Numba for CUDA

Required prerequisites

- GPU introduction: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
- CUDA Basics: CUDA C Programming Guide.
- For Numba:
  - Python programming: https://www.python.org/doc/
  - Numba basics: https://numba.pydata.org/numba-doc/latest/index.html
- To install python using Anaconda
  - https://docs.anaconda.com/anaconda/install/windows/

Numba

- Numba is a compiler for Python array and numerical functions that speeds up the applications with high performance functions written directly in Python.
- Numba generates optimized machine code from pure Python code using the LLVM compiler infrastructure. With a few simple annotations, array-oriented and math-heavy Python code can be just-in-time optimized to performance similar as C, C++ and Fortran, without having to switch languages or Python interpreters.

Numba for CUDA

- Numba supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model.
- Kernels written in Numba appear to have direct access to NumPy arrays.
- NumPy arrays are transferred between the CPU and the GPU automatically.

Installation

- Requirements
  - **Supported GPUs:** Numba supports CUDA-enabled GPU with compute capability 2.0 or above with an up-to-data Nvidia driver.
- Software
  - You will need the CUDA toolkit version 8.0 or later installed.
  - To install using Conda, type:
    - conda install cudatoolkit
  - If you are not using Conda or if you want to use a different version of CUDA toolkit, check here: https://numba.pydata.org/numba-doc/latest/cuda/overview.html

Missing CUDA features

- Numba does not implement all features of CUDA, yet. Some missing features are listed below:
  - dynamic parallelism
  - texture memory

CUDA Kernels

- CUDA has an execution model unlike the traditional sequential model used for programming CPUs.
    - In CUDA, the code you write will be executed by multiple threads at once (often hundreds or thousands).
    - Your solution will be modeled by defining a thread hierarchy of grid, blocks and threads.
- Numba’s CUDA support exposes facilities to declare and manage this hierarchy of threads.
    - The facilities are largely similar to those exposed by NVidia’s CUDA C language.
- Numba also exposes three kinds of GPU memory:
    - global device memory (the large, relatively slow off-chip memory that’s connected to the GPU itself), on-chip shared memory and local memory.
    - For all but the simplest algorithms, it is important that you carefully consider how to use and access memory in order to minimize bandwidth requirements and contention.

Kernel Declaration

- A *kernel function* is a GPU function that is meant to be called from CPU code (*).
- It gives it two fundamental characteristics:
    - kernels cannot explicitly return a value; all result data must be written to an array passed to the function (if computing a scalar, you will probably pass a one-element array);
    - kernels explicitly declare their thread hierarchy when called: i.e. the number of thread blocks and the number of threads per block (note that while a kernel is compiled once, it can be called multiple times with different block sizes or grid sizes).

Kernel Declaration

- Numba CUDA kernel format

@cuda.jit decorator from Numba performs the computation on GPU

```
@cuda.jit
def kernel_name(an_array):
    """
    Write kernel computation for each thread
    """
    # write code
```

Kernel Invocation

- A kernel is typically launched in the following way:
  - Instantiate the kernel proper, by specifying a number of blocks (or “blocks per grid”), and a number of threads per block. The product of the two will give the total number of threads launched. Kernel instantiation is done by taking the compiled kernel and indexing it with a tuple of integers.
  - Running the kernel, by passing it the input array (and any separate output arrays if necessary). By default, running a kernel is synchronous: the function returns when the kernel has finished executing and the data is synchronized back.

```
//a kernel invocation syntax
threadsperblock = 32
blockspergrid = (an_array.size + (threadsperblock - 1)) //
threadsperblock
Kernel_name[blockspergrid, threadsperblock](an_array)
```

Kernel invocation

- It might seem curious to have a two-level hierarchy when declaring the number of threads needed by a kernel. The block size (i.e. number of threads per block) is often crucial:
- On the software side, the block size determines how many threads share a given area of shared memory.
- On the hardware side, the block size must be large enough for full occupation of execution units; recommendations can be found in the CUDA C Programming Guide.
- More details on blocks and grids: https://numba.pydata.org/numba-doc/latest/cuda/kernels.html

Numba example

- Let us check a simple example of subtracting two matrix
- Suppose there are two 2D-matrix with all element values as 1(can be anything, selected 1 to subtract and get 0 result as shown in figure below)
- We can use GPU to perform subtraction of each element instead of CPU serial subtraction
- We need to import cuda from numba to use in the python file

$$\underset{A}{\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}} - \underset{B}{\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}} = \underset{C}{\begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array}}$$

Numba example

The example shows a kernel which adds elements from 2D array A and B corresponding to thread id

```
@cuda.jit
def kernel_op(A, B, C):

    #cuda.grid returns the absolute position of the current thread in the entire grid of blocks
    x, y = cuda.grid(2)

    if x >= C.shape[0] and y >= C.shape[1]:
        # Quit if (x, y) is outside of valid C boundary
        return

    # Each thread computes one element in the result matrix.
    C[x,y]=A[x,y]-B[x,y]
```

Numba example:

Initialize the matrices

```
# Initialite the data array
A = numpy.ones([48,48], dtype = float)
B = numpy.ones([48,48], dtype =float)
```

Copy host values to device

```
#copy the host variables to device
A_global_mem = cuda.to_device(A)
B_global_mem = cuda.to_device(B)
```

Create memory for result matrix in device

```
#Create memory for C in device
C_global_mem = cuda.device_array((48,48))
```

Configure the thread blocks

```
# Configure the blocks
threadsperblock = (TPB, TPB)
blockspergrid_x = int(math.ceil(A.shape[0] / threadsperblock[1]))
blockspergrid_y = int(math.ceil(B.shape[1] / threadsperblock[0]))
blockspergrid = (blockspergrid_x, blockspergrid_y)
```

Numba example

- Syntax to execute the kernel in the code

```
# Start the kernel
kernel_op[blockspergrid, threadsperblock](A_global_mem, B_global_mem, C_global_mem)
#copy the result to cpu
res = C_global_mem.copy_to_host()
```

Instructions to run the program

- Before running the program make sure that CUDA, python and Numba are properly installed and set up in path
- .ipynb files are the jupyter notebook files and need to be opened with Jupyter Notebook. It should be installed with Anaconda or you can install it manually (https://jupyter.org/install)
- .py files are the python files extracted from Jupyter Notebook and can be run as basic python programs

Exercise

- Review the code for Numba exercise
- Write a program for adding the vectors in CPU using loops and compare the results with using Numba for GPU CUDA

- Similar to the given example, write a python code with Numba to add two 1D lists