Kernel Implementation API:

This workbook shows how we use Numba as part of the DEC++ pipeline to write Python applications that will reap the most benefits from various features of the DECADES platform. Performance gains can be achieved through compilation for optimal memory access and optimal processor mapping, and the DECADES Numba Library allows for faster implementations of your Python algorithms.

To learn more about how to execute your kernels using the DEC++ pipeline please read the Decades Numba Pipeline (Decades_Numba_Pipeline.ipynb) document.

We will first give a short introduction on how to use Numba with Python. You can find more detailed information about how to use Numba here (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, int
64, jitclass
from time import time
from IPython.core.interactiveshell import Intera
ctiveShell
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 its use of LLVM to compile Python code (read 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 the LLVM architecture and we can insert 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)

The Python interface of Numba allows wrapping of a given Python function such that it will first be fed into the Numba compiler before execution. Wrapping a function involves placing a `@jit` annotation above it.

For example, let's start with a matrix addition function:

```
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
```

Now let's make this function Numba-compilable:

```python
@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
```

```python
X1 = np.random.uniform(low=0, high=1, size=1000000).reshape(1000, 1000)
X2 = np.random.uniform(low=0, high=1, size=1000000).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 the Numba compiler is the presence of the decorator *@jit(nopython=True)*. It's that simple! The *@jit()* decorator allows the function to be wrapped for the 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 synthaxes.

Here is an example of AOT mode:

```
from numba.pycc import CC

cc = CC('matrixSum')

@cc.export('sumf', 'f8[:,::1](f8[:,::1], f8[:,::1])')
@jit(nopython=True)
def matrixSum_AOT(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, `spec` :

```
from numba import jitclass,int32

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

@jitclass(spec)
class SparseGraph:
    def __init__(self, graph_indptr, graph_indic
es):
        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=Tr
ue, 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. (Decades_Numba_Pipeline.ipynb)](Decades_Numba_Pipeline.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:

For graph applications the best data structure to use with DECADES is sparse matrices. Since Numba does not support SciPy library, we included Numba implementations of some of the most popular SciPy data structures and functions (decribed in the next section):

SciPy like Sparse Matrix Classes ([wikipedia (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.

In addition, we have the following data structures in DEC_Numba_Lib for specialized 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 atributes.

- **BiPartiteGraph:** Graph object built as a numba-compiled class specialized for bipartite graphs with a triangular matrix structure for edge data.

For graph data that requires node_attribute data to be loaded, we have LoadSparseGraph function that loads edge data as well as node attributes into a graph object (G): the edge data is in CSR matrix format (G.indptr, G.indices, similar to SciPy csr matrix format) and the node attributes are included as an array(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.

Here is how you would load the data into one of these structures:

(i) Load data into DecSparseGraph:

You need to load data from an input directory that contains:

- adjacency list as: edge_list.tsv (tab separated file, with two columns, first: origin node, second: destination node)
- node attributes as: node_attributes.tsv

```
G1 = LoadDecSparseGraph(input_directory)
```

(ii) Load data into DecCSR:

You need to load the data using arrays required to build a sparse matrix (you can use scipy formed sparse matrix for this) `G2 = DecCSR(A_indptr, A_indices, A_data, A_shape)`

where either you can build the above arrays with numpy yourself as per description of a CSR from Wikipedia or use a scipy CSR object A and use A.indptr, A.indices, A.data, A.shape for your inputs.

If you have a dictionary data as a starting point, you can build the above arrays by going through the edge list and appending column indices and edge_weights into the corresponding positions in the initialized numpy arrays, where in the same loop increment corresponding element for the node(row) in the indptr numpy array, which needs to end up with a cumulatively summed number of edges per node. Here is an example:

```
    indptr = np.zeros(num_nodes+1)
    indices = np.zeros(num_edges)

    for n in range(num_nodes):
        if n in graph.keys():
            node = graph[n]
        else:
            node = []

        indptr[n+1] = len(node) + indptr[n]
        indices[int(num_edges):int(num_edges+len(nod
e))] = node
        num_edges = num_edges + len(node)

    indices = indices[0:int(num_edges+1)]
    data = np.ones(num_edges)

    indptr = indptr.astype('int64')
    indices = indices.astype('int64')
    data = data.astype('float64')

    dec_graph = DecCSR(indptr, indices, data, shape)
```

if you have an edge list you can use:

`G2 = DecCOO(A_rows, A_cols, A_data, A_shape)`

where you can enter the row indices, column indices and data as numpy arrays as well as the desired shape of the matrix (`2x1`). You can also load the data into a scipy COO first and then use the object elements A.row, A.col, A.data, A.shape as input for the DecCOO.

```
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, LoadDe
cSparseGraph, DecSparseGraphSpec, DecCSR

@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


# load data from an input directory that contain
s:
#       adjacency list as: edge_list.tsv  (tab s
eparated file, with two columns, first: origin n
ode, second: destination node)
#       node attributes as: node_attributes.tsv
G1 = LoadDecSparseGraph(input_directory)

# load data using arrays required to build a spa
rse matrix (you can use scipy formed sparse matr
ix for this)
G2 = DecCSR(A_indptr, A_indices, A_data, A_shape
)
# where either you can build the above arrays wi
th numpy yourself as per description of a CSR fr
om Wikipedia
# or use a scipy CSR object A and use A.indptr,
A.indices, A.data, A.shape)

#alternatively if you have an edge list you can
use:
G2 = DecCOO(A_rows, A_cols, A_data, A_shape)

# where you can enter the row indices, column in
dices and data as numpy arrays as well as the de
sired shape of the matrix (2x1)
# you can also load the data into a scipy COO fi
rst and then use the object elements A.row, A.co
l, A.data, A.shape as input for the DecCOO.


# you can now use this data structure in your Nu
mba-compilable function:
seeds = np.random.choice(num_nodes_connected, 5,
```

```
replace=False)
top_nominee, top_score = vertex_nomination_kerne
l_(G, seeds)
```

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) 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.

- *Elemenwise Operations of sparse matrices:*

elementwise_sparse_sparse, **elementwise_dense_sparse**, **elementwise_sparse_dense** (capable of handling elementwise multiplication of any combination of DecCSR, 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 (Decades_Numba_Pipeline.ipynb)](Decades_Numba_Pipeline.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.

- **Python default integer type:** Numba type of a python generated default integer is int64. You may need to cast it to a different type if this is not what you wanted in the numba code. You can do `np.int32(x)` if you want to cast it to int32 for instance.

- **Assigning values to output array:** When you construct a kernel to be launched by the `decades_kernel_launcher` , you will need to provide the return output as an initialized input. If this variable is an array, in order to assign values to the entire array you would normally do `returnarray = ...` However this will result in creation of a new object inside the numba function rather than changing its content. You therefore have to do `returnarray[:] = ...` . This is not an issue if you need to do this: `returnarray[i]=...` .

- **List of objects:** If you need to have a list of objects in your input arguments, the signature you need to use for that is `List(ObjectType)` . For loading a list of Numba classes, use `List(ObjectType, reflected=True)` .

- **Appending to numpy arrays:** It is very costly to do appending or concatenation operations with numpy arrays. Therefore we recommend using fixed size arrays (with an initial max size) and keep track of the end of the array using an integer variable.

- **Index out of range:** Numba unfortunately will let you assign a value out or range of an array and result in `core dump`. If you see this issue, inpect your code to be sure you are not trying to access a location out of range of an array.

- **Consistent variable types:** For any operation inside a Numba compiled function, you need to have consistent variable types. to give an example in `np.zeros((a, b))`, `a` and `b` must have the same integer type.

- **Lists versus numpy arrays:** Although Numba supports lists, note that it does not support list of lists. It would be wise to 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)