CUDA Python: Introduction to Numba and CuPy

Mahidhar Tatineni

SDSC HPC User Training Series

February 12, 2021

Overview

- **Numba** can compile Python code for execution on CUDA capable GPUs.
- Running native, compiled code is faster than interpreted code.
- Compilation at runtime (Just in Time - JIT)
- Keep flexibility of python and enable high performance aspects.
- **CuPy:** NumPy compatible array library, accelerated by CUDA.
- **CuPy** uses highly optimized libraries such as cuBLAS, cuDNN, cuRand, cuSolver, cuSPARSE, cuFFT and NCCL.

Numba & NumPy

- Numba is designed for array-oriented computing tasks just like NumPy.
- Numba disallows dynamic memory allocating features. Disables many NumPy APIs
- Supported NumPy features:
  - accessing ndarray attributes
  - scalar ufuncs
  - Indexing, slicing
- Unsupported features: array creation APIs, array methods, and functions that returns a new array.

Numba device management

- numba.cuda.gpus to list devices
- numba.cuda.select_device: Create a new CUDA context for the selected *device_id*. ***Only one context per thread.***

```
[mahidhar@exp-2-57 ~]$ module load gpu
[mahidhar@exp-2-57 ~]$ module load anaconda3
[mahidhar@exp-2-57 ~]$ python
Python 3.8.5 (default, Sep  4 2020, 07:30:14)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from numba import cuda
>>> print(cuda.gpus)
<Managed Device 0>
>>>
```

```
[mahidhar@exp-2-57 ~]$ module load gpu
[mahidhar@exp-2-57 ~]$ module load anaconda3
[mahidhar@exp-2-57 ~]$ python
Python 3.8.5 (default, Sep  4 2020, 07:30:14)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from numba import cuda
>>> print(cuda.gpus)
<Managed Device 0>, <Managed Device 1>
>>>
```

Numba: Memory management

- Global device memory
  - large, off-chip, slow
  - Numba will automatically transfer NumPy arrays to the device when kernel is invoked
- Shared memory (numba.cuda.shared.array(***shape, type***))
  - On-chip shared, available to all threads
- Local memory (numba.cuda.local.array(***shape, type***))
  - Private to each thread
  - Allocated once for duration of kernel
- Constant memory (numba.cuda.const.array_like(***arr***))
  - read only, cached and off-chip
  - accessible by all threads and host allocated

Numba: Data Transfer

- Automatic transfer NumPy arrays to the device – data moved back to host when kernel finishes.
  - Unnecessary transfer for read-only arrays.
  - Same elements might get transferred multiple times
- Manually allocate on device, move data to/from device:
  - numba.cuda.device_array(***shape*, *dtype=np.float*, *strides=None*, *order='C'*, *stream=0***)
  - numba.cuda.to_device(***obj*, *stream=0*, *copy=True*, *to=None***)
  - copy_to_host(***self*, *ary=None*, *stream=0***)
- Streams for asynchronous execution; methods for synchronization
- Pinned memory

Using CUDA kernels in Numba

- Kernels - GPU functions called from CPU code (Numba doesn't support device-side launches at present)
  - Don't explicitly return a value – results written to the array passed
  - Number of thread blocks and threads per block explicitly specified
  - Example call: *matmul[blockspergrid, threadsperblock](A, B, C)*
- Example Kernel:

```
@cuda.jit
def matmul(A, B, C):
    """Perform square matrix multiplication of C = A * B
    """
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

CUDA Ufuncs

- Ufuncs: Numpy universal functions, element by element basis
  - Example: np.sqrt(x)
- CUDA Ufuncs are analgous
  - support for passing intra-device arrays
  - stream for asynchronous mode
  - can all other device functions

```
@vectorize(['float32(float32, float32, float32)', 'float64(float64, float64, float64)'], target='cuda’)
def cu_discriminant(a, b, c):
        return math.sqrt(b ** 2 - 4 * a * c)
```

CUDA Generalized Ufuncs

- Regular Ufuncs – operations on scalar and return is scalar
- Generalized Ufuncs – can deal with subarray and return array
- Example:

```
from numba import guvectorize
@guvectorize(['void(float32[:,:], float32[:,:], float32[:,:])'], '(m,n),(n,p)->(m,p)', target='cuda’)
def matmulcore(A, B, C):
...
```

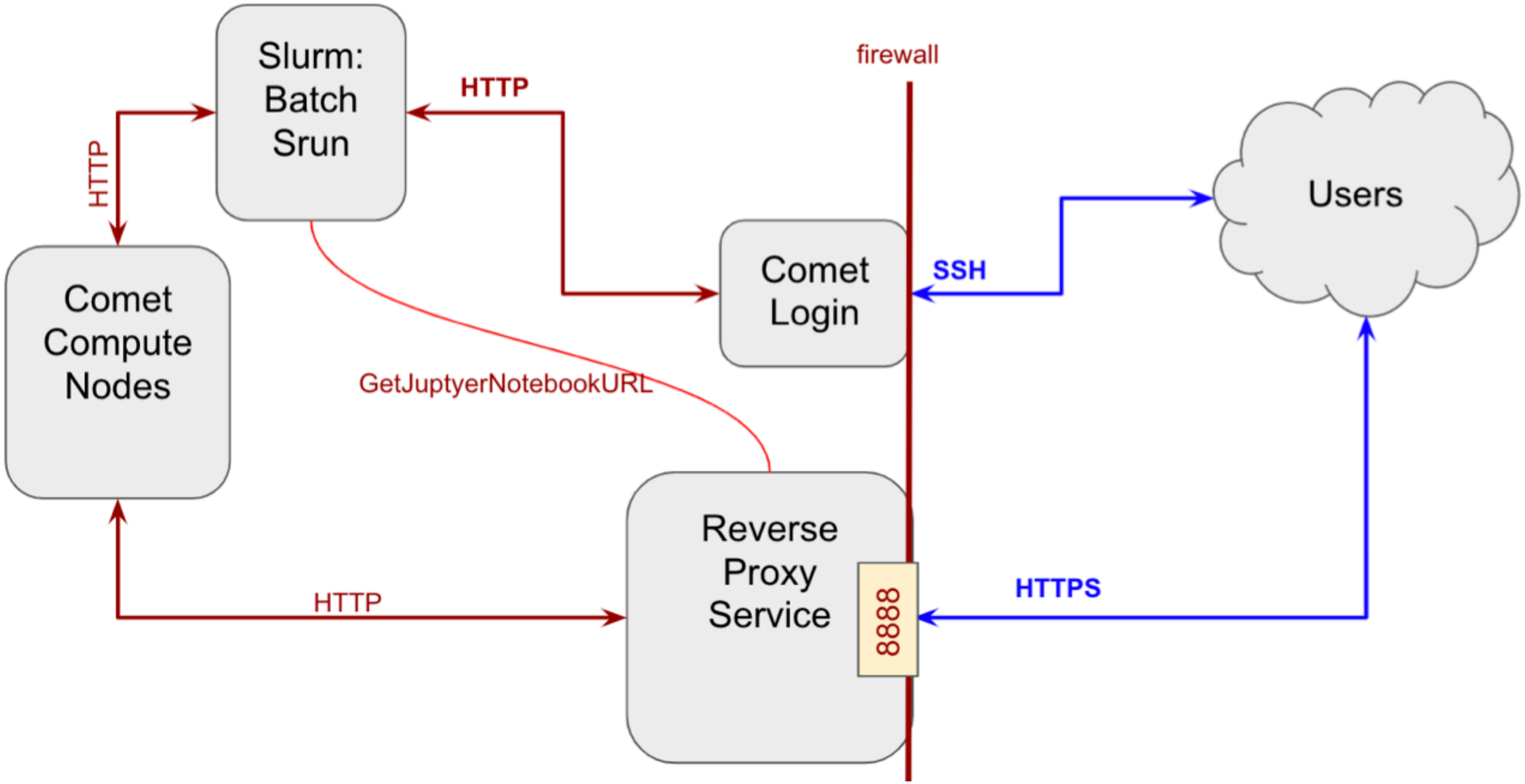
Running Examples on Expanse

- Examples using Jupyter notebooks
- SDSC developed reverse proxy service to secure notebooks
- Clone as follows:

git clone https://github.com/sdsc-hpc-training-org/reverse-proxy.git

- Central Anaconda install available on Expanse – don’t need to do the install in your directory.

Reverse Proxy Service for Jupyter Notebooks



Launching the notebook

- Clone the repos needed:

  git clone https://github.com/sdsc-hpc-training-org/reverse-proxy.git

  git clone https://github.com/sdsc-hpc-training-org/hpc-training-2021

  (Note: if you already cloned this, just pull to get the week4 changes)

- Launch the notebook from the reverse-proxy directory:

  module load gcc anaconda3

  ./start-jupyter -p gpu-shared -A use300 -t 00:30:00 -b ../hpc-training-2021/week4_cuda_python/notebook-expanse-gpu.sh

  (Change the info in red to your values)

Sample output for notebook launch

```
[mahidhar@login02 reverse-proxy]$ module load gcc anaconda3
[mahidhar@login02 reverse-proxy]$ ./start-jupyter -p gpu-shared -A use300 -t 00:30:00 -b ../hpc-training-2021/week4_cuda
_python/notebook-expanse-gpu.sh
Your notebook is here:
        https://staleness-apron-whacking.expanse-user-content.sdsc.edu?token=526b1001c1e80cab98b89bc61375e212
If you encounter any issues, please email help@xsede.org and mention the Reverse Proxy Service.
Your job id is 1271497
You may occasionally run the command 'squeue -j 1271497' to check the status of your job
[mahidhar@login02 reverse-proxy]$ squeue -u $USER
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
           1271497 gpu-share notebook mahidhar PD       0:00      1 (Priority)
[mahidhar@login02 reverse-proxy]$
```

Example notebooks

- computing_pi_solution.ipynb
- distance_matrix_solution.ipynb
- law_of_cosines.ipynb (Homework)

[The above 3 notebooks are from Abe Stern's talk]

- CuPy_examples.ipynb
- External example from Mark Harris (NVIDIA):

https://github.com/harrism/numba_examples/blob/master/mandelbrot_numba.ipynb

Note: Minor corrections needed on above notebook: print statements need braces, "autojit" changed to "jit".

Overview of CuPy

- CuPy: GPU array backend that implements a NumPy interface (subset). Some SciPy compatible features.
- Can you CuPy with Numba.
- cupy.ndarray is the core class (equivalent of numpy.ndarray)
- Function available for switching current device in use:
  - Cupy.cuda.Device.use()
- Function to move data arrays to device:
  - cupy.asarray()
  - Can be used to move data between devices
- Function to move data array from device to host:
  - cupy.asnumpy()

CuPy: Subset of NumPy supported

- Basic and advanced indexing (except for some with boolean masks)
- Data types:

  bool_, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float16, float32, flo
  at64, complex64, complex128
- Most array creation and manipulation routines
- All operators with broadcasting
- All universal functions (except those for complex numbers)
- Linear algebra functions (cuBLAS accelerated)
- Reduction along axes

CuPy: Performance enhancement options

- User-defined elementwise CUDA kernels
- User-defined reduction CUDA kernels
- Fusing CUDA kernels to optimize user-defined calculation
- Customizable memory allocator and memory pool
- Using cuDNN utilities

- Look at **CuPy_examples.ipynb** for example cases.

Useful Links

- Numba Documention:
  - https://numba.pydata.org/numba-doc/latest/cuda/
- NYU Numba tutorial:
  - https://nyu-cds.github.io/python-numba/05-cuda/
- CuPy Documentation:
  - https://docs.cupy.dev/en/stable/index.html
- CuPy and Numba tutorial:
  - https://carpentries-incubator.github.io/gpu-speedups/01_CuPy_and_Numba_on_the_GPU/index.html