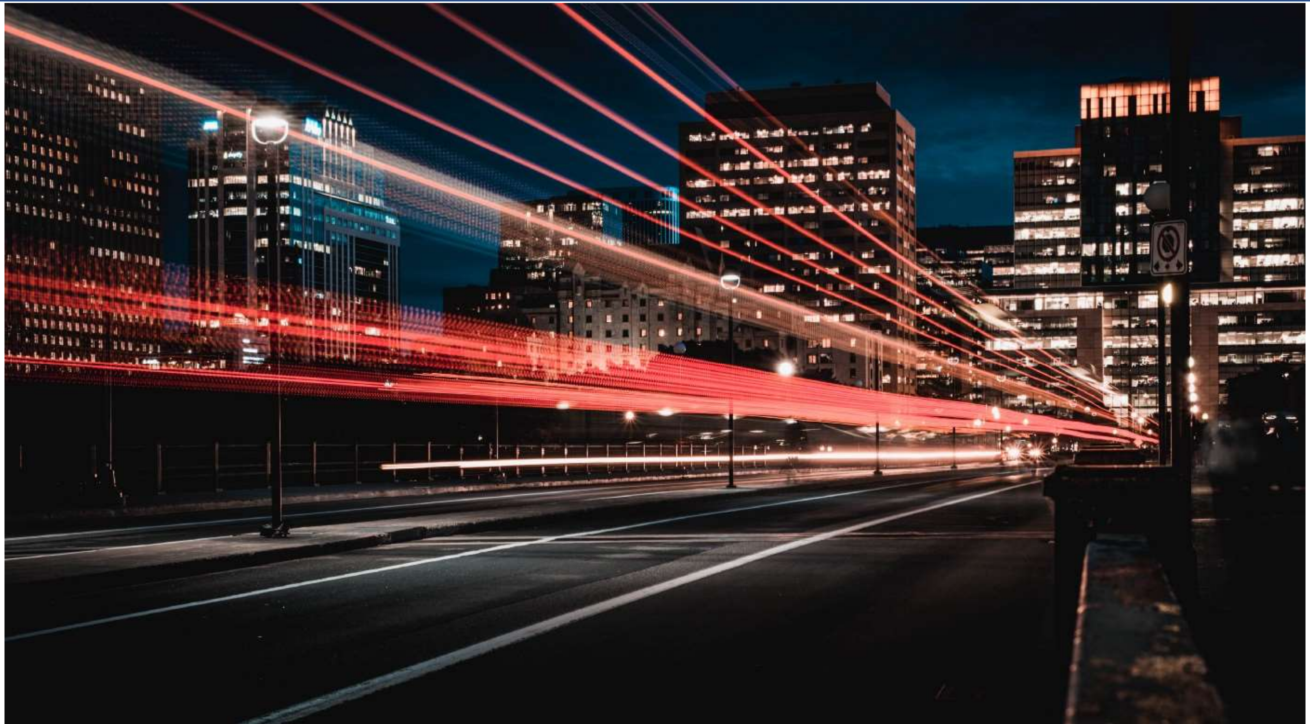
Parallel programming

Python Numba. Basics



Overview

- *Numba* package supports *CUDA GPU* programming by directly compiling *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.

Terminology

The main CUDA programming terms are listed below:

- host: the CPU
- device: the GPU
- host memory: the system main memory
- device memory: onboard memory on a GPU card
- kernels: a GPU function launched by the host and executed on the device
- device function: a GPU function executed on the device which can only be called from the device (i.e. from a kernel or another device function)

Setting up python numba

- You can install the NVIDIA bindings with:

$ conda install nvidia::cuda-python

- Or if you are using pip:

$ pip install cuda-python

- Easy to work in Google Colab

- Additional info:

https://numba.readthedocs.io/en/stable/cuda/overview.html

CUDA Kernels

A ***kernel function*** is a GPU function that is meant to be called from CPU code.

- kernels cannot explicitly return a value; all result data must be written to an array passed to the function
- kernels explicitly declare their thread hierarchy when called
  - the number of thread blocks
  - the number of threads per block
- While a kernel is compiled once, it can be called multiple times with different block sizes or grid sizes

Declaration/Invocation of the kernel

```python
@cuda.jit
def increment_by_one(an_array):
    """
    Increment all array elements by one.
    """
    # code elided here; read further for different implementations
```

```python
threadsperblock = 32
blockspergrid = (an_array.size + (threadsperblock - 1)) // threadsperblock
increment_by_one[blockspergrid, threadsperblock](an_array)
```

Blocks of threads

The block size – the number of threads per block - is often crucial:

- *Software side:* the block size determines how many threads access a given area of shared memory.
- *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)*

Positioning of threads and blocks

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

Positioning of threads and blocks

- ***Inside block/grid***
  - numba.cuda.threadIdx
  - numba.cuda.blockIdx
- ***Dimensions***
  - numba.cuda.blockDim
  - numba.cuda.gridDim
- ***Absolute positions***
  - numba.cuda.grid(ndim)
  - numba.cuda.gridsize(ndim)

Data transfer

- ***Allocate device array***
  - numba.cuda.device_array(…)
  - numba.cuda.device_array_like(…)
- ***Copy the data from host to device***
  - numba.cuda.to_device(…)
- ***Copy the data from device to host***
  - numba.cuda.copy_to_host(…)

Shared memory

- A limited amount of shared memory can be allocated on the device to speed up access to data.
- That memory will be shared (i.e. both readable and writable) amongst all threads belonging to a given block and has faster access times than regular device memory.
- It also allows threads to cooperate on a given solution. You can think of it as a manually-managed data cache.
- The memory is allocated once for the duration of the kernel

Shared memory and synchronization

- ***numba.cuda.shared.array(shape, type)***
  - Allocate a shared array of the given *shape* and *type* on the device.
  - The function must be called from the device
- ***numba.cuda.syncthreads()***
  - Synchronize all threads in the same thread block.
  - This function implements the pattern of barrier

Local memory

- Local memory is the memory area private to a thread:

  numba.cuda.local.array(shape, type)

- Using local memory helps to allocate some scratchpad area when scalar local variables are not enough.

- The memory is allocated once for the duration of the kernel

Constant memory

- Constant memory is an area of memory that is read only, cached and off-chip:

  numba.cuda.const.array_like(arr)

- Accessible by all threads
- Allocated from the host

References

- **Fundamental tutorial on numba:**

https://numba.readthedocs.io/en/stable/cuda/index.html

- **Selected pages:**

https://numba.readthedocs.io/en/stable/cuda/kernels.html

https://numba.readthedocs.io/en/stable/cuda/memory.html