RAY SUMMIT 2026

By Anyscale

NVIDIA and vLLM Full-Stack Collaboration for MoE Model Performance

Original talk: Siyuan Fu
Sr. AI Developer Technology Engineer, NVIDIA

Recap by Arisa Umparat

ARISA UMPARAT

AI ENGINEER | NLP RESEARCHER | DATA SCIENTIST



EXPERIENCE

- AI Engineer — Ascend Group
- Senior Data Scientist, Health Link Special Project — BDI
- AI Software Engineer Intern — BOL
- Analyst - Bangkok Bank
- Data Scientist — OSN
- Nuclear Medicine Technologist — Bumrungrad International Hospital

EDUCATION

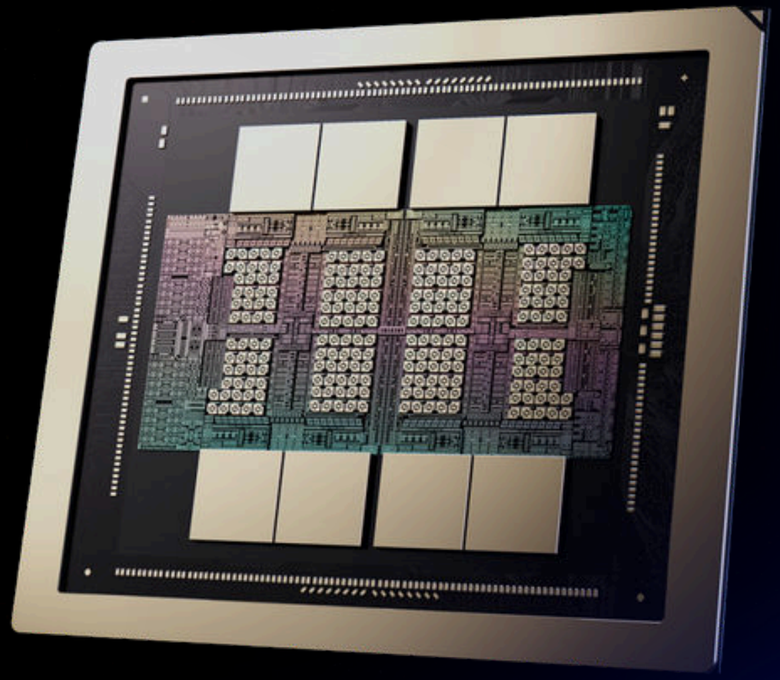
- M.Sc. in Statistics, Data Science — Chulalongkorn University
- B.Sc. in Radiological Technology — Mahidol University

RESEARCH & PUBLICATIONS

- WiML @ NeurIPS 2026 | *Where You Put the Space Matters: Boundary-Conditioned Whitespace Robustness*
- IEEE JCSSE 2023 | Conference Paper | *Improving Pre-Trained Models for Multi-Label Classification in Stack Overflow: A Imbalanced Data Handling Methods*

RAY SUMMIT 2026

ถ้าเราใช้ GPU ที่แรงที่สุด
LLM ของเราจะตอบเร็วสุดไหม ?

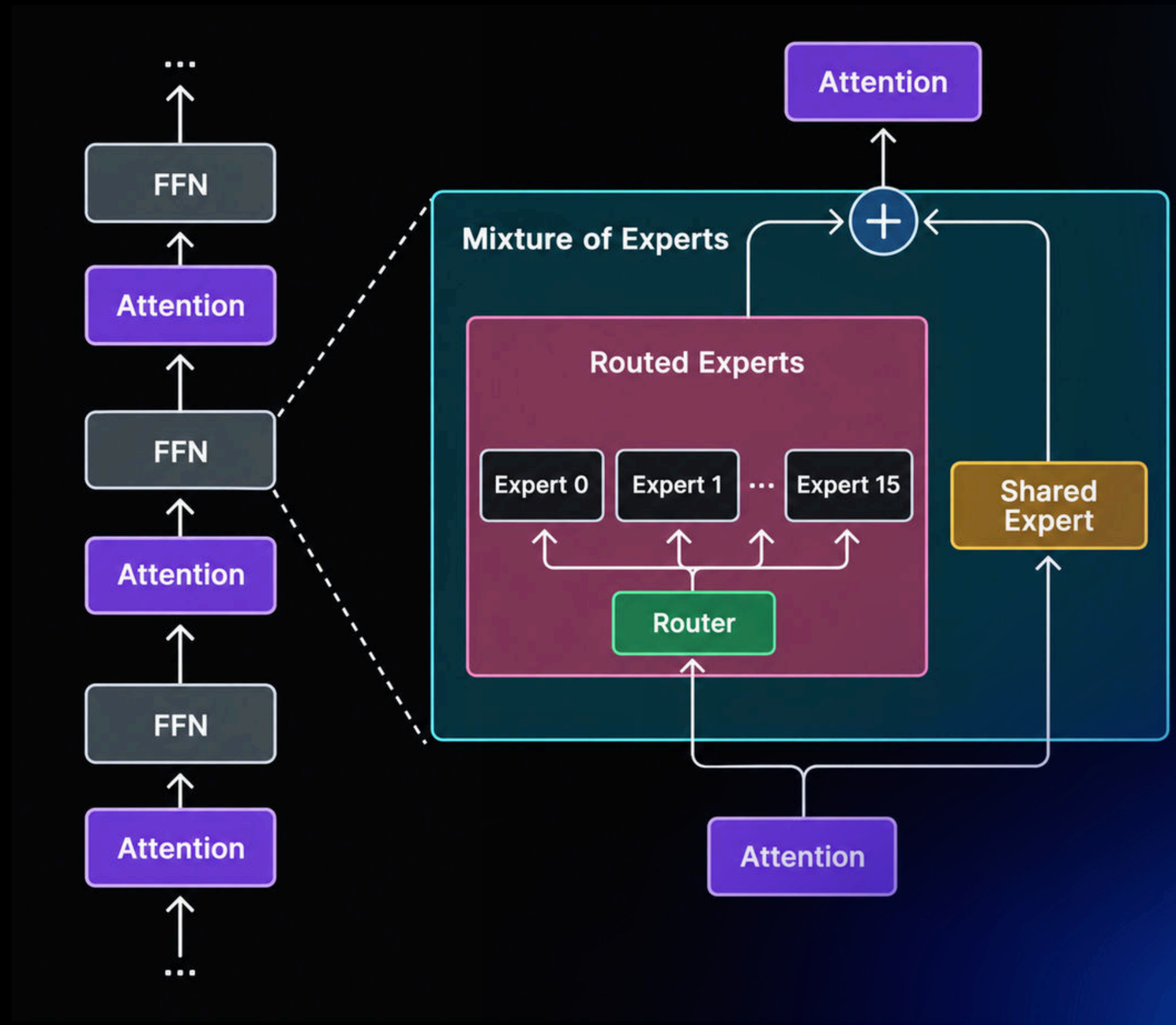


ไม่เสมอไป

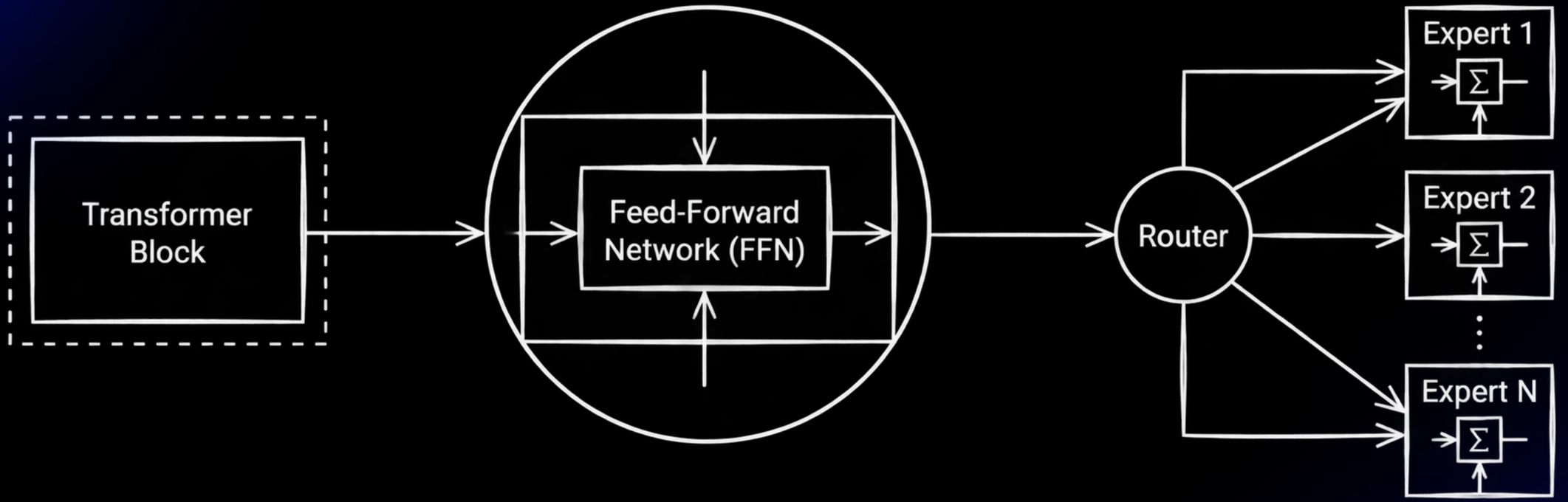
ความเร็วของ Inference ไม่ได้ขึ้นอยู่กับพลังการคำนวณของ GPU อย่างเดียว
โดยเฉพาะกับโมเดลขนาดใหญ่อย่าง Mixture of Experts

โมเดล MoE

Mixture of Experts



Mixture of Experts



เป็นสถาปัตยกรรมโมเดลที่มีโมดูลย่อยหลายตัวเรียกว่า Experts และมี Router คอยเลือกว่าจะส่งข้อมูลแต่ละ Token ไปให้ Expert ตัวไหนประมวลผล

เราแบ่งโมเดล ข้าม GPU อย่างไร?

Tensor Parallelism (TP)

แบ่ง computation ภายใน layer เดียว กระจายข้ามหลาย GPU และรวมผลด้วย collective communication

Expert Parallelism (EP)

กระจาย Experts ไปคนละ GPU Router ส่ง token ไปยัง Expert

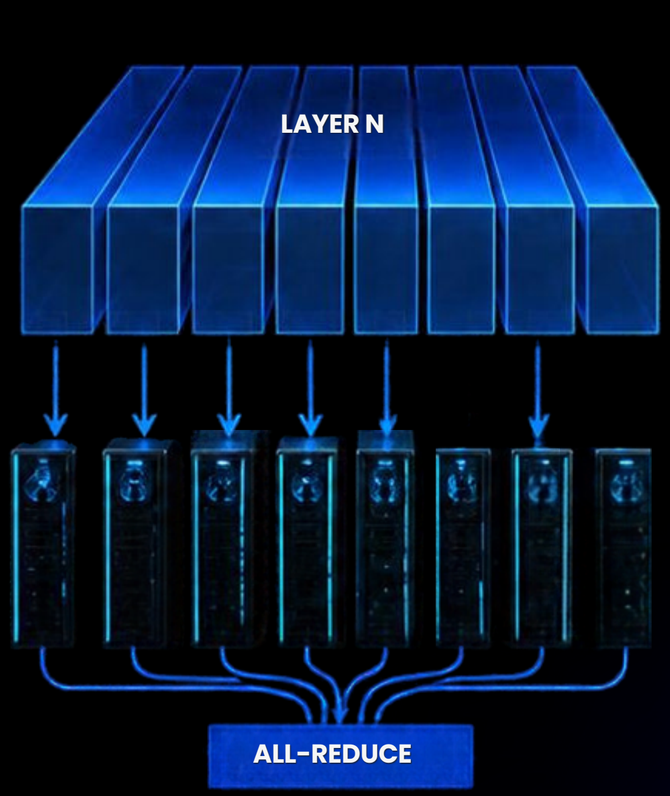
TP vs EP

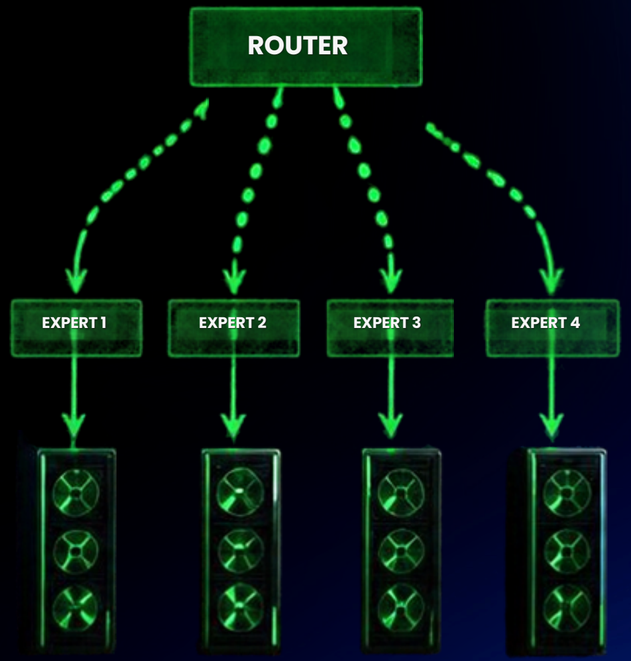
TENSOR PARALLELISM (TP8)

แบ่ง layer เดียวบน 8 GPUs

EXPERT PARALLELISM (EP4)

กระจาย Experts บน 4 GPUs

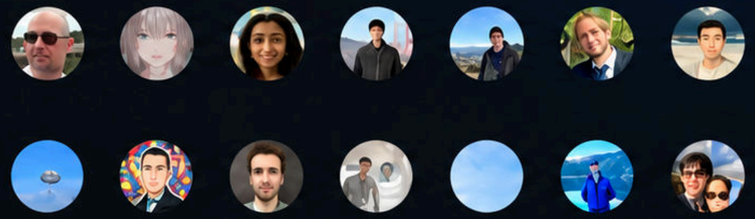




NVIDIA × vLLM Open-Source Community

60+

Nvidian vLLM contributors



110+

PRs merged during past 3 months

NVIDIA

vLLM

Day 0 Support

รองรับโมเดลใหม่ตั้งแต่วันเปิดตัว
vLLM ใช้งานโมเดลนั้นได้ตั้งแต่วันแรก

ลดเวลารอการรองรับโมเดล
ช่วยให้เริ่มทดสอบและ Deploy ได้เร็วขึ้น



Bottleneck 4 ระดับของระบบ AI

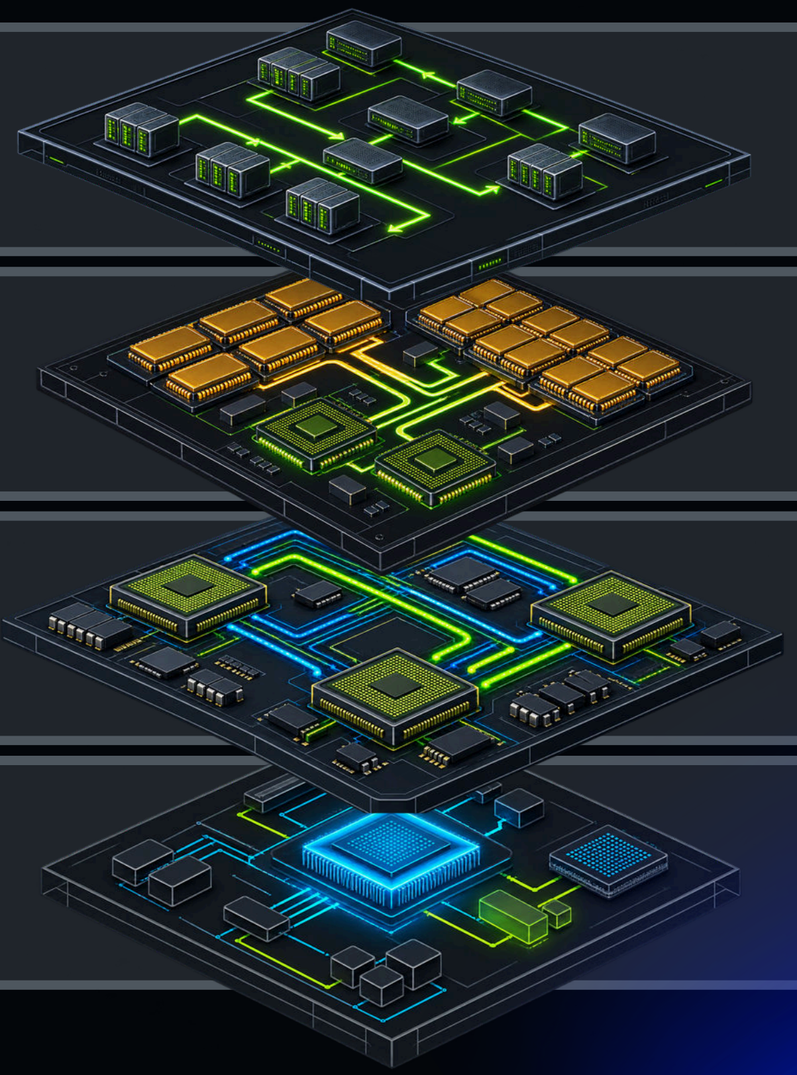
ขอบเขตปัญหา

Level 4 (Macro) — NVIDIA Dynamo
System Orchestration
จัดคำขอและกระจายงานทั้งระบบ

Level 3 — DeepSeek R1
Memory & Hardware
จัดการหน่วยความจำและฮาร์ดแวร์

Level 2 — MiniMax M2.5
Cross-GPU Communication
ลดเวลาสื่อสารระหว่าง GPU

Level 1 (Micro) — Kimi K3
Kernel Compute
เร่งการคำนวณภายใน GPU



The Full-Stack Optimization Matrix

MoE Models

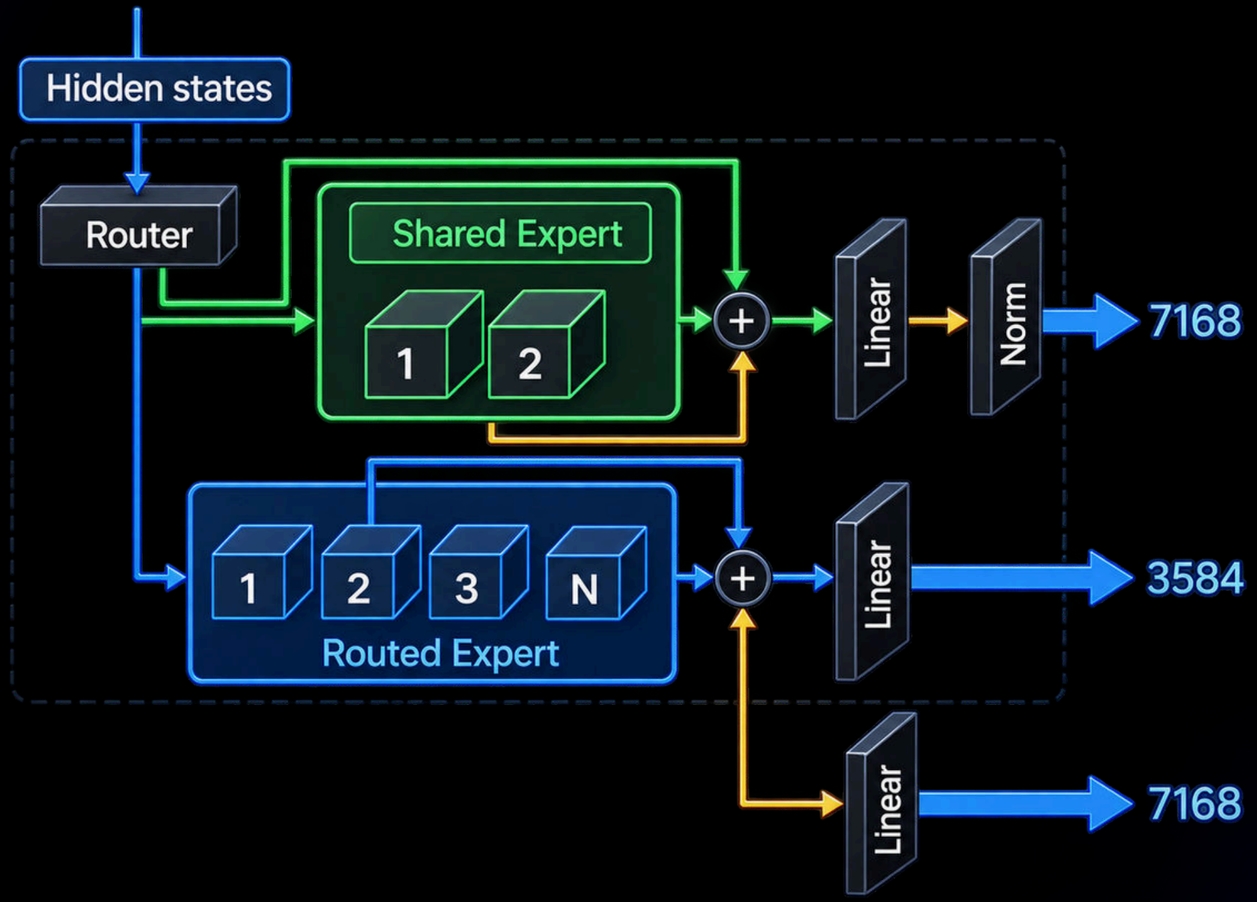
Kimi K3	**Bottleneck:** MoE Tail Collectives	**Solution:** Kernel Fusion
MiniMax M2.5	**Bottleneck:** Tiny Payloads, Big Syncs	**Solution:** One-shot All-Reduce
DeepSeek R1	**Bottleneck:** P/D Memory Imbalance	**Solution:** Disaggregated Serving & Offloading
Dynamo	**Bottleneck:** Request & State Chaos	**Solution:** KV-Aware Routing

Kimi K3

Latent MoE Optimization

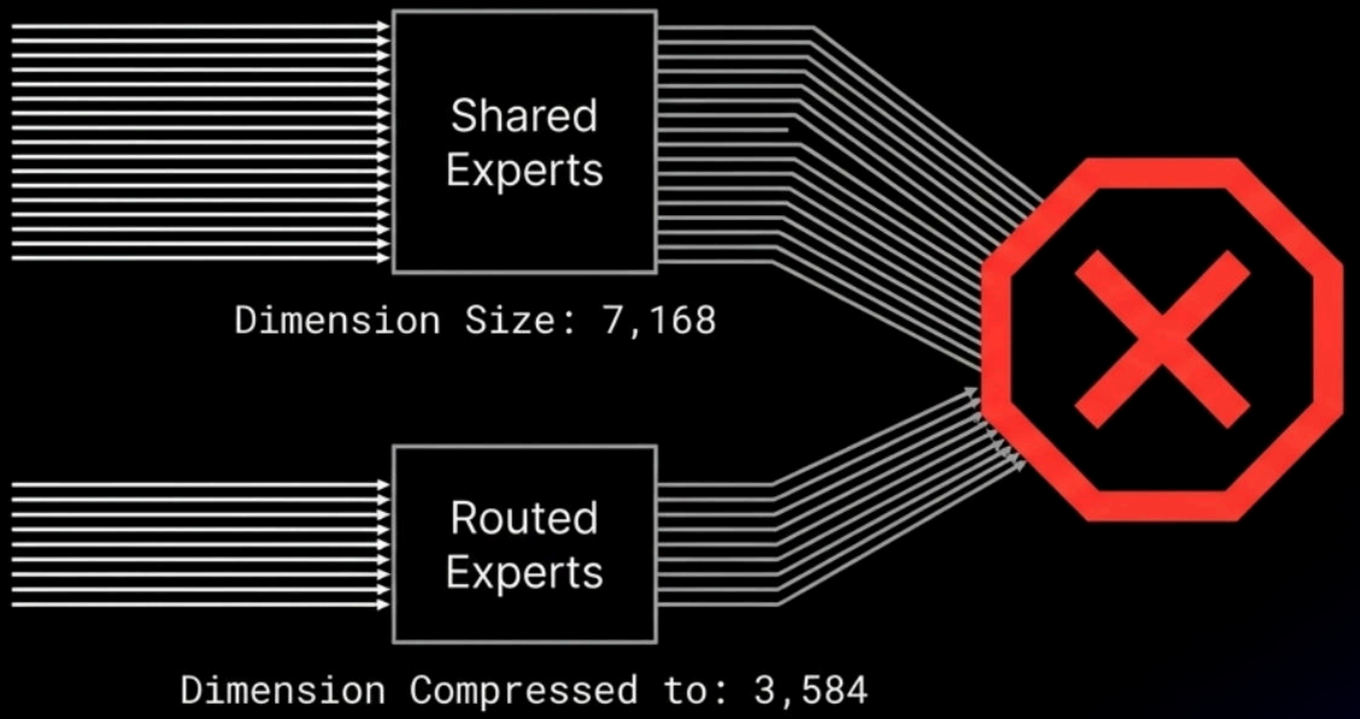
RAY SUMMIT 2026

ทำไม MoE tail ทำงานช้า ?



- Shared 7,168
- Routed 3,584

ขนาดข้อมูลไม่เท่ากัน



Bottleneck : MoE Tail Collectives

Kimi K3

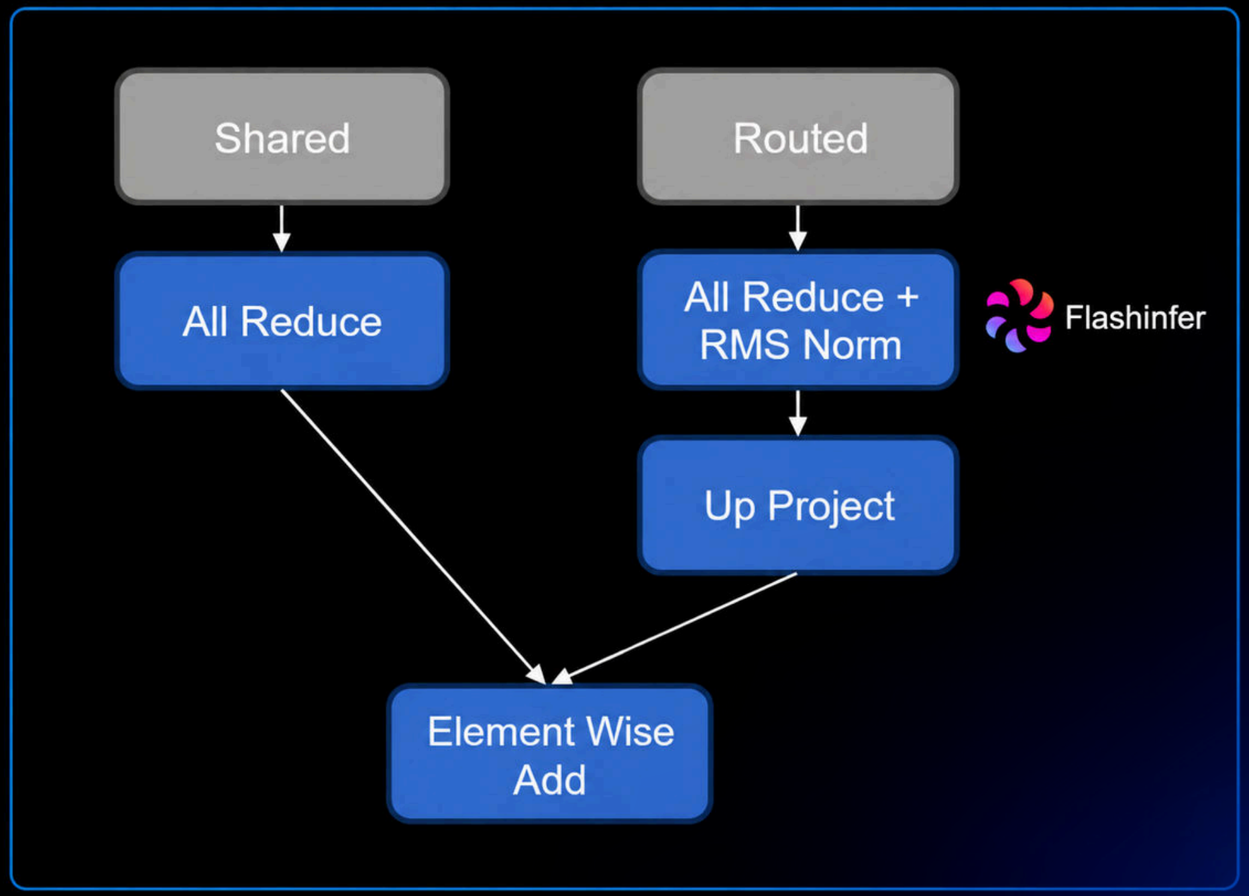
จุดที่ต้อง Optimize เพื่อ ลด latency ของ MoE tail

Shared branch

All Reduce

Routed branch

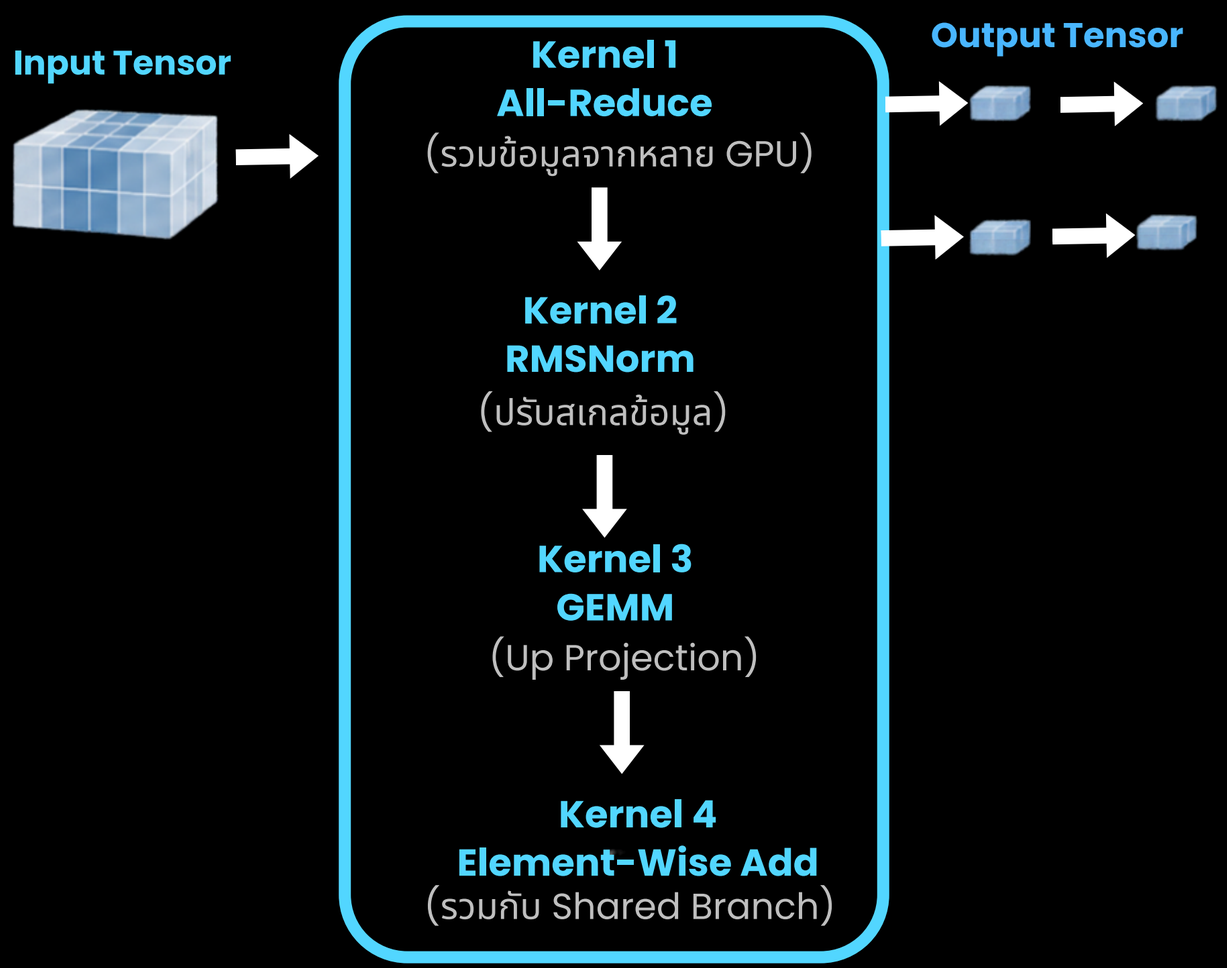
RMSNorm + Up Project



ก่อนทำ Kernel Fusion

เรียกหลาย Kernel แยกกัน

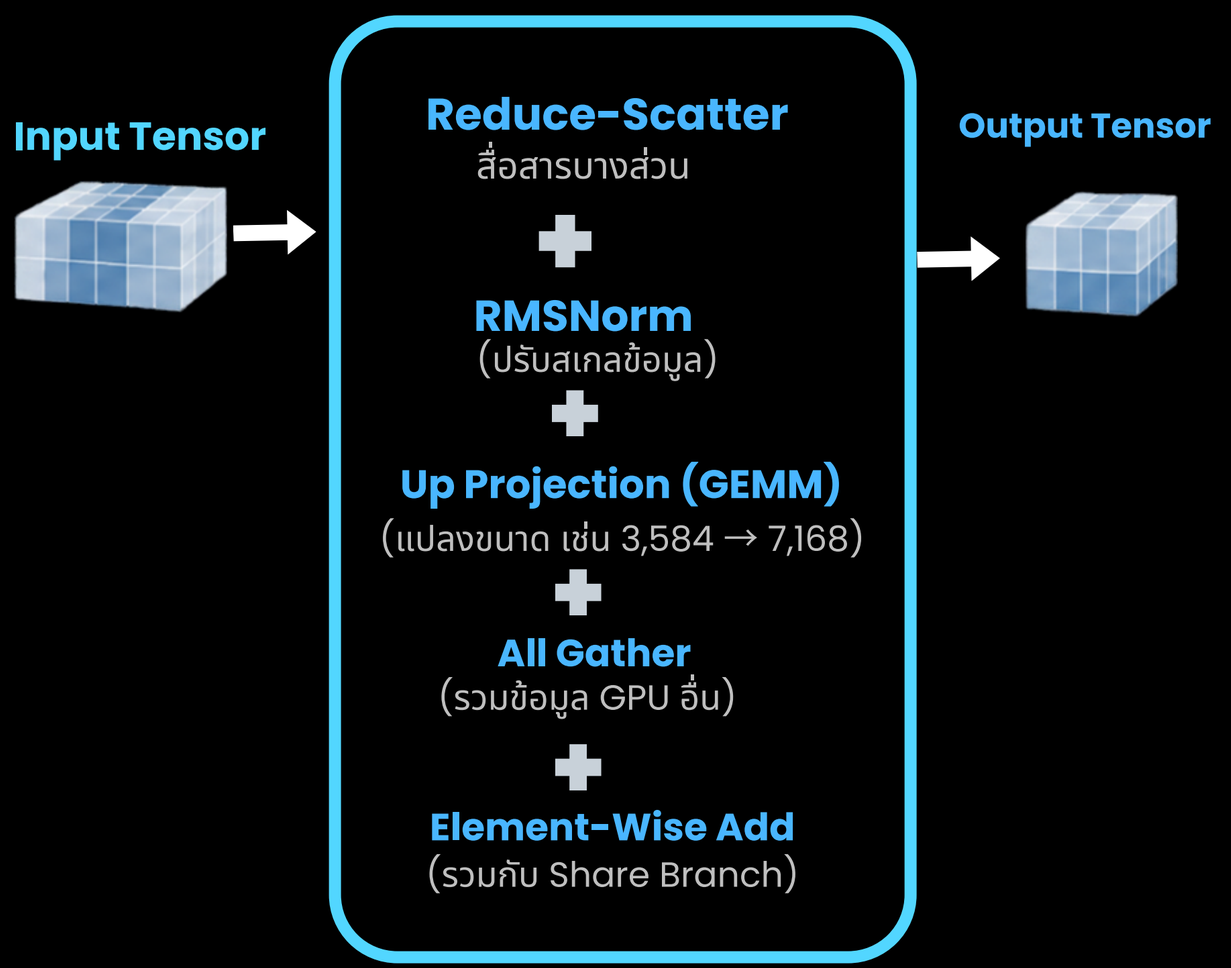
GPU Kernels ทำงานแยกกัน



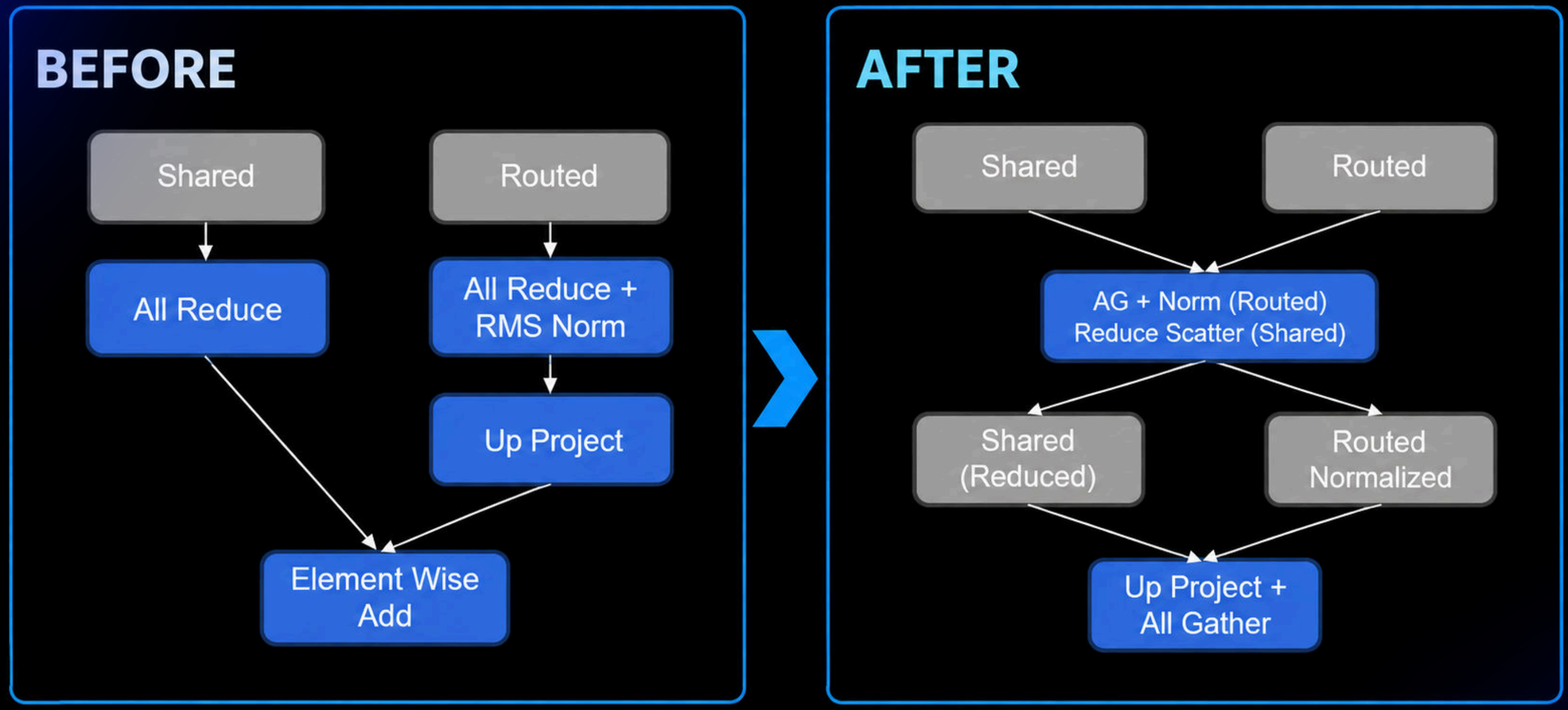
หลังทำ Kernel Fusion

รวมหลาย operation ใน Kernel เดียว (ลดการรอ)

Fused GPU Kernel



RESULT AFTER OPTIMIZATION

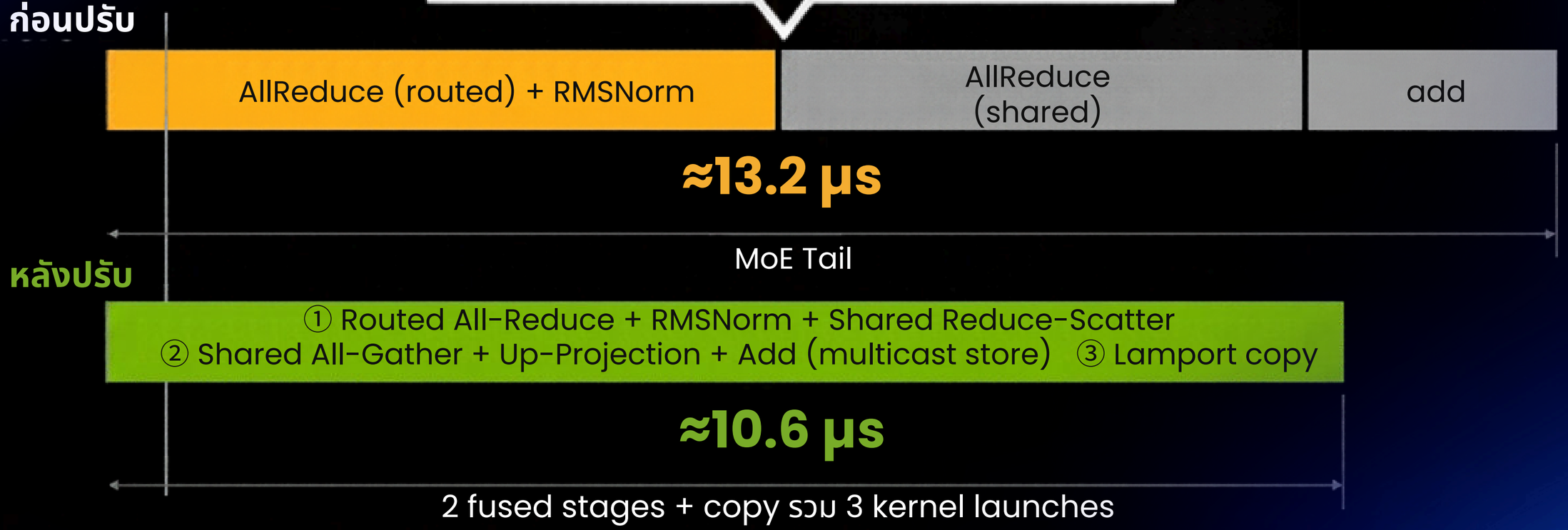


Split All Reduce | **Overlap communication + compute** | **Fuse All Gather + GEMM**

Kernel Fusion ด้วย FlashInfer

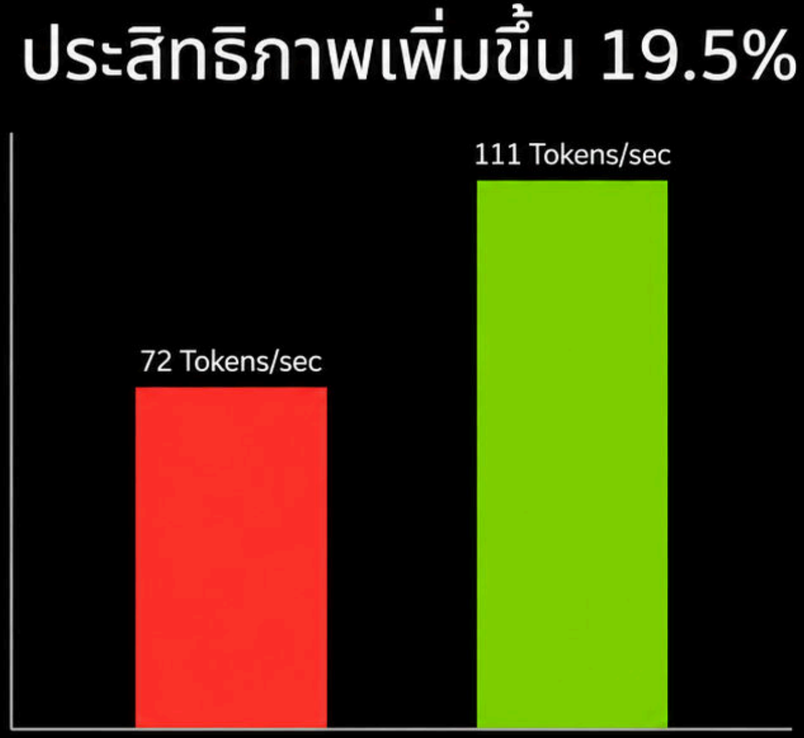
MoE tail เวลาลดลง

-19.5%



FlashInfer คือคลัง GPU kernels ที่ vLLM เรียกใช้เพื่อเร่ง Inference

72 to 111 Tokens/sec

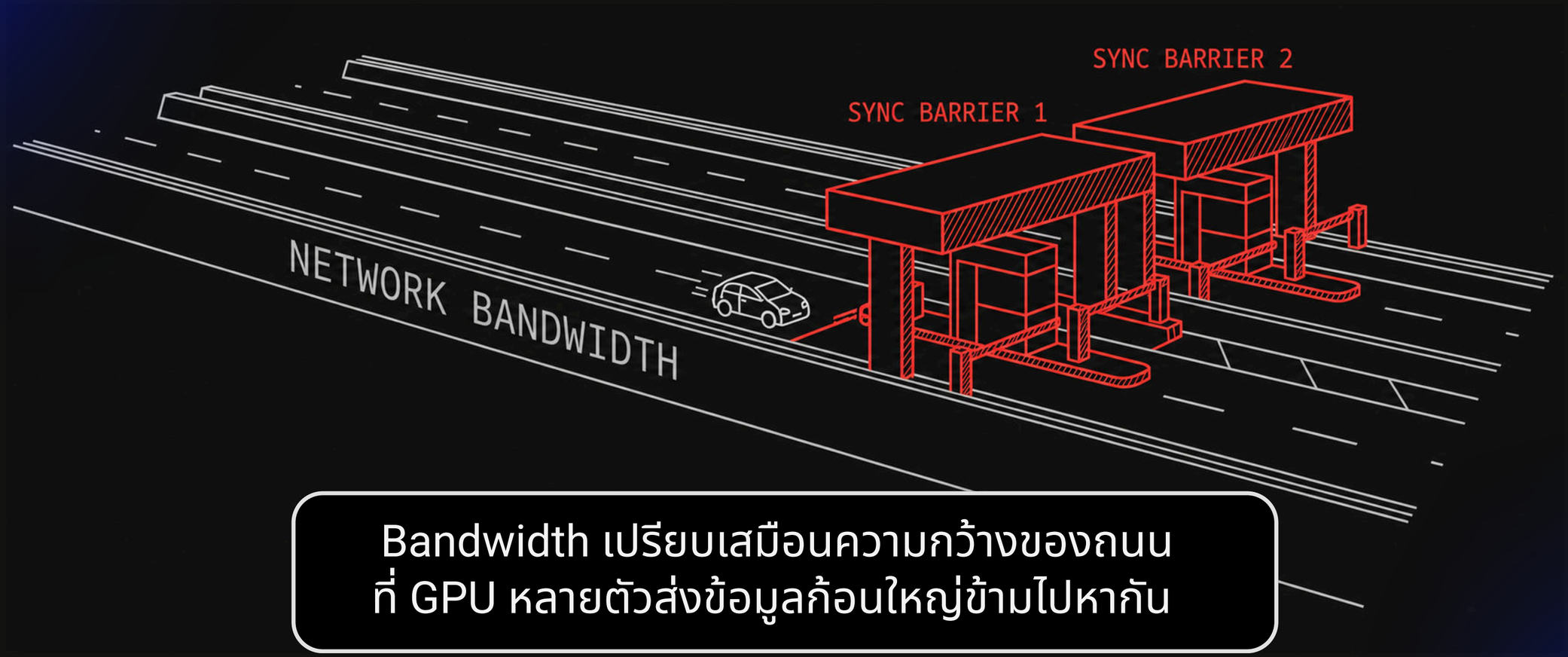


Before and After Kernel Fusion

MiniMax

การเพิ่มประสิทธิภาพ Full Heads RMSNorm
ด้วย One-Shot All Reduce

ข้อมูลมีขนาดเล็กแต่ทำไมยังทำงานช้า ?

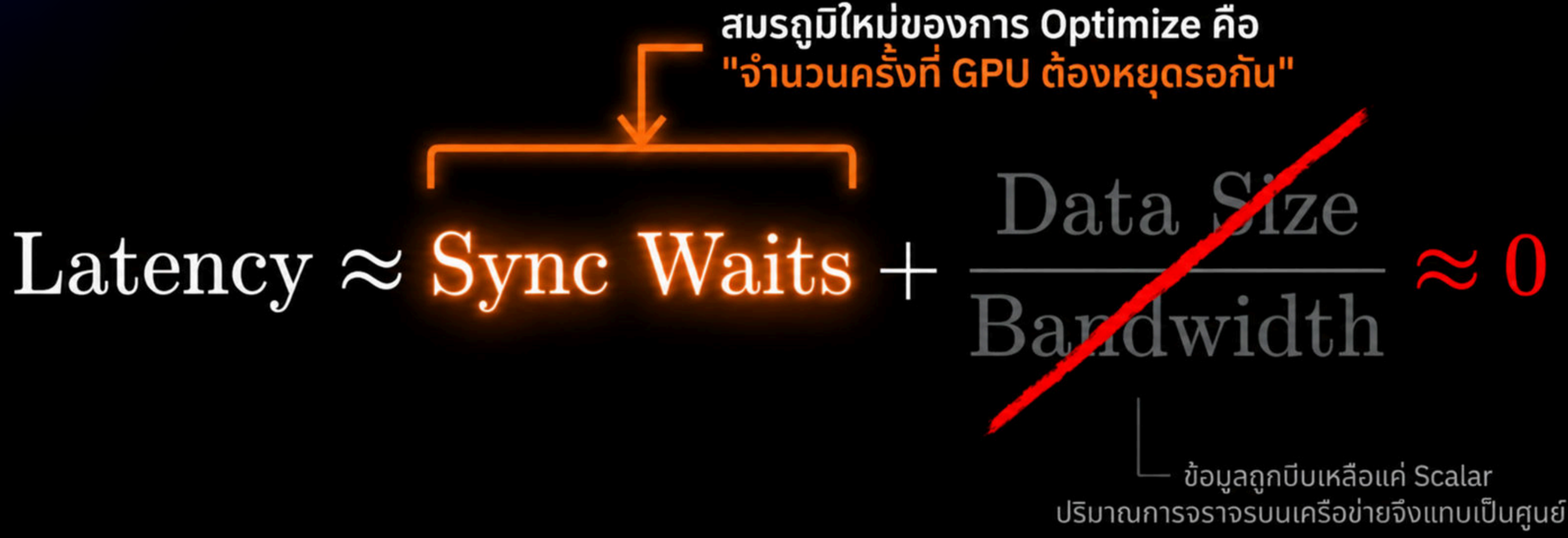


Bandwidth เปรียบเสมือนความกว้างของถนน
ที่ GPU หลายตัวส่งข้อมูลก้อนใหญ่ข้ามไปหากัน

แต่ MiniMax M2.5 ส่งข้อมูลที่เล็กมาก แค่เพียงเลขตัวเดียวต่อหัว เหมือนส่งกระดาษโน้ตแผ่นเล็กๆ แผ่นเดียว

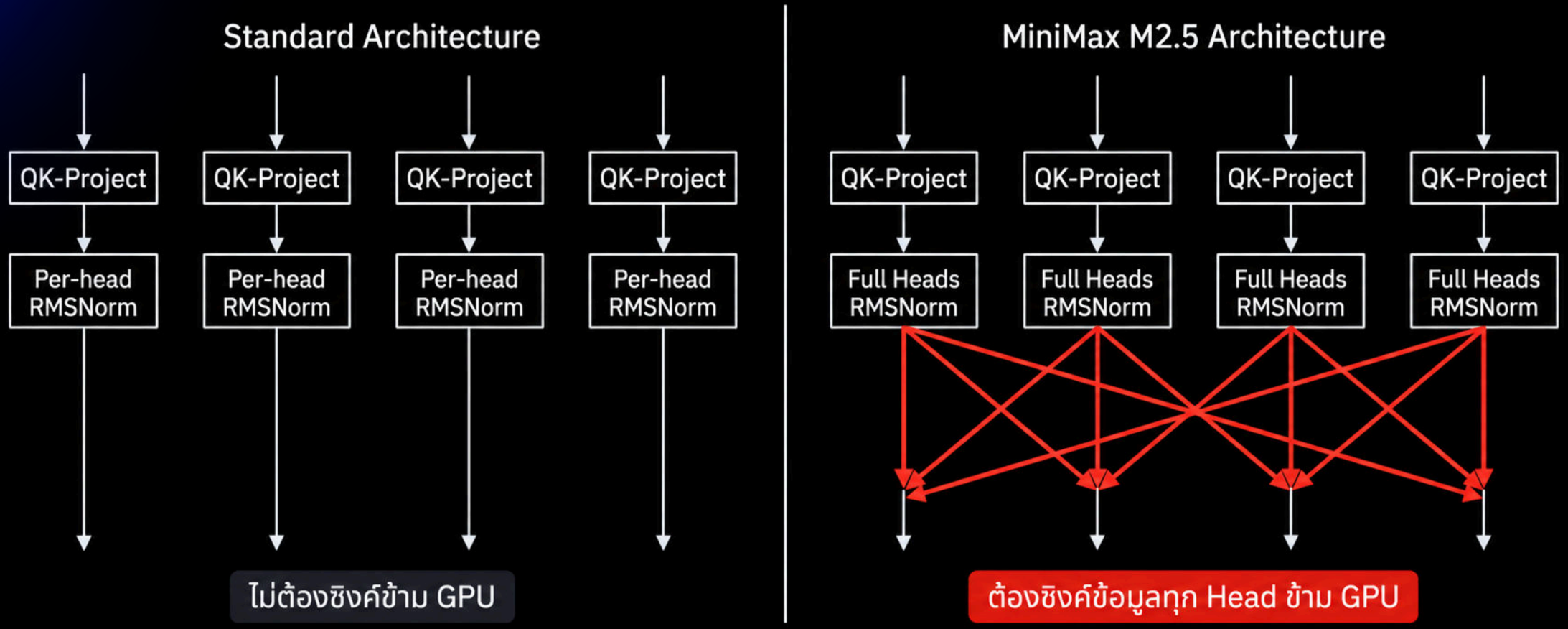
Latency เยอะ เกิดจากขั้นตอนไหน ?

เมื่อข้อมูลเล็กจนไม่เต็ม Bandwidth ปัญหาหลักจึงเหลือเพียง Sync Barrier



ไม่ได้เกี่ยวกับความกว้างของท่อส่ง (Bandwidth)

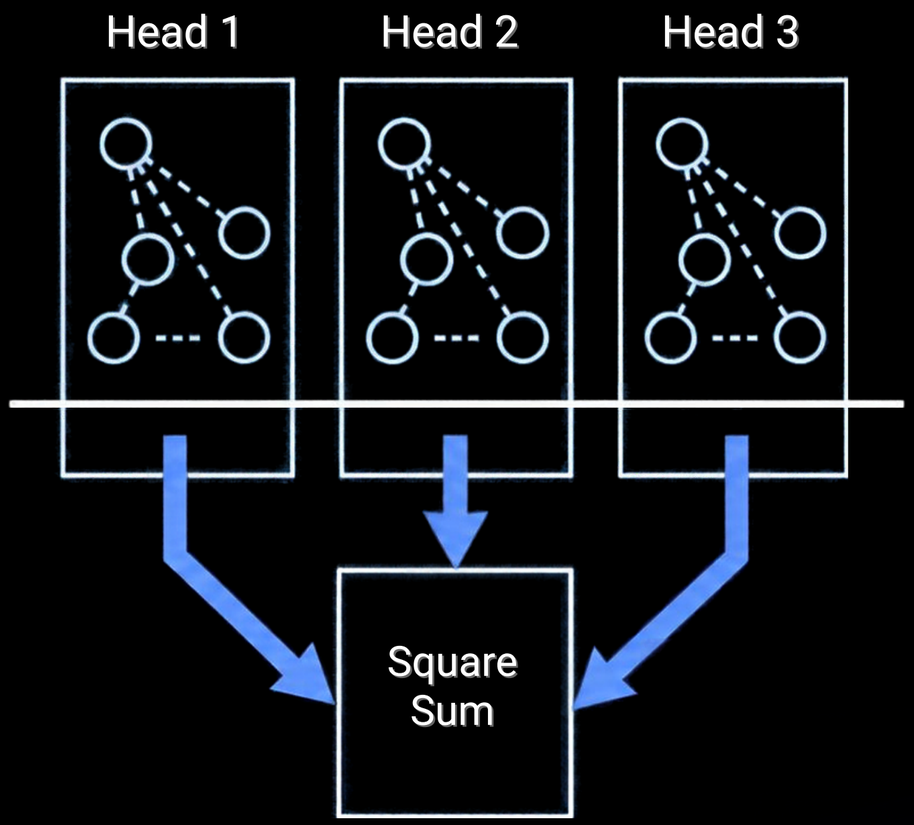
ทำไม MiniMax ถึงต่าง ?



โครงสร้างของ MiniMax บังคับให้ต้องหาค่าผลรวมกำลังสอง (Square Sum) จากทุก Head ที่กระจายอยู่บน GPU แต่ละตัว ทำให้เกิด All-Reduce Operation หลังขั้นตอน Attention อย่างหลีกเลี่ยงไม่ได้

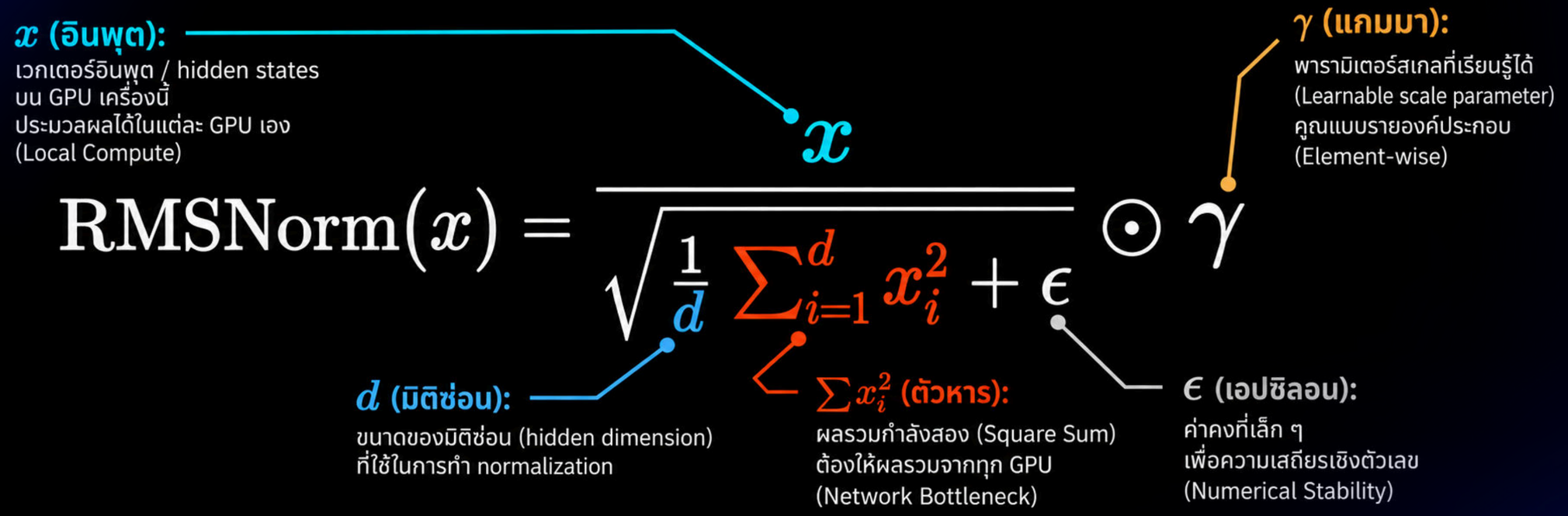
ปัญหาไม่ได้อยู่ที่ ‘ขนาดข้อมูล’ แต่อยู่ที่ ‘การรอ’

Bottleneck: Tiny Payloads, Big Syncs



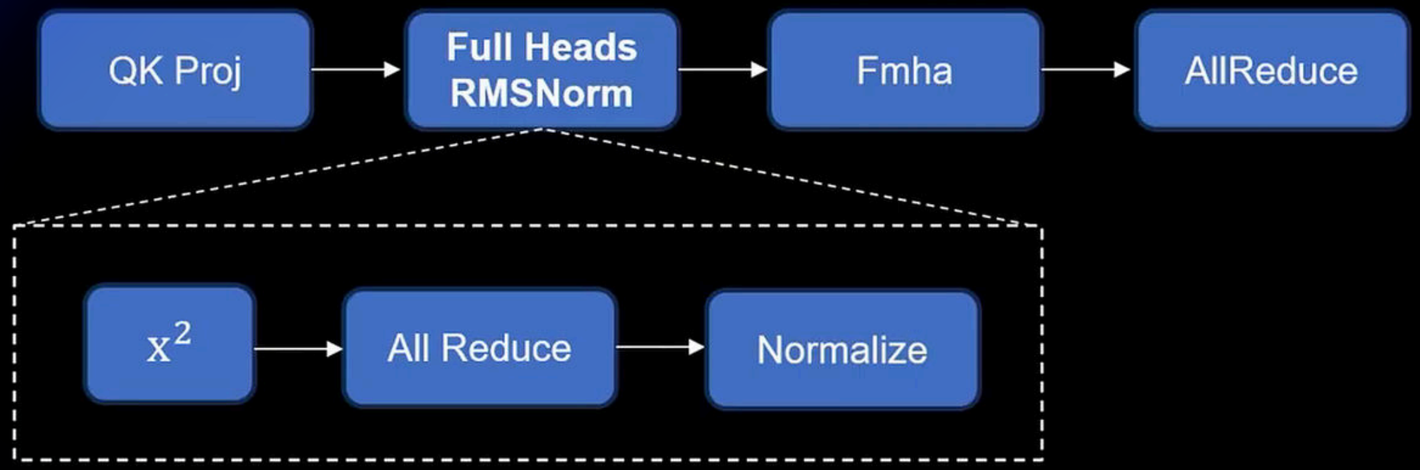
สิ่งที่ต้องซิงค์มีเพียงค่า Scalar (Square Sum) ตัวเดียวต่อ Head สำหรับเป็นตัวหารใน RMSNorm

สมการ RMSNorm



เราไม่จำเป็นต้องส่ง Vector x ทั้งหมด เพียงแค่คำนวณ Square Sum ไว้ล่วงหน้า
แล้ว All-Reduce ข้าม GPU เฉพาะค่า Scalar เท่านั้น

Bottleneck ของ RMSNorm

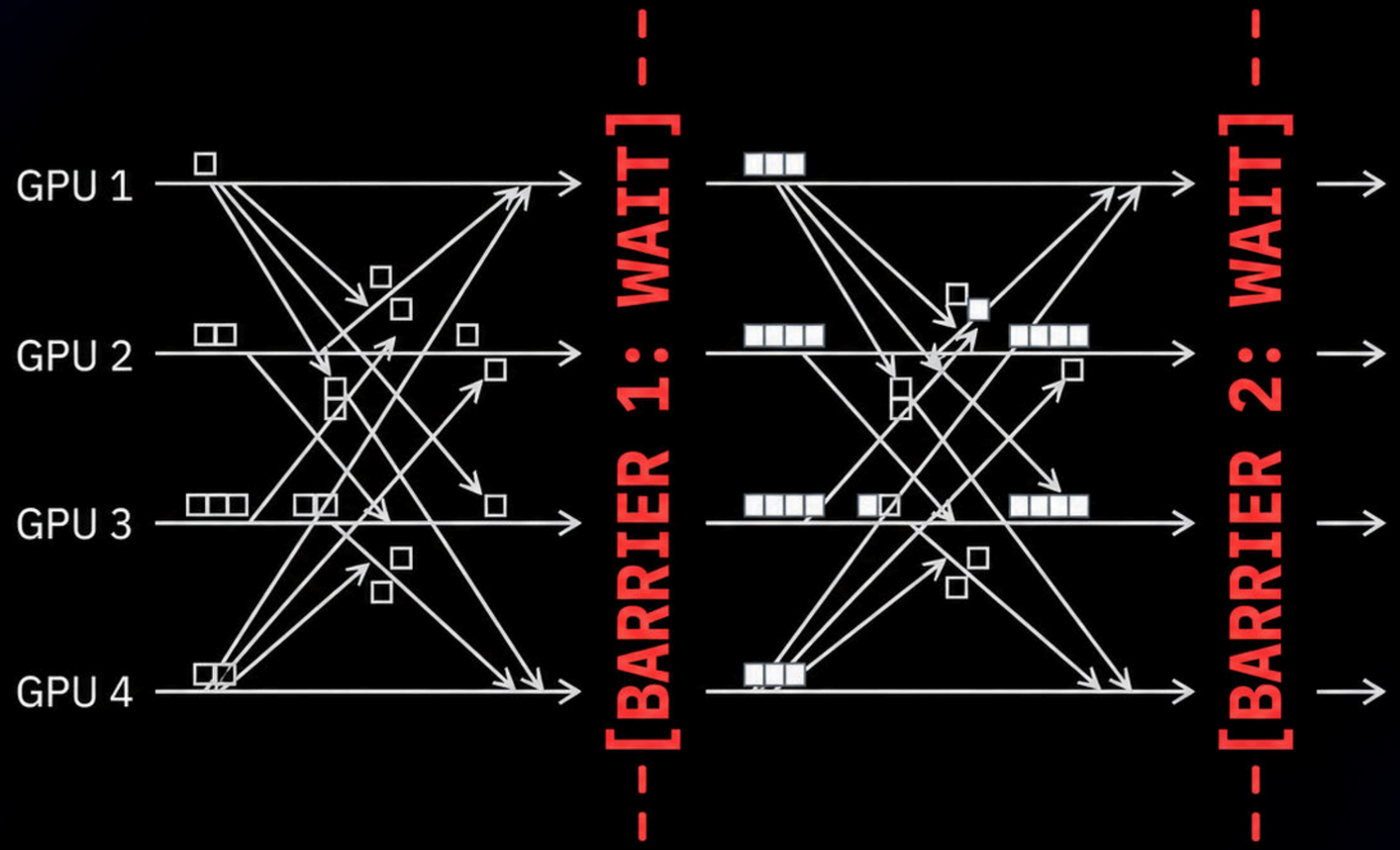


ทำไมต้องรอ?

1. QK Projection แบ่งตาม Rank
2. RMSNorm ใช้สถิติร่วมกัน
3. All Reduce มี Barrier

เพราะหลาย GPU ต้องรวมสถิติก่อน Normalize

แบบดั้งเดิม: Two-Shot All-Reduce



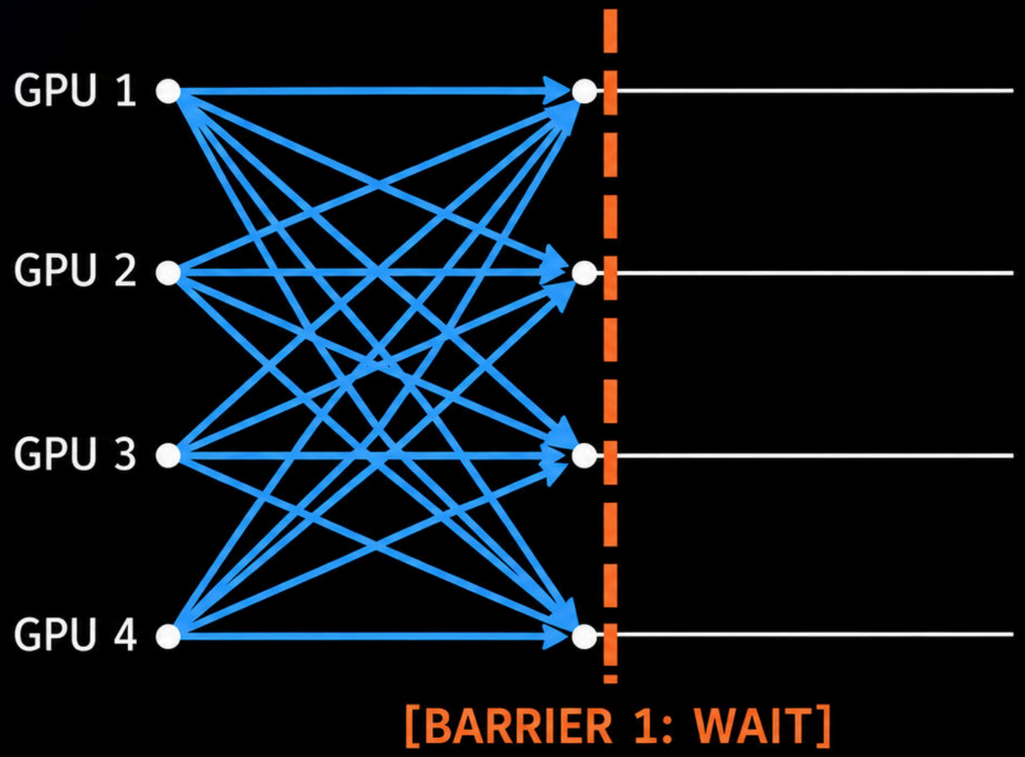
กระบวนการ:
Reduce-Scatter → All-Gather

ปริมาณข้อมูล (Traffic):
2(n-1)

จุดรอ (Sync Barriers):
2 ครั้ง

Latency รวม:
15.5 µs

แบบปรับแต่ง: One-Shot All-Reduce



กระบวนการ:
All-to-all → Local Reduce

ปริมาณข้อมูล (Traffic): n^2-n

จุดรอ (Sync Barriers): **1** ครั้ง

Latency รวม: 7.4 µs

Heuristic Rule Base สำหรับ Dynamic Routing

```
if batch-size < 40:
   protocol = One-Shot All-Reduce'  # Optimize for Latency (1 Sync)
else:
    protocol = 'Two-Shot All-Reduce'# Optimize for Bandwidth (2 Syncs)
```

เมื่อ Batch Size < 40:
ปริมาณจราจรข้อมูลน้อย (N^2-N)
เลือก One-Shot
ดึงข้อมูลตรง (1 Sync)
ประหยัดเวลาหน่วง

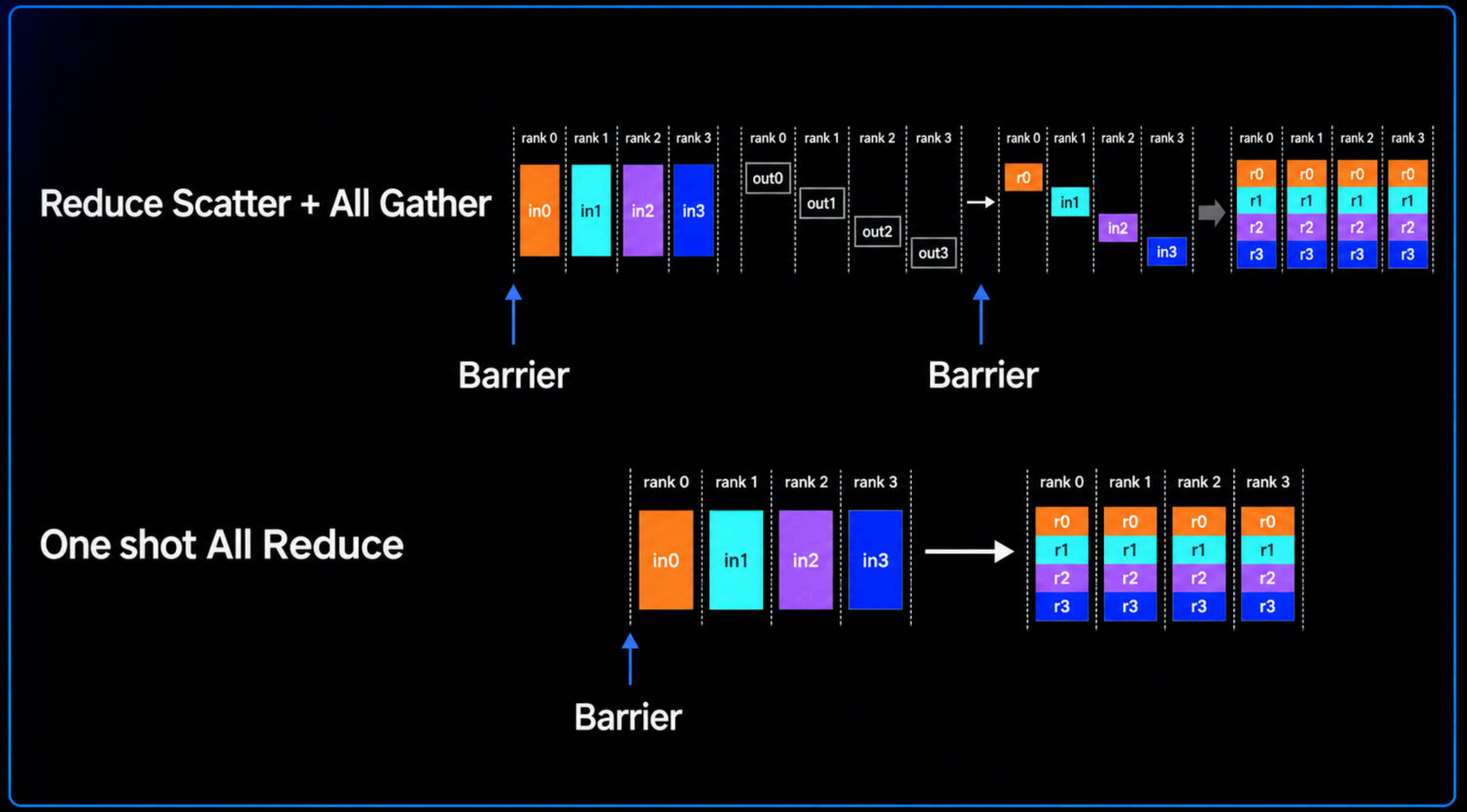


Batch Size = 40

เมื่อ Batch Size >> 40:
ปริมาณข้อมูลมหาศาล
สลับใช้ Two-Shot ทันที
เพื่อลดการจราจรเหลือ 2N-2
ยอมแลก 2 Syncs

FlashInfer ประเมินสภาวะของคอขวด (Bandwidth vs Latency)
และสลับอัลกอริทึมอัตโนมัติแบบ Real-Time โดยไม่ต้อง Hardcode

Two-shot vs One-shot

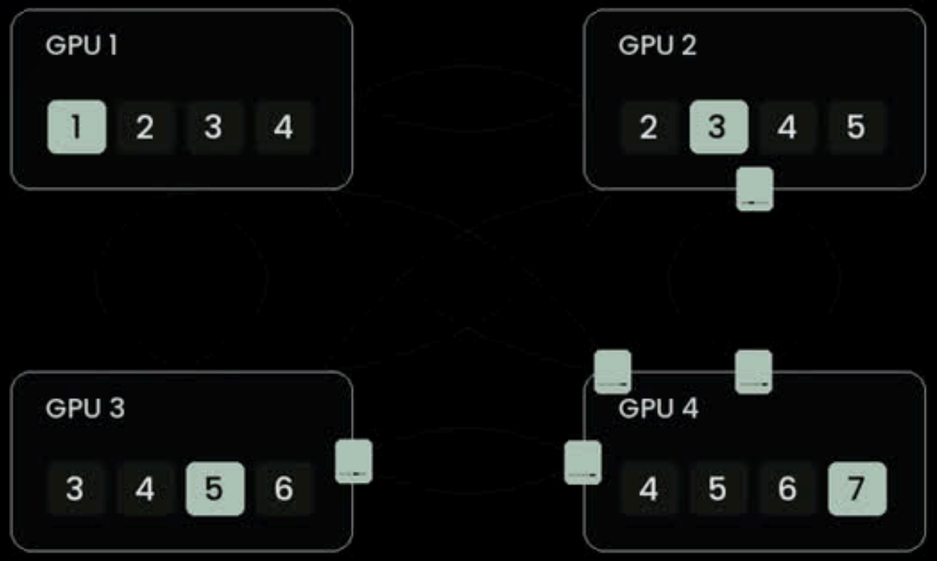


ลดจุดรอ จาก 2 Barriers → 1 Barrier

All-Reduce

Two-Shot

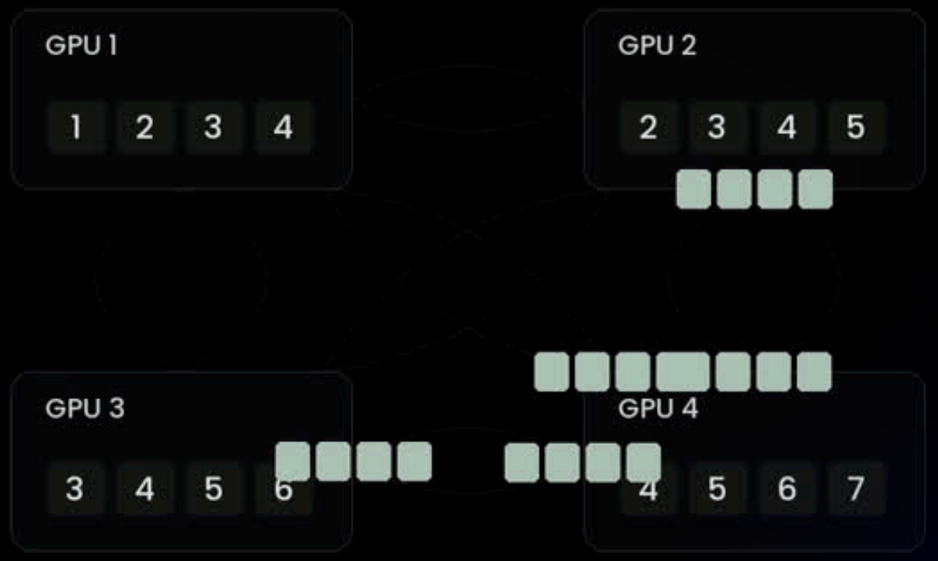
แบ่งกันบวก



6S · รอ 2 ครั้ง

One-Shot

ส่งทั้งชุด



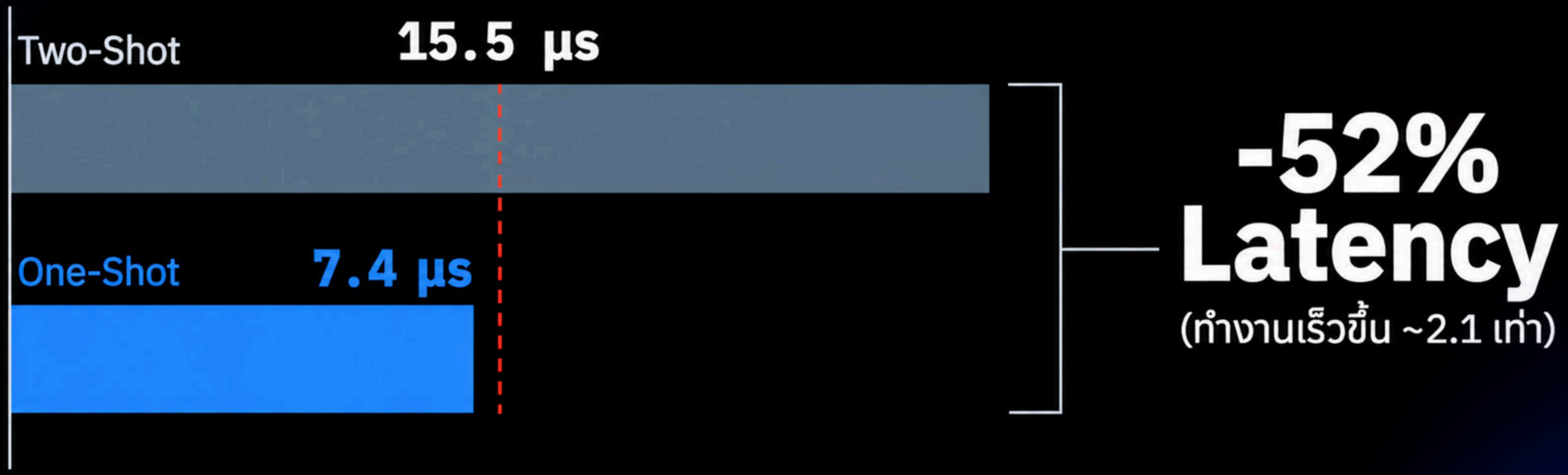
12S · รอ 1 ครั้ง

ภาพจำลอง · ข้อมูลส่งรวม · S = tensor เต็ม 1 ชุด

Compare Results

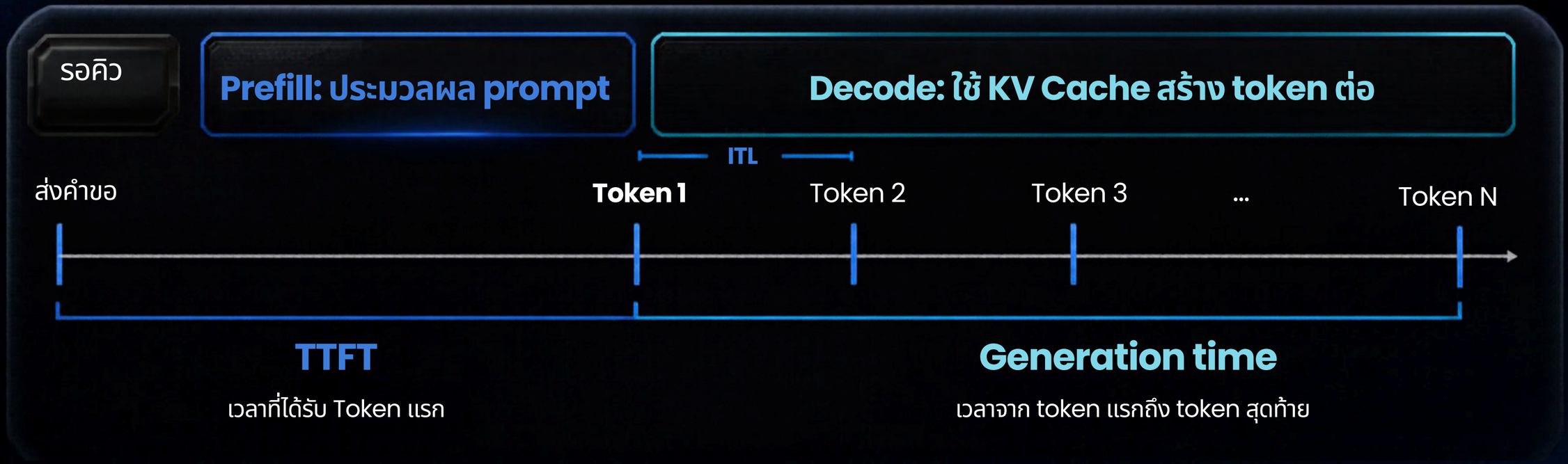
	Two-Shot	One-Shot
Traffic Size	**2(n-1) [WINNER]**	**n^2-n**
Sync Barriers	2 ครั้ง	**1 ครั้ง [WINNER]**
Latency	**15.5 µs**	**7.4 µs [WINNER]**
Use Case	Large Tensors	Tiny Scalars (Batch เล็ก)

Latency ลดลง 52 %



DeepSeek R1

PD Disaggregation



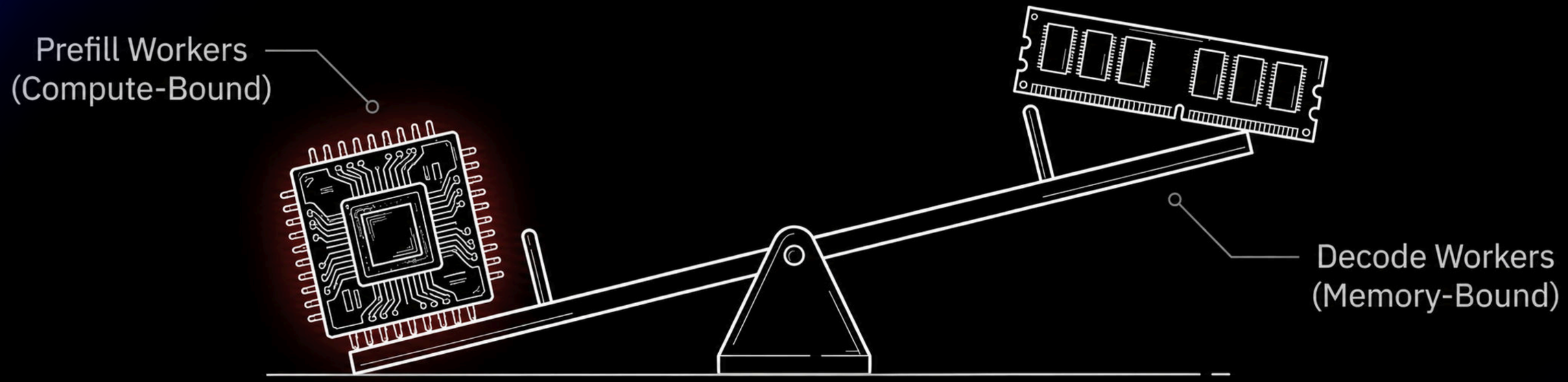
TTFT รวมเวลารอคิวและความหน่วงอื่น
ไม่ใช่เวลา Prefill อย่างเดียว

TPOT = เวลาเฉลี่ยต่อ token หลังตัวแรก
ตรวจนิยามของเครื่องมือที่ใช้วัด

LLM Inference ทำงานอย่างไร?



Bottleneck : P/D Memory Imbalance



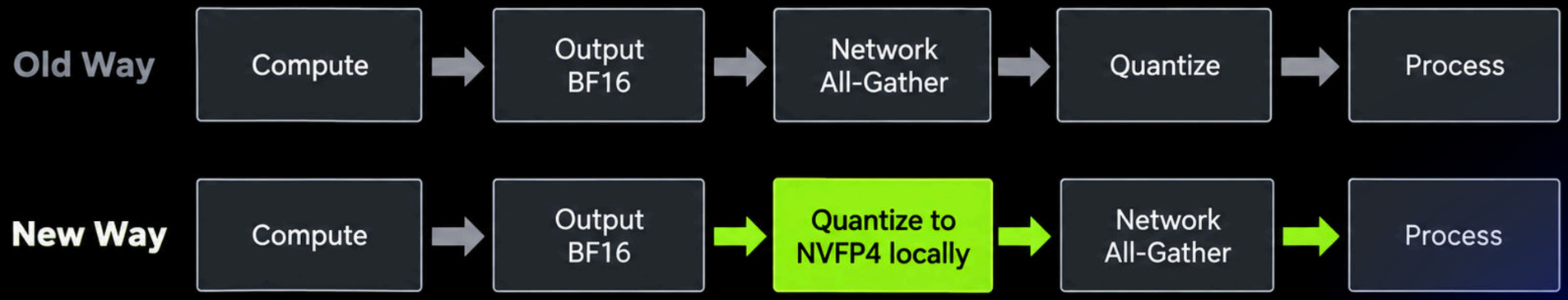
Prefill ใช้พลังคำนวณมหาศาล แต่เดิมใช้
Expert Parallelism (EP) ถึงระดับ EP4
จึงทำให้เกิด Latency สูงถึง 4,900

การโหลด Weights ของโมเดลใหญ่
ทำให้เกิดเวลาหยุดรอ

ลด Bottleneck ด้วย NVFP4 Quantize ก่อน All-Gather

สถาปัตยกรรมของ DeepSeek R1 ใช้ Data Parallel ร่วมกับ Expert Parallel ทำให้ต้องมีการทำ All-Gather ข้อมูล Tokens ก่อนสงเข้าชั้น MoE

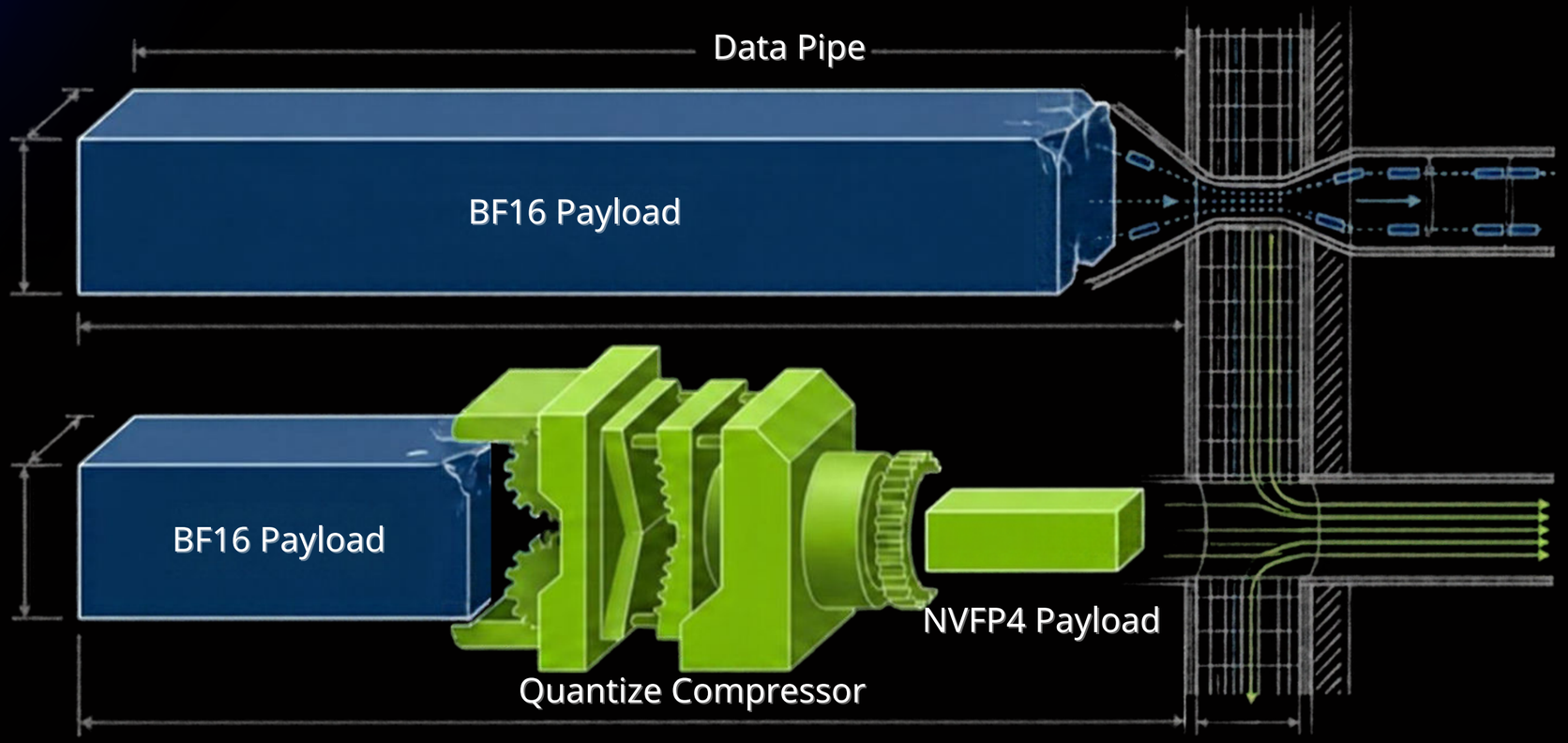
จุดเปลี่ยนสำคัญ: เนื่องจากระบบคำนวณรองรับความละเอียดแบบ NVFP4 ได้ เราจึงแปลง Hidden States จาก BF16 เป็น **NVFP4 'ก่อน'** ส่งข้ามเครือข่าย



ย้ายขั้นตอน Quantize มาไว้ก่อน Network เพื่อส่งข้อมูลที่เล็กลง

Traffic Network ลง 4 เท่า ประหยัด Bandwidth

NVFP4 Quantization ย่อขนาดไฟล์ให้เล็กลง 4 เท่าก่อนกดส่ง ช่วยลดปริมาณข้อมูล (Traffic Size)
ที่ต้องวิ่งข้าม GPU และ Node ลงถึง 4 เท่า ไม่โดนแย่ง Bandwidth งานส่งข้อมูลเลยเสร็จไว



ลดจำนวนเครื่องที่ต้องคุยกันลงจาก 4 เครื่องเหลือ 2 เครื่อง สายส่งก็โล่งขึ้น และประหยัด Bandwidth ได้ทั้งระบบ

V2 Weight Offloader & NVFP4

GB200 NVLink

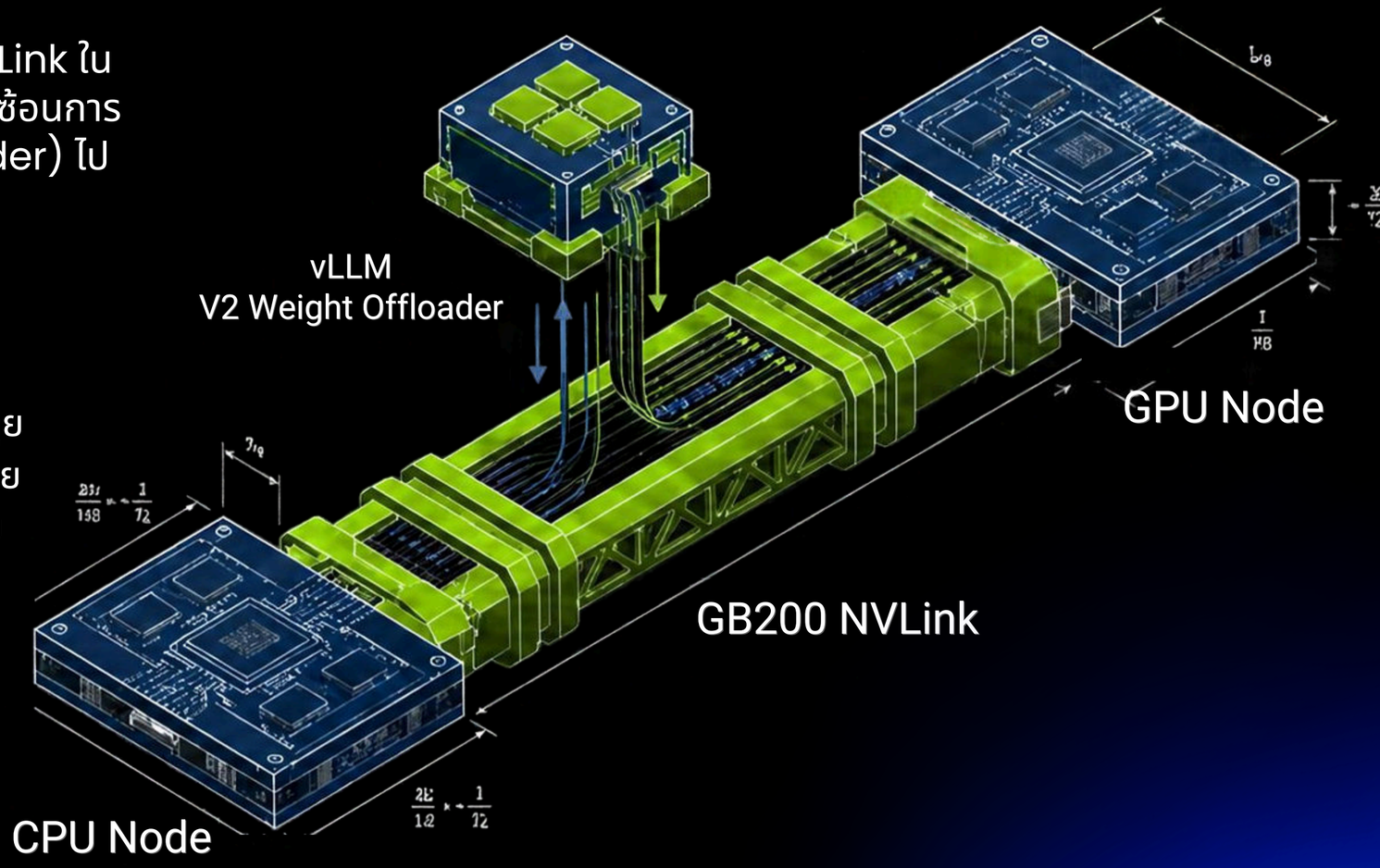
ใช้ Bandwidth ความเร็วสูงของทางด่วนไฮสปีด NVLink ในการรับส่งข้อมูลระหว่าง CPU และ GPU ทำให้สามารถซ้อนการโหลด Weight เลเยอร์ถัดไป (V2 Weight Offloader) ไปพร้อมๆ กับการคำนวณได้อย่างมีประสิทธิภาพ

ลดขนาด Expert Parallel (EP)

เพื่อประหยัดหน่วยความจำ โดยลดจำนวนเครื่องที่ต้องคุยกันลงจาก 4 เครื่องเหลือ 2 เครื่อง พอคุยข้ามเครื่องน้อยลง สายส่งก็โล่งขึ้น และประหยัด Bandwidth ทั้งระบบ

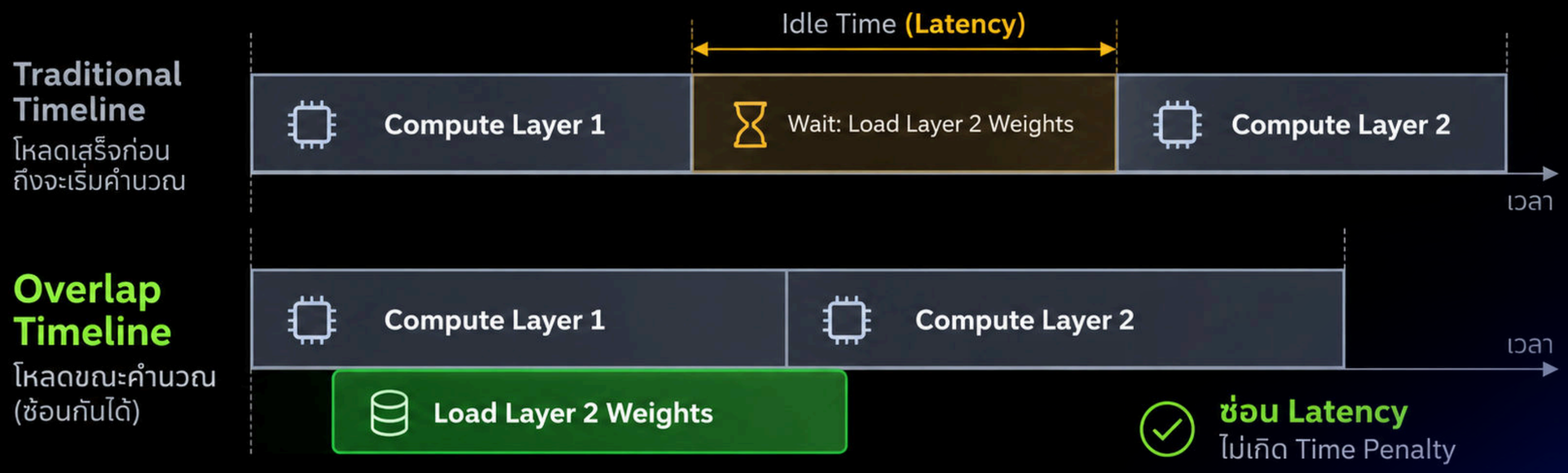
ลด Latency เกิน 50%

Prefill เป็น Compute-bound
การลด EP ช่วยลด Overhead การสื่อสารลงอย่างมาก Latency ลดลงจาก 4,900 µs เหลือเพียง 2,000 µs (ลดลงมากกว่า 50%)



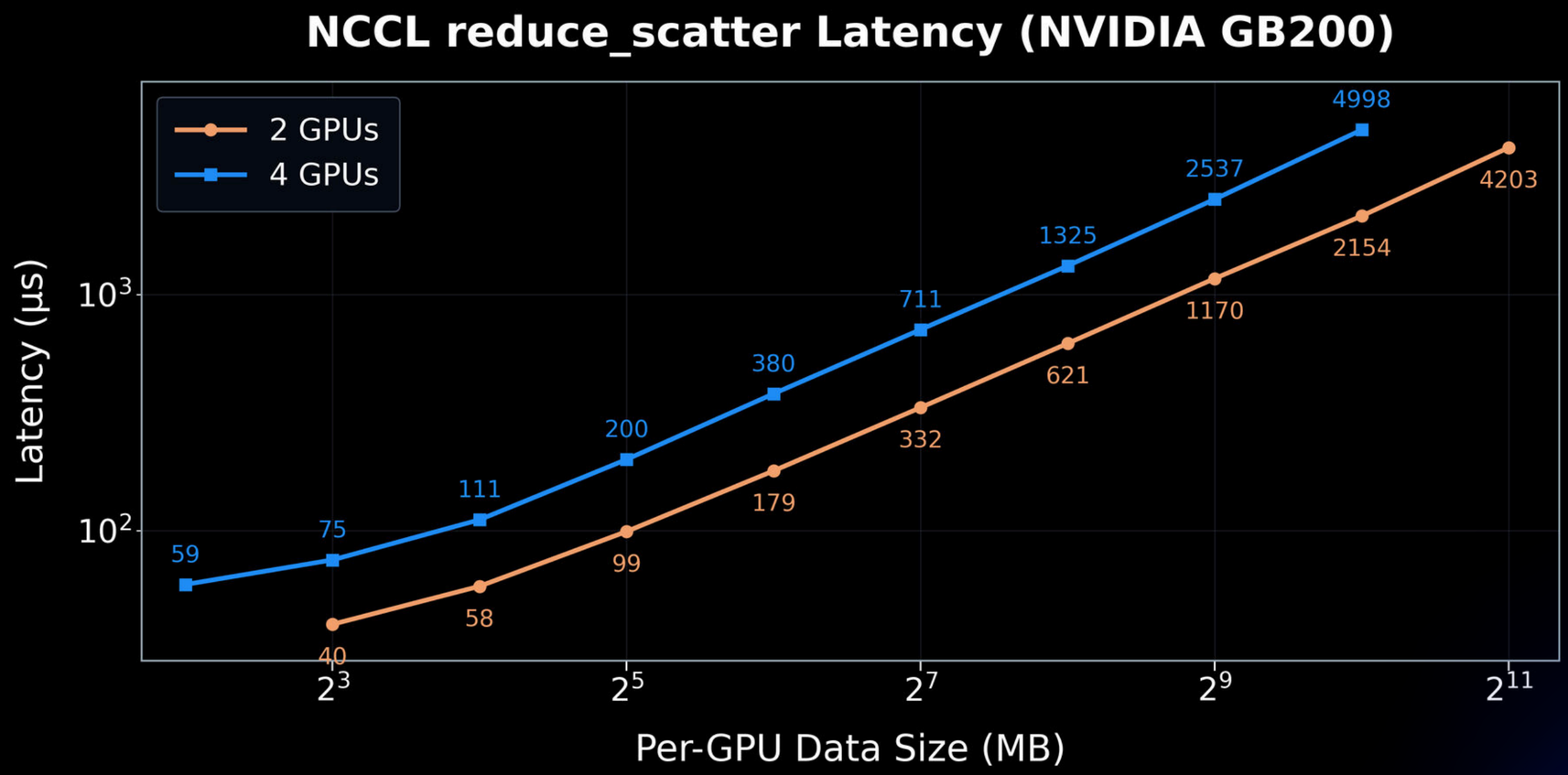
โหลด Weight แบบ overlap

การโหลดน้ำหนัก (Weights) ของเลเยอร์ถัดไป ซ้อนกับ (Overlap) กับการประมวลผล (Compute) ของเลเยอร์ปัจจุบันโดยสมบูรณ์ **ประหยัด VRAM** มหาศาลโดยไม่เกิด Time Penalty ช่วยให้เราสามารถขยายขนาด Expert Parallel (EP) ลงได้ในขั้นตอนต่อไป



NCCL Reduce Scatter Latency

DeepSeek R1 — V2 Weight Offloader

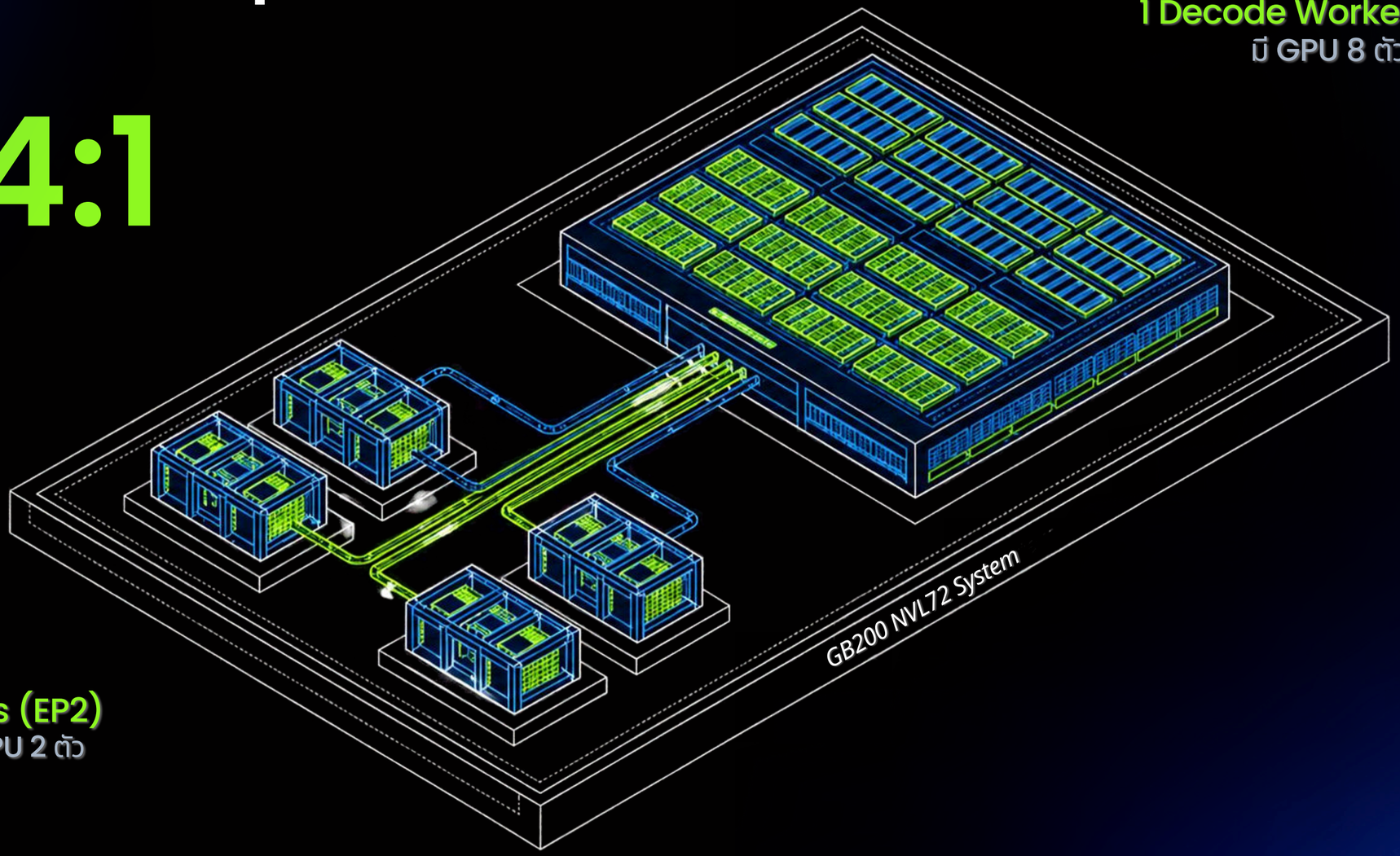


ผลวัด NVIDIA GB200 · เปรียบเทียบ 2 GPUs กับ 4 GPUs ตาม Per-GPU Data Size

Prefill–Decode Optimization

4:1

1 Decode Worker (EP8)
มี GPU 8 ตัว

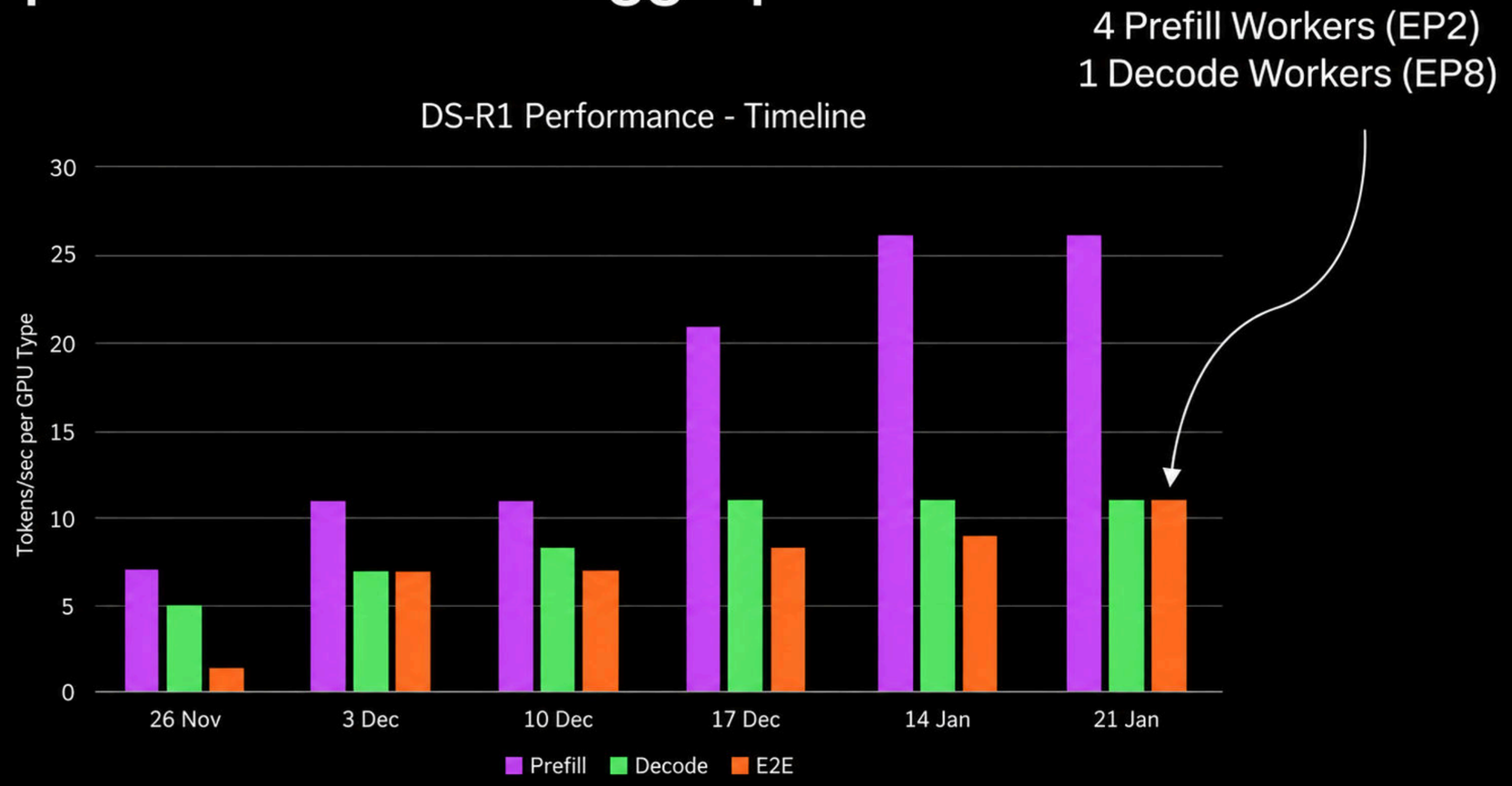


4 Prefill Workers (EP2)
แต่ละ worker มี GPU 2 ตัว

การจัดสรรทรัพยากรที่สมบูรณ์แบบที่สุดบนระบบ GB200 NVL72

Prefill–Decode Optimization

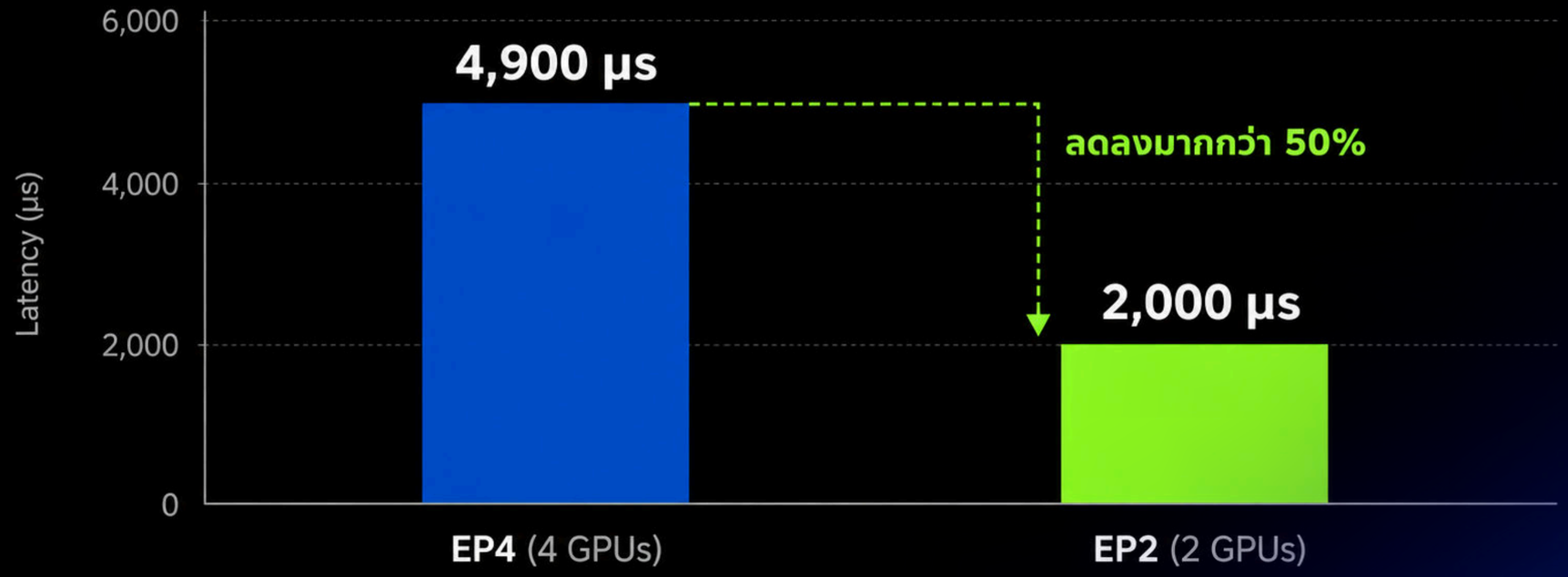
DeepSeek R1 – PD Disagg Opt



DeepSeek R1 เพิ่ม Throughput ด้วยการ Optimize Prefill และ Decode แยกกัน
แล้วปรับ Worker Allocation กับ Expert Parallelism ให้เหมาะกับแต่ละ Phase

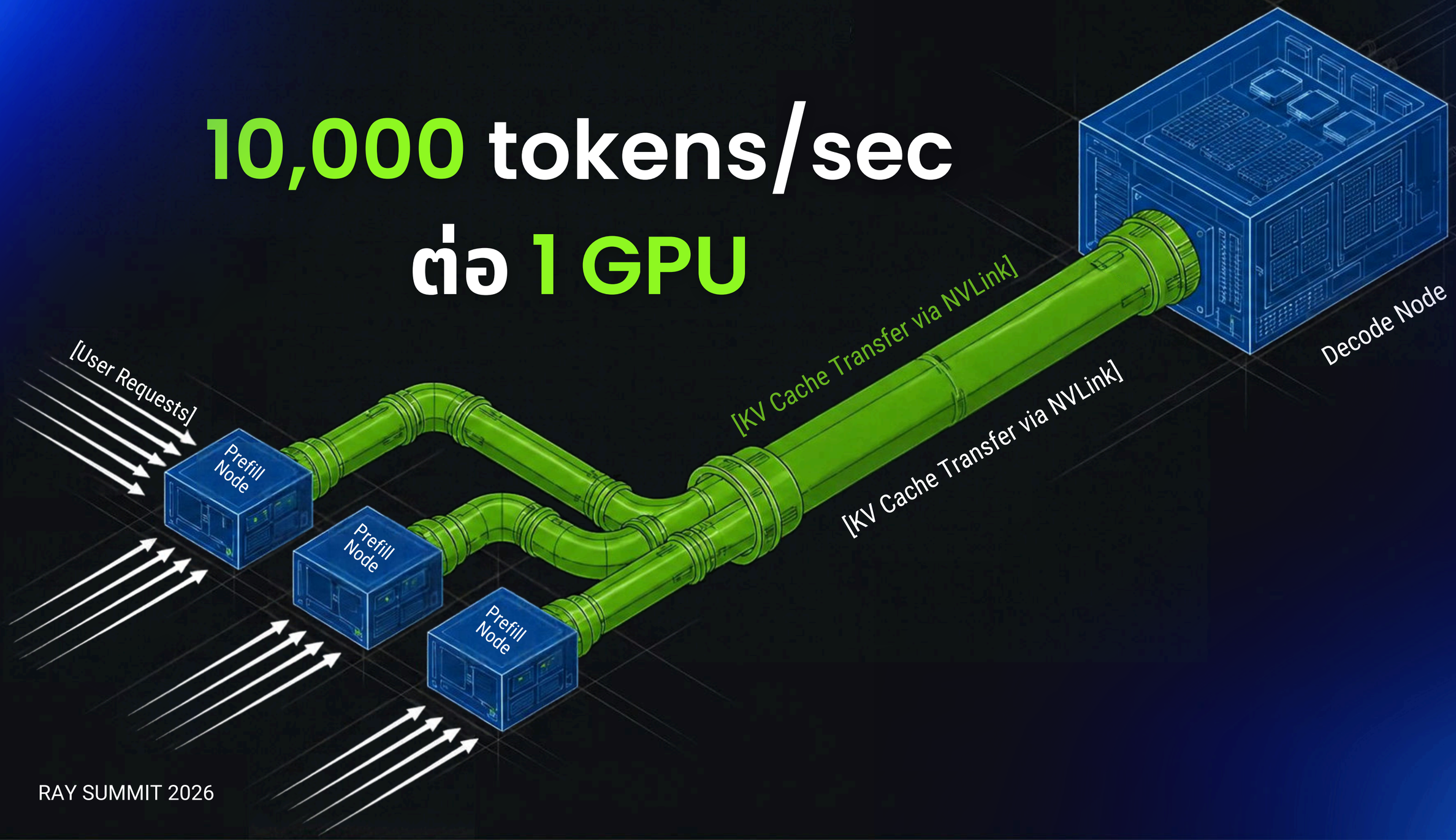
-50 % Latency

การลดการสื่อสารที่ไม่จำเป็นในช่วง Prefill ช่วยให้ Latency ลดลงอย่างชัดเจน
และลดคอขวดของระบบได้มาก



Overhead Latency ลดมากกว่า50 %

10,000 tokens/sec ต่อ 1 GPU



All-to-All Backend Options

--all2all-backend

allgather_reducescatter
deepep_high_throughput
deepep_low_latency
deepep_v2
flashinfer_all2allv
flashinfer_nvlink_one_sided
flashinfer_nvlink_two_sided
mori_high_throughput
mori_low_latency, naïve
nixl_ep
pplx

--all2all-backend

allgather_reducescatter

deepep_high_throughput

deepep_low_latency

flashinfer_all2all

Kimi MiniMax DeepSeek

Take Away

- We can split All Reduce into Reduce Scatter + All Gather, and hide their latency separately
- We can fuse All Gather with GEMM
- Use one-shot all reduce kernel when traffic is small
- Better use smaller parallel size in Prefill workers; consider weight offloading

NVIDIA Dynamo

Distributed Serving

RAY SUMMIT 2026

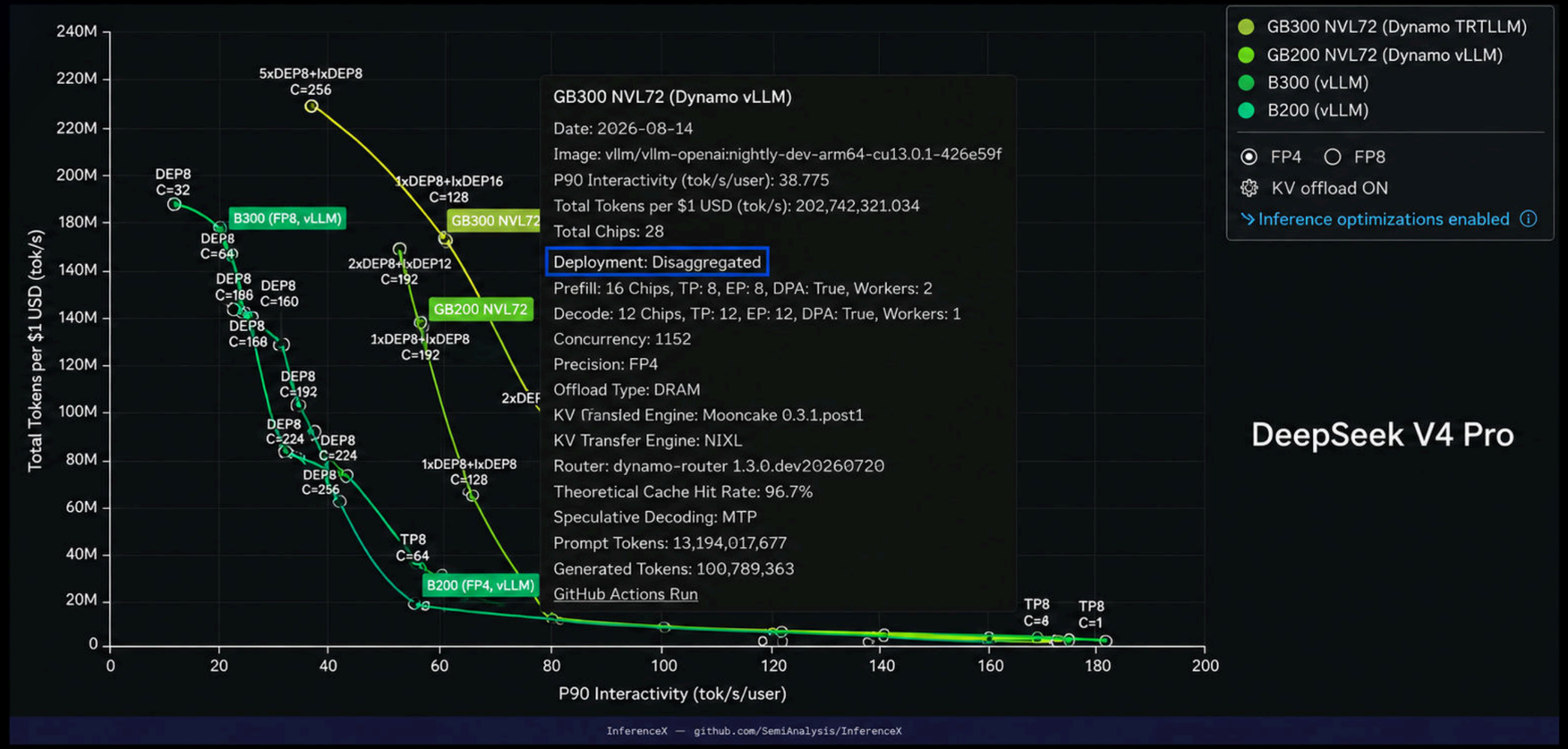
NVIDIA Dynamo

***OPENSOURCE Framework* เป็นตัว *LLM serving* เพื่อทำ *Scaling* ได้ง่ายขึ้น**

- Dynamo ลดเวลาที่ต้องรอสำหรับการ Responseแรกของ AI ที่เราจะเรียกว่า "time-to-first-token" หรือ TTFT
- สามารถรองรับการสนทนา AI หลายรายการในเวลาเดียวกันโดยไม่ Delay หรือมีน้อย
- จัดการหน่วยความจำได้ดี มีการย้ายข้อมูลที่ดีมากระหว่าง CPU, GPU, Storage เพื่อให้โมเดลไม่ติดขัดที่ต้องรอทรัพยากร
- ทำงานและเชื่อมกับ Tools ได้หลากหลาย (supports OPENAI Compatible Frontend แล้วก็ Backends: TRT-LLM, vLLM, SGLang or others)
- มีการ optimize KV cache เมื่อเราคุยกับ AI ซ้ำๆ Dynamo จะสามารถคำนวณและส่งข้อความ ใน Server ที่มีข้อทูลนี้อยู่แล้ว ทำให้การ Request เร็วขึ้น 3 เท่า
- Dynamo สามารถเก็บข้อมูลไว้ในหน่วยความจำหลายประเภท (จากหน่วยความจำ GPU ที่เร็วไปจนถึงพื้นที่เก็บข้อมูลที่ช้ากว่าอย่าง Storage) ทำให้สามารถรองรับการสนทนาที่ยาวขึ้นโดยไม่ ติด bottleneck

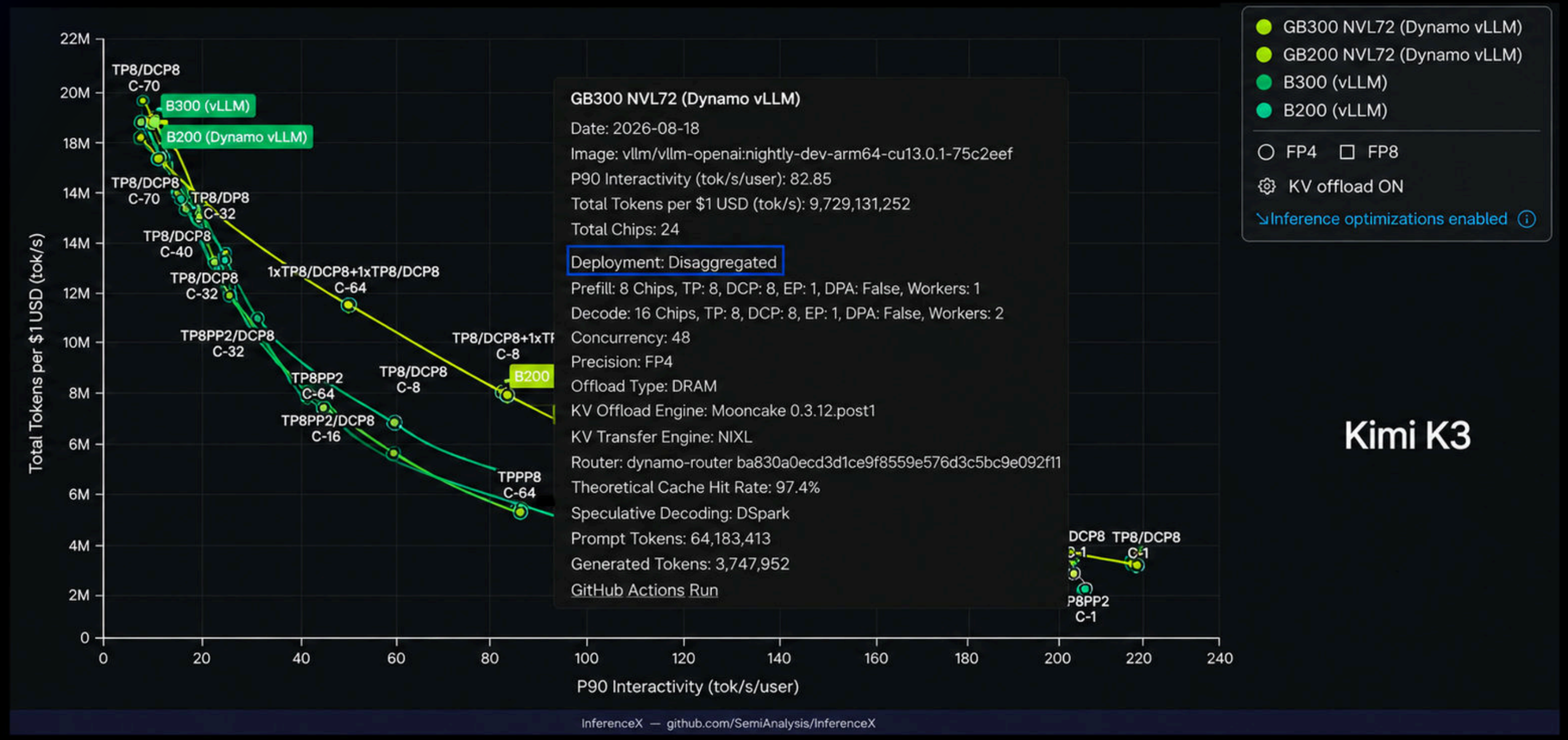


แสดงผลการประมวลผล Inference ของระบบ NVL72



การใช้ PD-Disagg (Prefill-Decode Disaggregation): สูตรระบบ (Recipe) ที่ให้ประสิทธิภาพสูงสุดบน NVL72 สำหรับโมเดลเหล่านี้ แทบทั้งหมดเลือกใช้การแยกส่วนประมวลผลระหว่าง Prefill และ Decode

Intro to Dynamo – Push InferenceX Result



ตัวอย่างเช่น Dynamo-vLLM แสดงให้เห็นว่าการใช้ Dynamo Suite เข้ามาจัดการระบบ
ร่วมกับ vLLM ช่วยให้การจัดคิวและดึงประสิทธิภาพของ Kimi K3 บน NVL72 ทำได้สูงสุด

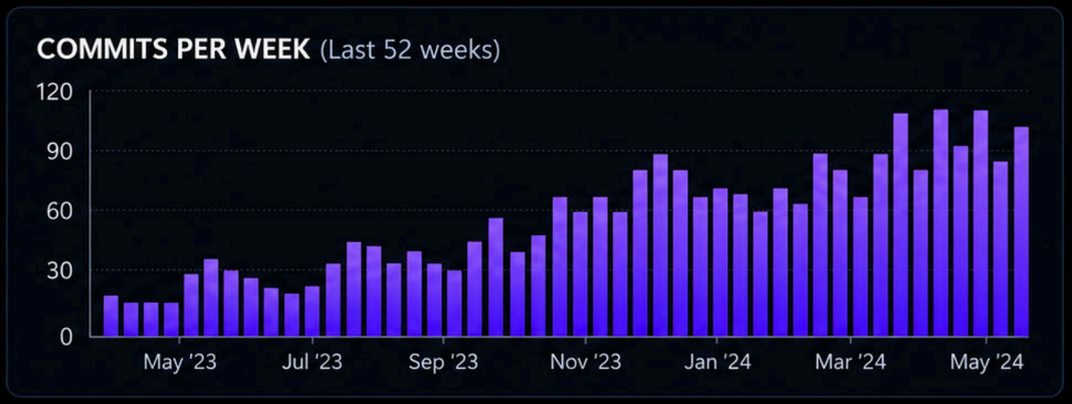
Open-Source Activity

Intro to Dynamo



2,428
COMMITS
In the last 52 weeks

8,902
STARS
and counting! 🚀



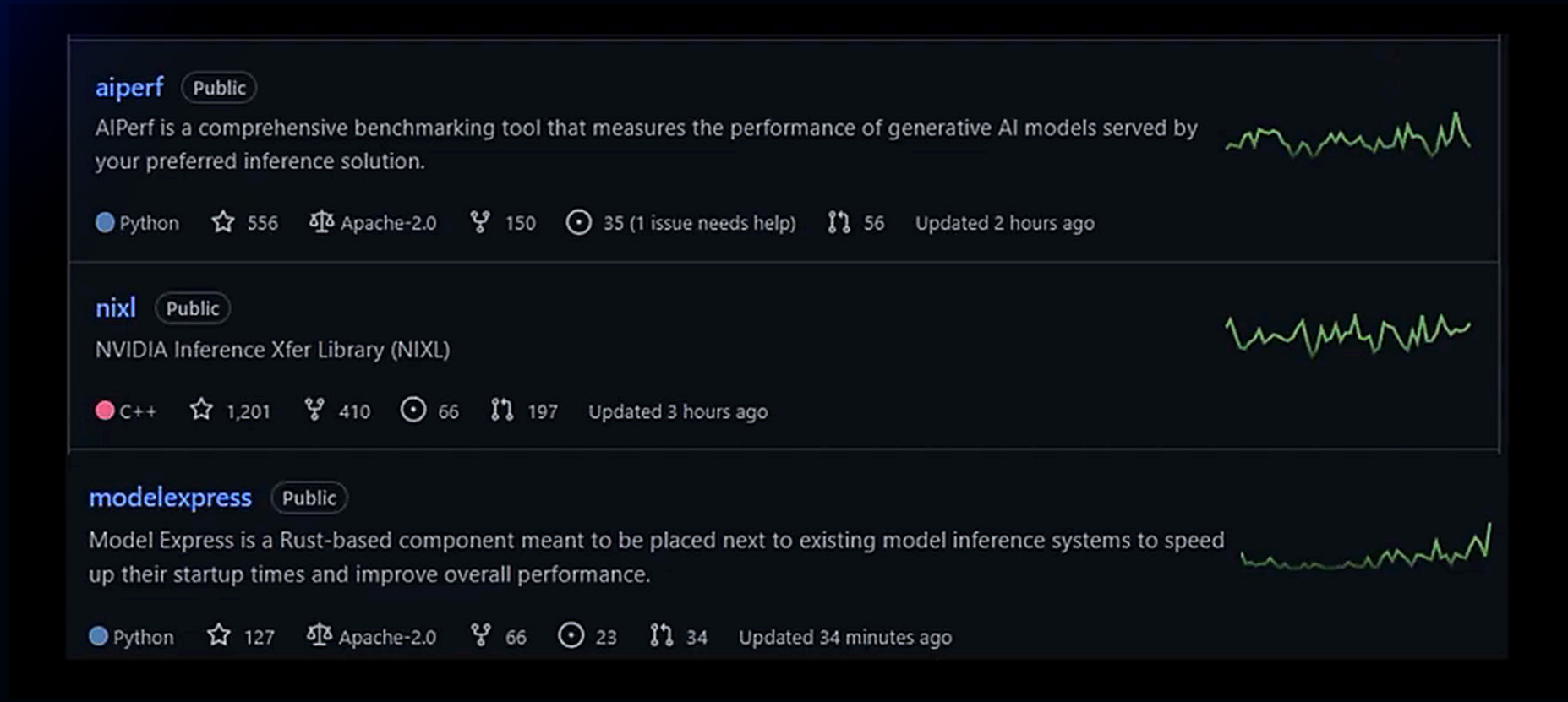
github.com/ai-dynamo/dynamo

Serving Ecosystem



ใน Ecosystem นี้ เราสามารถใช้ Dynamo ร่วมกับ Engine อย่าง vLLM, SGLang หรือ TensorRT-LLM และใช้ Kubernetes ช่วยจัดการการ Deploy Inference Engine ทำหน้าที่รันโมเดล ส่วน Dynamo ช่วยประสานงานและจัดการ Requests ระหว่าง Workers

Supporting Repositories



AI Perf ใช้สำหรับวัด Performance, NIXL ช่วยเรื่องการรับส่งข้อมูลระหว่าง Workers โดยเฉพาะ KV Cache และ Model Express ช่วยลดเวลาเตรียมโมเดลให้พร้อมทำงาน
จะเห็นว่าการทำ Distributed Serving ที่มีประสิทธิภาพต้องอาศัยหลายองค์ประกอบร่วมกัน ไม่ใช่แค่มี GPU ที่แรงหรือ Inference Engine ที่เร็วอย่างเดียว

NVIDIA Dynamo

ระบบแบบดั้งเดิมตอนนี้เราจะประมวลผล request AI ทั้งหมดในคราวเดียว ซึ่งไม่มีประสิทธิภาพ

Dynamo มีการแบ่ง workers อย่างมีประสิทธิภาพ
โดยแบ่ง workers ออกเป็น 2 ส่วน

- **Prefill** การประมวลผลข้อความเริ่มต้น (ใช้การคำนวณมาก)
- **Decode** การสร้างโทเค็น/คำใหม่แต่ละครั้ง (ต้องรวดเร็ว)



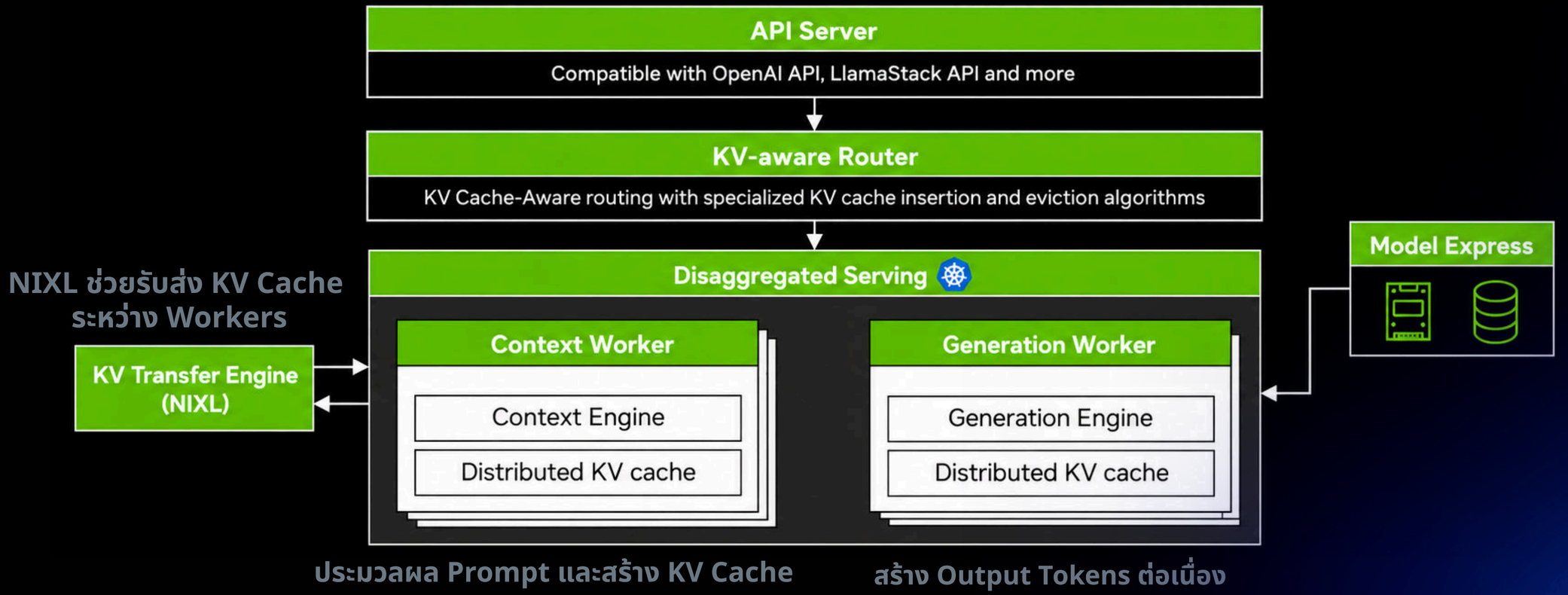
Distributed Serving

Optimize Request Routing + KV Cache

1 API Server รับ Request

2 KV Router เลือก Worker โดยพิจารณา Cache และ Load

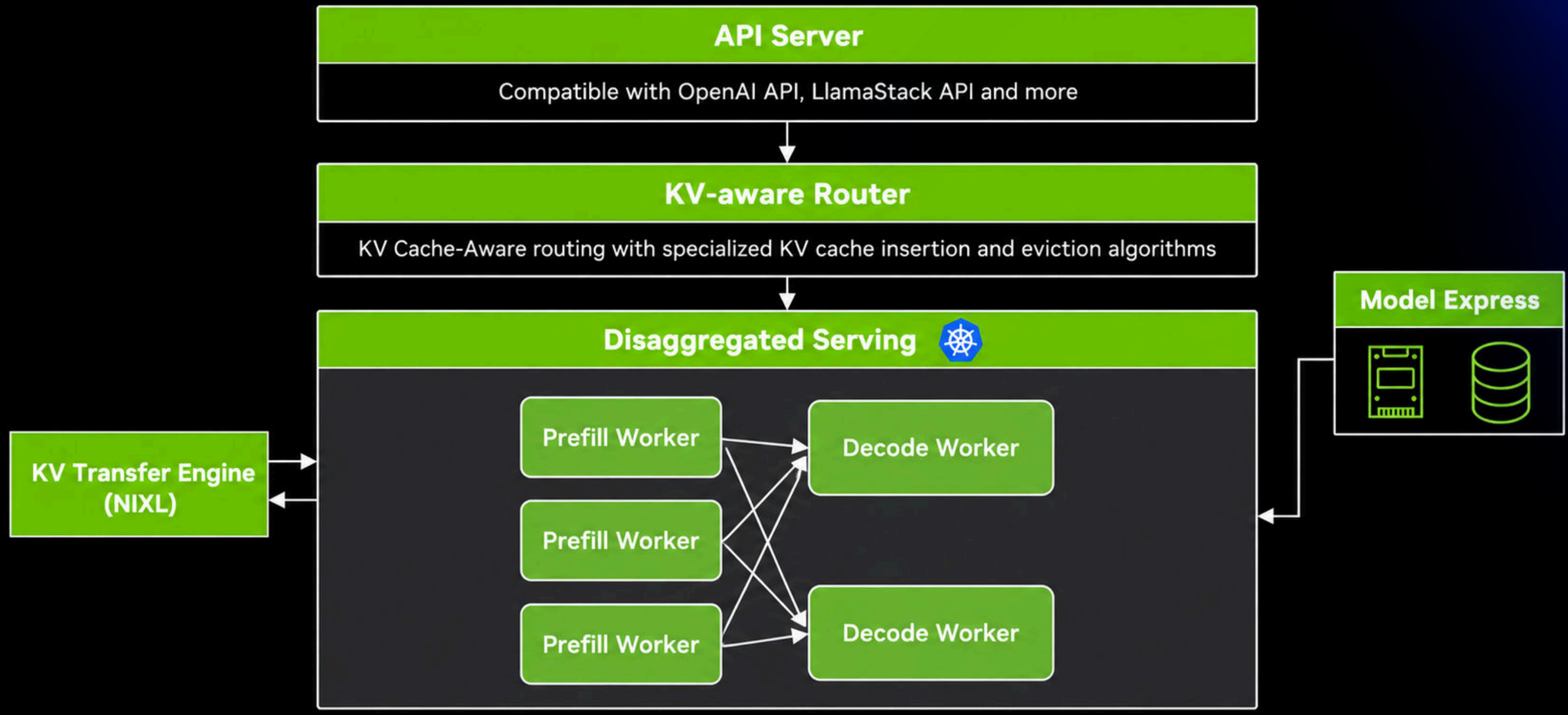
3 แยก Workers + ส่ง KV Cache



Architecture

Prefill / Decode แยกกัน

Scale worker แต่ละกลุ่มตาม workload

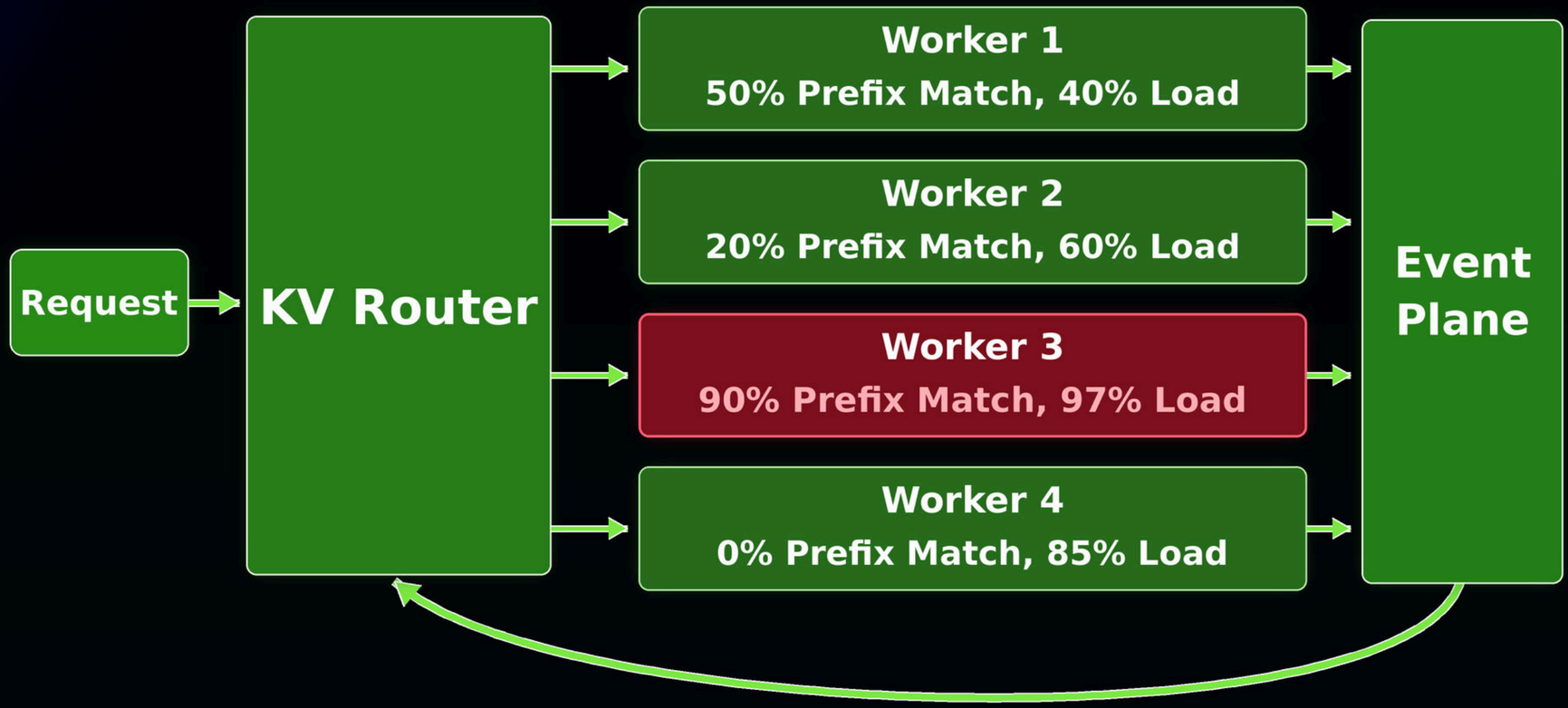


PREFILL
Prompt → KV Cache

KV TRANSFER
KV Cache Transfer

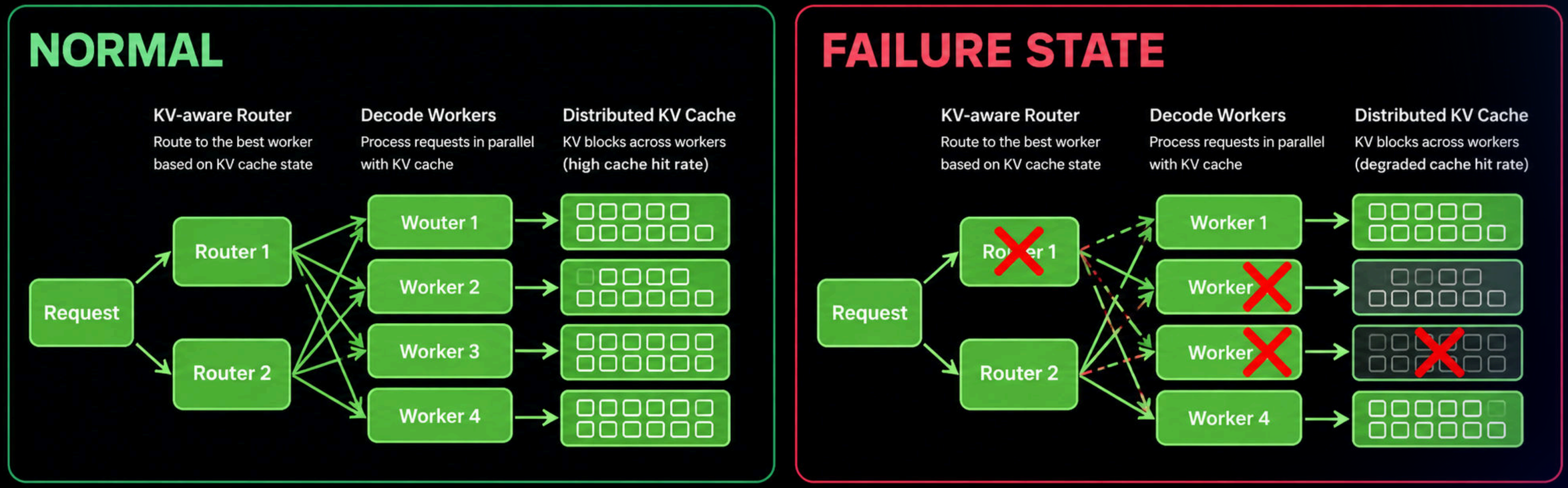
DECODE
Generate tokens

Dynamo เลือก Worker อย่างไร?



Bottleneck: Request & State Chaos

Fault Tolerance



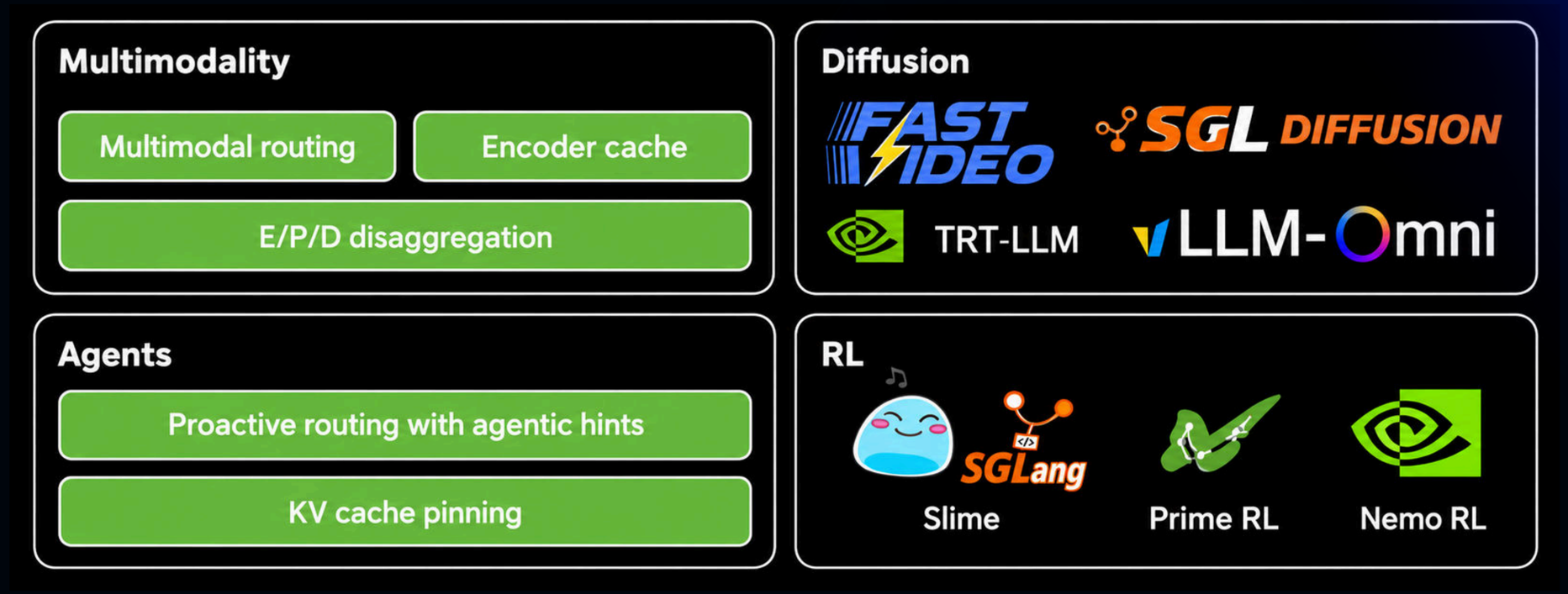
High availability
ระบบยังตอบได้

Cancellation
ยกเลิกงานที่ไม่ต้องทำ

Migration
ย้าย Request เมื่อมีปัญหา

Dynamo 2.0

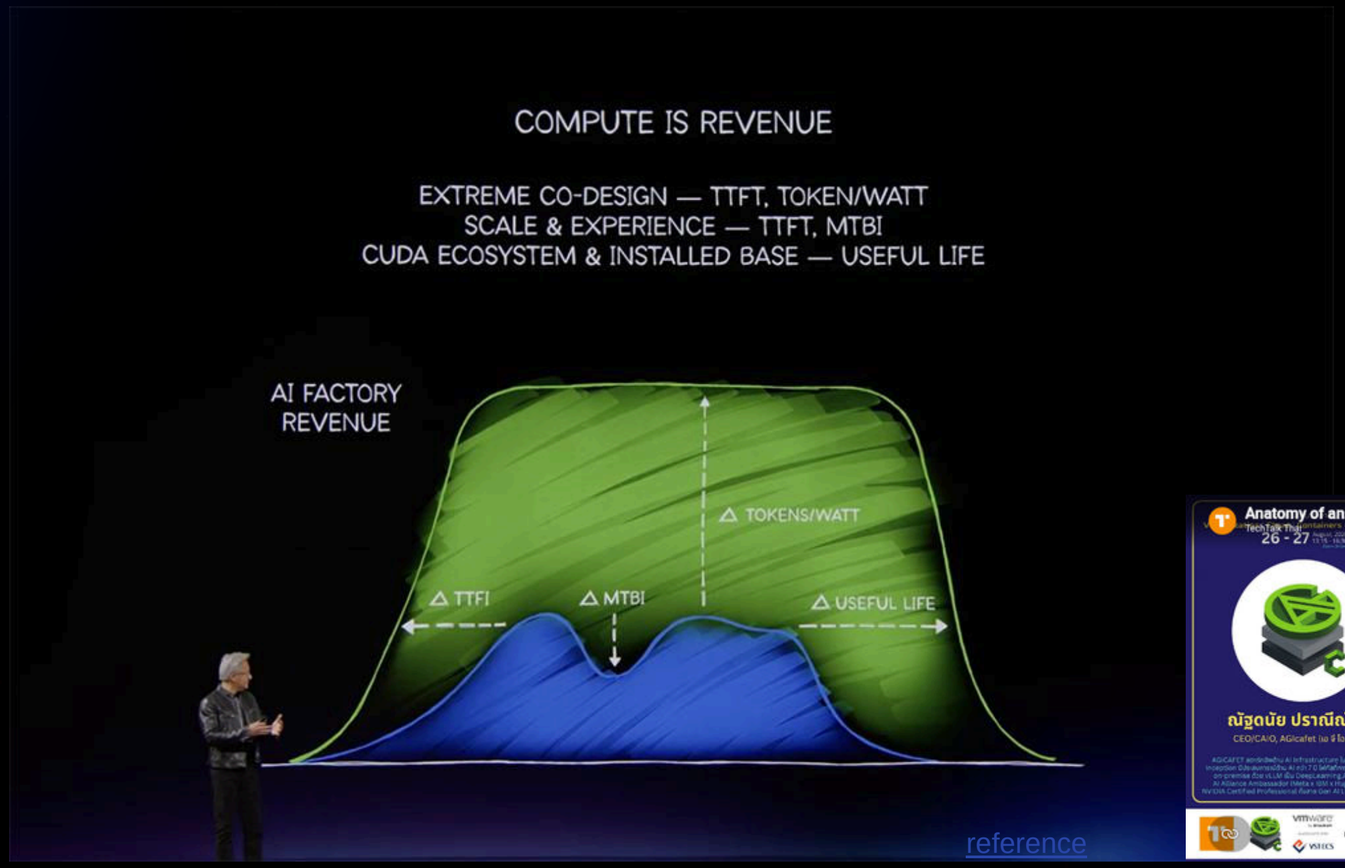
Original Roadmap



สรุปภาพรวมสถาปัตยกรรม

โมเดล / ระบบ (Model/System)	เทคนิคหลัก (Core Technique)	ฮาร์ดแวร์ / สเกล (Hardware/Scale)	ผลลัพธ์ (Impact)
MiniMax M2.5	Full Heads RMSNorm (Reduce Square Sum Scalar) + One-Shot All-Reduce	ระดับ Tensor Parallel	ลด Latency ลง 52%
DeepSeek R1	NVFPA Quantize ก่อน All-Gather + V2 Weight Offloader (ลด Prefill EP เหลือ EP2)	GB200 NVL72	ลด Traffic 4x, ท:ล >10k tokens/sec/GPU
Kimi K3	Latent MoE Kernel Fusion (Fuse All-Gather เข้า Up-Proj, Fuse Reduce-Scatter)	GB300 (TP8)	ความเร็วเพิ่มจาก 72 -> 111 tokens/sec
Dynamo Suite	KV Cache-Aware Router + Prefill/Decode Disaggregation + Fault Tolerance	ระดับ Cluster	Zero Wasted Compute / High Availability

Full-Stack Optimization เมื่อลด Latency ได้มากขึ้น
ได้ Token ที่สร้างขึ้น สร้างผลกำไรได้ คุ้มค่าทุกพลังงานต่อวัตต์





Jensen Huang believes that in the era of AI factories, every token generated can bring profit.
Performance per watt, reliability, and system lifespan are the core financial levers.