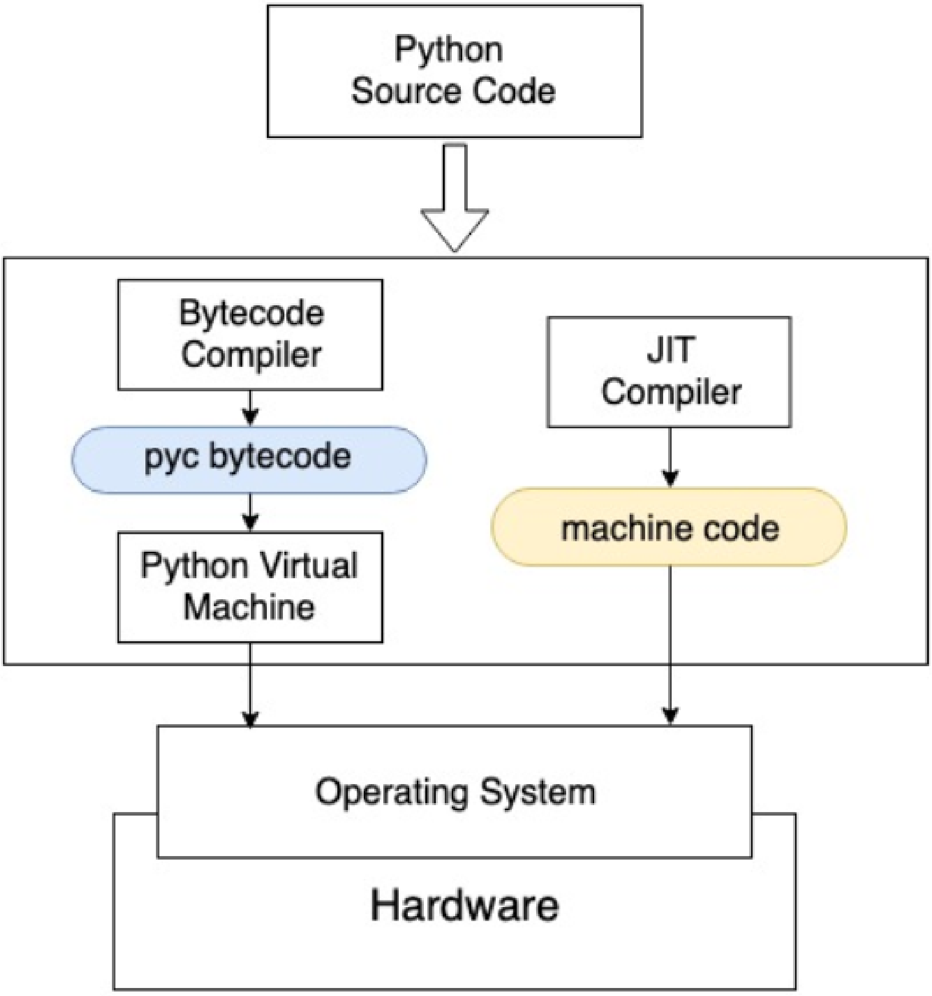
Numba Notes

Numba is a compiler for Python which takes the bytecode and compile it into the native language. Therefore, the interpreter is bypassed.



Reference: https://lulaoshi.info/gpu/python-cuda/numba

```
(.venv) erebus@DESKTOP-5E9UBUO:~/python_dev/numba_dev$ python monte_carlo_pi.py
python 100000 loop: 0.036972761154174805 result: 3.13572
numba 100000 loop: 0.3632771968841553 result: 3.1336
numba parallel 100000 loop: 0.6998176574707031 result: 3.13952
python 1000000 loop: 0.31112027168273926 result: 3.140024
numba 1000000 loop: 0.009532451629638672 result: 3.140808
numba parallel 1000000 loop: 0.0053675174713134766 result: 3.1431
python 10000000 loop: 3.127183198928833 result: 3.1415848
numba 10000000 loop: 0.09232878684997559 result: 3.1409232
numba parallel 10000000 loop: 0.0224611759185791 result: 3.1416296
python 100000000 loop: 29.81264090538025 result: 3.14154524
numba 100000000 loop: 0.9338881969451904 result: 3.14162352
numba parallel 100000000 loop: 0.15855145454406738 result: 3.14152764
```

Example of using numba to do monte carlo estimation of the value of *pi*.

In order to use numba in Windows Subsystem for Linux, *export NUMBA_CUDA_DRIVER="/usr/lib/wsl/lib/libcuda.so.1"* need to be added into the bash configure file.

Works well with Numba

- Numerical code, loop
- Large amounts of data
- Data-Parallel operations

Things that can be tricky to optimize, particularly on CUDA

- Code using lots of strings or dicts
- Inherently serial logic
- Calling lots of already-compiled code
- Code with a lot of object-oriented patterns and features
- Code that's already been heavily optimized using another tool / paradigm

Reference: https://www.youtube.com/watch?v=xes5ri5ccWY

```
from numba import cuda, njit, prange
import numba
import numpy as np
import pylab
from time import perf_counter

print(numba.__version__)
cuda.detect()
```
[1] ✓ 1.4s

```
0.53.1
Found 1 CUDA devices
id 0    b'NVIDIA GeForce RTX 2070 with Max-Q Design'                [SUPPORTED]
                      compute capability: 7.5
                           pci device id: 0
                              pci bus id: 1
Summary:
        1/1 devices are supported
True
```

Code Example

Generate gaussian in 2d, smooth and visualize it.

Setting the parameters

```
ITERATIONS = 20000
POINTS = 1000 # A grid of 1000 by 1000 points
```
[2] ✓ 0.2s

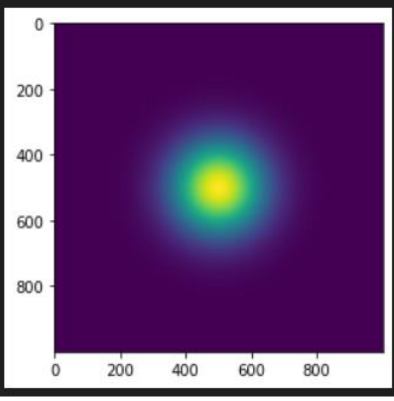
Using Numba's @njit decorator

@njit decorator

- Compiles the Python bytecode to native code
- Single-threaded on CPU

```python
@njit
def gauss2d(x, y):
    grid = np.empty_like(x)

    a = 1.0 / np.sqrt(2 * np.pi)

    for i in range(grid.shape[0]):
        for j in range(grid.shape[1]):
            grid[i, j] = a * np.exp(-(x[i, j]**2 / 2 + y[i, j]**2 / 2))

    return grid

X = np.linspace(-5, 5, POINTS)
Y = np.linspace(-5, 5, POINTS)
x, y = np.meshgrid(X, Y)

z = gauss2d(x, y)

pylab.imshow(z)
pylab.show()
```

[2] ✓ 1.1s



The meshgrid() function can be used to generate coordinates for all the combination in the given two vectors.

```python
import numpy as np

X = [1, 2, 3, 4]
Y = [5, 6, 7]
x, y = np.meshgrid(X,Y)
```

```
[[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
[[5 5 5 5]
 [6 6 6 6]
 [7 7 7 7]]
```

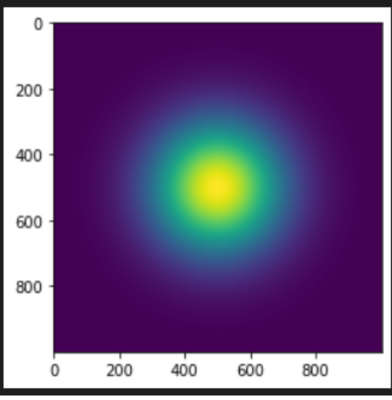
Parallelize execution on the CPU

- With parallel = True
- Use prange instead of range to mark the loop needed parallelism.

When the first call to the function will trigger compiler to compile the function which will cost lots of time. We can also pass cache=True to the njit which will then save the compiled object code and prevent recompile next time running the program.

```python
from time import perf_counter


@njit(parallel=True)
def smooth_jit(x0, x1):
    for i in prange(1, x0.shape[0] - 1):
        for j in range(1, x0.shape[1] - 1):
            x1[i, j] = 0.25 * (x0[i, j - 1] + x0[i, j + 1] + x0[i - 1, j] + x0[i + 1, j])

def run_cpu_jit():
    z0 = z.copy()
    z1 = np.zeros_like(z0)

    # Warm up JIT
    smooth_jit(z0, z1)

    start = perf_counter()

    for i in range(ITERATIONS):
        if(i % 2) == 0:
            smooth_jit(z0, z1)
        else:
            smooth_jit(z1, z0)

    end = perf_counter()

    time_cpu = end - start

    return z0, time_cpu

z_cpu, time_cpu = run_cpu_jit()

pylab.imshow(z_cpu)
pylab.show()
```

[10] ✓ 45.4s



If TBB version is too old or not exist, install TBB by *pip install tbb*.

Parallel implementation on the GPU

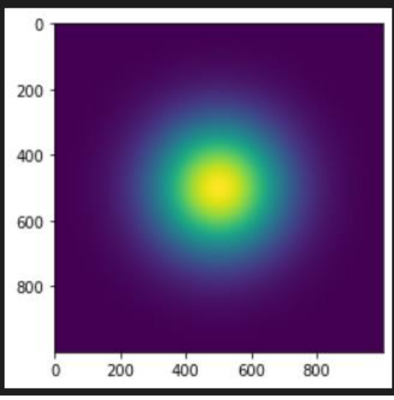
@cuda.jit decorator

- The index can be calculated form cuda.grid()
- Similar to CUDA C programming, the kernel indices need to checked if in bounds
- The data need to be copy to the device first otherwise, implicit copies will be invoke every time the kernel being called and executed.

```python
from time import perf_counter


@cuda.jit
def smooth_cuda(x0, x1):
    i, j = cuda.grid(2)

    i_in_bounds = (i > 0) and (i < (x0.shape[0] - 1))
    j_in_bounds = (j > 0) and (j < (x0.shape[1] - 1))

    if i_in_bounds and j_in_bounds:
        x1[i, j] = 0.25 * (x0[i, j - 1] + x0[i, j + 1] + x0[i - 1, j] + x0[i + 1, j])

def run_cuda_jit():
    # Copy to device
    z0 = cuda.to_device(z)
    z1 = cuda.device_array_like(np.zeros_like(z))

    # Warm up JIT
    blockdim = (16, 16)
    # Invoke one extra block for each dimention
    griddim = ((z0.shape[0] // blockdim[0]) + 1, (z0.shape[1] // blockdim[1]) + 1)
    smooth_cuda[griddim, blockdim](z0, z1)

    start = perf_counter()

    for i in range(ITERATIONS):
        if(i % 2) == 0:
            smooth_cuda[griddim, blockdim](z0, z1)
        else:
            smooth_cuda[griddim, blockdim](z1, z0)

    # Make sure the GPU is finished before we stop timing
    cuda.synchronize()

    end = perf_counter()

    time_cuda = end - start
    return z0.copy_to_host(), time_cuda

z_cuda, time_cuda = run_cuda_jit()

pylab.imshow(z_cuda)
pylab.show()
```

[12] ✓ 4.3s

...

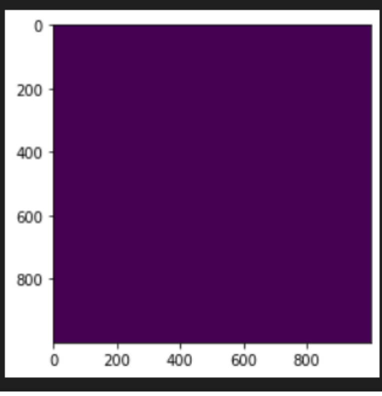


Check the difference between the solutions (should be fairly small)

```python
diff = np.abs(z_cpu - z_cuda)
pylab.imshow(diff)
pylab.show()
```

[13] ✓ 0.3s

...



Compare the performance

```
print(f"CPU time: {time_cpu:2.2f} seconds")
print(f"CUDA time: {time_cuda:2.2f} seconds")
```

[14] ✓ 0.3s

```
CPU time: 43.98 seconds
CUDA time: 3.67 seconds
```

Since the problem is an embarrassingly parallel workload, the performance of GPU is significantly better.