GPUs from Python

- The deep learning frameworks are on way to access GPUs from Python
- If those were the perfect way to express any algorithm, we would write all our CPU computations using them as well
  - So maybe they are not ideal, after all

MinPy

- A library tying together MXNet support and a Numpy API

- Ideally, it’s as simple as replacing

```
import numpy as np
```

with

```
import minpy as np
```

Some pitfalls exist when you’re exchanging numpy arrays with other modules, and using certain features.

GPU support

- Minpy will, by default, use MXNet when present
- So, in theory, you can just try running the same piece of code with `numpy` or `minpy` imported
- Important caveat
  - Functions on arrays that modify the array itself are generally not allowed
  - Whether that has a huge effect or not depends a lot on your coding style
- Many functions exist as method working on the array, and as separate functions in the `numpy` module

Autograd

- MinPy also has autograd support. One of their examples:

```
from minpy.core import grad

# define a function: f(x) = 5*x^2 + 3*x - 2
def foo(x):
    return 5*(x**2) + 3*x - 2

# f(4) = 90
print(foo(4))

# get the derivative function by `grad`: f'(x) = 10*x + 3
d_foo = grad(foo)

# f'(4) = 43.0
print(d_foo(4))
```

Autograd

- This works if the return value is a vector or matrix, rather than a scalar, as well.
- If you have a function taking multiple parameters, you need to specify the indices of the parameters to `grad`

```
def foo(x,y):
    return 5*(x**2) + 3*y - 2

d_foo = grad(foo, [0, 1])

print(d_foo(4,6))
```

- The related function `grad_and_loss` will create a function that returns a tuple of the gradient and the plain return value.

afnumpy

- `afnumpy` is a package with a similar goal as minpy, but focusing solely on numpy equivalents for GPU.
  – Developed here in Uppsala, a wrapper around the ArrayFire package for the actual GPU computations.
  – Since no autograd etc is involved, the performance might sometimes be better.
- Far less of a community than MinPy, and even MinPy is pretty small.

- Huge selling point of both: Small changes to existing Python code!

numba

- `numba` is a general package for compiling Python code
- It uses the LLVM compiler backend (which is also used for the Clang compiler) and converts Python code to something LLVM can understand.
- LLVM can generate Nvidia assembler.
  - As already mentioned, Clang can compile most Cuda code.
- Two main usage modes of numba for GPU:
  - ufunc-like objects
  - Cuda kernels in a Python subset

General numba

- Just put `@jit` at the function you want to go fast

```
from numba import jit

@jit
def f(x, y):
    return x + y
```

- If you call a Python function numba doesn't know about (like one of your own non-`@jit` functions), it will suddenly be slow again. You can put `nopython=True` to get an error in that case.

```
from numba import jit

@jit(nopython=True)
def f(x, y):
    return x + y
```

`nogil`, `fastmath`, `parallel`

- The standard Python implementation uses the global interpreter lock (GIL), basically meaning that only a single thread can run actual Python code at a time
  - Some libraries release this lock during large operations, allowing multiple threads to run efficiently
  - Releasing the lock in the jitted code allows for some synchronization errors that would not occur, or be very rare, in ordinary Python code.
- `fastmath` allows a set of speed-ups for floating point that compilers also tend to enable with flags with similar names, including things like allowing (a + b) + c be rearranged into a + (b + c), if that's beneficial.
- `parallel` instructs Numba to try to parallelize a single function if there is e.g. a long loop or a large array operation
  - You need to use `prange` rather than `range` to state that loop iterations are independent.
  - `@jit(nopython=True, parallel=True, nogil=True, fastmath=True)`

Eager compilation

- You can precompile the code if you specify beforehand what types it is intended for

```
@jit(int32(int32, int32))
def f(x, y):
    return x + y
```

ufuncs

- Numpy supports ufuncs, universal functions, which adhere to the overall conventions of numpy (including broadcasting) while iterating elementwise over one or multiple arrays
- These are normally written in some compiled language
- numba can support compiling an ordinary scalar function as a ufunc

```
@vectorize([int32(int32, int32),
            int64(int64, int64),
            float32(float32, float32),
            float64(float64, float64)])
def f(x, y):
    return x + y
```

Vectorization targets

- Vectorize supports several targets

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

- `cpu` – default, best for small data (1000 elements?)
- `parallel` – multiple CPU threads, for maybe 100000 elements
- `cuda` – transfer data to the GPU and back behind-the-scenes
- `roc` – transfer data to a supported AMD GPU in the same way

- ufuncs support many features out-of-the-box in numpy, like calling `myufunc.reduce(a,b)` in order to compute the ufunc over `a` and `b` and then sum the results

Actual Cuda kernels in numba

- You can also write explicit Cuda kernels in the numba Cuda subset of Python directly
  - In these, you get `numba.cuda.threadIdx`, `numba.cuda.blockDim`, `numba.cuda.blockIdx`, `numba.cuda.gridDim`, just like in the C++ CUDA
  - Also two helpers `numba.cuda.grid` and `numba.cuda.gridsize` to get absolute indices, combining the grid and block dimensions

Increment each element in an array by one

```
@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

threadsperblock = (16, 16)
blockspergrid_x = math.ceil(an_array.shape[0] / threadsperblock[0])
blockspergrid_y = math.ceil(an_array.shape[1] / threadsperblock[1])
blockspergrid = (blockspergrid_x, blockspergrid_y)
increment_a_2D_array[blockspergrid, threadsperblock](an_array)
```

- We’re using brackets rather than triple angle brackets for specifying grid and block size

Data transfers and memory

- Array parameters to kernels are transferred automatically to and from the GPU.
- For more efficient code, explicit allocations can be made using `numba.cuda.device_array`
- The kernel can access shared memory using `numba.cuda.shared.array`
- Read/writes to the same position shared memory from different threads need `numba.cuda.syncthreads()`, just like in “real” CUDA

Atomics

- A subset of atomics are supported, on indices within arrays, from `numba.cuda.atomic`
  - `add`
  - `compare_and_swap`
  - `max`
  - `min`

Reduction

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

sum_reduce = cuda.reduce(lambda a, b: a + b)
```

- Both of these create a callable object that will accept a full array, and combine all elements in the specified reduction operation.
  - The reduction operation does not have to be addition, of course, but any Python code combining elements.

Debugging

- Debugging Python code is far easier when it is normal Python
  - The numba Cuda support adds thread indices and other things that are simply not part of normal Python
  - numba provides a facility for running the kernel within the normal Python interpreter for this
- Involves setting the evinronment variable `NUMBA_ENABLE_CUDASIM` to 1 before loading numba

What to recommend?

- Do you have an existing, fast, numpy code that you want to run on GPU?
  - Try `MinPy`.
- Do you have an existing Python code with performance problems?
  - Try `numba`, both on CPU and GPU!
  - Numba also provides support for the proprietary AMD GPU API in a similar manner