EECS E6893 Big Data Analytics

HW4: GPU Programming, Deploying LLMs

Tutorial · GPU foundations

EECS E6893 · Big Data Analytics · Fall 2026 · TA: Yanhao Bai yb2630@columbia.edu

Agenda

GPU programming: SIMT, CUDA memory, Numba labs, speedup analysis

LLM fundamentals: tokens, attention, decoding, quantization (tutorial i)

Deploying & using LLMs: local inference, OpenAI-compatible serving, remote APIs (tutorial i)

Mini-Project: your dataset → EDA → meaningful analysis → launchpad proposal (tutorial i)

CPU vs. GPU

CPU: few powerful cores, high clock, large caches → good for sequential, branchy, latency-critical work

GPU: thousands of simple cores, massive parallelism, high memory bandwidth → good for data-parallel work

Rule of thumb: same operation over millions of elements = GPU; complex control flow = CPU

Cost: PCIe host↔device transfers can dominate small workloads

CUDA execution model

Thread: the smallest execution unit; runs the kernel on one data element

Warp: 32 threads executing in lockstep (SIMT); divergence is expensive

Block: group of threads that can synchronize and share shared memory

Grid: all blocks of one kernel launch

- Example: vector add on N elements → launch ceil(N/256) blocks × 256 threads
- Grid-stride loop: one kernel handles any N: for i in range(tid, N, gridDim.x*blockDim.x)

CUDA memory hierarchy

Registers: per-thread, fastest; too many per thread → spills, lower occupancy

Shared memory: per-block scratchpad; enables tiling (e.g., matrix multiply)

Global memory: device DRAM, large but slow; coalesced access = adjacent threads read adjacent addresses

Constant / texture / L2: cached read-only paths

Bank conflicts: threads hitting the same shared-memory bank serialize (32 banks)

Occupancy: active warps / max warps — limited by registers and shared memory per thread

Numba: GPU programming in Python

```
from numba import cuda
import numpy as np
@cuda.jit            # JIT-compile to GPU (PTX)
def add_kernel(a, b, c):
    i = cuda.grid(1)          # global thread index
    if i < a.size:
        c[i] = a[i] + b[i]
threads = 256
blocks = (a.size + threads - 1) // threads
add_kernel[blocks, threads](a, b, c)   # kernel launch
```

Q1.2: compare Python loop vs NumPy vs this kernel at N = 1e6..1e8, including H↔D transfer.

Colab GPU setup & checking the device

```
# Runtime → Change runtime type → T4 GPU (16 GB VRAM, free)
!pip install -q numba  # usually pre-installed
!nvidia-smi   # confirm GPU + driver
from numba import cuda
print(cuda.gpus)                # list devices
print(cuda.get_current_device())
dev = cuda.get_current_device()
print(dev.name, dev.compute_capability)
```

If VRAM is full: restart the session (Runtime → Restart).

Lab 1: FizzBuzz with @cuda.jit

```
# fizzbuzz rules: %3 → Fizz, %5 → Buzz, %15 → FizzBuzz
@cuda.jit
def fizzbuzz_kernel(start, stop, result):
    i = cuda.grid(1) + start
    if i > stop:
        return
    if i % 15 == 0:   result[i-start] = 3   # FizzBuzz
    elif i % 5 == 0:  result[i-start] = 2   # Buzz
    elif i % 3 == 0:  result[i-start] = 1   # Fizz
    else:             result[i-start] = 0
```

Map threads to array elements with an offset — no loop over the range at all.

Lab 2: Gaussian blur (2-D grid)

```
# threadIdx + blockIdx give a 2-D (x, y) → one thread per output pixel
@cuda.jit
def blur_kernel(img, out):
    x, y = cuda.grid(2)
    if x >= img.shape[0] or y >= img.shape[1]:
        return
    acc, cnt = 0.0, 0
    for dx in range(-1, 2):
        for dy in range(-1, 2):
            nx, ny = x+dx, y+dy
            if 0 <= nx < img.shape[0] and 0 <= ny < img.shape[1]:
                acc += img[nx, ny]; cnt += 1
    out[x, y] = acc / cnt
```

Screenshot kernel + output image. Explain the 2-D thread mapping and why it is faster than CPU loops.

HW4 grading (100 pts)

Part	Content	Points
1	GPU programming (concepts, vector add, 2 labs, bonus)	35
2	LLM fundamentals (attention by hand, concepts)	15
3	Deploying & using LLMs (local, quantization, serving, prompts)	30
4	Mini-Project (dataset, EDA, analysis, launchpad proposal)	20
—	Total (+5 bonus, capped at 100)	100

Submission checklist

One PDF: UNI_HW4.pdf — code screenshots + explanations + results

Part 1 GPU programming: answers, plots, lab screenshots

Part 2 LLM fundamentals: hand-computed attention matrices, definitions

Part 3 Deploying and Using LLMs: generation outputs, quantization table, server logs, cost table

Part 4 Mini-Project: dataset card, cleaning log, EDA plots, analysis + speedup, LLM summary, 3 findings

Due: Thursday, Nov 5, 2026

Thank you!

Questions? Bring them to the Office Hours.