Week 15-16

CUDA with numba python

1. What is CUDA?
2. What is Numba?
3. How do CUDA and Numba work together?
4. Basic CUDA concepts for parallel programming
5. Simple coding examples with Numba + CUDA
6. When to use it

1. What is CUDA?

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by NVIDIA . It allows developers to use NVIDIA GPUs for general purpose processing (GPGPU).

With CUDA, you can offload compute-intensive tasks from the CPU to the GPU, leading to significant speedups — especially for data-parallel operations.

2. What is Numba?

Numba is a just-in-time (JIT) compiler in Python that translates a subset of Python and NumPy code into fast machine code using the LLVM compiler infrastructure.

It supports:

- CPU-based JIT compilation
- GPU acceleration via CUDA

The key feature here is that you can write Python functions and compile them to run on the GPU using CUDA , with minimal changes and no need to write C/C++ or CUDA C code directly.

3. How do CUDA and Numba Work Together?

Numba provides a high-level interface to program NVIDIA GPUs using CUDA, directly from Python.

You can write a function in Python, decorate it with special Numba decorators like `@cuda.jit`, and Numba will compile and execute it on your GPU.

This makes it very convenient for Python users who want to harness GPU power without diving deep into CUDA C.

4. Basic CUDA Concepts

Here are some fundamental CUDA concepts:

CONCEPT	DESCRIPTION
Thread	The smallest unit of execution. Each thread runs the same kernel function.
Block	A group of threads. Threads in a block can communicate/synchronize.
Grid	A collection of blocks. Blocks are independent and cannot synchronize.
Kernel	A function that runs in parallel on the GPU. You launch kernels from the host (CPU).

Example analogy:

- Think of a thread as a worker
- A block is a team of workers
- A grid is a collection of teams

5. Simple Coding Examples with Numba + CUDA

Let's start with a basic example: adding two arrays element-wise using CUDA via Numba.

✅ Prerequisites:

Make sure you have:

```
pip install numba numpy
```

Also ensure you have an NVIDIA GPU and the appropriate drivers installed.

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

# Define the CUDA kernel
@cuda.jit
def add_kernel(a, b, c):
    i = cuda.grid(1)  # Get index in 1D grid
    if i < a.shape[0]:  # Prevent out-of-bounds access
        c[i] = a[i] + b[i]

# Host code
def main():
    n = 100000
    a = np.arange(n, dtype=np.float32)
    b = np.arange(n, dtype=np.float32)
    c = np.zeros(n, dtype=np.float32)

    # Allocate device memory and copy data
    d_a = cuda.to_device(a)
    d_b = cuda.to_device(b)
    d_c = cuda.device_array_like(c)

    # Set up grid/block sizes
    threads_per_block = 256
    blocks_per_grid = (n + threads_per_block - 1) // threads_per_block

    # Launch the kernel
    add_kernel[blocks_per_grid, threads_per_block](d_a, d_b, d_c)

    # Copy result back to host
    c = d_c.copy_to_host()

    print("Result (first 5):", c[:5])

if __name__ == '__main__':
```

Let's compare the performance of a simple array addition using:

1. A CPU-based Numba-optimized version
2. A GPU-based Numba CUDA version

You've already provided the CUDA GPU-based code (array addition with `@cuda.jit`). Now, we'll implement an equivalent CPU version using Numba's `@jit` decorator and measure both versions.

🧪 Goal

Compare the execution time of:

- CPU-based array addition (`@jit`)
- GPU-based array addition (`@cuda.jit`)

```python
from numba import jit, cuda
import numpy as np
from time import perf_counter

# CUDA kernel
@cuda.jit
def add_kernel_gpu(a, b, c):
    i = cuda.grid(1)
    if i < a.size:
        c[i] = a[i] + b[i]

# CPU kernel with JIT optimization
@jit(nopython=True)
def add_kernel_cpu(a, b, c):
    for i in range(a.size):
        c[i] = a[i] + b[i]
    return c

def main():
    n = 10_000_000
    print(f"Array size: {n}")

    # Host arrays
    a = np.arange(n).astype(np.float32)
    b = 2 * np.arange(n).astype(np.float32)
    c_cpu = np.zeros_like(a)

    # CPU timing
    start = perf_counter()
    add_kernel_cpu(a, b, c_cpu)
    end = perf_counter()
    print(f"CPU Time: {end - start:.4f} seconds")

    # GPU setup
    d_a = cuda.to_device(a)
    d_b = cuda.to_device(b)
    d_c = cuda.device_array_like(a)
```

```
    # Kernel-only timing
    start = perf_counter()
    threads_per_block = 256
    blocks_per_grid = (n + threads_per_block - 1) // threads_per_block
    add_kernel_gpu[blocks_per_grid, threads_per_block](d_a, d_b, d_c)
    cuda.synchronize()  # Wait for GPU to finish
    end = perf_counter()
    print(f"GPU (kernel only): {end - start:.4f} seconds")

    # Optional: Copy result back
    # c_gpu = d_c.copy_to_host()

if __name__ == '__main__':
    main()
```

Sample Output

```
Array size: 10000000
CPU Time: 0.0957 seconds
GPU (kernel only): 0.0155 seconds
```

🧠 Key Takeaways

FEATURE	CPU (`@JIT`)	GPU (`@CUDA.JIT`)
Execution Unit	Single-core / Multi-core CPU	Massively parallel GPU
Memory	RAM	Device memory (VRAM)
Overhead	Low (no transfer cost)	Includes device memory transfers
Performance	Good for small data	Excels at large-scale parallelism

💡 For small arrays (<10,000 elements), CPU may outperform GPU due to memory copy overhead.

🚀 Try It With Larger Data

Try increasing `n` to `100_000_000` and observe how the GPU scales better than the CPU.

🚀 Matrix Multiplication with CUDA (via Numba)

```
import numpy as np
from numba import cuda, float32
from time import perf_counter

# Define block size for CUDA
TPB = 16  # Threads per block (Typically 16x16 for matrices)

# CPU version for comparison
def matmul_cpu(A, B, C):
    for row in range(C.shape[0]):
        for col in range(C.shape[1]):
            tmp = 0.
            for k in range(A.shape[1]):
                tmp += A[row, k] * B[k, col]
            C[row, col] = tmp
    return C

# CUDA kernel using Numba
@cuda.jit
def matmul_gpu(A, B, C):
    row = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y
    col = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    if row < C.shape[0] and col < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[row, k] * B[k, col]
        C[row, col] = tmp

# Helper function to run GPU version
def run_matmul_gpu(A, B):
    rows, cols = A.shape[0], B.shape[1]
    C_gpu = np.zeros((rows, cols), dtype=np.float32)
    threads_per_block = (TPB, TPB)
    blocks_per_grid_x = (cols + TPB - 1) // TPB
    blocks_per_grid_y = (rows + TPB - 1) // TPB
    blocks_per_grid = (blocks_per_grid_x, blocks_per_grid_y)

    d_A = cuda.to_device(A)
```

```
    d_B = cuda.to_device(B)
    d_C = cuda.to_device(C_gpu)

    matmul_gpu[blocks_per_grid, threads_per_block](d_A, d_B, d_C)

    C_result = d_C.copy_to_host()
    return C_result

# Main test function
def main():
    # You can change this size to see different performance impacts
    N = 1024
    A = np.random.rand(N, N).astype(np.float32)
    B = np.random.rand(N, N).astype(np.float32)
    C_cpu = np.zeros((N, N), dtype=np.float32)

    print(f"Matrix size: {N}x{N}")

    # CPU Timing
    start = perf_counter()
    matmul_cpu(A, B, C_cpu)
    end = perf_counter()
    print(f"CPU Time: {end - start:.4f} seconds")

    # GPU Timing
    start = perf_counter()
    C_gpu = run_matmul_gpu(A, B)
    end = perf_counter()
    print(f"GPU Time: {end - start:.4f} seconds")

    # Check result accuracy
    assert np.allclose(C_cpu, C_gpu, atol=1e-4), "CPU and GPU results
do not match!"

if __name__ == '__main__':
    main()
```

📊 Sample Output

Depending on your hardware, you may get something like:

```
Matrix size: 1024x1024
CPU Time: 15.2345 seconds
GPU Time: 0.2134 seconds
```

This shows a significant speedup using the GPU!