Chapter 5: CUDA Kernels with Numba

Numba is an open-source JIT (Just In Time) compiler that translates a subset of Python and NumPy functions to optimized machine code. In the context of GPU's, Numba can be used to create kernel functions quickly and efficiently.

Numba Basics

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 have direct access to NumPy arrays. NumPy arrays are transferred between the CPU and the GPU automatically. Numba's integrated compilation system allows the creation of code using the characteristics of both the CPU and GPU in such a way that does not require many changes to the Python language.

Installation

Before setting up your Numba programming environment, first ensure that you have fulfilled the following prerequisites (if you followed the instructions for installing CuPy, you can skip these steps):

- CUDA-compatible GPU. (see https://developer.nvidia.com/cuda-gpus for a list of NVIDIA GPUs)
- CUDA-compatible NVIDIA Drivers.
- CUDA Toolkit

See installation instructions here: https://numba.pydata.org/numba-doc/latest/user/installing.html

Creating a Kernel Function with `@cuda.jit`

In Numba, the `@jit` decorator is used to specify a function to be optimized by the Numba just in time compiler. Within the context of GPU's we use a version called `@cuda.jit` to specify kernel functions to be optimized for execution by multiple threads on the GPU simultaneously.

```
from numba import cuda

@cuda.jit
def foo(input_array, output_array):
    # code block goes here
```

This should look very familiar to using numba on the CPU.

Launching a Kernel Function

Before running a kernel function, the number of blocks and threads per block need to be specified. This will define the execution grid's shape.

```
@cuda.jit
def foo(input_array, output_array):
    # Thread id in a 1D block
    thread_id = cuda.threadIdx.x
    # Block id in a 1D grid
    block_id = cuda.blockIdx.x
    # Block width, i.e. number of threads per block
    block_width = cuda.blockDim.x
    # Compute flattened index inside the array
    i = thread_id + block_id * block_width
    if i < an_array.size:  # Check array boundaries
        output_array[i] = input_array[i]
```

To call the **foo()** function, we have to specify the block and grid size.

```
input = np.asarray(range(10))
output = np.zeros(len(input))

block_threads = 32
grid_blocks = (input.size + (block_threads - 1)) // block_threads

foo[grid_blocks, block_threads](input, output)
```

For simple examples, the **cuda.grid()** function is a convenient way to manage threads, blocks and grids. The complete script can be re-written this way:

```
import numpy as np
from numba import cuda

input = np.asarray(range(10))
output = np.zeros(len(input))

@cuda.jit
def foo(input_array, output_array):
    i = cuda.grid(1)
    output_array[i] = input_array[i]
```

```
foo[1, len(input)](input, output)

output
```

Note: When a CUDA kernel executes, the call returns immediately before the kernel execution is complete. The kernel execution then needs to be synchronized in order to ensure the results are transferred back to the CPU. Without completing this step, you may run into memory errors where subsequent calls are trying to read or write to restricted memory. Use `cuda.synchronize()` to ensure data consistency.

Specifying the Number of Threads and Blocks

Don't worry too much about this now. Just take away the idea that we need to specify the number of times we want our kernel to be called, and that is given as two numbers which are multiplied together to give your overall grid size. This setup will ensure a grid size that has enough threads to handle the size of the data, even if that number is not an exact multiple of the threads per block.

Rules of thumb for threads per block:

- Optimal block size is usually a multiple of 32 (warp size).
- Profiling and benchmarking are required to determine the optimal value.

Getting Started:

- NSight's Occupancy Calculator: https://docs.nvidia.com/nsight-compute/NsightCompute/index.html#occupancy-calculator)
- Several sources recommend starting with a number between 128 and 256 to begin tuning.

Block and grid dimensions will affect CUDA performance. Larger blocks can lead to better utilization of the shared memory and reduce the overhead of launching many small blocks. However, excessively large blocks might reduce the number of blocks that can execute concurrently which will underutilize the GPU. Finding the right balance is necessary in order to take advantage of the GPU.

Numba with CuPy

CuPy's **`cupy.ndarray`** implements `__cuda_array_interface__`, which is the CUDA array interchange interface compatible with Numba v0.39.0 or later (see Numba's CUDA Array Interface for details). It means you can pass CuPy arrays to kernels JITed with Numba.

In this example, we use **cupy** arrays instead of **numpy** arrays:

```python
import cupy
from numba import cuda

@cuda.jit
def add(x_array, y_array, output_array):
        start = cuda.grid(1)
        stride = cuda.gridsize(1)
        for i in range(start, x.shape[0], stride):
                output_array[i] = x_array[i] + y_array[i]

a = cupy.arange(10)
b = a * 2
out = cupy.zeros_like(a)

add[1, 32](a, b, out)

print(out)  # => [ 0  3  6  9 12 15 18 21 24 27]
```

References

Numba for CUDA GPUs: https://numba.pydata.org/numba-doc/latest/cuda/index.html
CuPy's interoperability guide (includes Numba):
https://docs.cupy.dev/en/stable/user_guide/interoperability.html
Numba Github repository: https://github.com/numba/numba