Open in Colab

In [ ]:
```
from numba import cuda
from numba import jit
import numpy as np
# from numba import vectorize, int32, int64, float32, float64
import matplotlib.pyplot as plt

%matplotlib inline
```

In [ ]:
```
!nvidia-smi
```

```
Mon Nov  7 10:49:45 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   68C    P8    10W /  70W |      0MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

In [ ]:
```
# Note the use of an `out` array. CUDA kernels written with `@cuda.jit` do not return values,
# just like their C counterparts. Also, no explicit type signature is required with @cuda.jit
@cuda.jit
def add_kernel(x, y, out):

    # The actual values of the following CUDA-provided variables for thread and block indices,
    # like function parameters, are not known until the kernel is launched.

    # This calculation gives a unique thread index within the entire grid (see the slides above for
    idx = cuda.grid(1)          # 1 = one dimensional thread grid, returns a single value.
                                # This Numba-provided convenience function is equivalent to
                                # `cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x`

    # This thread will do the work on the data element with the same index as its own
    # unique index within the grid.
    out[idx] = x[idx] + y[idx]
```

In [ ]:
```
n = 4096
x = np.arange(n).astype(np.int32) # [0...4095] on the host
y = np.ones_like(x)               # [1...1] on the host

d_x = cuda.to_device(x) # Copy of x on the device
d_y = cuda.to_device(y) # Copy of y on the device
d_out = cuda.device_array_like(d_x) # Like np.array_like, but for device arrays

# Because of how we wrote the kernel above, we need to have a 1 thread to one data element mapping,
# therefore we define the number of threads in the grid (128*32) to equal n (4096).
threads_per_block = 128
blocks_per_grid = 32
```

In [ ]:
```
add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
cuda.synchronize()
print(d_out.copy_to_host()) # Should be [1...4096]
```

```
/usr/local/lib/python3.7/dist-packages/numba/cuda/dispatcher.py:488: NumbaPerformanceWarning: Grid s
ize 32 will likely result in GPU under-utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
[   1    2    3 ... 4094 4095 4096]
```

In [ ]:
```
from numba import vectorize

@vectorize(['int32(int32, int32)'], target='cuda') # Type signature and target are required for the
def add_ufunc(x, y):
    return x + y
```

Let us run some benchmarks

In [ ]:
```
%timeit np.add(x, y)   # NumPy on CPU
```

```
1.47 µs ± 24.1 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [ ]:
```
%timeit add_ufunc(x, y) # Numba on GPU
```

```
/usr/local/lib/python3.7/dist-packages/numba/cuda/dispatcher.py:488: NumbaPerformanceWarning: Grid s
ize 4 will likely result in GPU under-utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
1.15 ms ± 9.89 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [ ]:
```
%timeit add_kernel[blocks_per_grid, threads_per_block](x, y, d_out) # hand crafted kernel - data fr
```

```
/usr/local/lib/python3.7/dist-packages/numba/cuda/cudadrv/devicearray.py:885: NumbaPerformanceWarnin
g: Host array used in CUDA kernel will incur copy overhead to/from device.
  warn(NumbaPerformanceWarning(msg))
938 µs ± 18.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [ ]:
```
%timeit add_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out) # hand crafted kernel - dat
```

```
72.7 µs ± 3.2 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

2D-diffusion

The 2D-diffusion equation is known as

$$\frac{\partial u}{\partial t} = \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

The same scheme as in the case of 1D diffusion problem will be adopted to account for spatial derivatives in both x and y direction. It consinst of a forward difference in time and two second-order spatial derivatives.

$$\frac{u_{i,j}^{n+1} - u_{i,j}^n}{\Delta t} = \nu \frac{u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n}{\Delta x^2} + \nu \frac{u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n}{\Delta y^2}$$

Solving the discretized equation for $u_{i,j}^{n+1}$

$$\begin{aligned} u_{i,j}^{n+1} = u_{i,j}^n &+ \frac{\nu \Delta t}{\Delta x^2}(u_{i+1,j}^n - 2u_{i,j}^n + u_{i-1,j}^n) \\ &+ \frac{\nu \Delta t}{\Delta y^2}(u_{i,j+1}^n - 2u_{i,j}^n + u_{i,j-1}^n) \end{aligned}$$

In [ ]:
```
# @vectorize(['float32[:](int32, int32, float32, float32, float32)'], target='parallel')  #

@jit(nopython=True)
def step_cpu(nx, ny, fact, A_in,  A_out):
    # loop over all points in domain (except boundary)
    for x in range(1,nx-1):
        for y in range(1,ny-1):
            d2tdx2 = A_in[x-1][y] - 2*A_in[x][y] + A_in[x+1][y]
            d2tdy2 = A_in[x][y-1] - 2*A_in[x][y] + A_in[x][y+1]
            #update temperatures
            A_out[x][y] = A_in[x][y]+fact*(d2tdx2 + d2tdy2)
```

In [ ]:
```
@cuda.jit
def step_kernel_gpu(nx, ny, fact, A_in,  A_out):
    x, y = cuda.grid(2)
    # The above is equivalent to the following 2 lines of code:
    # x = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x
    # y = cuda.blockIdx.y * cuda.blockDim.y + cuda.threadIdx.y

    # loop over all points in domain (except boundary)
    if (x > 0 and y > 0 and x < nx-1 and y < ny-1):
    # if (x < nx and y < ny):
        d2tdx2 = A_in[x-1][y] - 2*A_in[x][y] + A_in[x+1][y]
        d2tdy2 = A_in[x][y-1] - 2*A_in[x][y] + A_in[x][y+1]
        #update temperatures
        A_out[x][y] = A_in[x][y]+fact*(d2tdx2 + d2tdy2)
```

In [ ]:
```
diffusivity = 1e-2 # thermal diffusivity
nstep = int(1e4)     # number of time steps
#Specify our 2D dimensions
nx = 256
ny = 256
```

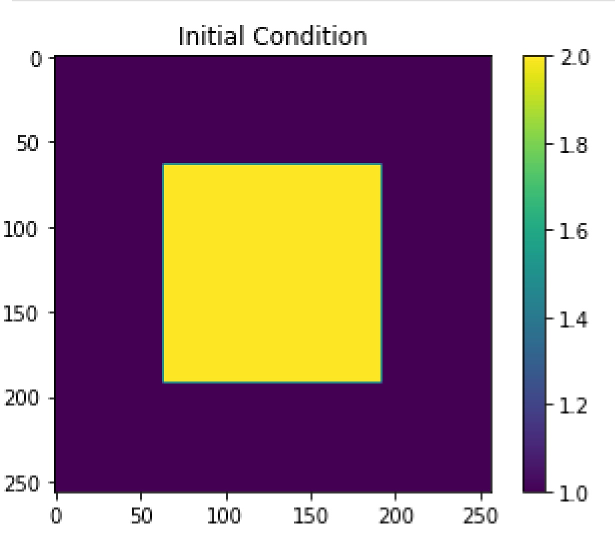
In [ ]:
```
threads_per_block = (32, 32) # 2D blocks, 32*32=1024 threads, this is the maximum number of threads
blocks = (8, 8) #2D grid
assert threads_per_block[0]*blocks[0] == nx and threads_per_block[1]*blocks[1] == ny

a1 = np.ones((nx, ny)).astype(np.float32) #create a nx * ny vector of 1's
a2 = np.ones((nx, ny)).astype(np.float32)

###Assign initial conditions
a1[int(nx/4):int(3*nx/4), int(ny/4):int(3*ny/4)] = 2
# a2[int(nx/4):int(3*nx/4), int(ny/4):int(3*ny/4)] = 2

d_a1 = cuda.to_device(a1)
d_a2 = cuda.to_device(a2)
```

In [ ]:
```
plt.imshow(d_a1.copy_to_host())
plt.colorbar()
plt.title('Initial Condition')
plt.show()
```



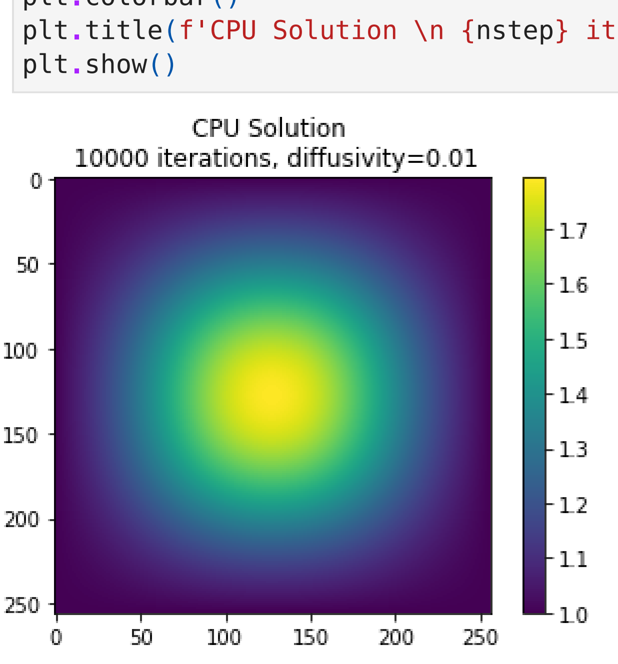
In [ ]:
```
def iterate_cpu(a1, a2):
    for i in range(0,nstep):
        step_cpu(nx, ny, diffusivity, a1,  a2)
        # // swap the temperature pointers
        tmp = a1
        a1 = a2
        a2 = tmp
```

In [ ]:
```
%timeit iterate_cpu(a1, a2)
```

```
3.18 s ± 17.8 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [ ]:
```
solution = a1

plt.imshow(solution)
plt.colorbar()
plt.title(f'CPU Solution \n {nstep} iterations, diffusivity={diffusivity}')
plt.show()
```



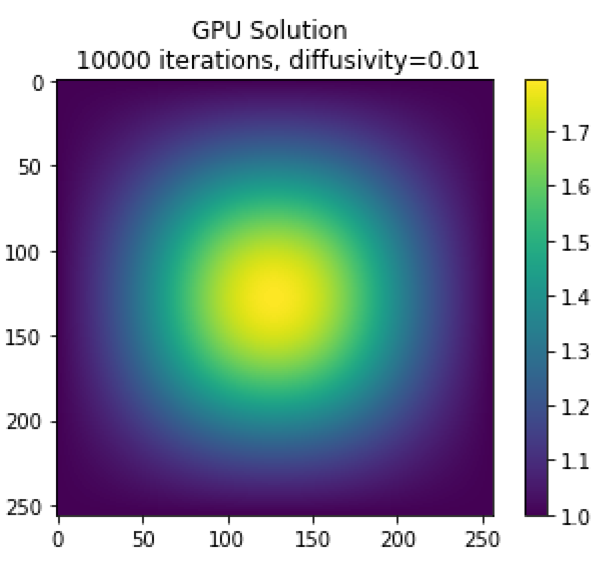
In [ ]:
```
def iterate_gpu(d_a1,  d_a2):
    for i in range(0,nstep):
        step_kernel_gpu[blocks, threads_per_block](nx, ny, diffusivity, d_a1,  d_a2)
        cuda.synchronize()
        # // swap the temperature pointers
        tmp = d_a1
        d_a1 = d_a2
        d_a2 = tmp
```

In [ ]:
```
%timeit iterate_gpu(d_a1,  d_a2)
```

```
/usr/local/lib/python3.7/dist-packages/numba/cuda/dispatcher.py:488: NumbaPerformanceWarning: Grid s
ize 64 will likely result in GPU under-utilization due to low occupancy.
  warn(NumbaPerformanceWarning(msg))
1.2 s ± 7.09 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [ ]:
```
solution = d_a1.copy_to_host()

plt.imshow(solution)
plt.colorbar()
plt.title(f'GPU Solution \n {nstep} iterations, diffusivity={diffusivity}')
plt.show()
```


GPU Solution
10000 iterations, diffusivity=0.01