

GPU systems and (LL) Models

Petr Lapukhov, NVIDIA

Disclaimer:

- Opinions are presenter's and not of the employer.
- Information from open sources.

Goals of this talk

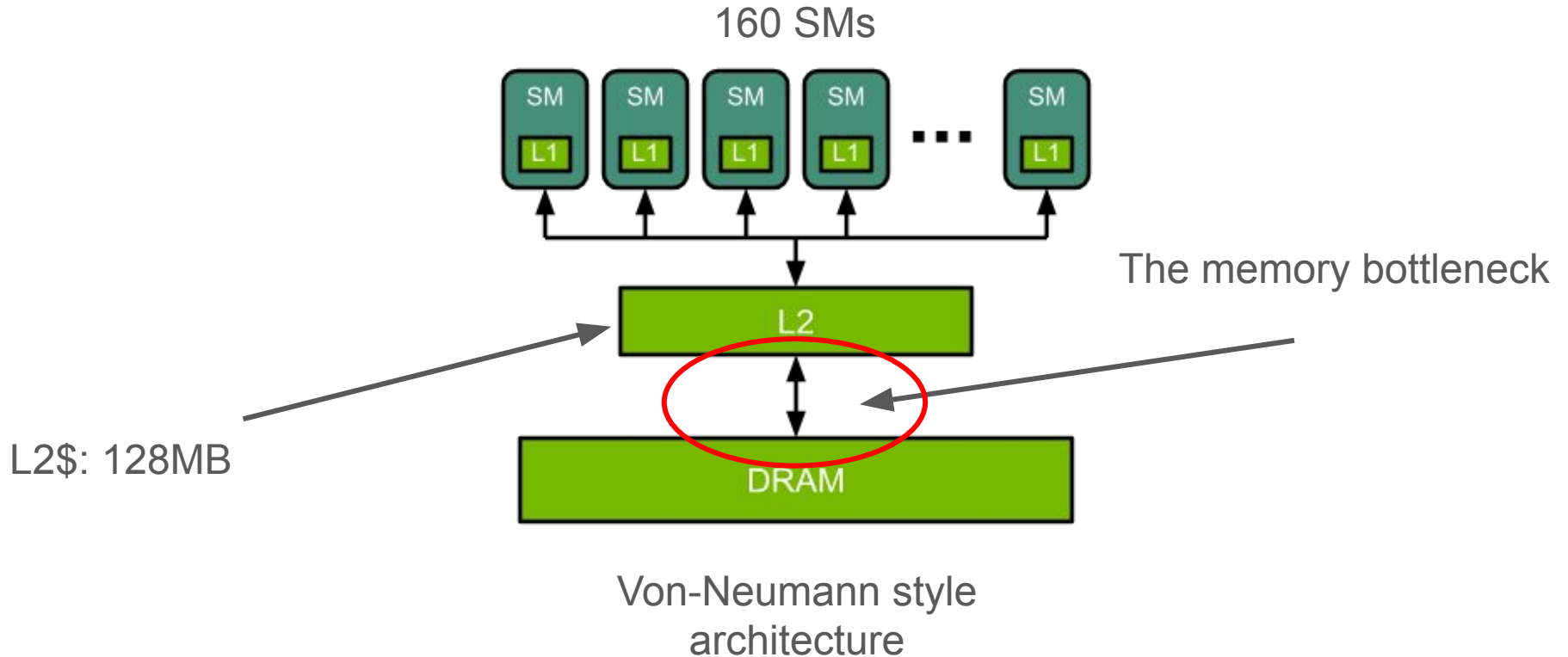
- A quick look at a system - NVL72 GB200
- Roofline “glance” on GPU performance
- What is LLM inference?
- Throughput vs. latency trade-off in inference
- Connecting “NVL domain” dimensions (size, bandwidth) with inference perf

NVL72 GB200



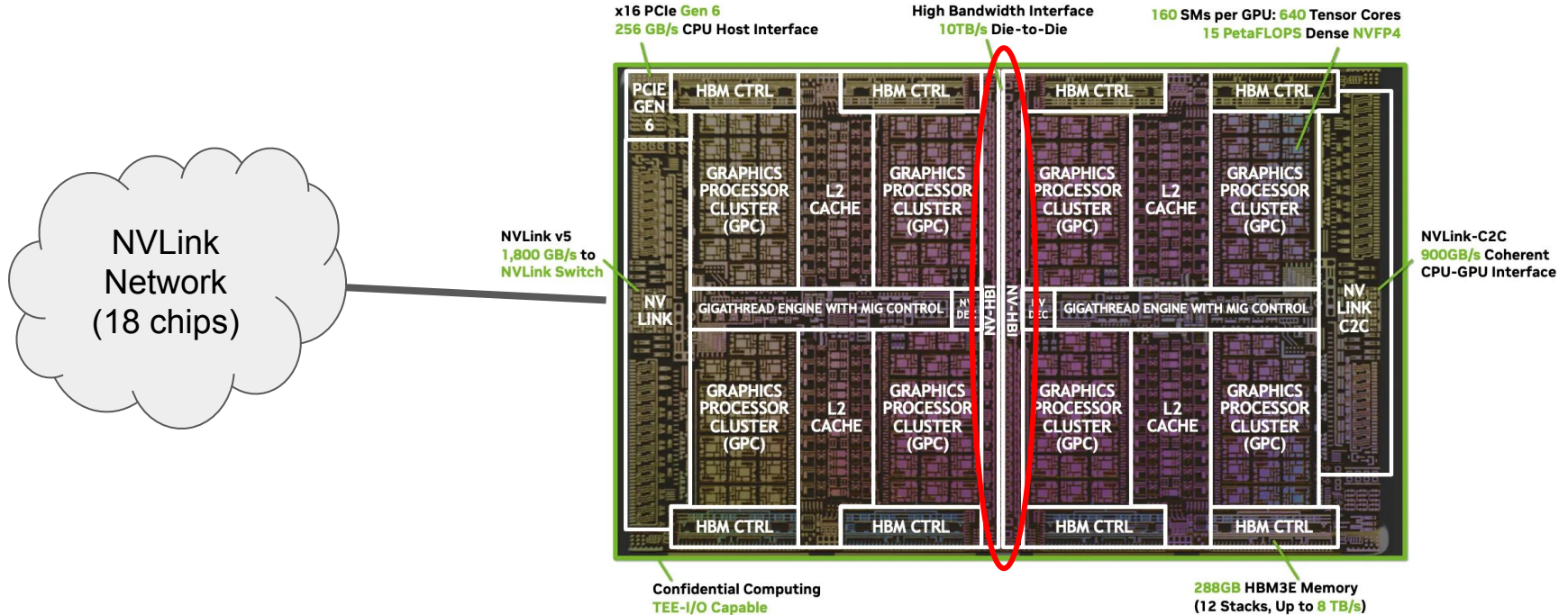
- 18 compute nodes
- 72 GPUs
- 18 switch chips
- 18x 400Gbps links per GPU
- ~2600 cables (200G lanes)

GPU primer (a very simple one)



Blackwell SM/package: 160 SMs, 128 cores/SM

NVIDIA Blackwell Ultra GPU



Blackwell GPU primer

FP4 FLOPs: **1e16**

MemBW byte/s: **8e12**

MemCap bytes: **~2e11**

NetBW byte/s: **9e11**

NVL domain: 72 GPUs

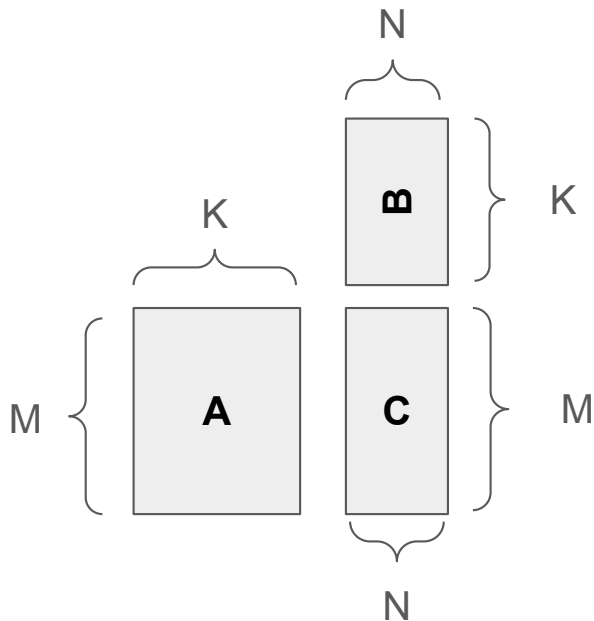
NVFP4 = 16x (E3M1) + (E4M3) (scaling)

Feature	Hopper	Blackwell
Manufacturing process	TSMC 4N	TSMC 4NP
Transistors	80B	208B
Dies per GPU	1	2
NVFP4 dense sparse performance	–	10 20 PetaFLOPS
FP8 dense sparse performance	2 4 PetaFLOPS	5 10 PetaFLOPS
Attention acceleration (SFU EX2)	4.5 TeraExponentials/s	5 TeraExponentials/s
Max HBM capacity	80 GB HBM (H100) 141 GB HBM3E (H200)	192 GB HBM3E
Max HBM bandwidth	3.35 TB/s (H100) 4.8 TB/s (H200)	8 TB/s
NVLink bandwidth	900 GB/s	1,800 GB/s 2-way
Max power (TGP)	Up to 700W	Up to 1,200W

Table 2. NVIDIA GPU chip comparison

Mat-mul: (C = A @ B) in FP4

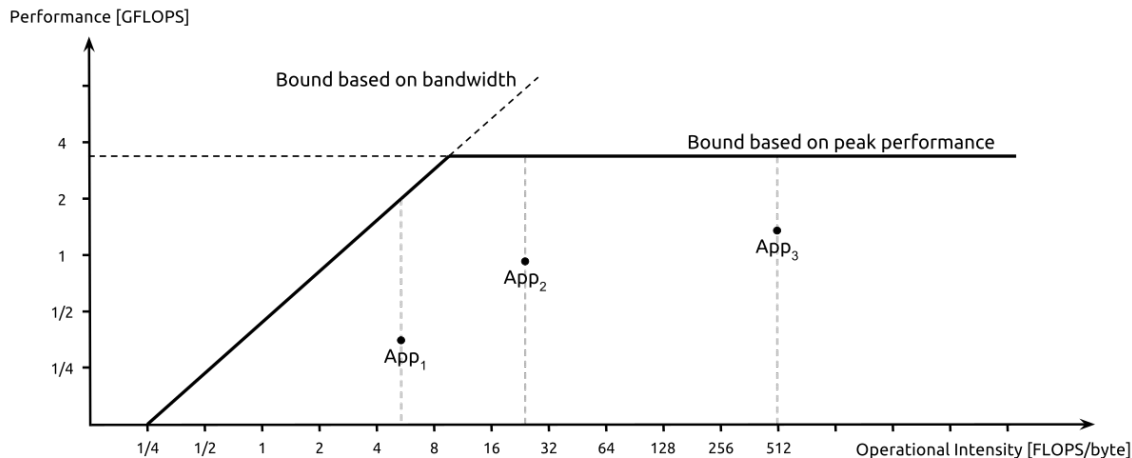
$$\text{Arithmetic Intensity} = \frac{\text{number of FLOPS}}{\text{number of byte accesses}} = \frac{2 \cdot (M \cdot N \cdot K)}{0.5 (M \cdot K + N \cdot K + M \cdot N)} = \frac{4 M \cdot N \cdot K}{M \cdot K + N \cdot K + M \cdot N}$$



Good case: [N×N] by [N×N] matrix
gives you arithmetic intensity of **4/3*N**

Tricky case: [N×N] by [N×1] (GEMV) -
intensity of **4N² / (2N + N²)** or **near constant**

Roofline performance model



To be **compute** bound:

$$\# \text{ ops} / \# \text{ bytes} > \text{BW}_{\text{math}} / \text{BW}_{\text{mem}}$$

Blackwell

Math/Mem : **1e16 / 8e12 = 1250**

Mem/Net: **8e12 / 9e11 ≈ 9**

NxNxN matmul:

N > 940 to be math limited on
blackwell in FP4

GEMV is problematic

Why do those ratios matter?

- $T_{\text{work}} = T_{\text{compute}} + T_{\text{mem}} + T_{\text{comms}}$
- Ideally we overlap compute with comms/memory
- Thus:

$$T_{\text{work}} = \max(T_{\text{compute}}, T_{\text{mem}}, T_{\text{comms}})$$

- Our goal is for T_{compute} to dominate - maximize system throughput
- **Memory bandwidth** is the bottleneck
- As we could see, it's fine for GEMM but bad for GEMV

Large language model (LLM) inference

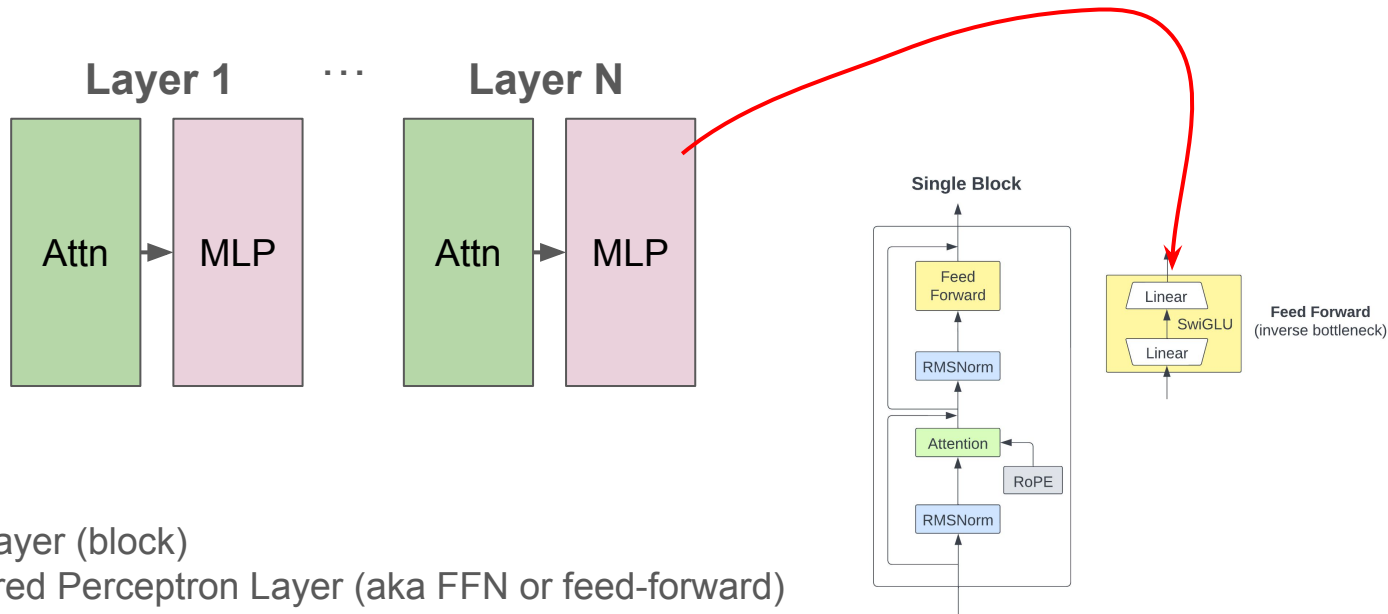
[1] Given: a deep **neural network** - think 70B - 671B “weights” - **lots of matrices in layers**

[2] **Inference:** supplying text **input** to the model and obtaining its “continuation” - output or generated text

Inference is a sequence of large or large mat-muls performed on “batch” of inputs sequences

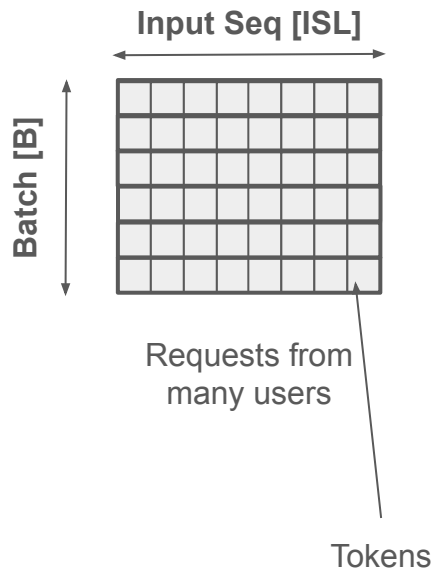
LLM token generation: the model (80 layers in LLama3-70B)

Model **weights** (matrices) are hosted on “one or more GPUs”



LLM token generation: input token batch

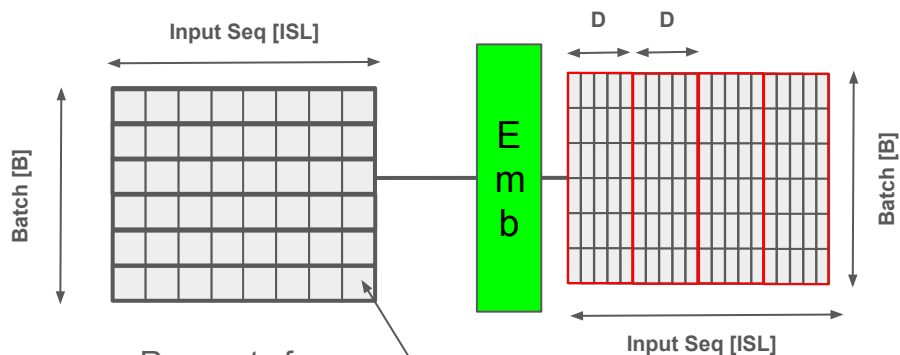
Input batch of tokens



- A “request” is a **sequence** of **tokens** - e.g., your question + history of chat
- Multiple **requests** get combined (on the fly) in a **batch of sequences of same length** (padded if needed)

LLM token generation: **tokens to vectors**

Input batch of token sequences → batch of sequences of vectors



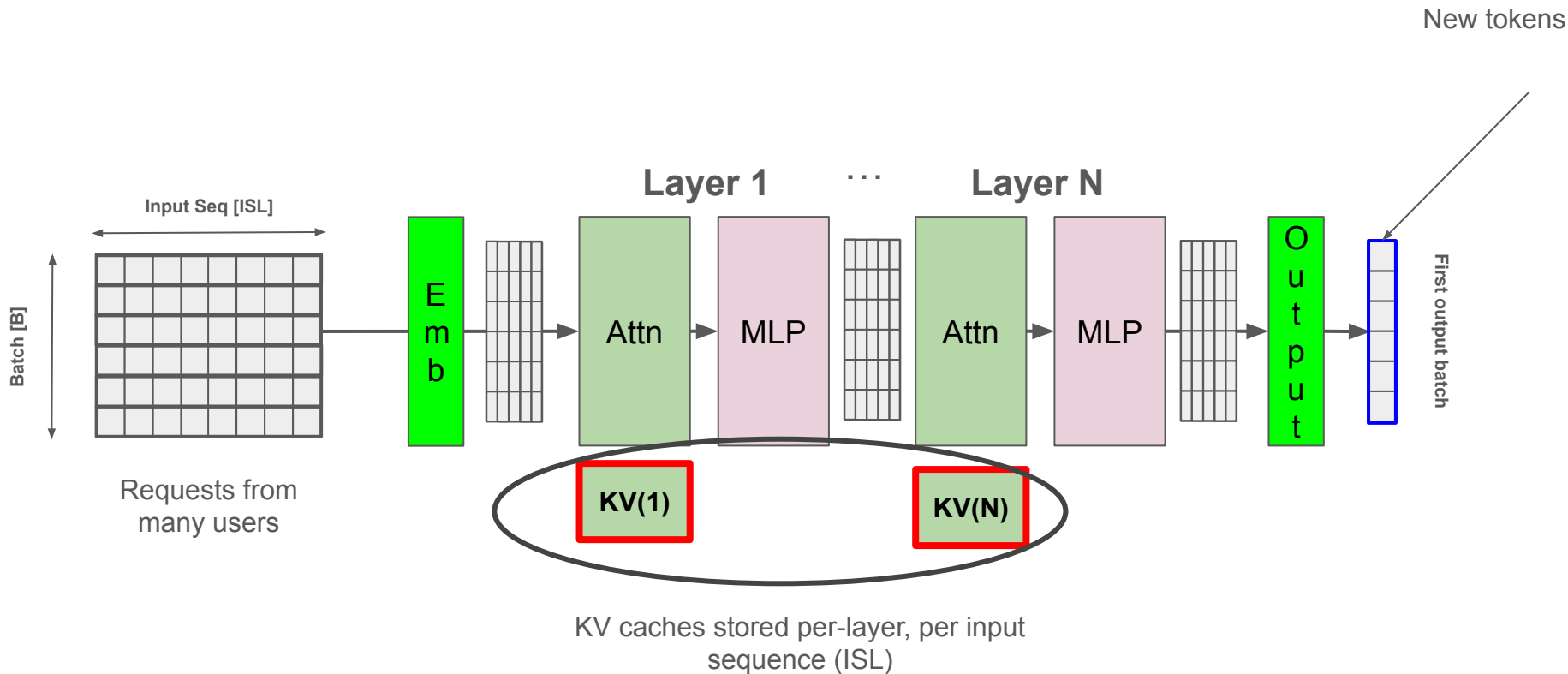
Requests from
many users

Tokens

Activations (batch of vectors)
 $B \times ISL \times D$ sized

- Embedding layer (vocabulary) maps tokens to “embeddings” - vectors of dimensions D (e.g., $D=8192$).
- Now we have a matrix of $B * ISL * D$ elements

LLM token generation: **prefill** (context) phase - KV gen



KV (key-value) cache generation (**prefill** phase)

[1] **Attention** layers models how tokens **interact inside input sequence**

[2] When calculating attention, every token “projects” in three vectors: **Key**, **Query**, **Value** using the weight matrices found in the **Attn** layer

[3] K & V vectors **remain the same** during computation, and can be cached [for each inputs sequence → **saves a ton of compute time**

KV cache is **materialized** when we process the input batch and remains during full pass

How big are the numbers? LLAMA3-70B example

hyperparam	value
n_{layers} (L)	80
d_{model} (D)	8,192
d_{ff} (F)	28,672
n_{heads} (N)	64
$n_{kv \text{ heads}}$ (K)	8
d_{kv} (H)	128
$n_{\text{embeddings}}$ (V)	128,256

Attn Size = $2DNH + 2DKH = 151 \text{ MB}$ / Layer

MLP Size = $3DF = 704 \text{ MB}$ / Layer (82%)

Emb = $2VD = 2.1 \text{ GB}$ / Model

KV/token =

= 2 KB / Layer = **160K FP8** / token

With large batches KV cache size dominates memory reqs

GPU memory capacity limits

Using LLAMA3 70B as a simple example:

[1] 70G weights (if using FP8)

[2] 160K per token for KV cache (using FP8)

[3] B200 HBM3e memory is ~192GB

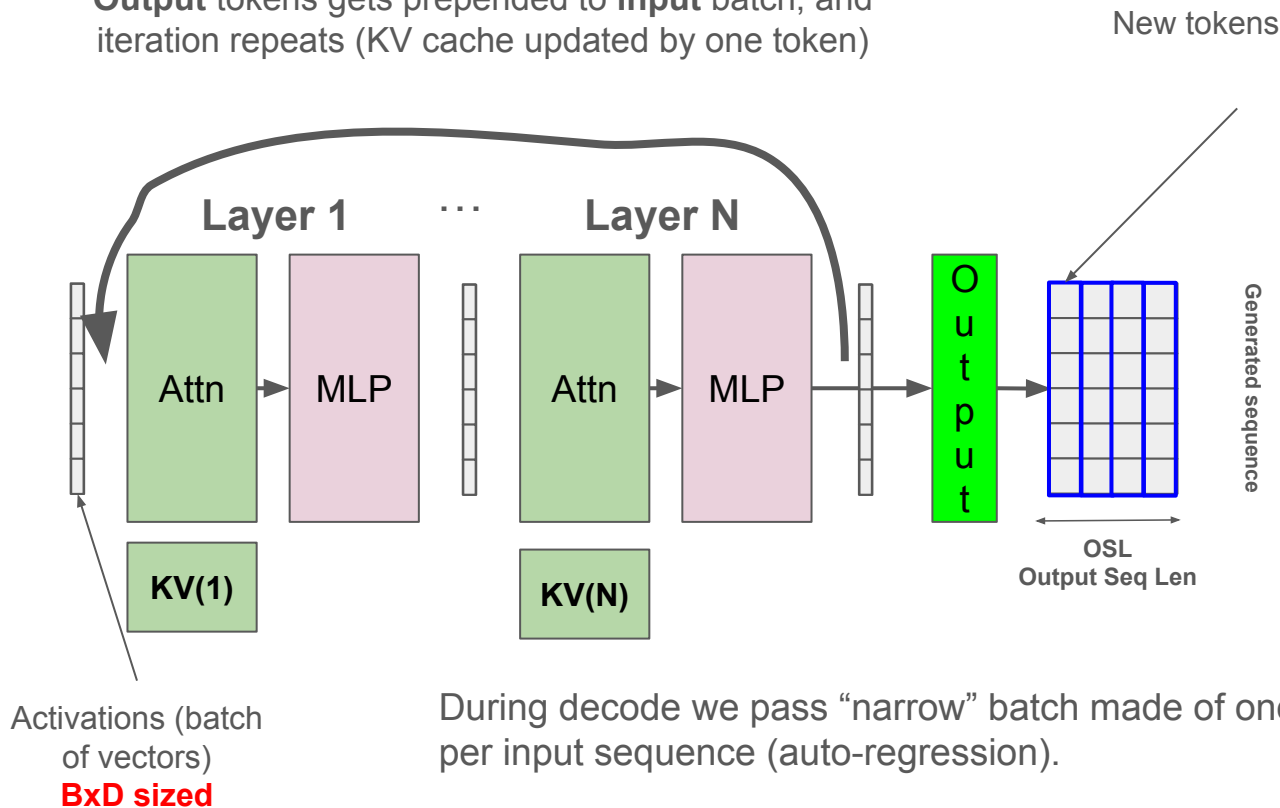
—

Single GPU can fit 750K tokens + model weight, which seems good, huh?

You would need **more GPUs** to fit larger models (405B) or DeepSeek v3 (671B)

LLM token generation: **decode** phase (after prefill)

Output tokens gets prepended to **input** batch, and iteration repeats (KV cache updated by one token)



Compute/memory per phase

B - batch size

S - sequence length

N - model size (MLP mostly)

	Prefill (full sequences)		Decode (1-token sequences)	
	Mem Bytes	FLOPs	Mem Bytes	FLOPs
Attn	$\theta(\mathbf{BS})$ (KV cache)	$\theta(\mathbf{BS}^2)$ (full seq)	$\theta(\mathbf{BS})$ (KV cache)	$\theta(\mathbf{BS})$ (single token)
MLP	$\theta(\mathbf{N})$	$\theta(\mathbf{BSN}_{\text{active}})$	$\theta(\mathbf{N})$	$\theta(\mathbf{BN}_{\text{active}})$

Compute/memory per phase

B - batch size

S - sequence length

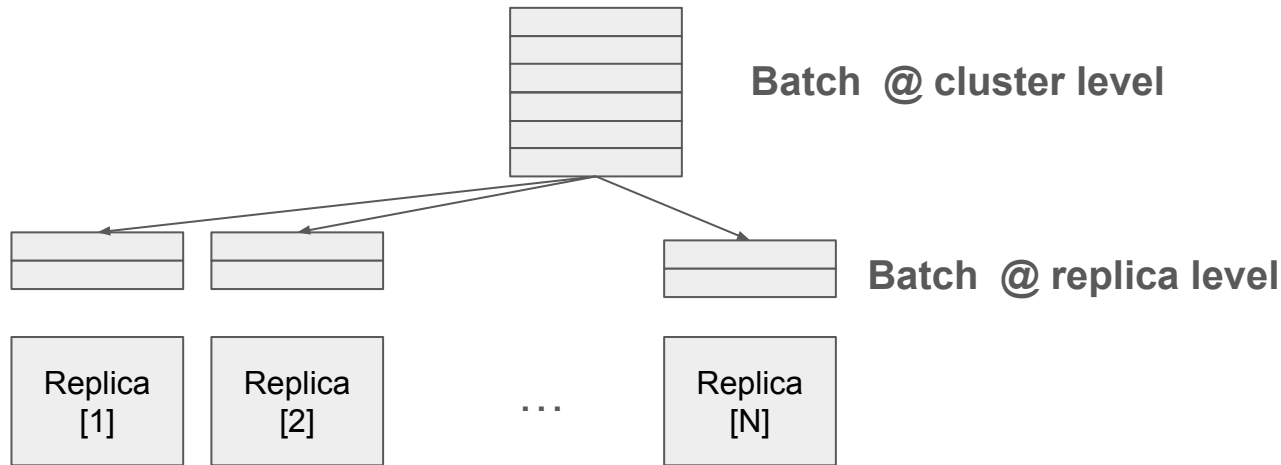
N - model size (MLP mostly)

	Prefill (full sequences)		Decode (1-token sequences)	
	Mem Bytes	FLOPs	Mem Bytes	FLOPs
Attn	$\theta(BS)$ (KV cache)	$\theta(BS^2)$ (full seq)	$\theta(BS)$ (KV cache)	$\theta(BS)$ (single token)
MLP	$\theta(N)$	$\theta(BSN_{\text{active}})$	$\theta(N)$	$\theta(BN_{\text{active}})$

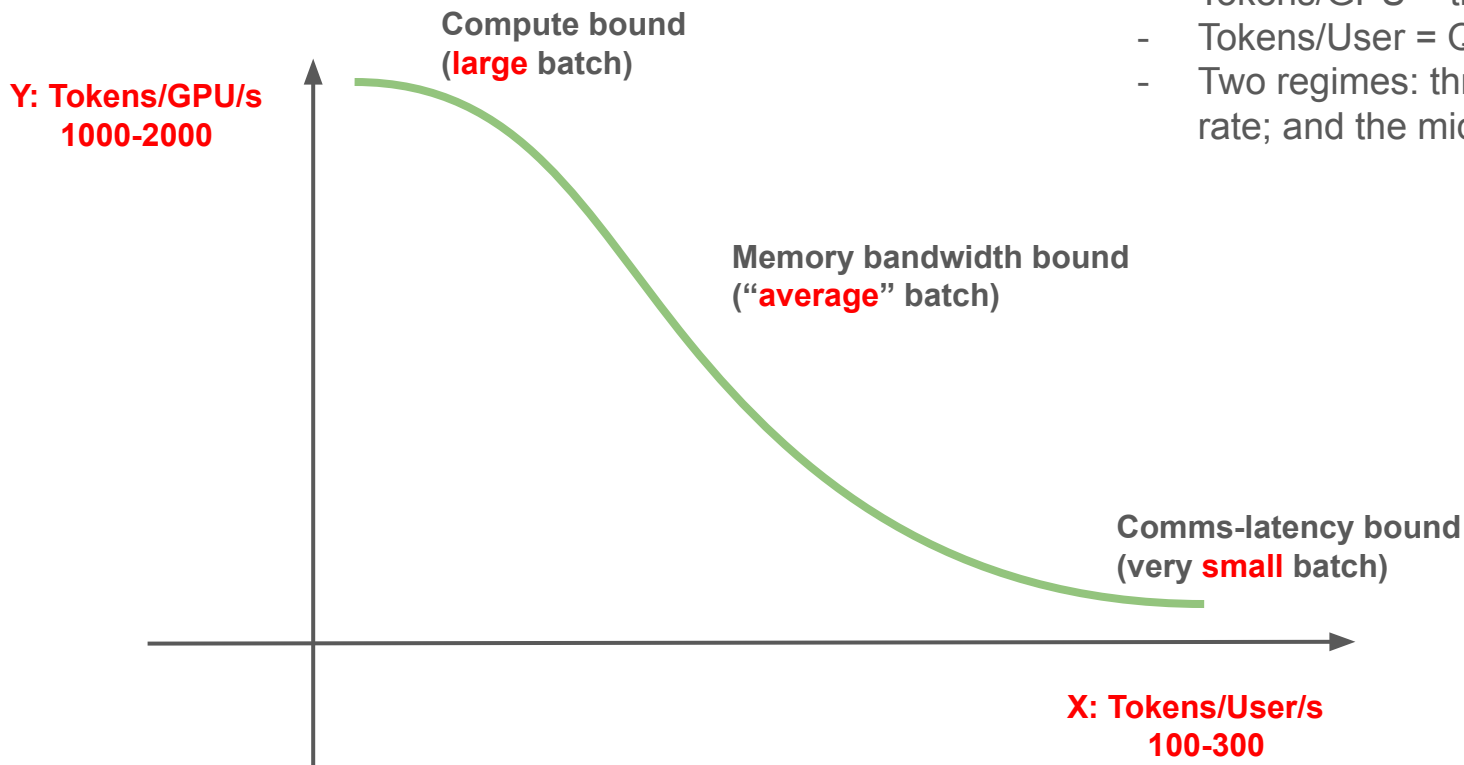
Is data-parallelism all you need?

- With **large enough batch** we're compute bound (hurray!)
- So why not **replicate the model** and keep batch large enough?
- This leads to a classical scale-out system architecture!

Problem: QoS or token-rate-per-user. Batch completes “all at once”... everyone has to wait longer



Latency-throughput curve



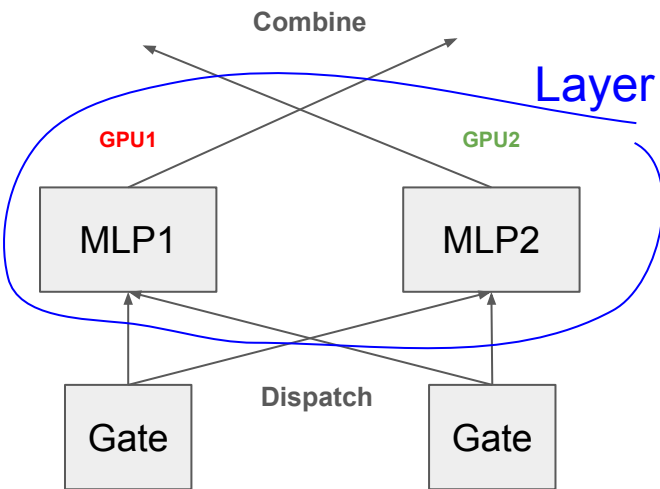
- Tokens/GPU = throughput = \$
- Tokens/User = Quality
- Two regimes: throughput vs decode rate; and the middle

High-decode-rate: Model parallelism to the rescue?

- Now the **batch is small** and so we're not compute bound anymore
- Memory time dominates
- Can we split **model weights** across more GPUs to yield higher aggregate memory bandwidth?

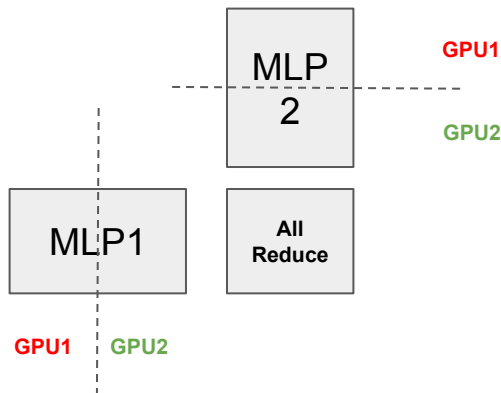
Model parallelism(s) types: splitting model across GPUs

Expert Parallel (EP)



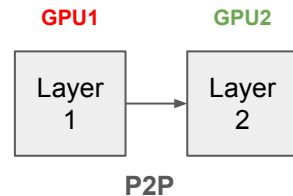
- Split MLP weights too
- Dispatch "tokens" (activations) to experts
- Multiply activations by weights locally
- Combine afterwards

Tensor Parallel (TP)



- Split **weights** of matrices
- **Per-GPU** matrix multiplication
- All-reduce to gather partial results across GPUs

Pipeline (PP)

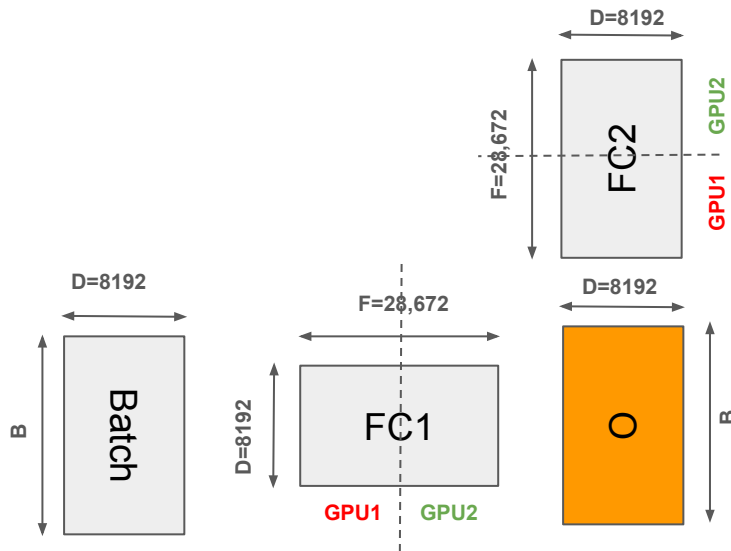


- Split layers of LLM among GPUs
- Pass activations along (whole batch)
- Improves **throughput**, but not latency

Scaling across **Y** GPUs (tensor parallel in MLP)

Single **MLP (FFN) layer** of LLAMA3

- Two big matrices (FC1 and FC2) for mul
- Input batch of activations
- Tensor-parallelism across **Y** GPUs



$$O = B @ FC1 @ FC2$$

Scaling across **Y** GPUs (tensor parallel in MLP)

Flops: **1e16**

MemBW byte/s: **8e12**

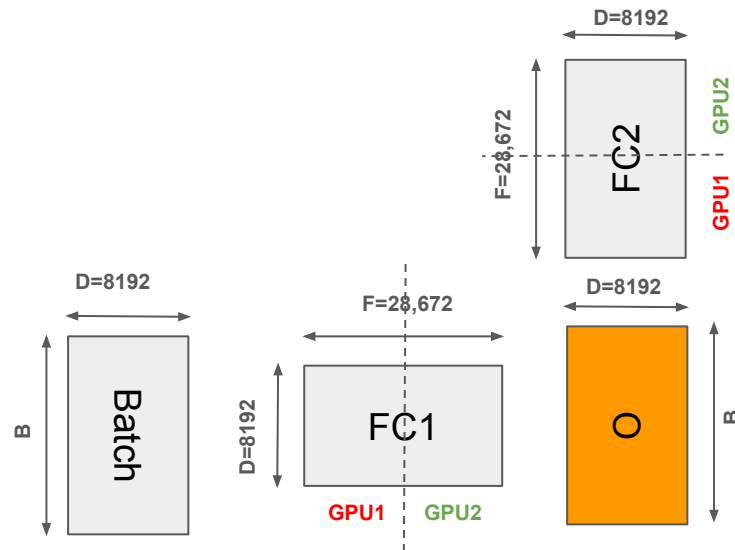
NetBW byte/s: **9e11**

For **B = 16** (per GPU):

- Compute time: $4BDF / \text{Flops} / Y = 0.37\mu\text{s} / Y$
- Memory load time: $2DF / \text{MemBw} / Y = 58\mu\text{s} / Y$
- Comms time = $2BD / \text{NetBw} = 0.14\mu\text{s}$ (in FP8)

Comms < Mem \rightarrow

$$Y < \frac{F * \text{NetBw}}{B * \text{MemBw}}$$

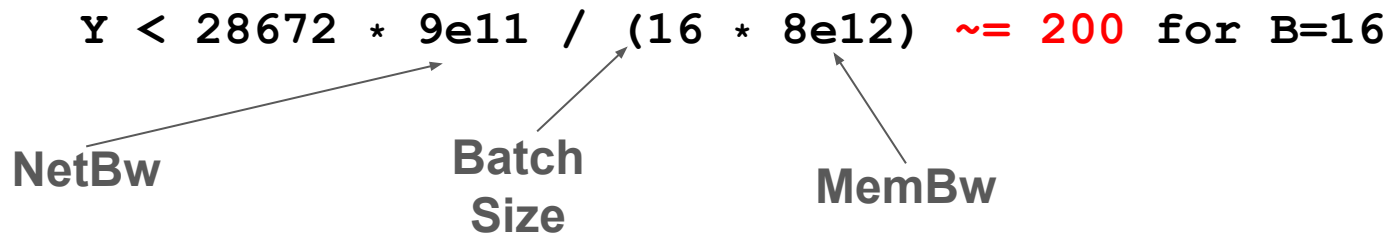


$$O = B @ \text{FC1} @ \text{FC2}$$

The practical useful NVL domain size is...

$$Y < 28672 * 9e11 / (16 * 8e12) \approx 200 \text{ for } B=16$$

NetBw Batch Size MemBw



- Y gets smaller with **lower NVL bandwidth**
- Y gets bigger with **lower compute!**
- Y gets smaller with **larger batch**

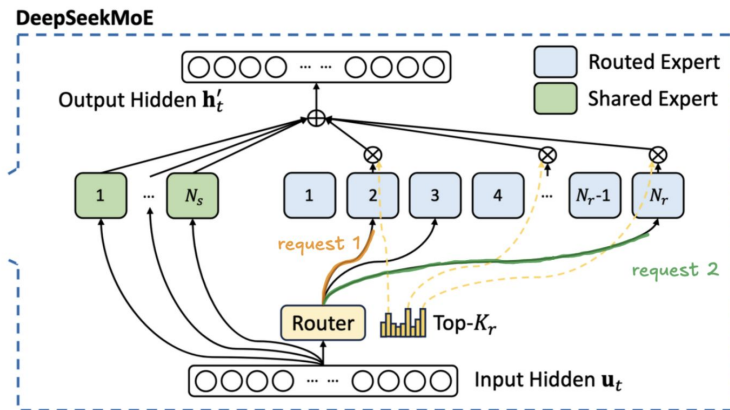
A correction for latency

- We assumed small batch ($B = 16 = 130$ KB activations in LLAMA-3).
- We ignored **latency** - but it plays a big role:
 - E.g., $130\text{K}/900\text{GB/s} = 0.14\mu\text{s}$
 - But GPU to GPU SM latency is $O(1\mu\text{s})$
- For a small batch **propagation latency** dominates - you can't have high radix without low latency!
- $B=128$ gets us into $1\mu\text{s}$ latency, resulting in NVL domain size of 25!

Isn't LLAMA3-70B a toy model now?

- Yes the glory days are over - MoE with “fine-grained” experts rule the world
- Compare: DeepSeek v3 has 37B activated params and 671B total params.
 - Compute wise it activates less per cycle than LLAMA!
- Ergo: your memory pressure gets worse with sparse models

Compute $\sim K/E*N$ ← where N is your number of MLP params, K number of activated experts and E is total number of experts



The final words

- A simple dimension such as NVL domain size has non-trivial rationale
- However this is not all there is to it
- For example, redundancy and resiliency are a bit thing: $64+8=72$ for a reason
- Aside from perf, there are major physical constraints: cable packaging and signal integrity
- Still, this strange detour into roofline models should give you an idea

Reading list

<https://jax-ml.github.io/scaling-book/>

<https://huggingface.co/spaces/nanotron/ultrascale-playbook>

<https://huggingface.co/spaces/HuggingFaceTB/smol-training-playbook>

<https://docs.nvidia.com/deeplearning/performance/index.html>

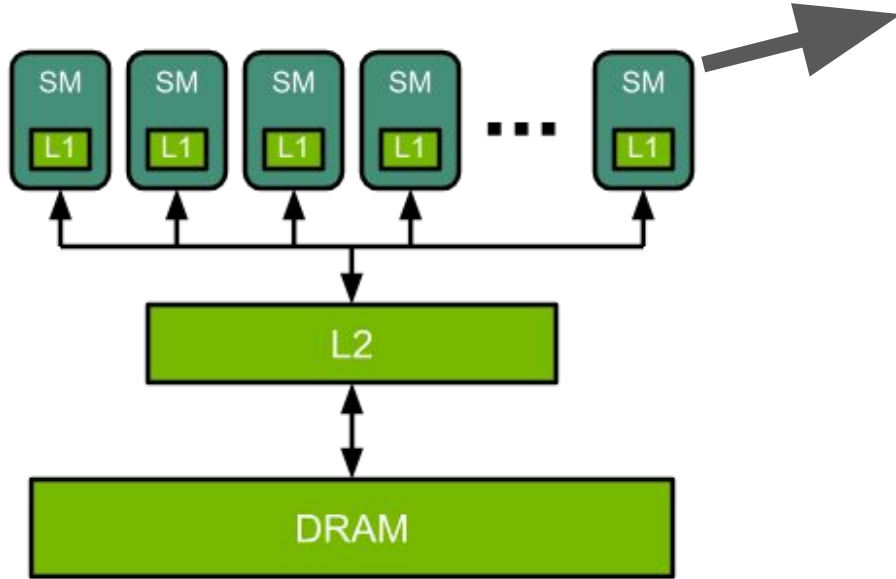
Distributed inference & network

It's desirable to map EP/TP parallelisms onto **scale-up network**

- In **model parallelism** we exchange activations
- Latency of exchange is critical for token generation rate

$$\text{Latency} \approx \text{Propagation Delay} + \text{Batch} / \text{Bandwidth}$$

GPU primer



Von-Neumann style architecture



A bit on terminology

[1] Batch, sequence, tokens ← We just covered

[2] **Activation** - a token converted into **vector** representation for matrix ops

[3] Vector size - “hidden dimension” of the model (e.g., 8192, 7168)

[4] **Embedding/Output** - First and Last layers of model; Convert tokens to activations and activations to tokens (Vocabulary-sized layers).

- **Between** the Layers we pass activation matrices (sized to batch)
- **Inside** the layers we **multiply weights by activations** and apply non-linear transformations

LLaMa3 dimensions

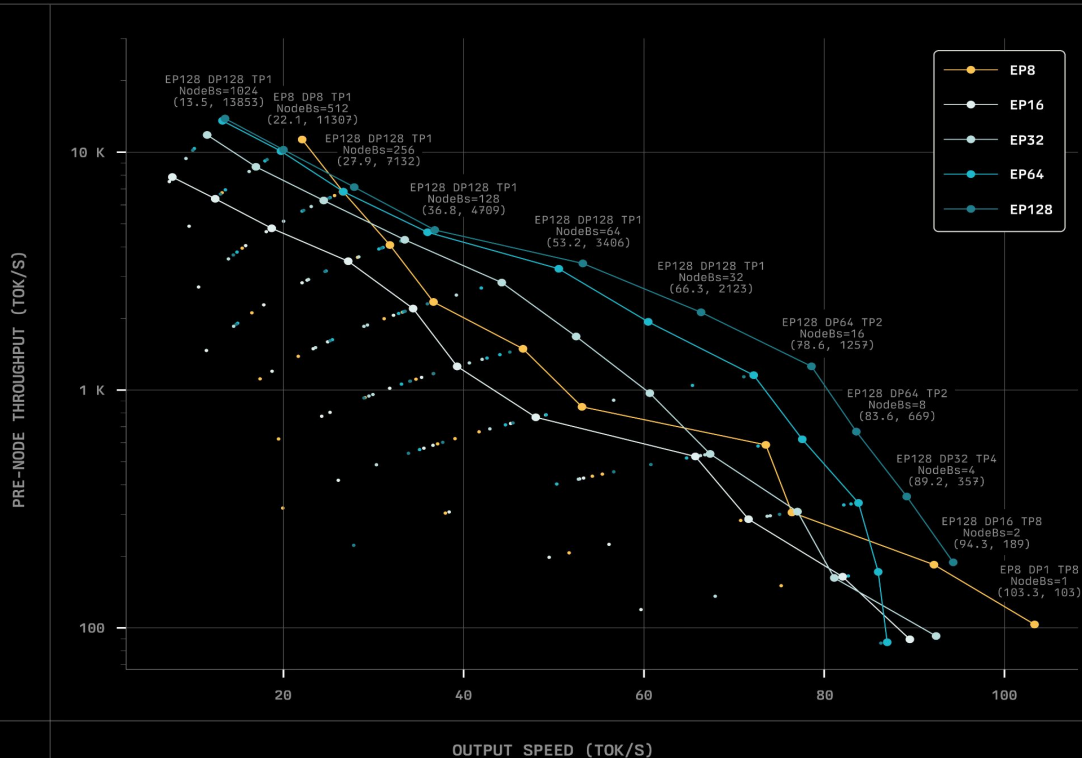
hyperparam	value
n_{layers} (L)	80
d_{model} (D)	8,192
d_{ff} (F)	28,672
n_{heads} (N)	64
$n_{\text{kv heads}}$ (K)	8
d_{qkv} (H)	128
$n_{\text{embeddings}}$ (V)	128,256

Component	Params per layer	Training FLOPs per layer
MLP	3DF	18BTDF
Attention	4DNH	24BTDNH + 12BT ² NH
Other	D	BTD
Vocab	DV (total, not per-layer)	12BTDV

Latency-throughput tradeoff

LATENCY-THROUGHPUT PARETO FRONTIER

From Perplexity Blog



- This is for Hopper-based system - up to 16 H200 machines
- EP/TP parallelism traffic maps on the scale-up or scale-out fabric
- Highest decode rate when we map on scale-up (EP8/TP8) only
- Reason: bandwidth and latency

Distributed inference & scale-up network

Scale-up networks: a bit of history

Precursors

- **CPU to CPU** interconnects: UPI, QPI, HyperTransport ...

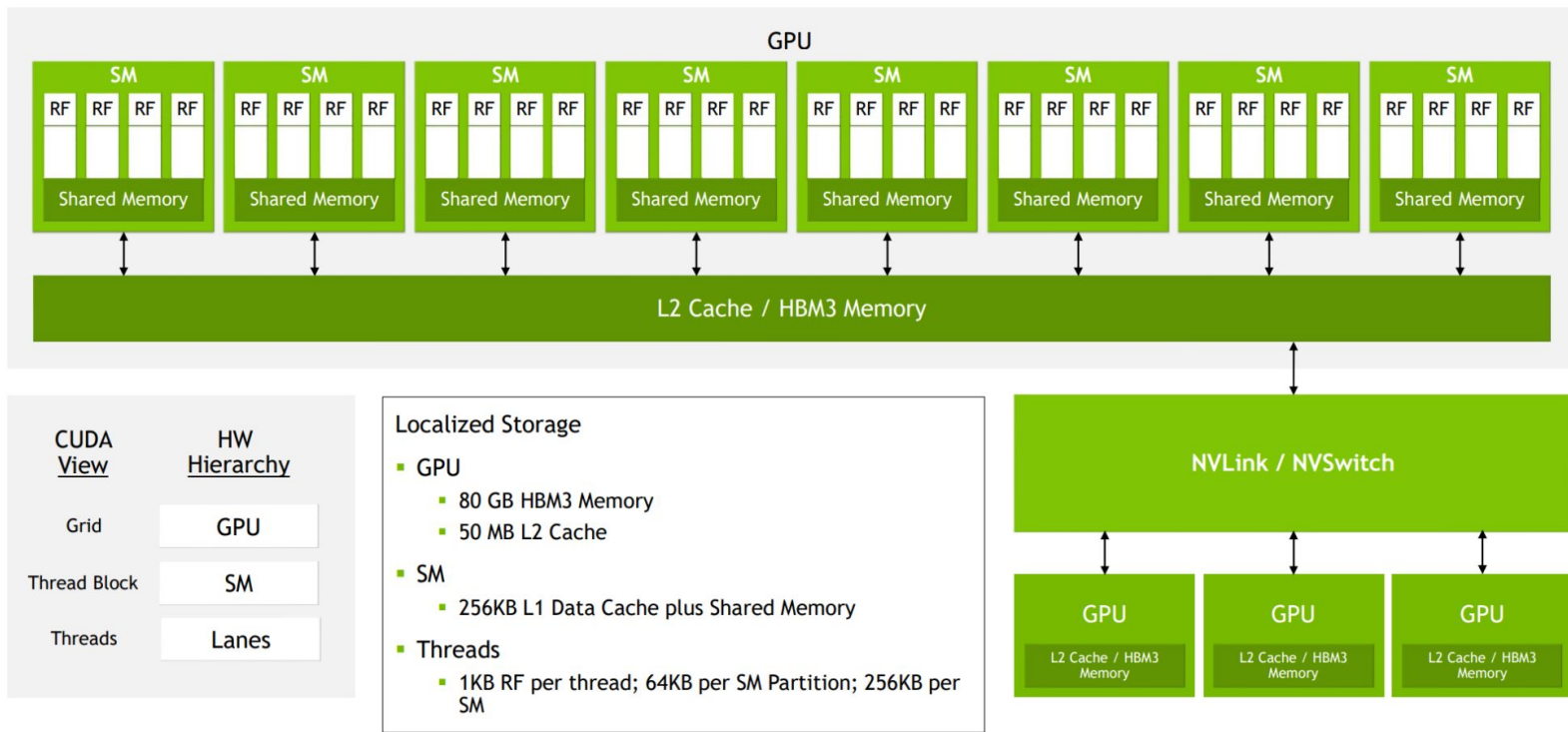
^^ Cache coherent and hence highly non-trivial to scale

- **GPU to GPU**: NVLink (first in Pascal generation)

^^ Bandwidth-oriented, no global cache coherency, local coherency only

Why not just use PCIe (NTB or CxL to scale)?

Hopper (H100) memory arch w/ NVLink



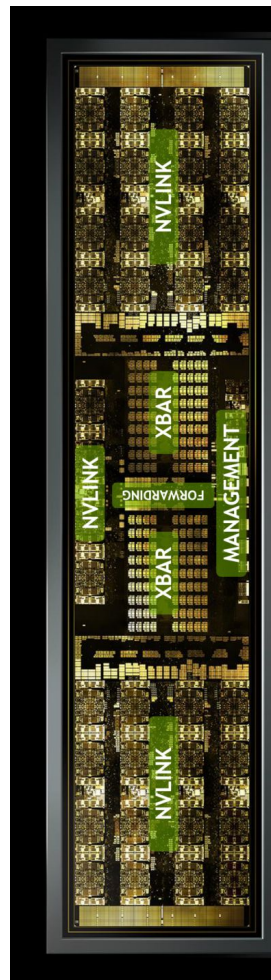
NVLink evolution

- Pascal was P2P links only
- Volta added first “NV switch” (still supported P2P)
- NV switch - routes memory transaction packets

NVlink directly extends internal GPU memory subsystem with memory addr translations

- Load/Store operations (Mem2Reg, LDG/STG) ← **sync**
- TMA (Tensor Memory Accelerator) ← **async**
- Copy Engines (think bulk data movements) ← **async**

Special “stuff:” Multicast operations and in-network reductions (Hopper and onwards)

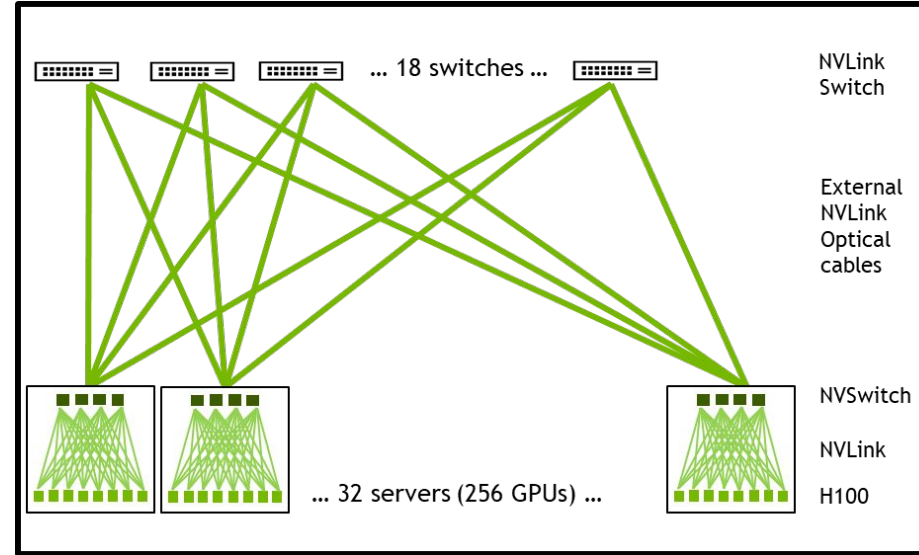
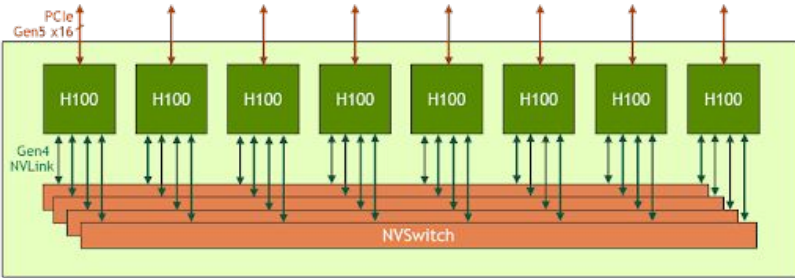


Scale-up “fabric” dimensions

	Ampere	Hopper	Blackwell
Lane speed	50Gbps	100Gbps	200Gbps
GPU NVL bandwidth	300GB/s	450GB/s	900GB/s
Fabric scale	8 (board)	8 (board)	72 (rack, 18 boards)

bandwidth is in unidirectional units

Scaling-out the scale-up?



NVLink switches can stack up if needed

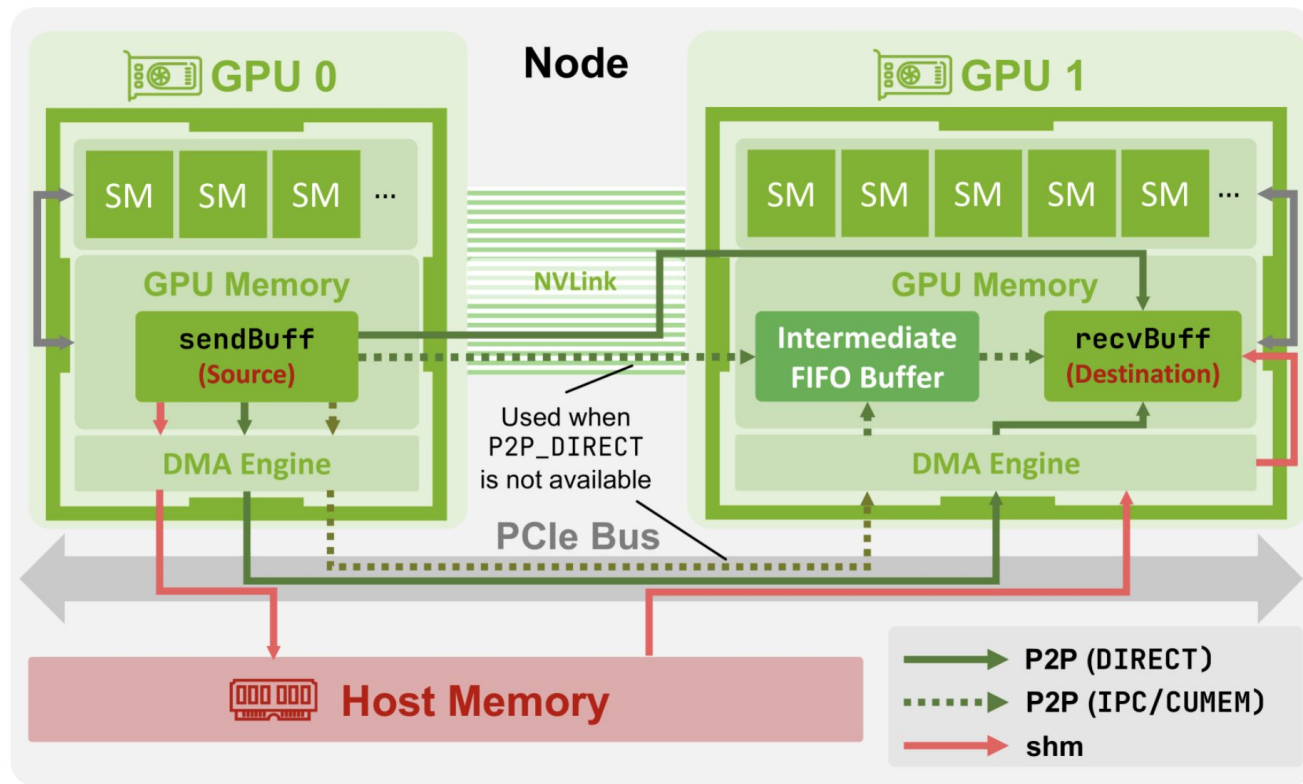
Contrast: scale-up vs scale-out?

Scale-out = RDMA (RoCE/IB)

- [1] NVLink ~9x more bandwidth per GPU (e.g., 900GB/s vs 100GB/s)
- [2] Transport “in silicon” - die/package area is of paramount concern
- [3] Direct memory access via pointer-based operations - easier to program (?)
- [4] Power efficiency is essential, hence “dense scale” and copper
- [5] Collective offload (reductions, multicast) ← c.f. InfiniBand

Primary difference: bandwidth, latency of direct operations, power efficiency

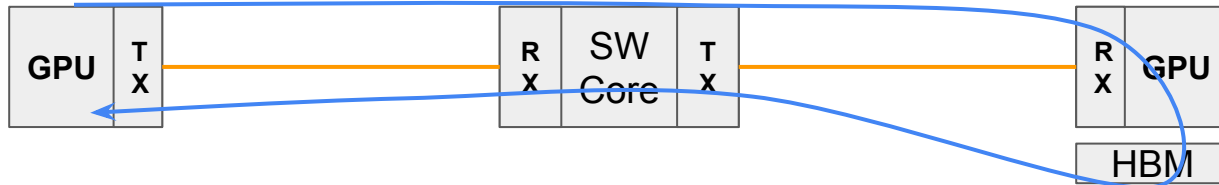
NCCL over NVLink



- Low-latency comms are driven by SM (streaming multiprocessors)
- CUDA kernels launched by NCCL library push and poll data between GPUs
- Producer/consumer model with synchronization

The hard part: latency

- Static: propagation delays, data-link layer
- Dynamic: queueing and FEC effects
- Error propagation: non-trivial in memory-semantic protocols
- Comm-type: reads require round-trip; write & poll is faster



- 2x GPU die crossing (from SM to remote L2 cache/HBM)
- 2x TX/RX FEC processing
- 2x wire propagation
- 1x switch core crossing and queueing

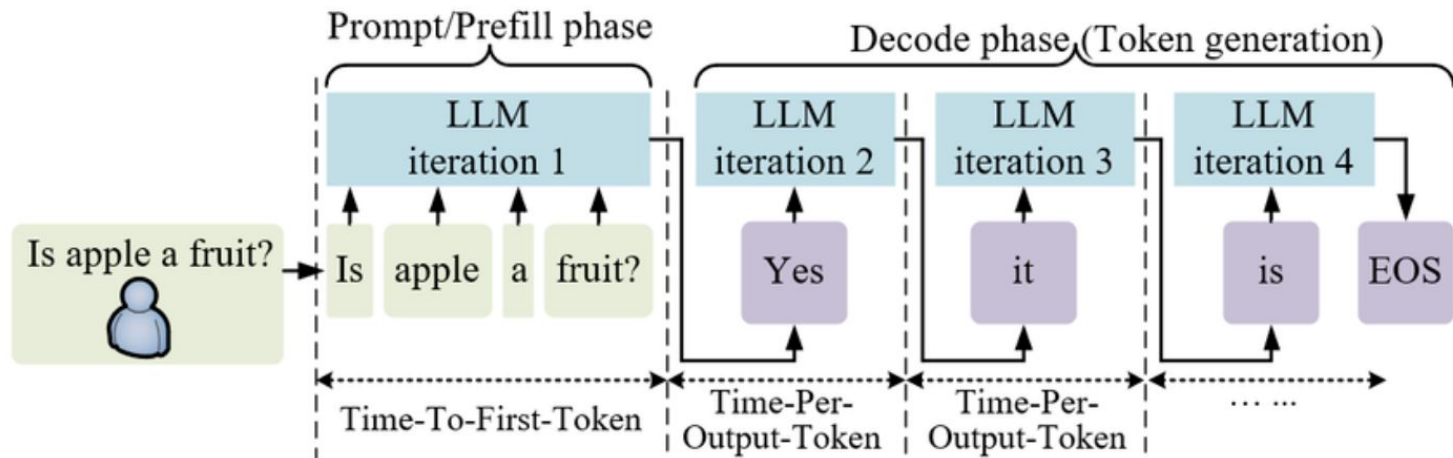
What's besides NVLink?

- **UALink** standard (no implementations yet?) → Ethernet SerDes
- Broadcom **SUE** → Ethernet based (WIP?)
- Both documented openly online

Tricky part: you need compute engine to implement the transport

So why is scale-up a big deal for inference?

Performance metrics for inference



TTFT and TPOT (ITL) are critical inference metrics

Total query completion time = TTFT + OSL * TPOT

So how scale-up helps... how?

- **Low-latency:** needed for all-reduce in the $\max(\text{tokens/user})$ regime - for dense matrix multiplications
- **Larger radix/scale:** helps with sharding and boosts normalized memory bandwidth (=more shards)
- **High bandwidth:** needed for latency on mid-to-large batches

NOTE: more “sharding” means more exposed comms latency (less compute to hide)

Disaggregated inference

Prefill & Decode aka “Context” vs “Generation”

Recall inference has two phases

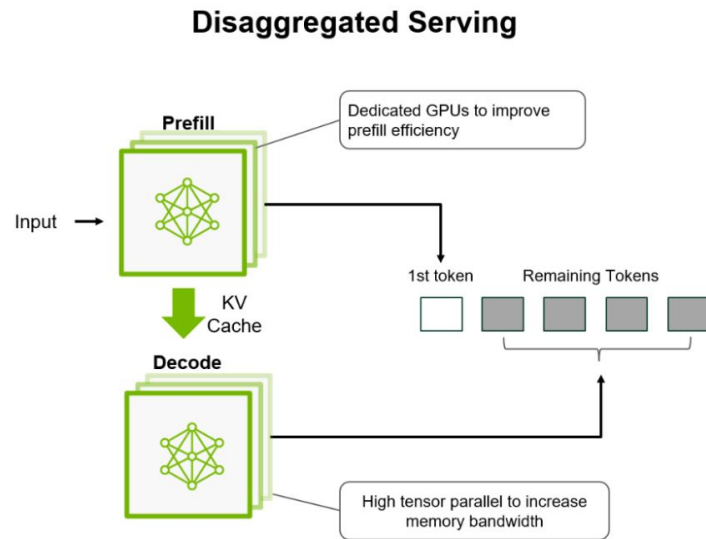
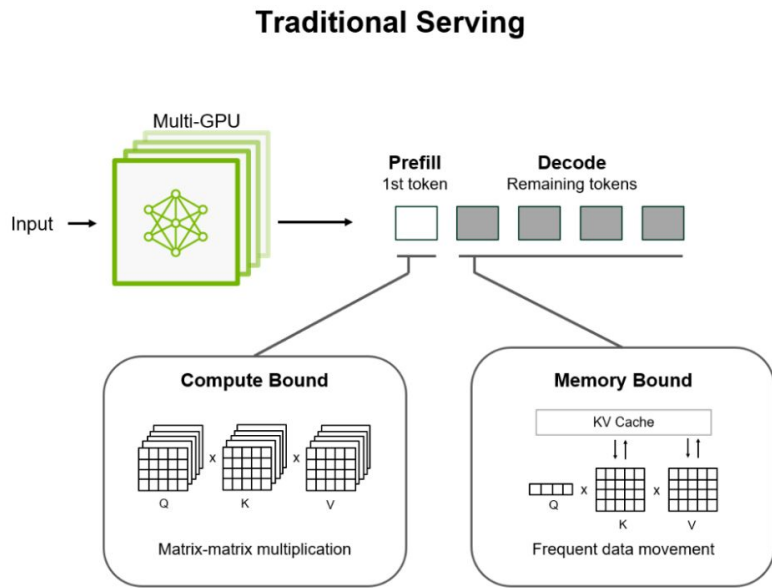
- Prefill aka Context: process the initial batch (attention)
- Decode/Generation: generate one token a time per sequence

Prefill is almost entirely **compute bound**

Decode requires higher memory bandwidth and low comm latency

It is not uncommon to see two phases “**disaggregated**”

Prefill/Decode (P/D) split



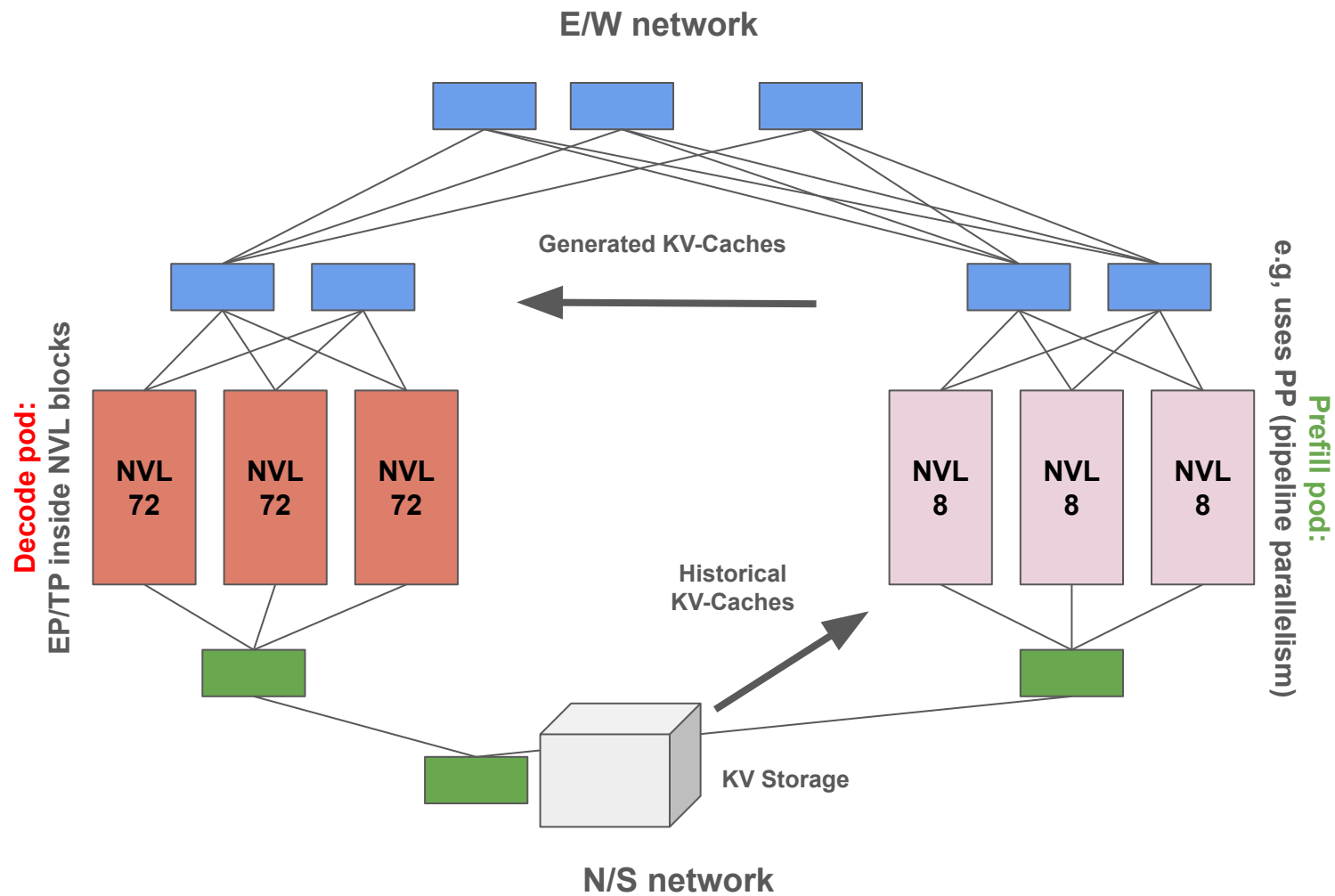
How does P/D disagg works again?

- Same model, but hosted twice
- Different distributed partitioning strategies
- Global scheduler (router) directs queries
- Prefill produces **KV cache**
- Decode takes the KV cache and emits tokens

What's KV Cache storage?

- There are two kinds of “KV cache”
- First is the “active” that is used in running computations
- Second is “inactive” which encodes your old sessions
- Alternatively: can you your “indexed” code base

KV cache “offline” storage is critical for performance on very large contexts (e.g. 100K+ tokens of history or shared context)



Recap

- LLM inference is distributed matrix multiplication (TP/EP/PP)...
- KV cache creates memory capacity pressure
- Throughput vs. latency tradeoffs are non-trivial
- Distributed inference runs over scale-up - whether possible
- Disaggregated inference specializes “pods” for prefill/decode
- E/W network transfers KV caches
- Inactive KV cache storage becoming essential for large contexts