Parallel Computing with Numba

By: Shuhao Lai

Date: 7/22/19

Setup and Getting Started

Download and install the Anaconda python distribution that includes many popular packages (Numpy, Scipy, Matplotlib, iPython, etc) and “conda”, a package manager. Once Anaconda is installed, install the required CUDA packages with “conda install numba cudatoolkit”.

Numba is a Python compiler that can compile Python code for execution on CUDA-capable GPUs or multicore CPUs

Cudatoolkit provides a development environment for creating GPU-accelerated applications.

If you are using Google Colab, numba and cudatoolkit are already installed.

Google Colab specific: in order to use @vectorize for ufuncs (not important if you don’t know what they are now), I need to configure the environment variables. To find the path for “libdevice” and “libnvvm.so”, use the following code:

```
!find / -iname 'libdevice'
!find / -iname 'libnvvm.so'

# Output:
# /usr/local/cuda-9.2/nvvm/lib64/libnvvm.so
# /usr/local/cuda-9.2/nvvm/libdevice
```

Next, set the environment variables.

```
import os
os.environ['NUMBAPRO_LIBDEVICE'] = "/usr/local/cuda-9.2/nvvm/libdevice"
os.environ['NUMBAPRO_NVVM'] = "/usr/local/cuda-9.2/nvvm/lib64/libnvvm.so"
```

It is important to know that there are two basic approaches to GPU programming in Numba:

1. ufuncs/gufuncs
2. CUDA Python kernels

Numba Basics

Numba is a just-in-time, type-specializing, function compiler for accelerating numerically-focused Python.

- **function compiler**: Numba compiles Python functions to turin it into a (usually) faster function.
- **type-specializing**: Numba speeds up your function by generating a specialized implementation for the specific data types you are using. Python functions are designed to operate on generic data types, which makes them very flexible, but also very slow. In practice, you only will call a function with a small number of argument types, so Numba will generate a fast implementation for each set of types.

- **just-in-time**: Numba translates functions when they are first called. This ensures the compiler knows what argument types you will be using. This also allows Numba to be used interactively in a Jupyter notebook just as easily as a traditional application
- **numerically-focused**: Currently, Numba is focused on numerical data types, like int, float, and complex. There is very limited string processing support, and many string use cases are not going to work well on the GPU. To get best results with Numba, you will likely be using NumPy arrays.

Essentially, with Numba, it is possible to write standard Python functions and run them on a CUDA-capable GPU.

Numba Miscellaneous Details

```
from numba import jit
import math

@jit
def hypot(x, y):
    # Implementation from https://en.wikipedia.org/wiki/Hypot
    x = abs(x);
    y = abs(y);
    t = min(x, y);
    x = max(x, y);
    t = t / x;
    return x * math.sqrt(1+t*t)
```

The above code uses a CPU decorator so the function is parallelized on the CPU. The first time we call the method, the compiler is triggered and compiles a machine code implementation for float inputs.

Use the timeit Python module to benchmark code or use the %timeit command from Jupyter.

```
%timeit hypot.py_func(3.0, 4.0)
```

The following link explains the timeit output:

https://stackoverflow.com/questions/28218648/how-to-interpret-the-returned-string-of-timeit

Essentially, timeit runs the function in a loop until it exceeds 0.2ms, then it takes the best of three runs.

Function.inspect_types() printed annotated version of source code.

Taking a look at the data types can sometimes be important in GPU code because the performance of float32and float64 computations will be very different on CUDA devices. An accidental upcast can dramatically slow down a function.

Numba doesn't support dictionaries. However, if you try to compile a function that uses dictionaries, it will fall back to "object mode", which does not do type-specialization. Use

```
@jit(nopython=True)
```

to force the compile in "nopython mode" so only compilation with type-specialization it allowed.

Universal Functions (Ufuncs)

Maximum throughput is achieved when you are computing the same operations on many different elements at once so that you can compute them in parallel. This is known as data parallel.

Ufuncs (universal functions) broadcast a scalar function over array inputs. They are functions that can take NumPy arrays of varying dimensions (or scalars) and operate on them element-by-element.

Ufuncs are naturally data parallel.

Numba can create compiled ufuncs. You implement a scalar function of all the inputs, and Numb a will figure out the broadcast rules for you. Generating a ufunc that uses CUDA requires giving an explicit type signature ad setting the target attribute.

```
from numba import vectorize

@vectorize(['int64(int64, int64)'], target='cuda')
def add_ufunc(x, y):
    return x + y
```

A lot of things just happened! Numba automatically:

- Compiled a CUDA kernel to execute the ufunc operation in parallel over all the input elements.
- Allocated GPU memory for the inputs and the output.
- Copied the input data to the GPU.
- Executed the CUDA kernel with the correct kernel dimensions given the input sizes.
- Copied the result back from the GPU to the CPU.
- Returned the result as a NumPy array on the host.

The result of the above code utilizing CUDA is still very slower because of the following:

- **Our inputs are too small**: Our test inputs have only 4 and 16 integers, respectively. Benefits of parallelism is minimal.
- **Our calculation is too simple**: Sending a calculation to the GPU involves quite a bit of overhead compared to calling a function on the CPU. If our calculation does not

involve enough math operations (often called "arithmetic intensity"), then the GPU will spend most of its time waiting for data to move around.

- **We copy the data to and from the GPU**: While including the copy time can be realistic for a single function, often we want to run several GPU operations in sequence. In those cases, it makes sense to send data to the GPU and keep it there until all of our processing is complete.
- **Our data types are larger than necessary**: Our example uses int64 when we probably don't need it. Scalar code using data types that are 32 and 64-bit run basically the same speed on the CPU, but 64-bit data types have a significant performance cost on the GPU. Basic arithmetic on 64-bit floats can be anywhere from 2x (Pascal-architecture Tesla) to 24x (Maxwell-architecture GeForce) slower than 32-bit floats. NumPy defaults to 64-bit data types when creating arrays, so it is important to set the dtype attribute or use the ndarray.astype() method to pick 32-bit types when you need them.

```
import math  # Note that for the CUDA target, we need to use the scalar functions from the math module, not NumPy

SQRT_2PI = np.float32((2*math.pi)**0.5)  # Precompute this constant as a float32.  Numba will inline it at compile time.

@vectorize(['float32(float32, float32, float32)'], target='cuda')
def gaussian_pdf(x, mean, sigma):
    '''Compute value of a Gaussian probability density function at x with given mean and sigma.'''
    return math.exp(-0.5 * ((x - mean) / sigma)**2) / (sigma * SQRT_2PI)

# Evaluate the Gaussian a million times!
x = np.random.uniform(-3, 3, size=1000000).astype(np.float32)
mean = np.float32(0.0)
sigma = np.float32(1.0)
```

This method is still slower than the gaussian_pdf function from scipy.stats. Details discussed later.

Ufuncs that use special functions (exp, sin, cos, etc) on large data sets run especially well on the GPU.

You can also create normal functions that are only called from other functions running on the GPU, also known as device functions.

Created with the numba.cuda.jit decorator:

```
from numba import cuda
```

```
@cuda.jit(device=True)
def polar_to_cartesian(rho, theta):
    x = rho * math.cos(theta)
    y = rho * math.sin(theta)
    return x, y  # This is Python, so let's return a tuple
```

```
@vectorize(['float32(float32, float32, float32, float32)'], target='cuda')
def polar_distance(rho1, theta1, rho2, theta2):
    x1, y1 = polar_to_cartesian(rho1, theta1)
    x2, y2 = polar_to_cartesian(rho2, theta2)

    return ((x1 - x2)**2 + (y1 - y2)**2)**0.5
```

Compared to Numba on the CPU (which is already limited), Numba on the GPU has more limitations. Supported Python includes:

- if/elif/else
- while and for loops
- Basic math operators
- Selected functions from the math and cmath modules
- Tuples

Memory Management

```
from numba import vectorize
import numpy as np

@vectorize(['float32(float32, float32)'], target='cuda')
def add_ufunc(x, y):
    return x + y

n = 100000
x = np.arange(n).astype(np.float32)
y = 2 * x
```

The above code is inefficient for CUDA because host (CPU) and device (GPU) bandwidth is used inefficiently. Better to involve host ← → device transfers at the end of computations.

Use the following to copy host data to GPU and return a CUDA device array:

```
from numba import cuda

x_device = cuda.to_device(x)
y_device = cuda.to_device(y)
```

Device arrays can be passed to CUDA functions just like NumPy arrays, but without the copy overhead:

To eliminate device → host bandwidth, create the output buffer with the numba.cuda.device_array() function:

```
out_device = cuda.device_array(shape=(n,), dtype=np.float32)  # does not initialize the contents, like np.empty()
```

And then we can use a special out keyword argument to the ufunc to specify the output buffer:

```
%timeit add_ufunc(x_device, y_device, out=out_device)
```

CUDA C Programming Guide from ToolKit Manual

CUDA C extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*, as opposed to only once like regular C functions.

A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<<...>>>

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

```
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
```

```
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

a kernel can be executed by multiple equally-shaped thread blocks,

Threads can form a one-dimensional, two-dimensional, or three-dimensional block of threads.

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional *grid* of thread blocks

The following is an example with a grid of blocks.

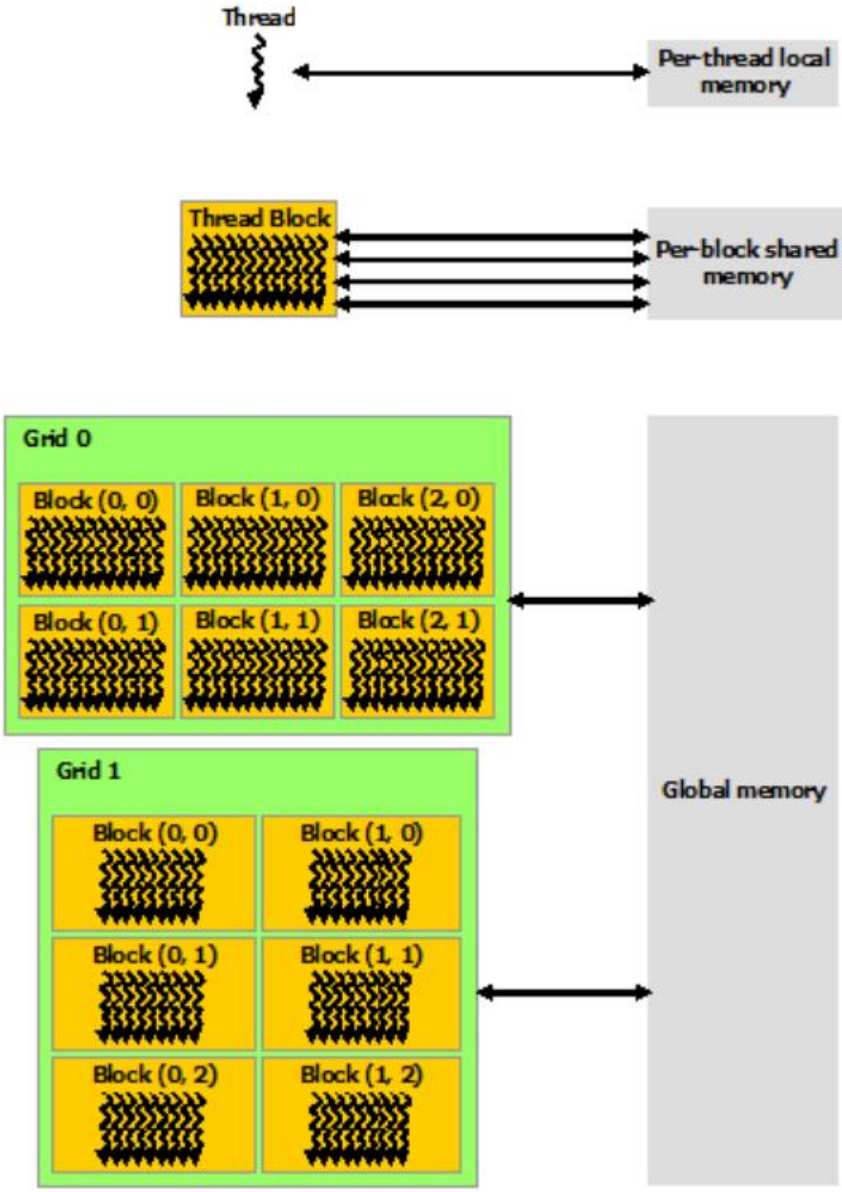
```
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Definitions:

- blockDim.x,y,z gives the number of threads in a block, in the particular direction
    - blockDim.y gives the number of threads in a block in the y direction.
- gridDim.x,y,z gives the number of blocks in a grid, in the particular direction
- blockDim.x * gridDim.x gives the number of threads in a grid (in the x direction, in this case)

one can specify synchronization points in the kernel by calling __syncthreads().

Memory layout of threads:



the CUDA programming model assumes that the CUDA threads execute on a physically separate *device* that operates as a coprocessor to the *host* running the C program. The CUDA programming model also assumes that both the host and the device maintain their own separate memory spaces in DRAM. Unified Memory provides *managed memory* to bridge the host and device memory spaces. Managed memory is accessible from all CPUs and GPUs in the system as a single, coherent memory image with a common address space. Elminates the need to explicitly mirror data on host and device.

Writing CUDA Kernels Guide from GitHub Notes

Deciding thread grid heuristics:

- the size of a block should be a multiple of 32 threads, with typical block sizes between 128 and 512 threads per block.
- the size of the grid should ensure the full GPU is utilized where possible. Launching a grid where the number of blocks is 2x-4x the number of "multiprocessors" on the GPU is a good starting place. Something in the range of 20 - 100 blocks is usually a good starting point.
- The CUDA kernel launch overhead does depend on the number of blocks, so we find it best not to launch a grid where the number of threads equals the number of input elements when the input size is very big. We'll show a pattern for dealing with large inputs below.

```
from numba import cuda

@cuda.jit # The kernel decorator.
def add_kernel(x, y, out):
    tx = cuda.threadIdx.x # this is the unique thread ID within a 1D block
    ty = cuda.blockIdx.x  # Similarly, this is the unique block ID within the 1D grid

    block_size = cuda.blockDim.x  # number of threads per block
    grid_size = cuda.gridDim.x    # number of blocks in the grid

    start = tx + ty * block_size
    stride = block_size * grid_size

    # assuming x and y inputs are same length
    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]
```

Most of the function is spent figuring out how to turn the block and grid indices and dimensions into unique offsets into the input arrays. The for-loop is used because we did not assume there is a thread for every element in the array. Further, by using a loop with stride equal to the grid size, we ensure that all addressing within warps is unit-stride, so we get maximum memory coalescing (refers to combining multiple **memory** accesses into a single transaction). Also, Thread indices beyond the length of the input (x.shape[0], since x is a NumPy array) automatically skip over the for loop.

Calling the function:

```
import numpy as np

n = 100000
```

```
x = np.arange(n).astype(np.float32)
y = 2 * x
out = np.empty_like(x)

threads_per_block = 128
blocks_per_grid = 30

add_kernel[blocks_per_grid, threads_per_block](x, y, out)
print(out[:10])
```

Note that in this example, the arguments are passed to the kernel as full NumPy arrays, which is not optimal. Changes will be made in the later method.

Simple off-set/index finding with Numba helper functions:

```
@cuda.jit
def add_kernel(x, y, out):
    start = cuda.grid(1)      # 1 = one dimensional thread grid, returns a single value
    stride = cuda.gridsize(1) # ditto

    # assuming x and y inputs are same length
    for i in range(start, x.shape[0], stride):
        out[i] = x[i] + y[i]
```

Speed up by allocating device arrays

```
x_device = cuda.to_device(x)
y_device = cuda.to_device(y)
out_device = cuda.device_array_like(x)
%timeit add_kernel[blocks_per_grid, threads_per_block](x_device, y_device, out_device); out_d
evice.copy_to_host()
```

The semicolon is there to separate two commands.

CUDA kernel execution is designed to be asynchronous with respect to the host program. This means kernels return immediately, allowing the CPU to continue executing while the GPU works in the background. Only host←→device memory copies or an explicit synchronization call will force the CPU to wait until previously queued CUDA kernels are complete.

When you pass host NumPy arrays to a CUDA kernel, Numba has to synchronize on your behalf, but if you pass device arrays, processing will continue.

The following forces GPU to finish before CPU can continue.

```
cuda.synchronize()
```

Always be sure to synchronize with the GPU when benchmarking CUDA kernels!

The following is okay because it runs in the background so we only time the set up.

```
# GPU input/output arrays, no synchronization (but force sync before and after)
cuda.synchronize()
%time add_kernel[blocks_per_grid, threads_per_block](x_device, y_device, out_device)
cuda.synchronize()
```

Avoid hazards by making sure each thread has exclusive responsibility for unique subsets of output array elements, and/or to never use the same array for both input and output in a single kernel call.

CUDA provides "atomic operations" which will read, modify and update a memory location in one, indivisible step. To prevent synchronization problems if the steps were split up. BELOW IS A GREAT EXAMPLE.

```
@cuda.jit
def thread_counter_race_condition(global_counter):
    global_counter[0] += 1  # This is bad

@cuda.jit
def thread_counter_safe(global_counter):
    cuda.atomic.add(global_counter, 0, 1)  # Safely add 1 to offset 0 in global_counter array
```

Troubleshooting and Debugging

Can print from CUDA kernals; however, output printed from a CUDA kernel will not be captured by Jupyter, so you will need to debug with a script you can run from the terminal.

The following allows step-by-step debugging on a single thread. But output is messy.

```
@cuda.jit
def histogram(x, xmin, xmax, histogram_out):
    nbins = histogram_out.shape[0]
    bin_width = (xmax - xmin) / nbins

    start = cuda.grid(1)
    stride = cuda.gridsize(1)

    ### DEBUG FIRST THREAD
    if start == 0:
        from pdb import set_trace; set_trace()
    ###

    for i in range(start, x.shape[0], stride):
```

```
    bin_number = np.int32((x[i] + xmin)/bin_width)

    if bin_number >= 0 and bin_number < histogram_out.shape[0]:
        cuda.atomic.add(histogram_out, bin_number, 1)
```

The following decorator gives source file and problematic line when debugging.

```
@cuda.jit(debug=True)
def histogram(x, xmin, xmax, histogram_out):
```

```
cuda-memcheck python debug/ex3a.py
```

Output can look like:

```
========= CUDA-MEMCHECK
========= Invalid __global__ write of size 4
=========     at 0x00000ba8 in /Users/sseibert/continuum/conferences/gtc2017-numba/debug/
ex3a.py:17:cudapy::__main__::histogram$241(Array<float, int=1, C, mutable, aligned>, float, fl
oat, Array<int, int=1, C, mutable, aligned>)
```

The highlighted portion is the source file and line.

Extra Topics

Random Number Generator

```
import numpy as np
from numba import cuda
from numba.cuda.random import create_xoroshiro128p_states, xoroshiro128p_uniform_float32

threads_per_block = 64
blocks = 24
rng_states = create_xoroshiro128p_states(threads_per_block * blocks, seed=1)

@cuda.jit
def compute_pi(rng_states, iterations, out):
    """Find the maximum value in values and store in result[0]"""
    thread_id = cuda.grid(1)

    # Compute pi by drawing random (x, y) points and finding what
    # fraction lie inside a unit circle
    inside = 0
    for i in range(iterations):
        x = xoroshiro128p_uniform_float32(rng_states, thread_id)
        y = xoroshiro128p_uniform_float32(rng_states, thread_id)
```

```
        if x**2 + y**2 <= 1.0:
            inside += 1

    out[thread_id] = 4.0 * inside / iterations


out = np.zeros(threads_per_block * blocks, dtype=np.float32)
compute_pi[blocks, threads_per_block](rng_states, 10000, out)
print('pi:', out.mean())
```

Shared Memory

Shared memory is a section of memory that is visible at the block level. Different blocks cannot see each other's shared memory, and all the threads within a block see the same shared memory.

The actual code and benefits are difficult to understand. Not too important right now.

Generalized Universal Functions (Gufuncs)

In regular ufuncs, the elementary function is limited to element-by-element operations, whereas the generalized version (gufuncs) supports “sub-array” by “sub-array” operations.

Example decorators:

```
@guvectorize(['(float32[:], float32[:])'], # have to include the output array in the type signature
             '(i)->()',                    # map a 1D array to a scalar output
             target='cuda')
def l2_norm(vec, out):
```

```
@ guvectorize (['void(float32[:,:], float32[:,:], float32[:,:])'],

             '(m,n),(n,p)->(m,p)', target='cuda')

def matmulcore(A, B, C):
```

Example complete code:

```
from numba import guvectorize
import math
```

```
@guvectorize(['(float32[:], float32[:])'], # have to include the output array in the type signature
             '(i)->()',                    # map a 1D array to a scalar output
             target='cuda')
def l2_norm(vec, out):
    acc = 0.0
    for value in vec:
        acc += value**2
```

```
    out[0] = math.sqrt(acc)


angles = np.random.uniform(-np.pi, np.pi, 10)
coords = np.stack([np.cos(angles), np.sin(angles)], axis=1)
print(coords)
```

Timing

Use the timeit module in Python.

The following are possible methods and classes to use timeit:

```
timeit.timeit(stmt='pass', setup='pass', timer=<default timer>, number=1000000)

timeit.repeat(stmt='pass', setup='pass', timer=<default timer>, repeat=3, number=1000000)

timeit.default_timer()
```

stmt (statement) is the code that is being tested. The setup process is executed once and not time. This prevents things like imports from influencing the time. The number defines how many times to run the statement. Note, you don't get a time for each run of the statement. Repeat represents the number of times to run the test; each test consists of the statement executed "number" of times, and, at the end, a time is printed.

The timeit.default_timer() returns an object that provides the best clock available on your platform and version of Python automatically. You can use it as follows:

```
from timeit import default_timer as timer #from timeit file, import default_timer object.

start = timer() #same as start = timeit.default_timer()
# ...
end = timer()
print(end - start) # Time in seconds, e.g. 5.38091952400282
```

An example of using the other methods:

```
def test():
    """Stupid test function"""
    L = []
    for i in range(100):
        L.append(i)

if __name__ == '__main__':
    import timeit
    print(timeit.timeit("test()", setup="from __main__ import test"))
```

Note that this example uses the default timer and number parameters. Further the statement and setup are strings. Also note that the setup = “from __main__ import test” is necessary for timeit to recognize the add method.

The best practice is to get the minimum of the time overall several tests, like three tests.

```
import timeit

def add(a, b):
    return a + b

if __name__ == '__main__':
    times = timeit.repeat(stmt='add(1, 1)', setup='from __main__ import add', repeat=3, number=1000000)
    print(times)
    print(min(times))


[0.092061278, 0.09276553399999998, 0.09605395100000003]
0.092061278
```

CUDA-Enabled GPUs

The GeForce mx150, which is on the Asus Vivobook s14, is not CUDA-enabled as seen in the official Nvidia website https://developer.nvidia.com/cuda-gpus.

CuDNN

CuDNN is a library that uses Cuda to implement primitive deep neural network functions. There is a Python wrapper for CuDNN but it has new syntax to learn. Further, it is early in development.

The best option is to continue using Numba when building networks from scratch. Eventually, move to machine learning frameworks, which usually employ cuDNN in their GPU versions.

Good to Know

Although CUDA kernel launches are asynchronous, all GPU-related tasks placed in one stream (which is the default behavior) are executed sequentially.

Further, waiting for the GPU processes in queue to finish from methods like cuda.synchronize() is not expensive. Also, transferring data from host to device or vice versa is not expensive relative to time saved on large computations.

Call “cuda.device_array(shape)” to create an empty array directly in device. Elements are not guaranteed to be zeros.

The method “cuda.device_array_like(arr)” to create the output_device simply calls “cuda.device_array()” with information from the array. Thus, “cuda.device_array”, “cuda.device_array_like”, “cuda.to_device” all work to create device arrays.

References

- https://github.com/ContinuumIO/gtc2017-numba/blob/master/1%20-%20Numba%20Basics.ipynb
- https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction
- https://docs.python.org/2/library/timeit.html#
- https://stackoverflow.com/questions/11888772/when-to-call-cudadevicesynchronize