CPSC 4240/5240: Parallel Programming Techniques

Lecture 22: Homework Discussion and Parallelism for LLMs

Lecturer: Quanquan C. Liu

Spring 2026

Final Project Presentations

- Final project presentations schedule has been posted
- See the course announcement for the schedule
- 5 minutes to present:
  - **The Problem:** Briefly introduce the specific problem you are studying.
  - **The Methods:** Explain the parallel programming methods and techniques you plan to use to study it.
  - **The Progress:** Summarize what you have accomplished so far.

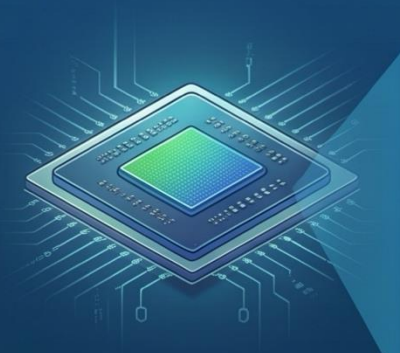
What is NVIDIA CUB?

- An open-source C++ template library for CUDA programming
- A collection of highly optimized, reusable **parallel primitives**
- **Hierarchy of Primitives:**
  - **Device-wide**: Algorithms that process data across the entire GPU
  - **Block-wide**: Primitives for threads within a single Thread Block
  - **Warp-wide**: Primitives for a single Warp of 32 threads
- **Common operations**: Reduction, Prefix Sum (Scan), Sorting

CUB (CUDA Unbound): Simplified Hierarchy & Common Operations.

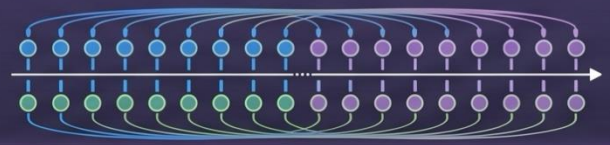
THE HIERARCHY OF PRIMITIVES.

DEVICE-WIDE PRIMITIVES



BLOCK-WIDE COLLECTIVES

WARP-WIDE COLLECTIVES



- DeviceReduce
- DeviceScan
- DeviceSort
- BlockReduce
- BlockScan
- BlockLoad
- WarpReduce
- WarpScan
- WarpShuffle

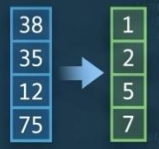
COMMON OPERATIONS.

REDUCTION

- WarpReduce
- BlockReduce
- DeviceReduce

PREFIX SUM (SCAN)

- WarpScan
- BlockScan
- DeviceScan



SORTING

- DeviceSort

HISTOGRAMS

- DeviceHistogram

Why use NVIDIA CUB?

- Writing efficient GPU primitives (like Prefix Sum/Scan) is **notoriously difficult**
- Achieving optimal memory bandwidth and block synchronization is a PhD-level task
- NVIDIA CUB (CUDA Unbound) provides state-of-the-art, highly optimized primitives

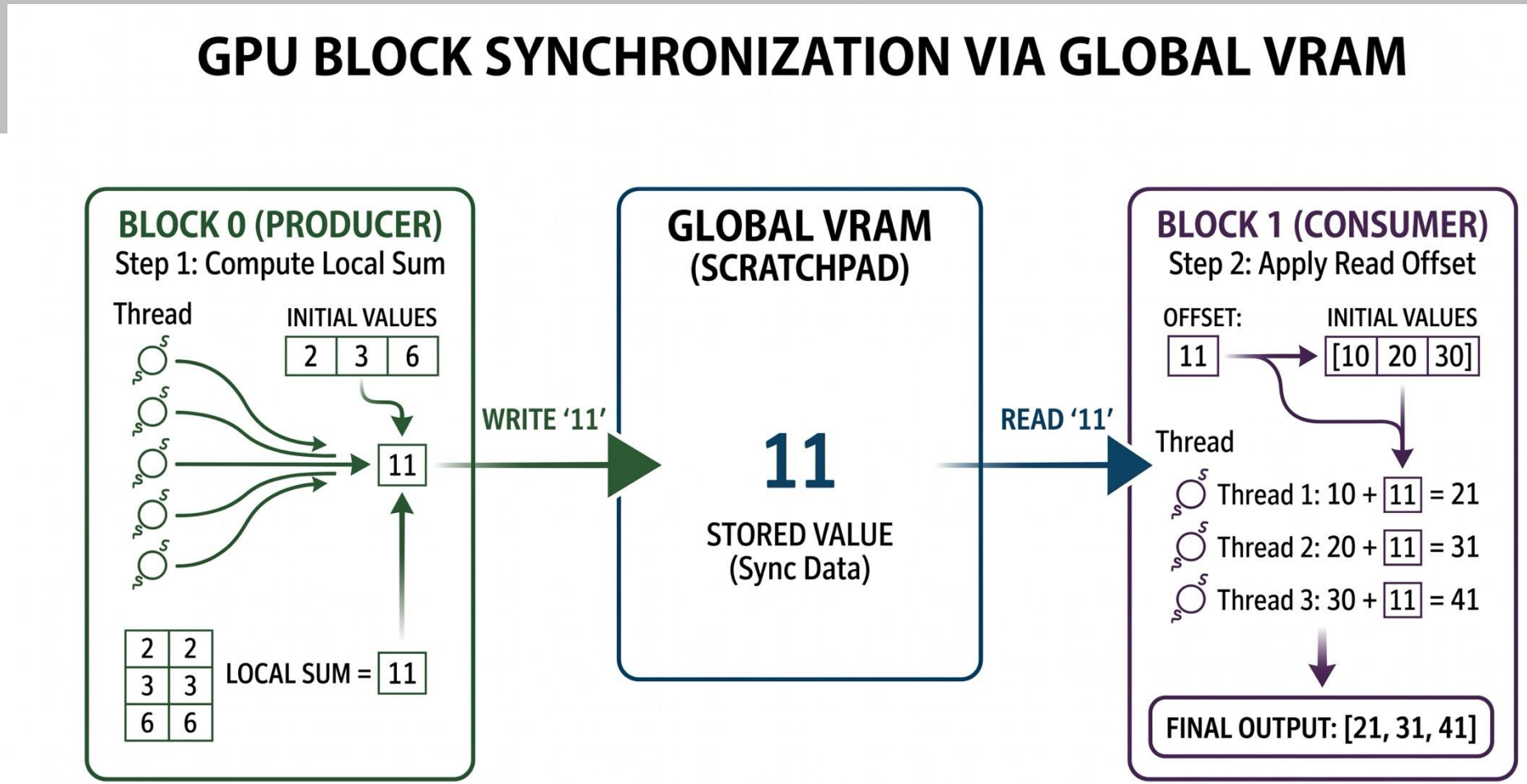
NVIDIA CUB Uses Temporary Storage

- Global GPU operations require communication across **independent Thread Blocks**
- To synchronize these blocks and store intermediate algorithm states, CUB requires **temporary Video RAM (VRAM)**
- The developer must manage this "Scratchpad" memory
- Leads us to a specific API rule you must follow when using CUB
- There is **no built-in hardware mechanism** to pause and synchronize all blocks across the entire GPU simultaneously

NVIDIA CUB Uses Temporary Storage

- Blocks must use the GPU's Global Memory (VRAM) to pass intermediate states
- **Local Computation**: Blocks compute localized results in fast Shared Memory
- **Global Sync**: Blocks write their local totals to the VRAM Scratchpad so downstream blocks can read them

NVIDIA CUB Uses Temporary Storage



Exclusive Sum

- Inclusive Sum:
  - The value at index i is the sum of all elements up to **and including** index I

Input Array: [3, 1, 7, 0, 4]

Math: [3, (3 + 1), (3 + 1 + 7), (3 + 1 + 7 + 0), (3 + 1 + 7 + 0 + 4)]

Result: [3, 4, 11, 11, 15]

- Exclusive Sum:
  - The value at index i is the sum of all elements **strictly before** index i

Exclusive Sum

- Exclusive Sum:
  - The value at index i is the sum of all elements **strictly before** index i
  - Excludes value at index i
  - The first element of an exclusive sum is always 0

Input Array: [3, 1, 7, 0, 4]
Math: [0, (3), (3 + 1), (3 + 1 + 7), (3 + 1 + 7 + 0)]
Result: [0, 3, 4, 11, 11]

Why is ExclusiveSum Important for GPUs?

- Used for **memory allocation and array compaction**
- Suppose each thread computes a variable number of results and needs to write to a single global output array
- Eg.
  - Thread 0 has **3** results to write
  - Thread 1 has **1** result
  - Thread 2 has **7** results
- If they all try to write to the output array at the same time, they will overwrite each other
- How does Thread 2 know exactly where in the output array it should start writing?

Why is ExclusiveSum Important for GPUs?

- How does Thread 2 know exactly where in the output array it should start writing?
- Run an ExclusiveSum on the counts!
- The result array [0, 3, 4, ...] immediately gives every thread the starting index in the global array:
  - Thread 0 starts writing at index **0**
  - Thread 1 starts writing at index **3**
  - Thread 2 starts writing at index **4**

Managing the CUB Memory API: Two-Pass API

- Manually Manage CUB Memory Using Two-Pass API:
  - First pass: find out how much memory CUB needs
    - Pass **nullptr** into the function the function skip the math and return number of bytes to execute the function in **temp_bytes**

```
size_t temp_bytes = 0;

// Pass 1: Pass nullptr for storage. CUB only calculates the required size.
cub::DeviceScan::ExclusiveSum(nullptr, temp_bytes, d_in, d_out, num_d_in);
```

Managing the CUB: Two-Pass API

- Pass two:
  - Now that we know how many bytes are required, we move to the second pass
  - Use standard cudaMalloc to allocate a buffer of that exact size
  - Call ExclusiveSum again but pass in newly allocated d_temp_storage pointer
  - Fully executes the scan given the pointer to the full storage
  - Need to cudaFree when done

```
size_t temp_bytes = 0;

// Pass 1: Pass nullptr for storage. CUB only calculates the required size.
cub::DeviceScan::ExclusiveSum(nullptr, temp_bytes, d_in, d_out, num_d_in);
```

```
void* d_temp_storage = nullptr;

// Allocate the exact amount of scratchpad memory requested
cudaMalloc(&d_temp_storage, temp_bytes);

// Pass 2: Pass the allocated storage. CUB executes the math.
cub::DeviceScan::ExclusiveSum(d_temp_storage, temp_bytes, d_in, d_out, num_d_in);

// Free up the temporary storage!
// cudaFree(d_temp_storage);
```

Atomics in CUDA

- Two frontier vertices **may share a neighbor**
- Two threads **may try to decrement the same degree** at the same time
- Overlapping neighbor deletions cause race conditions
- Atomics enforce hardware serialization for localized contention
- Without atomics, updates can be lost
- **atomicSub** makes concurrent decrements safe

Parallelization in LLMs

From Graphs to LLMs

- Graphs:
  - Irregular memory access
  - Dynamic frontiers
  - Sparse updates
- LLMs:
  - Dense tensor operations
  - Attention kernels
  - KV-cache management
  - Multi-GPU communication

From Graphs to LLMs

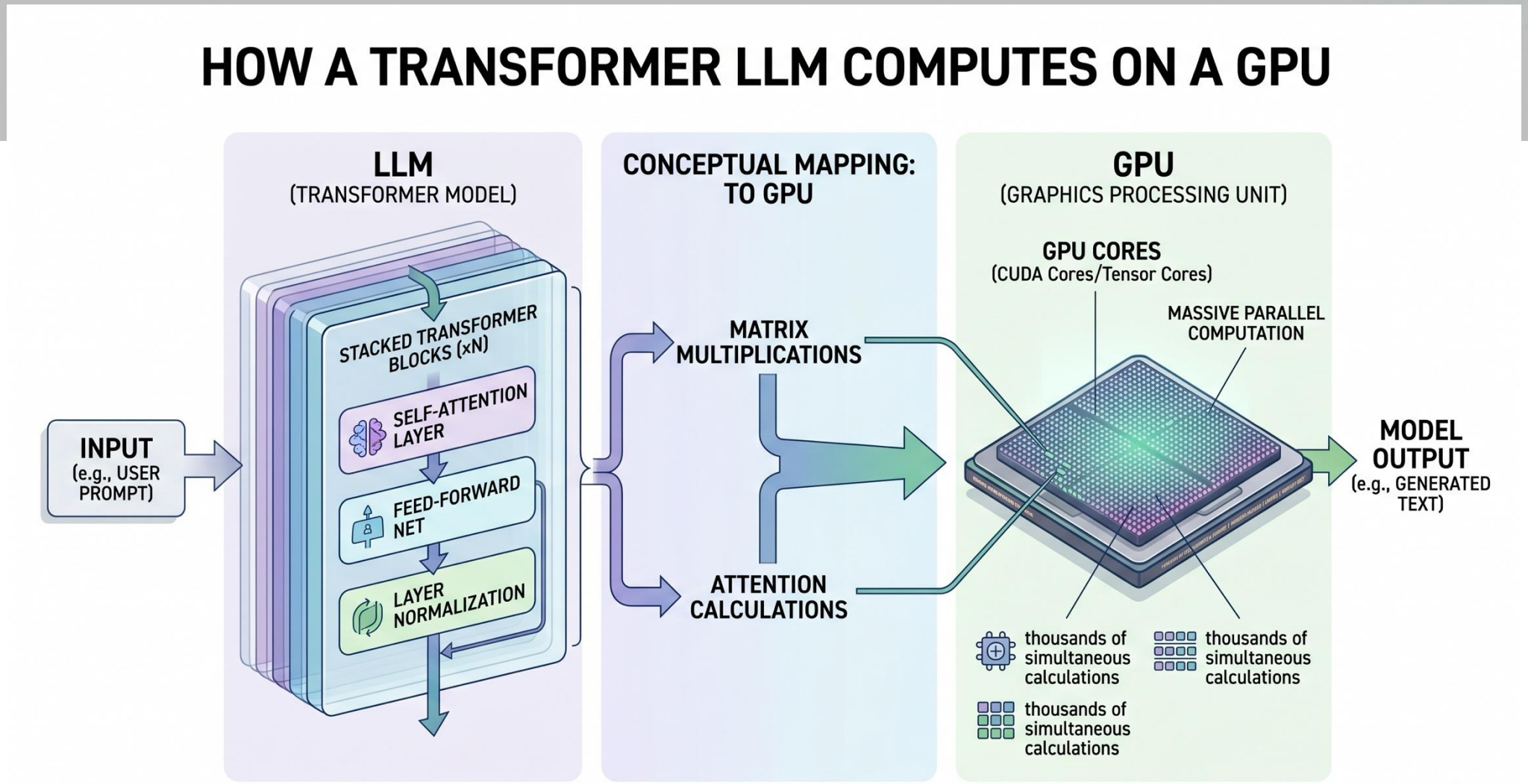
- Graphs:
  - Irregular memory access
  - Dynamic frontiers
  - Sparse updates
- LLMs:
  - Dense tensor operations
  - Attention kernels
  - KV-cache management
  - Multi-GPU communication

Common Theme: organize work for the GPU memory system

Why GPUs are such a good fit for LLMs

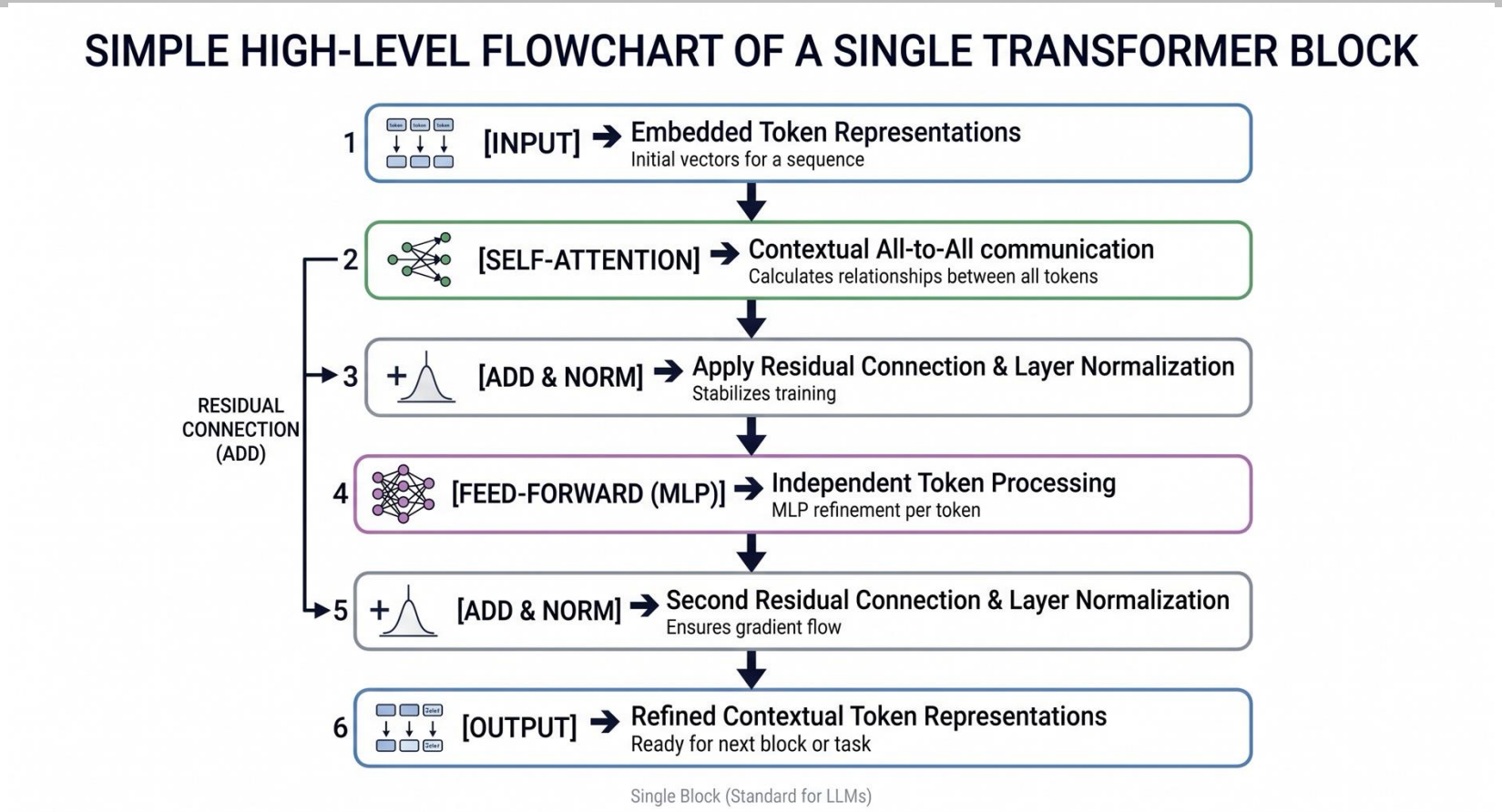
- Transformers are dominated by **large tensor operations**
- Linear layers are **matrix multiplications**
- Attention is built from **tensor products** plus softmax-style operations
- GPUs are **optimized for high-throughput tensor math**

Why GPUs are such a good fit for LLMs



The Computational View of Transformers

- Each LLM layer is dominated by two compute regions: **Attention and MLP**



The Computational View of Transformers

- Each LLM layer is dominated by two compute regions: **Attention and MLP**
- Each transformer layer is mainly:
- Attention block
  - Q, K, V projections
  - Scaled dot-product attention
- MLP / feed-forward block
  - Large matrix multiplies
  - Nonlinearities
- Normalization and residual connections

Parallelism in Attention

- The **Attention mechanism** figures out which words are important by generating raw scores
- Those scores must be converted into **percentages**; a probability distribution **using Softmax**
- To calculate a percentage, you need the **sum of the entire row**
- If you have a massive context window of 100,000 tokens, how do you sum that array efficiently on a GPU without bottlenecking?

Parallel Scan/Parallel Reduction

Two Phases of LLM Inference: Prefill and Decode

- Generating text is not a single, uniform computational process
- It is split into **two distinct phases** with entirely different hardware demands
- A two-stage pipeline.
  - **Stage 1 [Prefill]:** Input text ("Explain quantum physics")
    Stage 2 [Decode]: Feedback loop generating one word at a time ("Quantum", then "is", then "the"...)

Prefill and Decode

- **Prefill: Processing the Prompt**
  - Process the prompt
  - Many tokens available at once
  - **Lots of parallel work: Compute-Bound**
- **Decode: Generating the Output**
  - Generate new tokens step by step
  - Often one token per sequence at a time
  - Less arithmetic intensity
  - **Memory and scheduling become more important: Memory-Bound**

Why LLMs need multi-GPU parallelism

- **Why one GPU is often not enough**
  - Model parameters may not fit in one GPU's memory
  - Activations, optimizer state, and KV cache also consume memory
  - One GPU may not provide enough throughput
  - Large LLM systems need both:
    - **Memory scaling**
    - **Compute scaling**

Why LLMs need multi-GPU parallelism

- **Why multiple parallelism strategies exist**
  - Different bottlenecks appear in different settings
  - Training and inference have different bottlenecks
  - No single strategy solves everything
  - Real systems often combine several forms of parallelism

Data Parallelism

- **Data parallelism**
  - Replicate the model on multiple GPUs
  - Split the batch across GPUs
  - Each GPU computes on different data
  - Synchronize gradients or updates across devices
- **Strengths**
  - Simple mental model
  - Scales throughput when the model already fits
- **Limits**
  - Does not reduce per-GPU model memory footprint
  - Communication cost grows with training scale

Sharded Data Parallelism

- **Sharded data parallelism**
  - Do not fully replicate all parameters on every GPU all the time
  - Shard model states across devices
  - Gather what is needed for computation
  - Reshard to save memory
- **Why it matters**
  - Reduces memory overhead
  - Makes larger models trainable
  - Trades memory savings for extra communication complexity

Tensor Parallelism

- **Tensor parallelism**
  - Partition large weight tensors across GPUs
  - Each GPU computes part of the same layer
  - Combine partial results to produce the full output
- **Typical use**
  - Very large linear layers
  - Attention projections (massive matrix multiplication)
  - Models too large for one GPU even within a single layer
- **Tradeoff**
  - Reduces per-GPU memory
  - Increases communication inside each layer

Pipeline Parallelism: Split the Model by Depth

- **Pipeline parallelism**
  - Divide the model into layer groups
  - Place different groups on different GPUs
  - Send microbatches through the stages like an assembly line
- **Why use it**
  - Helps when the full depth of the model will not fit on one GPU
  - Allows different devices to work on different stages simultaneously
- **Tradeoff**
  - Can introduce pipeline bubbles
  - Scheduling becomes more complex

Real Systems Combine Strategies

- **Real LLM systems are hybrid**
  - Data parallel + tensor parallel
  - Tensor parallel + pipeline parallel
  - Sharded data parallel + tensor parallel
  - Context parallel + expert parallel + pipeline parallel
- **Main idea**
  - Match the parallelism strategy to the bottleneck
  - Memory bottleneck ≠ communication bottleneck ≠ throughput bottleneck
  - Large systems often need several strategies at once

What is NVIDIA CUB?