Fundamentals of Accelerated Computing with Python

Notes inspired by NVIDIA DLI

April 9, 2021

1 Introduction to CUDA python with Numba

1.1 Compile for the CPU

The Numba compiler is typically enabledd by applying a **function decorator** to a Python function.

Decorators are function modifiers that transform the Python function they decorate.
Example:

```
from numba import jit
...
@jit
def func(x,y):
...
```

Numba saves the original Python implementation of the function using the following attribute:

```
.py_func()
```

1.2 Benchmarking

In a Jupyter notebook, use the `\%timeit` magic function to measure the speed of the code.

to annotate code with data types use

```
.inspect_types()
```

1.3 Object and nopython Modes

As of right now, Numba annot compile some Python code, such as dictionaries, and some functions.

If non-compilable code is encountered, Numba will fall back to **Object Mode** which does not do type-specialization.

You can force nopython mode by passing the `nopython` argument to the decorator:

```
@jit(nopython=True)
```

Using nopython mode is the recommended and best practice way to use jit as it leads to best performance

1.4 Introduction to Numba for the GPU with NumPy Universal Functions (ufuncs)

1.4.1 Quick overview of ufuncs

ufuncs are functions that can take NumPy arrays of varying dimensions, or scalars, and operate on them element by element.

Arrays of different, but compatible dimensions can also be combined via a technique called broadcasting.

1.5 Making ufuncs for the GPU

Numba has the ability to create *compiled* ufuncs. For instance, you can implement a scalar function to be performed on all the inputs, decorate it with `@verctorize`, and Numba will figure out the broadcasting rules.

We can generate a ufunc that uses CUDA on the GPu with the addition of giving an **explicit type signiture** and setting the target *attribute*

```
'return_value_type(argument1_value_type, argument2_value_type, ...)'
```

For instance, the ufunc below expects two int64 values and also returns an int64 value:

```
# Type signature and target are required for the GPU
@vectorize(['int64(int64, int64)'], target='cuda')
def add_ufunc(x, y):
    return x + y
```