

# Kernel Implementation API:

This workbook is about how we use Numba with DEC++ pipeline to write Python apps that will reap the most benefits from accelerations that can be gained through the compilation for optimal memory access and optimal processor mapping and how you can utilize the Decades Numba Library for faster implementation of your algorithms.

To learn more about how to execute your kernels using the DEC++ pipeline please read the [kernel\\_execution](#) ([kernel\\_execution.ipynb](#)) document.

We will first give a short introduction to how to use Numba with Python, please find more detailed information about how to use Numba [here \(https://numba.pydata.org\)](https://numba.pydata.org). We will then describe the DECADES Numba Library in detail to help you get familiarized with this toolbox.

Finally, we will have a short section for tips and tricks for adapting your code to Numba.

Let's first load a few useful libraries. Note that unless you launch the Jupyter notebook from a computer with the full environment, you cannot execute all of the cells below successfully.

```
import numpy as np
import numba
from numba import jit, njit, float32, int32, int64, jitclass
from time import time
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

%load_ext autoreload
%autoreload 2
```

## 1. Numba:

Numba is a specialized compiler for Python apps for optimizing numerically heavy code. The feature of Numba that we are most interested in is that Numba compiles a Python code by using an LLVM compiler library (more about LLVM [here \(https://llvm.org\)](https://llvm.org)). It therefore at some point during compilation generates an LLVM IR file. This is useful for us, because DEC++ compiler is also based on LLVM architecture and we can hook DEC++ compilation into the Numba compilation path to optimize Python code for specialized hardware.

Given the limited set of variable types and functions Numba can handle at the moment, not every Python function can be Numba-compiled. We will use Numba only for functions that rely on numerical computations and that have heavy cpu usage. You can find out more about Numba [here \(http://numba.pydata.org/numba-doc/latest/user/5minguide.html\)](http://numba.pydata.org/numba-doc/latest/user/5minguide.html).

Python interface of Numba allows wrapping of a given Python function such that it will first be fed into the Numba compiler before execution.

You can see an example here:

```
@jit(nopython=True)
def matrixSum_numba(A, B):
    m, n = A.shape
    X = np.zeros_like(A)

    for i in range(m):
        for j in range(n):
            X[i, j] = A[i, j] + B[i, j]

    return X
```

```
def matrixSum(A, B):
    m, n = A.shape
    X = np.zeros_like(A)

    for i in range(m):
        for j in range(n):
            X[i, j] = A[i, j] + B[i, j]

    return X
```

```

X1 = np.random.uniform(low=0, high=1, size=10000
00).reshape(1000, 1000)
X2 = np.random.uniform(low=0, high=1, size=10000
00).reshape(1000, 1000)

t = time()
Z = matrixSum(X1, X2)
print('compute time without numba', time() - t)
t = time()
Z = matrixSum_numba(X1, X2)
print('compute time with numba', time() - t)

```

```

compute time without numba 0.3348517417907715
compute time with numba 0.19780993461608887

```

Notice that the only difference in writing the code with and without Numba compiler is the presence of the decorator `@jit(nopython=True)`. That simple! The `@jit()` decorator allows the function to be wrapped in Numba compiler.

## 1.1 nopython = True

Numba can compile in two modes: object mode and the nopython mode. When the `nopython=True` is set, it will run in *nopython* mode and the code will be compiled entirely without the involvement of the Python Interpreter.

This is an important setting for us. When `nopython=True` is not set, in the event that Numba fails to compile, `@jit()` will use the Python interpreter without giving an error. For our applications we always want to make sure that Numba compiles the function entirely.

As a quick note, `@jit(noython=True)` can also be written shortly as `@njit()`.

## 1.2 @jit vs @cc

Another mode of compilation for Numba is the distinction between just-in-time (JIT) and ahead-of-time (AOT) compiler and these have different decorators and syntaxes.

See example below for AOT mode:

```
from numba.pycc import CC

cc = CC('matrixSum')

@cc.export('sumf', 'f8[:,::1](f8[:,::1], f8[:,::1])')
@jit(nopython=True)
def matrixSum(A,B):
    m,n = A.shape
    X = np.zeros_like(A)

    for i in range(m):
        for j in range(n):
            X[i,j] = A[i,j] + B[i,j]

    return X

cc.compile()
```

While in JIT mode, Numba will compile the function on the fly during execution. AOT mode on the other hand will generate an binary executable that can be imported later. In the AOT mode, the compiler must know all possible versions of input and output variable types for compilation. In the case of JIT however, Numba can dynamically determine the variable type based on the input type provided.

Although we would have liked to use the AOT, this functionality of Numba is not as well developed as the JIT mode. We will therefore be using the JIT, however we will provide all the input and output variable types just as it is done in the AOT mode. This will ensure that when we introduce the DEC++ compiler into the Numba pipeline it will run smoothly.

Now let's take a look at what that looks like:

```

from numba import int32, njit
from numba.types import UniTuple

@njit(UniTuple(int32, 2)(int32, int32), nogil=True)
def k_n_to_i_j(k, n):
    #function definition here
    #
    return i, j

```

The `nogil=True` setting is to make sure that Python's Global Interpreter Lock is released and Numba and DEC++ can now run the code on multiple threads.

### 1.3 @jitclass

The last type of decorator we will talk about is `jitclass()` which is a Numba compiler for class type objects. As usual all the variables belonging to a class must be specified in the decorator. Here is an example of Numba compiled class:

```

from numba import jitclass, int32

spec = [('indptr', int32[:]), ('indices', int32[:])]

@jitclass(spec)
class SparseGraph:
    def __init__(self, graph_indptr, graph_indices):
        self.indptr = graph_indptr
        self.indices = graph_indices

```

## 2. DEC++ Pipeline:

Adding the DECADES pipeline is fairly simple. We simply need to import the DEC\_Pipeline module and then use `pipeline_class=DEC_pipeline` in our decorator settings. Let's now see the same code above with the DEC++ pipeline:

```
import numpy as np
from numba import jitclass, int32, njit, float32
from numba.types import UniTuple
from DEC_Pipeline import DEC_Pipeline

@njit(UniTuple(int32, 2)(int32, int32), nogil=True, pipeline_class=DEC_Pipeline)
def k_n_to_i_j(k, n):
    #function definition here
    #
    return i, j
```

The above code will now be compiled through DEC++. You can learn more about specific kernel execution settings [here](#). ([kernel\\_execution.ipynb](#))

## 3. DEC\_Numba\_Lib :

To facilitate rapid development in Python using the DECADES pipeline, we developed a DECADES Numba Library focusing on the graph applications. We call this library DEC\_Numba\_Lib. You can import this library using `import DEC_Numba_Lib`. There are four different types of support we provide through this library:

### 3.1 Load Data:

Currently , we provide two functions `LoadDecSparseGraph` and `LoadDecBipartiteGraph` to load graph data into Numba compiled graph class objects, both of which take data directory path as input. We plan on building on this library to further enable rapid development with Python for Numba DEC++ compilation.

With `LoadDecSparseGraph` the edge and `node_attribute` data are loaded into a graph object (G) that has the edge data in CSR matrix format (G.indptr, G.indices, similar to SciPy csr matrix format) as well as node attributes (G.node\_attr).

`LoadDecBipartiteGraph` on the other hand loads the edge data from a bipartite graph to a triangular matrix.

Feel free to use these functions as well as the Numba-compiled `DecSparseGraph` and `DecBipartiteGraph` jitclasses they return as a basis to develop more complex graph data structures and Numba-compiled graph applications you might be developing.

### 3.2 Data structures:

We have the following data structures in `DEC_Numba_Lib`: Graph Data Classes:

- **DecSparseGraph:** Graph object (see above) built as a Numba-compiled class with CSR matrix structure for adjacency matrix as well as node attributes.
- **BiPartiteGraph:** Graph object built as a numba-compiled class specialized for bipartite graphs with a triangular matrix structure for edge data.

Here is an example using `DecSparseGraph` module of DECADES Numba Library:

```
import numpy as np
from numba import njit
from numba.types import Tuple

from DEC_Pipeline import DEC_Pipeline
from DEC_Numba_Lib import DecSparseGraph, LoadDecSparseGraph, DecSparseGraphSpec

@njit(Tuple((int32, float32))(DecSparseGraphSpec(), int32[:], int32), nogil=True, pipeline_class=DEC_Pipeline)
def vertex_nomination_kernel_(G, seeds):
    # function definition here
    #
    return top_nominee, top_score

G = LoadDecSparseGraph(input_directory)
seeds = np.random.choice(num_nodes_connected, 5, replace=False)
top_nominee, top_score = vertex_nomination_kernel_(G, seeds)
```



Because Numba does not support SciPy library, we included Numba implementations of some of the most popular SciPy data structures and functions (described in the next section):

SciPy like Sparse Matrix Classes ([wikipedia](https://en.wikipedia.org/wiki/Sparse_matrix) ([https://en.wikipedia.org/wiki/Sparse\\_matrix](https://en.wikipedia.org/wiki/Sparse_matrix))):

- **DecCSR:** Compressed Sparse Row format with row-wise stacked vector form matrix data. The class object has `indptr`, `indices`, `data`, `shape` and `size` instance variables.
- **DecCSC:** Compressed Sparse Column format with column-wise stacked vector form matrix data. The class object has `indptr`, `indices`, `data`, `shape` and `size` instance variables.
- **DecCOO:** Coordinate list format (row index, column index and data are stored in 3 different vectors). The class object has `rows`, `cols`, `data`, `shape` and `size` instance variables.

For all the data structures we provide special functions to retrieve class signatures to be used in the `njit` decorators of functions that would use these data as input (e.g. the spec for the `DecSparseGraph` class can be retrieved with `DecSparseGraphSpec()`). This removes the overhead of heavy variable typing prior to function calls and results in cleaner code.

You can load the graph data using `LoadDecSparseGraph` and build these different sparse matrices using instance variables of the resulting `DecSparseGraph`.

In the next section we describe the special functions in the `DEC_Numba_Lib` targeting these data structures such as conversion between different data structures and basic matrix operations.

```

import numpy as np
from numba import njit
from numba.types import Tuple

from DEC_Pipeline import DEC_Pipeline
from DEC_Numba_Lib import DecSparseGraph, LoadDecSparseGraph, DecSparseGraphSpec

njit(Tuple((int32, float32))(DecSparseGraphSpec(
), int32[:], int32), nogil=True, pipeline_class=DEC_Pipeline)
def vertex_nomination_kernel_(G, seeds):
    # function definition here
    #
    return top_nominee, top_score

G = LoadDecSparseGraph(input_directory)
seeds = np.random.choice(num_nodes_connected, 5,
replace=False)
top_nominee, top_score = vertex_nomination_kernel_(G, seeds)

```

### 3.3 Kernel Functions:

Matrix format conversion:

- **coo\_to\_csr**
- **csr\_to\_coo**
- **csr\_to\_csc**
- **csc\_to\_csr**

Matrix operations:

- *Transpose*: **transpose\_csc**, **transpose\_csr**, **transpose\_coo**
- *Dot Product*: **dot** This function is capable of handling any combination of DecCSR, DecCSC, DecCOO and dense numpy matrices. Please note however, due to a Numba [bug \(https://github.com/numba/numba/issues/3590\)](https://github.com/numba/numba/issues/3590) this function currently is on hold. You can however access the underlying specific implementations for different combinations of matrix data types directly inside the DEC\_Numba\_Lib module.
- *Hadamard Product*: **multiply** (capable of handling

elementwise multiplication of any combination of DecCSR, DecCSC, DecCOO and dense numpy matrices)

- *Elementwise operations of numpy matrices:* Although numpy is supported, we also provide the numba compiled elementwise operations of numpy matrices in this library (**dec\_mul\_scalar**(scalar-matrix multiplication), **dec\_add**(elementwise addition), **dec\_subt**(elementwise subtraction), **dec\_div**(elementwise division), **dec\_dot**(dot product), **dec\_int64\_max**(find maximum))

Other miscellaneous operations:

- **dec\_minimum\_spanning\_tree:** finds the minimum spanning tree given a DecCSR graph data structure using Kruskal's algorithm.
- **eliminate\_zeros:** eliminates explicit zeros from DecCSR Matrix.
- **list\_intersection:** find intersection of two lists, especially useful for common neighbor search.
- Triangular matrix helper functions: **TriDenseGraph\_k\_to\_i\_j**, **TriDenseGraph\_num\_ele\_i\_rows**)

### 3.4 Synchronization:

Finally, we also provide the **PyDECADES\_barrier** function to synchronize all the threads in order to preserve memory access order in the [multi-threaded operations \(kernel\\_execution.ipynb\)](#). This can be achieved by inserting `PyDECADES_barrier()` in the main kernel Python app to the point in the program where all the threads must complete operations in order to proceed to the next set of instructions.

## Appendix 2 Tips and tricks with Numba:

- **Most common error you will encounter:** By far the most common error you will come across is going to be due to variable type settings. You will get an error similar to

"TypeError: No matching definition for argument type(s) array(float32, 1d, C), array(int64, 1d, C), array(int64, 1d, C)" in such cases. You need to make sure that Python and Numpy variables are cast into specific data types your Numba function expects. Numba does not give very good description of the issues in the error, so you might need to do some detective work to understand which variable or function it is complaining about.

- **Lists versus numpy arrays:** Although Numba supports lists, note that it does not support list of lists. It would be a wise to in general use numpy arrays instead of lists. This way, you will be less prone to coming across errors and bugs related to whether or not Numba can handle the python functions associated with lists. There may however be situations where using lists is much more convenient, in those cases make sure the types of the variables inside the list are compatible with operations you will do with them.
- **Variable types in Numba:** Since you will need to declare the type of the input and output variables of your njit function, let's talk about different variable types you might have to deal with. As we said above, safest type of variables to use in Numba are Numpy variables where most of the methods associated with these variables are supported.

Commonly used variable types are as follows:

int32	integer variable
int32[:]	integer array
int32[:,:]	2D integer array
int32[:,::1]	2D integer array, packed row-wise
...	...

where int32 can be replaced by int8, int16, int64, float32 and float64. The output variables must be wrapped in a Numba Tuple if there are more than one.

You can also use jitclasses as input and output variables and in that case you need to capture the class signature by adding `.class_type.instance_type` at the end of a class name and use this signature as variable type.

For more information on different variable types of Numba, please refer [here](https://numba.pydata.org/numba-doc/dev/reference/types.html) (<https://numba.pydata.org/numba-doc/dev/reference/types.html>).

