Python: CUDA

Spring 2025

What is CUDA?

CUDA is a parallel computing platform and programming model that uses graphics processing units (GPUs) to speed up applications. CUDA allows to dramatically speed up computing applications

How CUDA works:

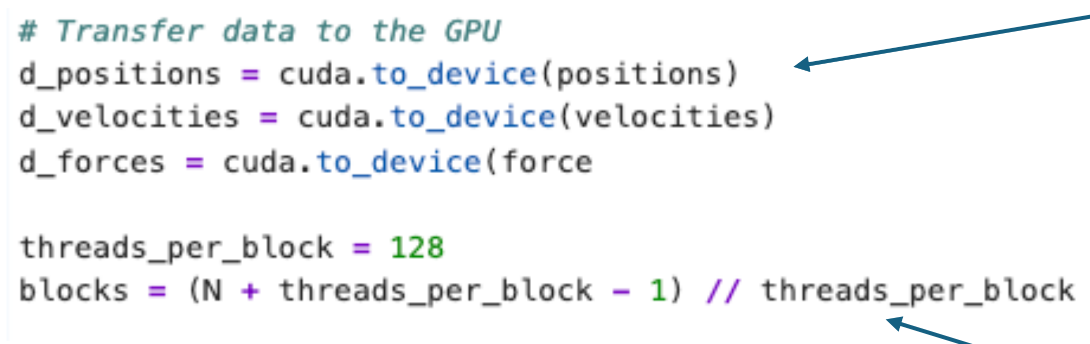
- Use CUDA to write functions called CUDA kernels that run on GPUs.
- The CPU copies data from the main RAM to the GPU's memory.
- The CPU tells the GPU to execute the CUDA kernel in parallel.
- The GPU executes the code in a block that organizes threads into a grid.
- The GPU copies the final result back to the main memory.

A simple molecular dynamics simulation in Python that uses CUDA for GPU acceleration using Numba.

In this example, we simulate a set of particles interacting via a Lennard-Jones potential. Note that this code is meant as a demonstration - it uses a naïve algorithm without neighbor lists or other common optimizations found in production MD codes.

```python
import numpy as np
from numba import cuda
import math

# Simulation parameters
N = 1024          # Number of particles
dim = 3           # 3D simulation
box = 10.0        # Size of the simulation box (assumed cubic)
mass = 1.0        # Mass of each particle
epsilon = 1.0     # Depth of the Lennard-Jones potential well
sigma = 1.0       # Finite distance at which the inter-particle potential is zero
cutoff = 2.5 * sigma  # Cutoff distance for the potential
dt = 0.005        # Time step
nsteps = 1000     # Number of simulation steps

# Initialize positions and velocities
positions = np.random.rand(N, dim).astype(np.float32) * box
velocities = (np.random.rand(N, dim).astype(np.float32) - 0.5) * 0.1
forces = np.zeros((N, dim), dtype=np.float32)
```

Transfer data to the GPU



```python
# Transfer data to the GPU
d_positions = cuda.to_device(positions)
d_velocities = cuda.to_device(velocities)
d_forces = cuda.to_device(force

threads_per_block = 128
blocks = (N + threads_per_block - 1) // threads_per_block
```

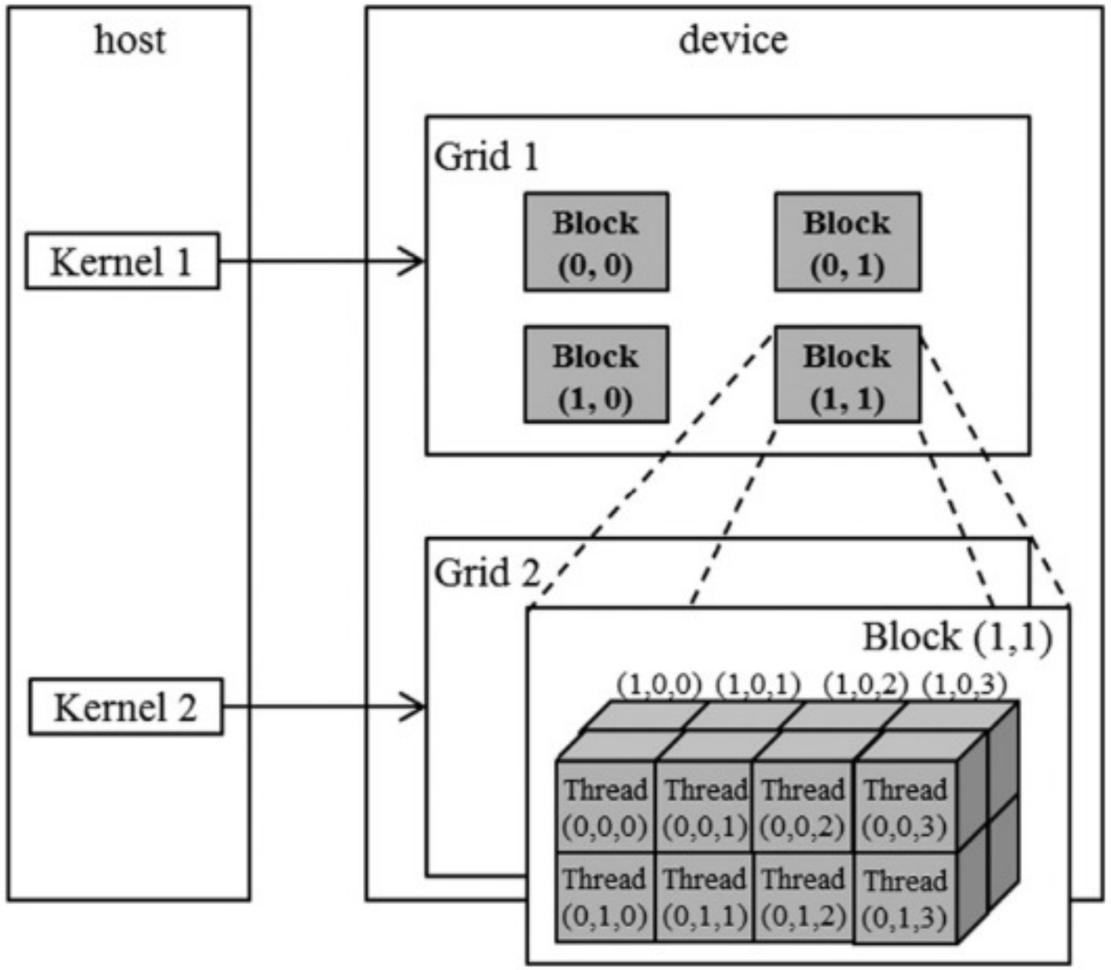
cuda.to_device() is a function in Numba's CUDA module that transfers data from the host (CPU) memory to the device (GPU) memory. It takes a NumPy array (or similar array-like object) as input, allocates space for it on the GPU, copies the data over, and returns a device array (typically a DeviceNDArray) that you can use in CUDA kernels.

threads_per_block sets the number of threads that will run within each block when you launch a CUDA kernel. Common Choice is 128 (or 256/512) as a block size because it often provides good occupancy and performance on many GPU architectures.

CUDA Grid and Block Organization

- **Threads:** The smallest unit of execution in CUDA.
- **Blocks:** Threads are grouped into blocks. Each block runs on a single Streaming Multiprocessor (SM) and can share fast on-chip memory.
- **Grids:** Blocks are further organized into a grid when a kernel is launched.

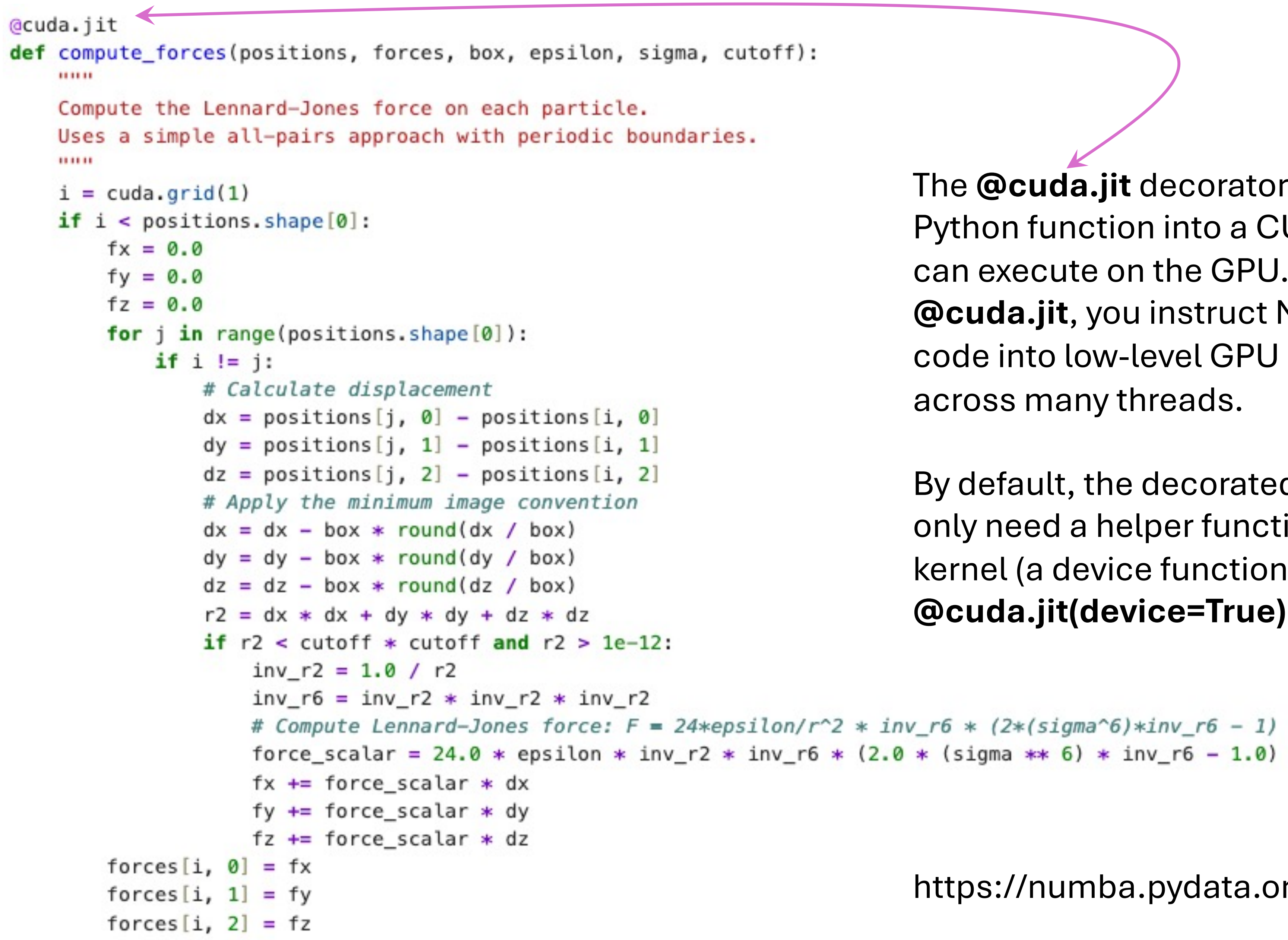
When launching a kernel, you typically specify both the number of blocks and the number of threads per block.



https://christianjmills.com/posts/cuda-mode-notes/lecture-002/

https://harmanani.github.io/classes/csc447/Notes/Lecture15.pdf

Just-In-Time (JIT)

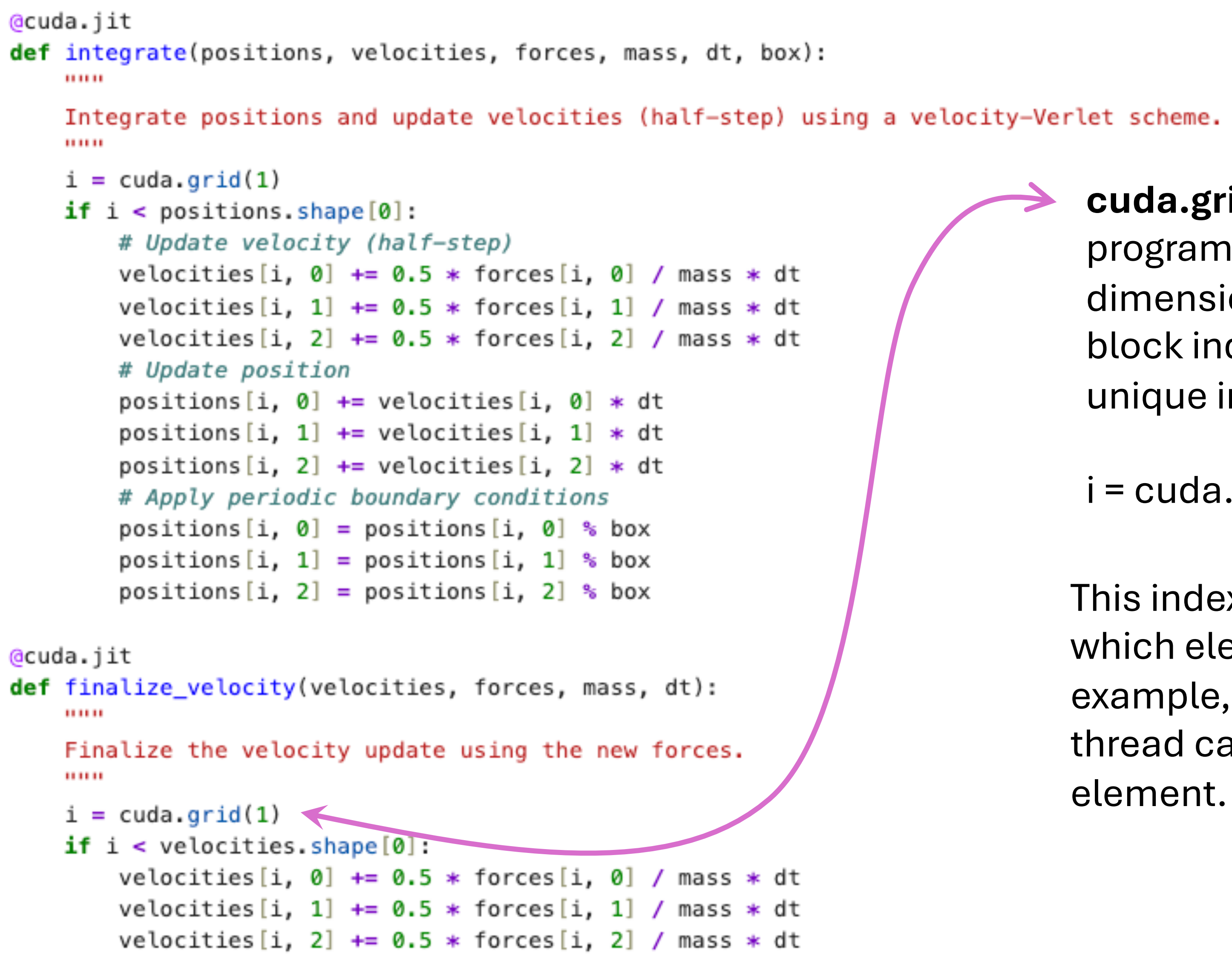


```python
@cuda.jit
def compute_forces(positions, forces, box, epsilon, sigma, cutoff):
    """
    Compute the Lennard-Jones force on each particle.
    Uses a simple all-pairs approach with periodic boundaries.
    """
    i = cuda.grid(1)
    if i < positions.shape[0]:
        fx = 0.0
        fy = 0.0
        fz = 0.0
        for j in range(positions.shape[0]):
            if i != j:
                # Calculate displacement
                dx = positions[j, 0] - positions[i, 0]
                dy = positions[j, 1] - positions[i, 1]
                dz = positions[j, 2] - positions[i, 2]
                # Apply the minimum image convention
                dx = dx - box * round(dx / box)
                dy = dy - box * round(dy / box)
                dz = dz - box * round(dz / box)
                r2 = dx * dx + dy * dy + dz * dz
                if r2 < cutoff * cutoff and r2 > 1e-12:
                    inv_r2 = 1.0 / r2
                    inv_r6 = inv_r2 * inv_r2 * inv_r2
                    # Compute Lennard-Jones force: F = 24*epsilon/r^2 * inv_r6 * (2*(sigma^6)*inv_r6 - 1)
                    force_scalar = 24.0 * epsilon * inv_r2 * inv_r6 * (2.0 * (sigma ** 6) * inv_r6 - 1.0)
                    fx += force_scalar * dx
                    fy += force_scalar * dy
                    fz += force_scalar * dz
        forces[i, 0] = fx
        forces[i, 1] = fy
        forces[i, 2] = fz
```

The **@cuda.jit** decorator in Numba is used to compile a Python function into a CUDA kernel or device function that can execute on the GPU. By decorating your function with **@cuda.jit**, you instruct Numba to translate your Python code into low-level GPU code that can be run in parallel across many threads.

By default, the decorated function becomes a kernel. If you only need a helper function that is called from within a kernel (a device function), you can specify: **@cuda.jit(device=True)**

https://numba.pydata.org/numba-doc/0.13/CUDAJit.html

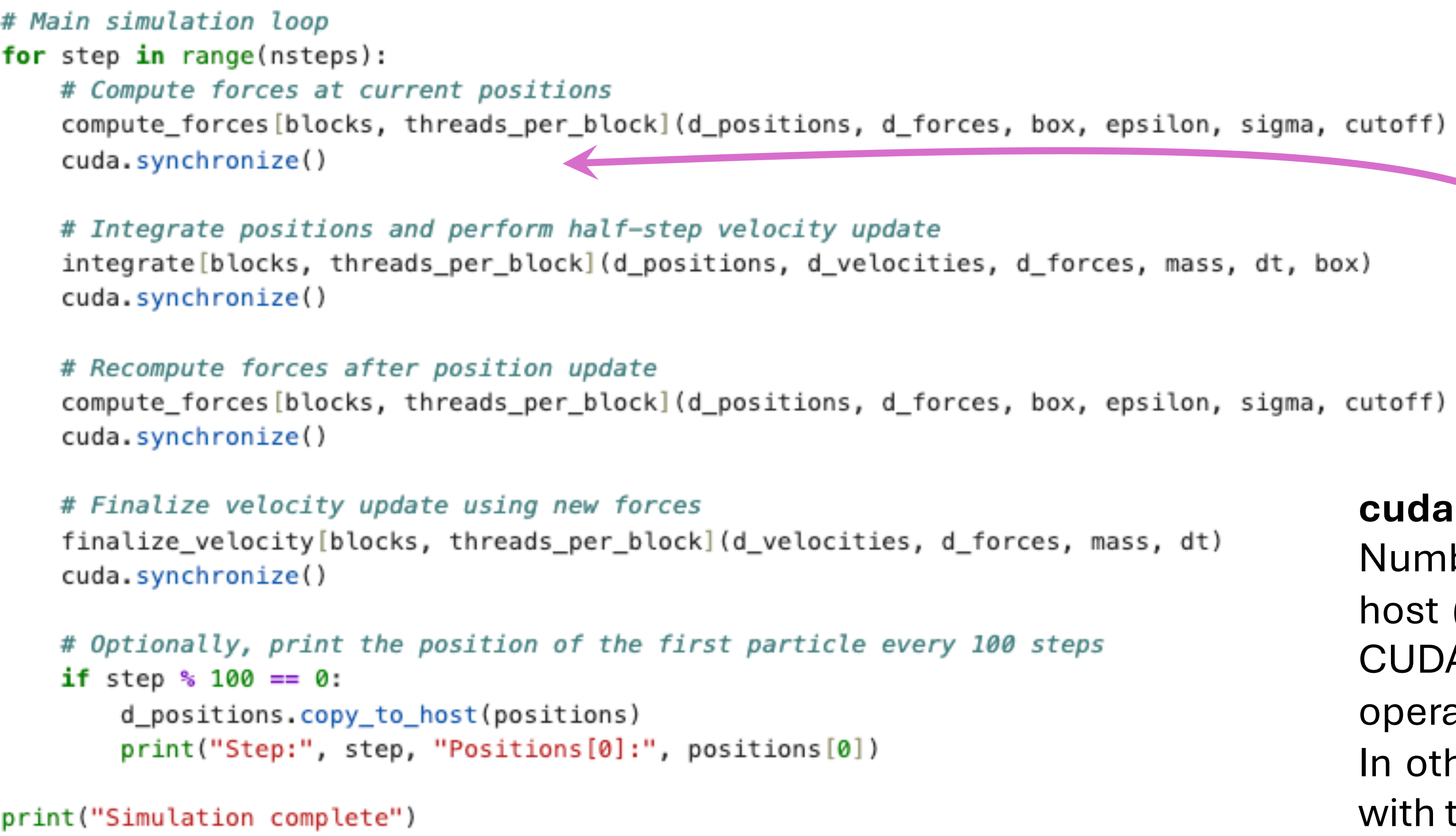


```python
@cuda.jit
def integrate(positions, velocities, forces, mass, dt, box):
    """
    Integrate positions and update velocities (half-step) using a velocity-Verlet scheme.
    """
    i = cuda.grid(1)
    if i < positions.shape[0]:
        # Update velocity (half-step)
        velocities[i, 0] += 0.5 * forces[i, 0] / mass * dt
        velocities[i, 1] += 0.5 * forces[i, 1] / mass * dt
        velocities[i, 2] += 0.5 * forces[i, 2] / mass * dt
        # Update position
        positions[i, 0] += velocities[i, 0] * dt
        positions[i, 1] += velocities[i, 1] * dt
        positions[i, 2] += velocities[i, 2] * dt
        # Apply periodic boundary conditions
        positions[i, 0] = positions[i, 0] % box
        positions[i, 1] = positions[i, 1] % box
        positions[i, 2] = positions[i, 2] % box

@cuda.jit
def finalize_velocity(velocities, forces, mass, dt):
    """
    Finalize the velocity update using the new forces.
    """
    i = cuda.grid(1)
    if i < velocities.shape[0]:
        velocities[i, 0] += 0.5 * forces[i, 0] / mass * dt
        velocities[i, 1] += 0.5 * forces[i, 1] / mass * dt
        velocities[i, 2] += 0.5 * forces[i, 2] / mass * dt
```

cuda.grid is a convenience function in Numba's CUDA programming model that computes the global, one-dimensional index for the current thread. It combines the block index and the thread index within the block to yield a unique index across the entire grid.

i = cuda.blockIdx.x * cuda.blockDim.x + cuda.threadIdx.x

This index is commonly used in CUDA kernels to determine which element of an array the thread should process. For example, in an element-wise operation on an array, each thread can use its unique **i** to access the corresponding element.



```python
# Main simulation loop
for step in range(nsteps):
    # Compute forces at current positions
    compute_forces[blocks, threads_per_block](d_positions, d_forces, box, epsilon, sigma, cutoff)
    cuda.synchronize()

    # Integrate positions and perform half-step velocity update
    integrate[blocks, threads_per_block](d_positions, d_velocities, d_forces, mass, dt, box)
    cuda.synchronize()

    # Recompute forces after position update
    compute_forces[blocks, threads_per_block](d_positions, d_forces, box, epsilon, sigma, cutoff)
    cuda.synchronize()

    # Finalize velocity update using new forces
    finalize_velocity[blocks, threads_per_block](d_velocities, d_forces, mass, dt)
    cuda.synchronize()

    # Optionally, print the position of the first particle every 100 steps
    if step % 100 == 0:
        d_positions.copy_to_host(positions)
        print("Step:", step, "Positions[0]:", positions[0])

print("Simulation complete")
```

cuda.synchronize() is a function in Numba's CUDA module that blocks the host (CPU) until all previously launched CUDA kernels and asynchronous operations on the GPU have completed. In other words, it synchronizes the CPU with the GPU.

CUDA operations are asynchronous by default. This means that after you launch a kernel or initiate a data transfer, the host code continues executing without waiting for the GPU operation to finish. Calling **cuda.synchronize()** forces the host to wait until all preceding GPU tasks have been completed.

Synchronization is often used when you need to accurately measure the execution time of a kernel or verify that all GPU operations have finished before accessing or transferring data back to the host. It can also help with debugging.

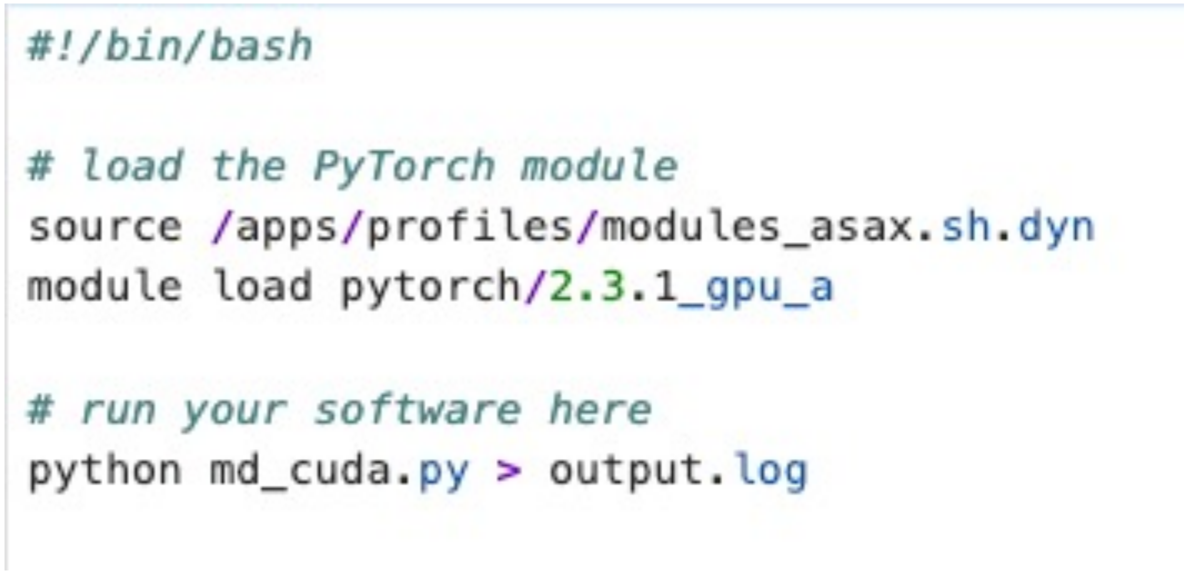
```python
# Main simulation loop
for step in range(nsteps):
    # Compute forces at current positions
    compute_forces[blocks, threads_per_block](d_positions, d_forces, box, epsilon, sigma, cutoff)
    cuda.synchronize()

    # Integrate positions and perform half-step velocity update
    integrate[blocks, threads_per_block](d_positions, d_velocities, d_forces, mass, dt, box)
    cuda.synchronize()

    # Recompute forces after position update
    compute_forces[blocks, threads_per_block](d_positions, d_forces, box, epsilon, sigma, cutoff)
    cuda.synchronize()

    # Finalize velocity update using new forces
    finalize_velocity[blocks, threads_per_block](d_velocities, d_forces, mass, dt)
    cuda.synchronize()

    # Optionally, print the position of the first particle every 100 steps
    if step % 100 == 0:
        d_positions.copy_to_host(positions)
        print("Step:", step, "Positions[0]:", positions[0])

print("Simulation complete")
```

The **.copy_to_host()** method is used on a device array (an array stored in GPU memory) to transfer its data back to the host (CPU) memory as a NumPy array. This is typically done after performing computations on the GPU when you need to access or further process the results on the CPU.

copy_to_host() returns a NumPy array containing the data from the GPU, making it accessible for further processing in your Python code.

Run it on Alabama HPC

`myproject.sh`



```
#!/bin/bash

# load the PyTorch module
source /apps/profiles/modules_asax.sh.dyn
module load pytorch/2.3.1_gpu_a

# run your software here
python md_cuda.py > output.log
```

Make the script executable like this

```
chmod +x myproject
```

Then submit the script to the queue like this

```
run_gpu myproject.sh
```