, pp):
        return quantum

    def resolve_a(self, pp):
        return quantum
```

# TYPING FILIGREE PIXEL STRUCTURES (POINTER)

```cpp
// C++ Type (from Filigree)
class Image
{
  public:
    Image(std::string filename, /* other params */);
    ~Image();

    void to_greyscale();
    void to_binary();
    void to_weighted_binary();

    // Other methods omitted...

  private:
    rmm::mr::device_memory_resource *_mr;
    rmm::cuda_stream_view _stream;

    Magick::PixelPacket *_pixels;

    size_t _width;
    size_t _height;
    size_t _alloc_size;
};
```

```python
# Numba type
class PixelPacketPointer(types.RawPointer):
    def __init__(self):
        super().__init__(name="PixelPacket*")


pixel_packet_pointer = PixelPacketPointer()

# Reuse the r, g, b, a attributes
@cuda_registry.register_attr
class PixelPacketPointerAttrs(PixelPacketAttrs):
    key = pixel_packet_pointer

    # For dereferencing (pp a PixelPacketPointer):
    # pixel_packet = pp.values
    def resolve_values(self, pp):
        return pixel_packet
```

# TYPING THE FILIGREE IMAGE CLASS

```cpp
// C++ Type (from Filigree)
class Image
{
  public:
    Image(std::string filename, /* … */);
    ~Image();

    void to_greyscale();
    void to_binary();
    void to_weighted_binary();

    // Other methods omitted...

  private:
    rmm::mr::device_memory_resource *_mr;
    rmm::cuda_stream_view _stream;

    Magick::PixelPacket *_pixels;

    size_t _width;
    size_t _height;
    size_t _alloc_size;
};
```

```python
# Numba typing
class FiligreeImage(types.Type):
    def __init__(self):
        super().__init__(name="FiligreeImage")

# E.g. for: pixel = image[x, y]
@cuda_registry.register_global(operator.getitem)
class Image_getitem(CallableTemplate):
    def generic(self):
        def typer(image, indices):
            if not isinstance(image, FiligreeImage):
                return None
            if (not isinstance(indices, types.BaseTuple)
                    or len(indices) != 2):
                return None
            return signature(pixel_packet_pointer, image, indices)

        return typer
```

# DATA MODELS: PIXEL PACKET DATA MODEL

```cpp
// C++ Types (from ImageMagick):

typedef unsigned short Quantum;

struct PixelPacket {
  Quantum blue;
  Quantum green;
  Quantum red;
  Quantum opacity;
};
```

```python
# Numba data model

@register_model(PixelPacket)
class PixelPacketModel(models.StructModel):
    def __init__(self, dmm, fe_type):
        members = [
            ("b", quantum),
            ("g", quantum),
            ("r", quantum),
            ("a", quantum)
        ]
        super().__init__(dmm, fe_type, members)
```

# DATA MODELS: IMAGE DATA MODEL

```cpp
// C++ Type (from Filigree)
class Image
{
  public:
    Image(std::string filename, /* … */);
    ~Image();

    void to_greyscale();
    void to_binary();
    void to_weighted_binary();

    // Other methods omitted...

  private:
    rmm::mr::device_memory_resource *_mr;
    rmm::cuda_stream_view _stream;

    Magick::PixelPacket *_pixels;

    size_t _width;
    size_t _height;
    size_t _alloc_size;
};
```

```python
# Numba data model

@register_model(FiligreeImage)
class FiligreeImageModel(models.StructModel):
    def __init__(self, dmm, fe_type):
        members = [
            ("pixel_ptr", pixel_packet_pointer),
            ("width", types.uint64),
            ("height", types.uint64),
        ]
        super().__init__(dmm, fe_type, members)
```

# LOWERING PIXEL PACKET ATTRIBUTES

```python
# Numba data model

@register_model(PixelPacket)
class PixelPacketModel(models.StructModel):
    def __init__(self, dmm, fe_type):
        members = [
            ("b", quantum),
            ("g", quantum),
            ("r", quantum),
            ("a", quantum)
        ]
        super().__init__(dmm, fe_type, members)
```

```python
# Numba lowering

make_attribute_wrapper(PixelPacket, 'r', 'r')
make_attribute_wrapper(PixelPacket, 'g', 'g')
make_attribute_wrapper(PixelPacket, 'b', 'b')
make_attribute_wrapper(PixelPacket, 'a', 'a’)


# User code:

r, g, b, a = pp.r, pp.g, pp.b, pp.a
```

# LOWERING IMAGE GETITEM

- cgutils:
  - Code generation utilities for common patterns.
- In this example:
  - create_struct_proxy
  - unpack_tuple
  - gep_inbounds

```
# User code

def fun(image):
    x, y = cuda.grid(2)
    pixel = image[x, y]
```

```
# Numba lowering

@cuda_lower(operator.getitem, filigree_image, types.UniTuple)
def filigree_image_getitem(context, builder, sig, args):
    image_arg, index_arg = args
    image_ty, index_ty = sig.args
    image = cgutils.create_struct_proxy(image_ty)(context, builder,
                                                  value=image_arg)
    x, y = cgutils.unpack_tuple(builder, index_arg, count=2)

    x = context.cast(builder, x, index_ty[0], types.uint64)
    y = context.cast(builder, y, index_ty[1], types.uint64)
    offset = builder.add(builder.mul(y, image.width), x)

    ptr = builder.bitcast(image.pixel_ptr,
                          context.get_value_type(pixel_packet).as_pointer())
    current_pixel = cgutils.gep_inbounds(builder, ptr, offset)
    return current_pixel
```

# LOWERING SETTING OF PIXEL ATTRIBUTES

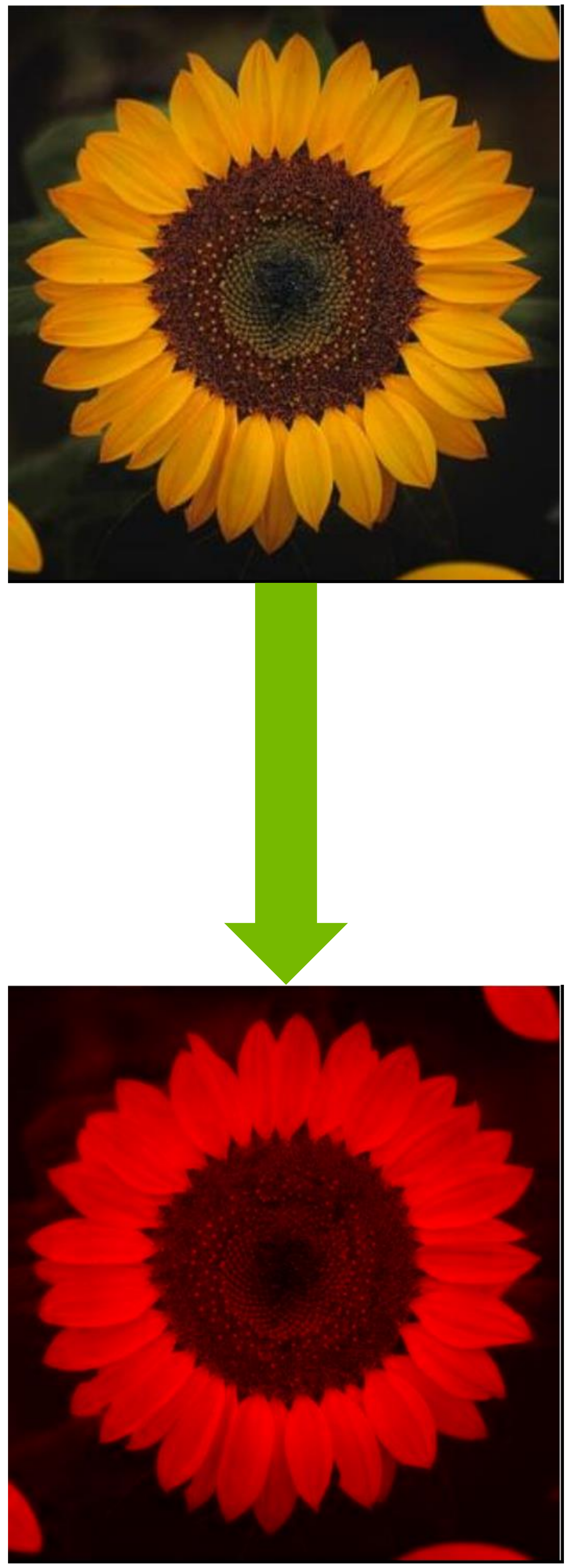
```python
# User code

def set_blue(image):
    x, y = cuda.grid(2)
    pixel = image[x, y]
    pixel.b = 127
```

```python
# Numba lowering

@cuda_lower_registry.lower_setattr_generic(pixel_packet_pointer)
def pixel_packet_pointer_set_attr(context, builder, sig, args, attr):

    # Load pixel data into struct proxy
    base_idx = context.get_constant(types.intp, 0)
    value_ptr = cgutils.gep_inbounds(builder, args[0], base_idx)
    data = builder.load(value_ptr)
    values = cgutils.create_struct_proxy(pixel_packet)(context, builder, value=data)

    # Set value. Alternative to:
    # if attr == 'r':
    #     values.r = args[1]
    # elif attr == 'g':
    #     values.g = args[1]
    setattr(values, attr, args[1])

    # Store struct value back to memory
    builder.store(values._getvalue(), value_ptr)
```

# SUPPORTED USE CASE SO FAR

- Given our typing, lowering, and data model, we can compile:

```python
@cuda.jit
def red_filter(pixel):
    # Strip out green and blue components
    return pixel.r, 0, 0, pixel.a

@cuda.jit
def my_transform(image):
    x, y = cuda.grid(2)
    pixel = image[x, y]
    r, g, b, a = red_filter(pixel.values)
    pixel.r = r
    pixel.g = g
    pixel.b = b
    pixel.a = a

image = Image('sunflower.png')
my_transform[grid_dim, block_dim](image)
```



- Need to add:
  - Mechanism to pass the Image object to a kernel
  - API for pixel-centric kernel launch

# PYTHON LAUNCH API - PASSING IN IMAGES

```python
# User code

def red_filter(pixel):
    # Strip out green and blue components
    return pixel.r, 0, 0, pixel.a

image = Image('sunflower.png')
Image.apply_located_pixel_udf(red_filter)


# Adaptor class

class FiligreeImageHandler:
    def prepare_args(self, ty, val, **kwargs):
        # Transform argument if it's an image
        # Otherwise, passthrough unchanged.
        if isinstance(val, api.Image):
            # Type + values match data model
            ty = types.UniTuple(types.uint64, 3)
            val = (val.pixel_ptr, val.width, val.height)
        return ty, val

filigree_image_handler = FiligreeImageHandler()

@cuda.jit(extensions=[filigree_image_handler])
```

```python
# From Filigree Python API:

class Image:
    def __init__(self, filename):
        # _lib is the Python C API wrapper
        # for libfiligree
        self._lib_image = _lib.Image(filename)

    @property
    def pixel_ptr(self):
        return self._lib_image.get_pixel_ptr()

    @property
    def width(self):
        return self._lib_image.get_width()

    @property
    def height(self):
        return self._lib_image   \
                   .get_height()
```

# PYTHON LAUNCH API - GENERATING THE APPLY KERNEL

```python
# User code

def red_filter(pixel):
    # Strip out green and blue components
    return pixel.r, 0, 0, pixel.a

image = Image('sunflower.png')
Image.apply_located_pixel_udf(red_filter)
```

```python
# API Implementation

def apply_located_pixel_udf(image, udf):
    device_function = cuda.jit(device=True)(udf)

    @cuda.jit(extensions=[filigree_image_handler])
    def apply_udf(image):
        x, y = cuda.grid(2)

        if x < image.width and y < image.height:
            pixel_ptr = image[x, y]
            r, g, b, a = device_function(pixel_ptr.values, x, y)
            pixel_ptr.r = r
            pixel_ptr.g = g
            pixel_ptr.b = b
            pixel_ptr.a = a

    nthreads_x = 16
    nthreads_y = 8
    nblocks_x = (image.width // nthreads_x) + 1
    nblocks_y = (image.width // nthreads_y) + 1
    grid_dim = (nblocks_x, nblocks_y)
    block_dim = (nthreads_x, nthreads_y)

    apply_udf[grid_dim, block_dim](image)
```

# LAUNCH API DESIGN CONSIDERATIONS

- Caching options:
  - `functools.lru_cache`:
    - Simple, but fails to cache user-generated functions
    - Just add `@lru_cache` decorator to kernel generation
  - Bytecode-based caching:
    - More complex, handles users generating functions
    - Used in cuDF and Numba’s on-disk cache
- Launch options:
  - Python: Used in Filigree, cuDF
  - C++: Used in PyOptiX, cuDF
    - Use `cuda.compile_ptx` to get PTX and invoke from C++
- See example repository for links

# CONCLUDING POINTS

- Evaluation
- Summary
- Resources

# PTX COMPARISON: RED FILTER KERNEL

```
.visible .entry my_transform(
    .param .u64 my_transform_param_0,
    .param .u64 my_transform_param_1,
    .param .u64 my_transform_param_2
)
{
    .reg .pred  %p<4>;
    .reg .b16   %rs<2>;
    .reg .b32   %r<9>;
    .reg .b64   %rd<11>;


    ld.param.u64    %rd3, [my_transform_param_0];
    ld.param.u64    %rd4, [my_transform_param_1];
    ld.param.u64    %rd5, [my_transform_param_2];
    mov.u32     %r1, %tid.x;
    mov.u32     %r2, %ctaid.x;
    mov.u32     %r3, %ntid.x;
    mad.lo.s32  %r4, %r3, %r2, %r1;
    mov.u32     %r5, %ctaid.y;
    mov.u32     %r6, %ntid.y;
    mov.u32     %r7, %tid.y;
    mad.lo.s32  %r8, %r6, %r5, %r7;
    cvt.u64.u32     %rd1, %r4;
    setp.ge.u64     %p1, %rd1, %rd4;
    cvt.u64.u32     %rd2, %r8;
    setp.ge.u64     %p2, %rd2, %rd5;
    or.pred     %p3, %p1, %p2;
    @%p3 bra    $L__BB0_2;

    cvta.to.global.u64  %rd6, %rd3;
    mul.lo.s64  %rd7, %rd2, %rd4;
    add.s64     %rd8, %rd7, %rd1;
    shl.b64     %rd9, %rd8, 3;
    add.s64     %rd10, %rd6, %rd9;
    mov.u16     %rs1, 0;
    st.global.u16   [%rd10], %rs1;
    st.global.u16   [%rd10+2], %rs1;

$L__BB0_2:
    ret;

}
```

```
.visible .entry my_transform(
    .param .u64 my_transform_param_0,
    .param .u64 my_transform_param_1,
    .param .u64 my_transform_param_2
)
{
    .reg .pred  %p<4>;
    .reg .b16   %rs<2>;
    .reg .b32   %r<9>;
    .reg .b64   %rd<11>;


    ld.param.u64    %rd3, [my_transform_param_0];
    ld.param.u64    %rd4, [my_transform_param_1];
    ld.param.u64    %rd5, [my_transform_param_2];
    mov.u32     %r1, %ntid.x;
    mov.u32     %r2, %ctaid.x;
    mov.u32     %r3, %tid.x;
    mad.lo.s32  %r4, %r2, %r1, %r3;
    mov.u32     %r5, %ntid.y;
    mov.u32     %r6, %ctaid.y;
    mov.u32     %r7, %tid.y;
    mad.lo.s32  %r8, %r6, %r5, %r7;
    cvt.u64.u32     %rd1, %r4;
    setp.gt.u64     %p1, %rd1, %rd4;
    cvt.u64.u32     %rd2, %r8;
    setp.gt.u64     %p2, %rd2, %rd5;
    or.pred     %p3, %p1, %p2;
    @%p3 bra    $L__BB0_2;

    mul.lo.s64  %rd6, %rd2, %rd4;
    add.s64     %rd7, %rd6, %rd1;
    cvta.to.global.u64  %rd8, %rd3;
    shl.b64     %rd9, %rd7, 3;
    add.s64     %rd10, %rd8, %rd9;
    mov.u16     %rs1, 0;
    st.global.u16   [%rd10], %rs1;
    st.global.u16   [%rd10+2], %rs1;

$L__BB0_2:
    ret;

}
```

# PTX COMPARISON: HIGHLIGHT_CENTER DEVICE FUNCTION

```
.visible .func  (.param .b32 func_retval0) highlight_center_py(
    .param .b64 highlight_center_py_param_0,
    .param .b32 highlight_center_py_param_1,
    .param .b32 highlight_center_py_param_2,
    .param .b32 highlight_center_py_param_3,
    .param .b32 highlight_center_py_param_4,
    .param .b32 highlight_center_py_param_5,
    .param .b32 highlight_center_py_param_6
)
{
    .reg .pred  %p<3>;
    .reg .b16   %rs<5>;
    .reg .b32   %r<2>;
    .reg .f64   %fd<11>;
    .reg .b64   %rd<15>;


    ld.param.u64    %rd1, [highlight_center_py_param_0];
    ld.param.u16    %rs1, [highlight_center_py_param_1];
    ld.param.u16    %rs2, [highlight_center_py_param_2];
    ld.param.u16    %rs3, [highlight_center_py_param_3];
    ld.param.u16    %rs4, [highlight_center_py_param_4];
    ld.param.u32    %rd2, [highlight_center_py_param_5];
    ld.param.u32    %rd3, [highlight_center_py_param_6];
    add.s64     %rd4, %rd2, -155;
    setp.lt.u64     %p1, %rd2, 155;
    mov.u64     %rd5, 155;
    sub.s64     %rd6, %rd5, %rd2;
    selp.b64    %rd7, %rd6, %rd4, %p1;
    add.s64     %rd8, %rd3, -162;
    setp.lt.u64     %p2, %rd3, 162;
    mov.u64     %rd9, 162;
    sub.s64     %rd10, %rd9, %rd3;
    selp.b64    %rd11, %rd10, %rd8, %p2;
    mul.lo.s64  %rd12, %rd7, %rd7;
    mul.lo.s64  %rd13, %rd11, %rd11;
    add.s64     %rd14, %rd13, %rd12;
    cvt.rn.f64.s64  %fd1, %rd14;
    sqrt.rn.f64     %fd2, %fd1;
    div.rn.f64  %fd3, %fd2, 0dC064400000000000;
    add.f64     %fd4, %fd3, 0d3FF0000000000000;
    cvt.rn.f64.u16  %fd5, %rs3;
    mul.f64     %fd6, %fd4, %fd5;
    cvt.rn.f64.u16  %fd7, %rs2;
    mul.f64     %fd8, %fd4, %fd7;
    cvt.rn.f64.u16  %fd9, %rs1;
    mul.f64     %fd10, %fd4, %fd9;
    st.f64  [%rd1], %fd6;
    st.f64  [%rd1+8], %fd8;
    st.f64  [%rd1+16], %fd10;
    st.u16  [%rd1+24], %rs4;
    mov.u32     %r1, 0;
    st.param.b32    [func_retval0+0], %r1;
    ret;
}
```

```
.visible .func highlight_center(
    .param .b64 highlight_center_param_0,
    .param .align 2 .b8 highlight_center_param_1[8],
    .param .b32 highlight_center_param_2,
    .param .b32 highlight_center_param_3
)


{
    .reg .pred  %p<3>;
    .reg .b16   %rs<5>;
    .reg .b32   %r<16>;
    .reg .f64   %fd<11>;
    .reg .b64   %rd<2>;


    ld.param.u32    %r1, [highlight_center_param_2];
    ld.param.u32    %r2, [highlight_center_param_3];
    ld.param.u16    %rs1, [highlight_center_param_1+6];
    ld.param.u16    %rs2, [highlight_center_param_1];
    ld.param.u16    %rs3, [highlight_center_param_1+2];
    ld.param.u16    %rs4, [highlight_center_param_1+4];

    add.s32     %r3, %r1, -155;
    mov.u32     %r4, 155;
    sub.s32     %r5, %r4, %r1;
    setp.lt.s32     %p1, %r3, 0;
    selp.b32    %r6, %r5, %r3, %p1;
    add.s32     %r7, %r2, -162;
    mov.u32     %r8, 162;
    sub.s32     %r9, %r8, %r2;
    setp.lt.s32     %p2, %r7, 0;
    selp.b32    %r10, %r9, %r7, %p2;
    mul.lo.s32  %r11, %r6, %r6;
    mad.lo.s32  %r12, %r10, %r10, %r11;
    cvt.rn.f64.u32  %fd1, %r12;
    sqrt.rn.f64     %fd2, %fd1;
    div.rn.f64  %fd3, %fd2, 0dC064400000000000;
    add.f64     %fd4, %fd3, 0d3FF0000000000000;
    cvt.rn.f64.u16  %fd5, %rs4;
    mul.f64     %fd6, %fd4, %fd5;
    cvt.rzi.u32.f64     %r13, %fd6;
    ld.param.u64    %rd1, [highlight_center_param_0];
    st.u16  [%rd1+4], %r13;
    cvt.rn.f64.u16  %fd7, %rs3;
    mul.f64     %fd8, %fd4, %fd7;
    cvt.rzi.u32.f64     %r14, %fd8;
    st.u16  [%rd1+2], %r14;
    cvt.rn.f64.u16  %fd9, %rs2;
    mul.f64     %fd10, %fd4, %fd9;
    cvt.rzi.u32.f64     %r15, %fd10;
    st.u16  [%rd1], %r15;
    st.u16  [%rd1+6], %rs1;
    ret;
}
```

# SUMMARY AND RESOURCES

- Using Numba to compile Python code for CUDA solves the ***“Impedance mismatch”*** problem:
  - Python Users can ***extend accelerated applications***
  - By writing their UDFs in Python they sustain their own ***productivity*** and ***workflow***
  - And they can get ***high performance code*** generated equivalent to that from CUDA C++ with NVCC.

- Github Repository for this talk: 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
- 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
- Numba documentation:
  - 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
- Contact:
  - Numba real-time chat: https://gitter.im/numba/numba
  - Numba Discourse forums: https://numba.discourse.group/
  - Email / Twitter: gmarkall@nvidia.com / https://twitter.com/gmarkall

# SUMMARY AND RESOURCES

- Using Numba to compile Python code for CUDA solves the ***"Impedance mismatch"*** problem:
  - Python Users can ***extend accelerated applications***
  - By writing their UDFs in Python they sustain their own ***productivity*** and ***workflow***
  - And they can get ***high performance code*** generated equivalent to that from CUDA C++ with NVCC.

- Github Repository for this talk: 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
- 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
- Numba documentation:
  - 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
- Contact:
  - Numba real-time chat: https://gitter.im/numba/numba
  - Numba Discourse forums: https://numba.discourse.group/
  - Email / Twitter: gmarkall@nvidia.com / https://twitter.com/gmarkall

# SUMMARY AND RESOURCES

- Using Numba to compile Python code for CUDA solves the ***“Impedance mismatch”*** problem:
  - Python Users can ***extend accelerated applications***
  - By writing their UDFs in Python they sustain their own ***productivity*** and ***workflow***
  - And they can get ***high performance code*** generated equivalent to that from CUDA C++ with NVCC.

- Github Repository for this talk: 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
- 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
- Numba documentation:
  - 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
- Contact:
  - Numba real-time chat: https://gitter.im/numba/numba
  - Numba Discourse forums: https://numba.discourse.group/
  - Email / Twitter: gmarkall@nvidia.com / https://twitter.com/gmarkall


NVIDIA