DECADES Numba Examples

Here we will walk through two simple examples in their entirety for the Numba flow through DECADES.

We will first do a matrix multiplication and then a reduction.

Before reading this guide, we hope that you have gone over:

1. Intro to DECADES programming (Introduction.ipynb) document
2. Numba Kernel programming guide (Numba.ipynb)
3. Numba Kernel execution guide (Numba_Decades_Examples.ipynb)

Location

Both examples and their inputs can be found in the docker at: (/decades/applications/DEC_Kernel_Execution_Examples/)

Matrix Multiplication

Ideally, if your workload was simply matrix multiplication, you should use the TensorFlow programming method for DECADES, as this can be computed using accelerators. However, matrix multiplication also makes a useful example.

Lets compute C = A x B where A and B (and hence C) are square matrices with dimension (s x s).

Kernel

We will start by writing our kernel. Recall from the Kernel Execution guide:

1. kernels have no return value, except through arguments.

2. need to be specified as an @njit numba kernel.
3. take in a tid and num_tiles arg
4. are compiled through the DEC_Pipeline

```
import numpy as np
# include necessary packages
from DEC_Pipeline import DEC_Pipeline, DEC_Optio
ns, decades_launch_kernel
from numba import int32, njit

@njit((int32[:,:],int32[:,:],int32[:,:], int32,
int32, int32), nogil=True, pipeline_class=DEC_Pi
peline)
def mat_mul(C, A, B, s, tid, num_tiles):
    for i in range(s):
        for j in range(s):
            val = 0
            for k in range(s):
                val += A[i,k] * B[k,j]
            C[i,j] = val
```

We hope this algorithm is not too difficult to follow. It can be found through any straightfoward google search on the subject

Using Threads

The for loops are a good indication that this algorithm can be parallelized. It is straight forward to use `tid` and `num_tiles` to do this. For example, the outer most for loop can be computed in parallel. We can simply stride through this loop using `tid` and `num_tiles` and it will be valid for any number of tiles (including 1). Here is that kernel updated to use threads:

```
import numpy as np
import sys # for command line args

# include necessary DEC_Pipeline packages
from DEC_Pipeline import DEC_Pipeline, DEC_Optio
ns, decades_launch_kernel
from numba import int32, njit


@njit((int32[:,:],int32[:,:],int32[:,:], int32,
int32, int32), nogil=True, pipeline_class=DEC_Pi
peline)
def mat_mul(C, A, B, s, tid, num_tiles):
    for i in range(tid, s, num_tiles):
        for j in range(s):
            val = 0
            for k in range(s):
                val += A[i, k] * B[k, j]
            C[i, j] = val
```

Launching Kernel

Recall from kernel execution documentation that we can now set some compiler options. For now, we will use the preset config (so that we can evaluate later using the evaluation programs, as documented in the evaluation guide (evaluation_guide.ipynb)

Note that you will need to have these in a python file that can be executed from a terminal in order for the evaluator to run, so please don't copy paste these to a python console.

We will first read in some data. We could hard code some data, but the evaluation programs require a single file input. We will take as input a file with a single integer that describes the size of a matrix. In more realistic workflows, these inputs might be a graph structure representing a social media network

Then we generate some data for A and B. Make the array for C. And finally launch the kernel, and check the answer.

```
# enter the mat_mul description we had above her
e.

if __name__ == "__main__":

    DEC_Options.preset_config()


    with open(sys.argv[1],'r') as f:
        matrix_size = int(f.read())


        A = np.random.randint(2048, size=(matrix
_size, matrix_size), dtype=np.int32)
        B = np.random.randint(2048, size=(matrix
_size, matrix_size), dtype=np.int32)
        C = np.zeros(matrix_size * matrix_size,
dtype=np.int32)

        t = decades_launch_kernel(mat_mul, A, B,
C, matrix_size)

        C_cpu = np.dot(A,B)
        assert(np.array_equal(C, C_cpu))
        print("kernel execution time (seconds):
" + str(t))
```

Running the program

To run the program, save it with the name "mat_mul.py" (this is important for the evaluation program) and then simply run it like a regular python script (creating an input file).

$echo\ "\ 32\ "> thirty_two.txt$ python mat_mul.py thirty_two.txt

Reduction

We will now do a similar example for a reduction kernel. We will first do a thread local reduction, followed by a sequential reduction

The arg will be a single array A and we will return the sum of all elements in A.

Remember!!! Only one kernel is allowed per script: this will need to be saved in a different script.

A reduction typically returns a single value, but we cannot return value directly. It is stored in an argument.

Kernel

A simple single threaded variant looks like this

```
import numpy as np
# include necessary packages
from DEC_Pipeline import DEC_Pipeline, DEC_Optio
ns, decades_launch_kernel
from numba import njit, int32


@njit((int32[:],int32[:], int32, int32), nogil=T
rue, pipeline_class=DEC_Pipeline)
def sum_reduction(A, return_arg, tid, num_tiles)
:
    return_value = 0
    for i in range(len(A)):
        return_value += A[i]
    return_arg[0] = return_value
```

Using Threads

We now show a parallel variant, which chunks up the data and performs a sequential reduction. Note that we need to synchronize the decades tile between the parallel thread local reduction and the sequential final reduction.

This can be done with the function PyDECADES_barrier as described in the Numba (Numba.ipynb) documentation.

```
import numpy as np
# include necessary packages
from DEC_Pipeline import DEC_Pipeline, DEC_Optio
ns, decades_launch_kernel
from DEC_Numba_Lib import PyDECADES_barrier

@njit((int32[:],int32[:], int32[:], int32, int32
), nogil=True, pipeline_class=DEC_Pipeline)
def sum_reduction(A, tile_local, return_arg, tid
, num_tiles):
    return_value = 0

    # ceiling division
    chunk_size = ((len(A) - 1) / num_tiles) + 1
    start_chunk = tid * chunk_size
    end_chunk = start_chunk + chunk_size

    # so we don't overflow
    end_chunk = min(len(A), end_chunk)

    # get a thread local reduction
    for i in range(start_chunk, end_chunk):
        tile_local[tid] += A[i]

    # synchronize decades tiles:
    PyDECADES_barrier()

    # sequentially finish reduction
    if tid == 0:
        for i in range(num_tiles):
            return_arg[0] += tile_local[i]
```

Launching the kernel

Similar to the above, we will use the preset config.

But we need to know how many threads the DECADES preset provides so that we can provide the `tile_local` array with enough space. For this we use the `DEC_Options.get_num_tiles()` function

```
# enter the ret_cpu description we had above her
e.

if __name__ == "__main__":
    DEC_Options.preset_config()

    num_tiles = DEC_Options.get_num_tiles()

    # Read in file so that we can use evaluator
scripts
    with open(sys.argv[1],'r') as f:
        vector_size = int(f.read())


    A = np.random.randint(2048, size=(vector_siz
e),dtype = np.int32)
    thread_local = np.zeros(num_tiles, dtype=np.
int32)
    return_value = np.zeros(1, dtype = np.int32)

    t = decades_launch_kernel(sum_reduction, A,
thread_local, return_value)

    ret_cpu = sum(A)
    assert(ret_cpu==return_value[0])
    print("kernel execution time (seconds): " +
str(t))
```

Running the Program

to run the program, create a text file to read in and execute the python script

$echo$ " 256 "> $two_f ifty_s ix.txt$ python mat_mul.py

Using the Evaluator

To continue these examples with using the evaluator, please see the evaluation documentation (Evaluation_Guide.ipynb)