

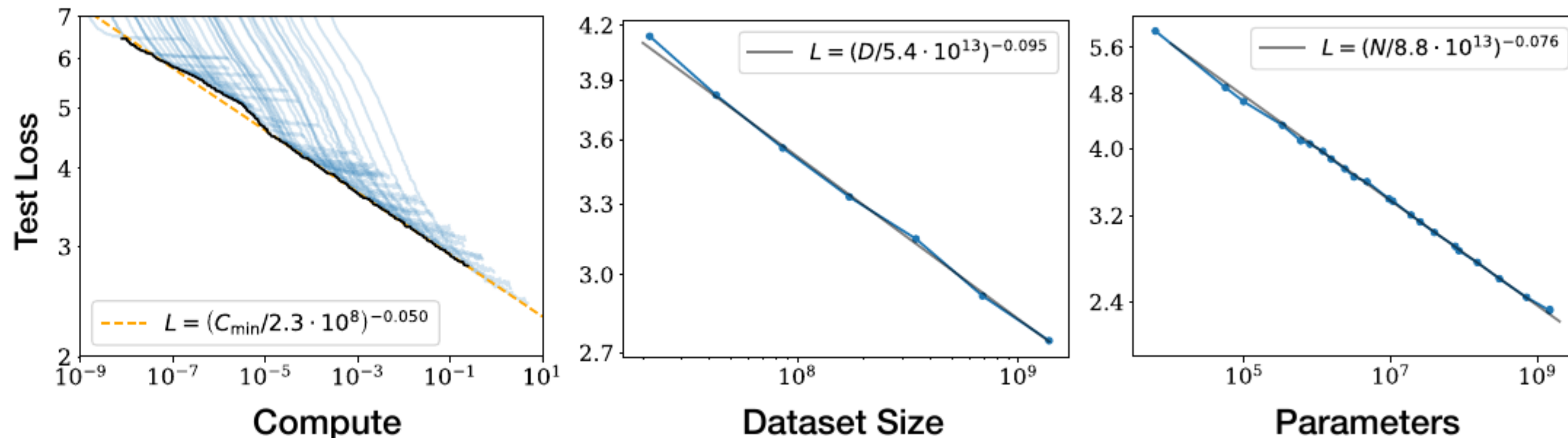
AI Infrastructure

Build scalable ML systems through end-to-end optimization across algorithms, system software, and hardware

Li Shang
lishang@slai.edu.cn

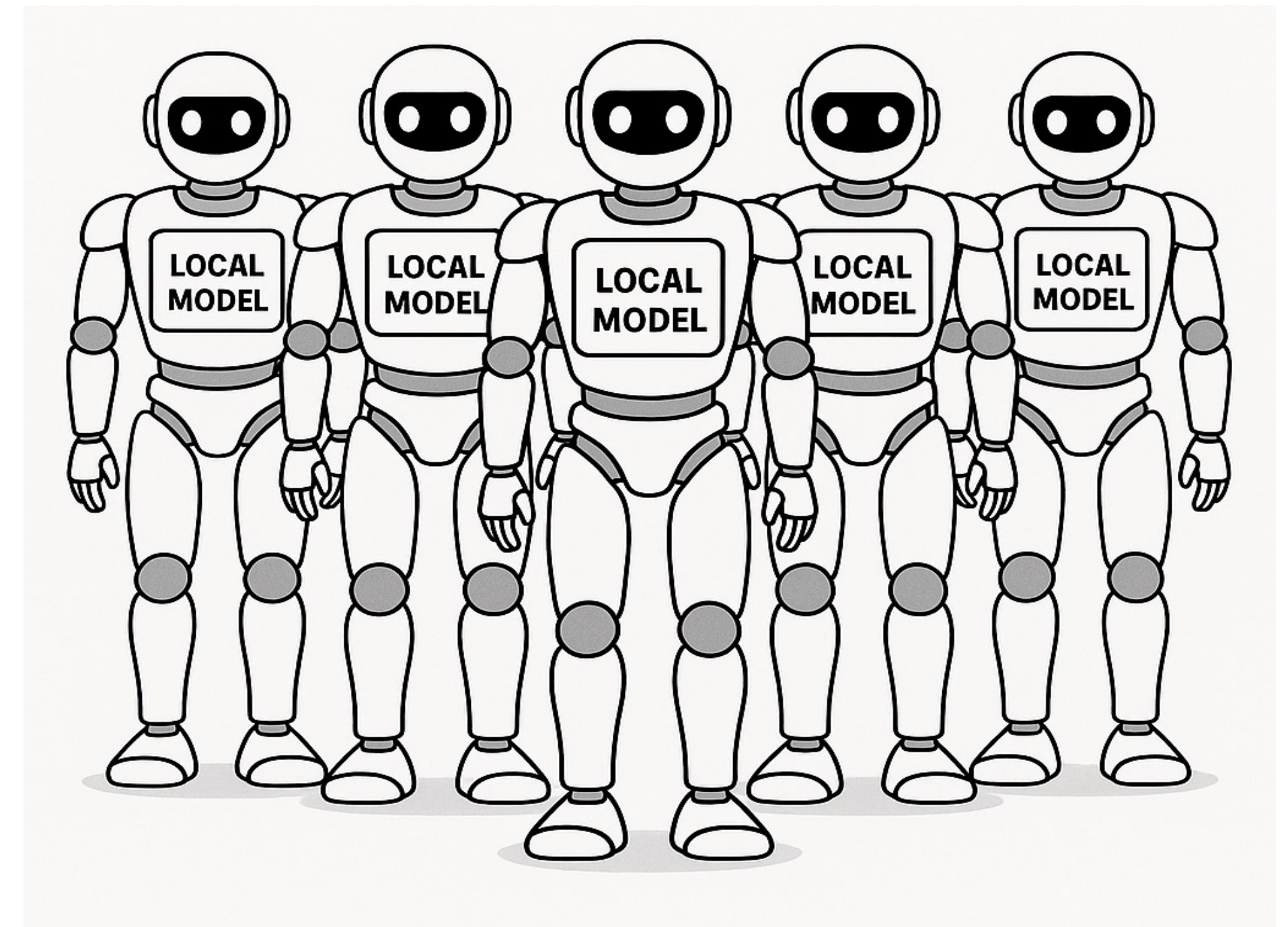
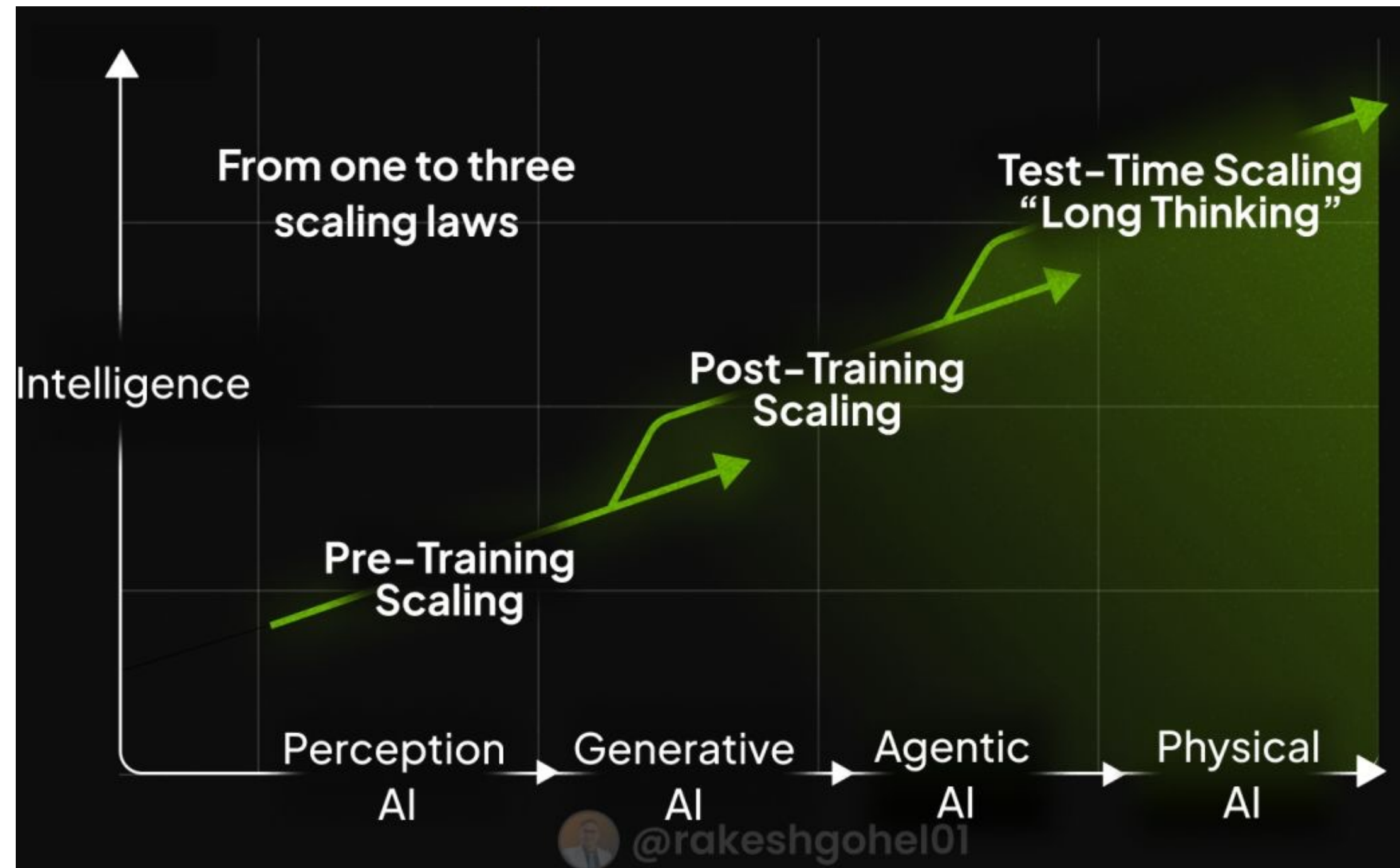
Scalability: The first principle of LLMs

- Large language models begin with a deliberately weak prior: the meaning of a word is determined by its context.
- The opportunity is empirical. The infrastructure question is how far we can extend it before cost or information loss becomes the bottleneck.



Kaplan et al. (2020), Fig. 1 — language-model test loss as model size, data, and training compute vary.

Mission Statement



Scalable intelligence



AI Infra



Wearables



Humanoid Robots

Machine Learning Systems



Cybercab

Acknowledgement

AI infrastructure is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blog posts, research talks, tutorial videos, and other materials shared by the research community.

Part of the course material was created by LLM itself.

More importantly, for each and every of us, LLM shall be heavily involved in daily learning.

Why is computer system design challenging?

–Tom Jerry

Lesson 0: Good system design starts with predicting the future workload.

–Tom Jerry

Lesson 1: Machine intelligence is about learning compact representations in high-dimensional space

–Tom Jerry

The blessing of high dimension

A wide latent space lets many simple features coexist with less interference.

01 Simple features

An individual feature, such as a concept, relation, syntactic role, or task constraint, may require only one direction or a small subspace.

The challenge is to represent many features together without excessive interference.

02 Width reduces interference

As d grows, generic normalized feature directions in a d -dim space become closer to orthogonal.

Typical cosine overlap $\propto \frac{1}{\sqrt{d}}$

Squared overlap $\propto \frac{1}{d}$

03 Combinations stay expressive

A language model can place many low-dimensional features in one shared space, keep them distinct, and combine them for each token and context.

The individual features remain simple while their combinations express rich meaning.

High-dimensional geometry gives low-dimensional features room to remain distinct and combine dynamically.

Lesson 2: Tensor operation is the primary workload.

–Tom Jerry

Why matrix multiplication fits GPUs

B

$$\begin{bmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,n-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n-1,0} & b_{n-1,1} & \cdots & b_{n-1,n-1} \end{bmatrix}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,n-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1,0} & a_{n-1,1} & \cdots & a_{n-1,n-1} \end{bmatrix} \begin{bmatrix} \\ \\ \\ \end{bmatrix}$$

A

C

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

1

Massive parallelism

The n^2 output elements are independent. GPU threads can compute many row-column dot products at the same time.

2

High arithmetic density

Dense $n \times n$ multiplication performs about $2n^3$ operations on $O(n^2)$ data. Tiling reuses values from A and B across many multiply-accumulate operations.

Independent outputs keep the GPU parallel. Data reuse keeps it computing.

Transformer training memory in one calculation

Example: d = 2,048 hidden units, N = 24 layers, B = 8 sequences, L = 1,024 tokens, h = 16 heads, V = 50,000 vocabulary entries

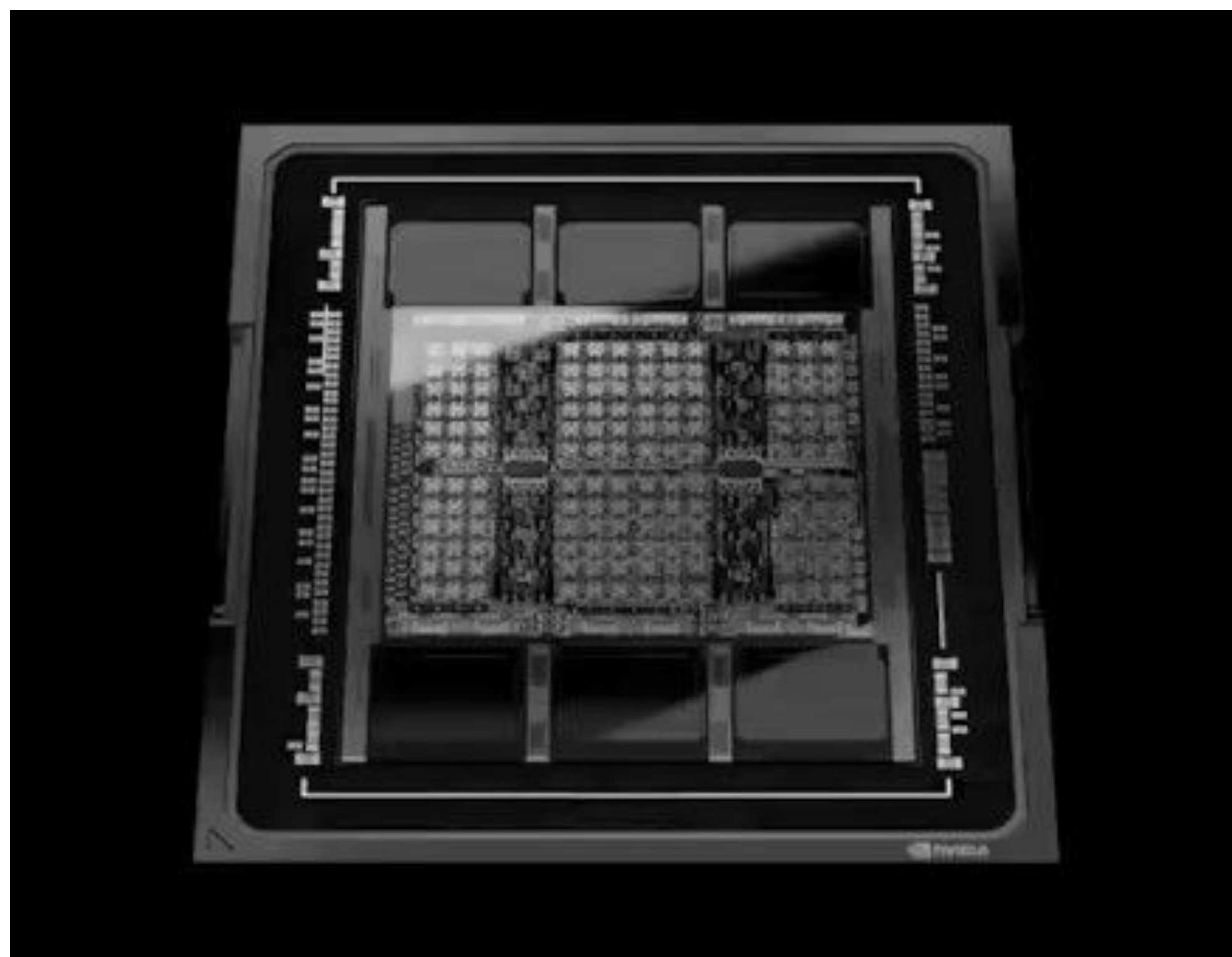
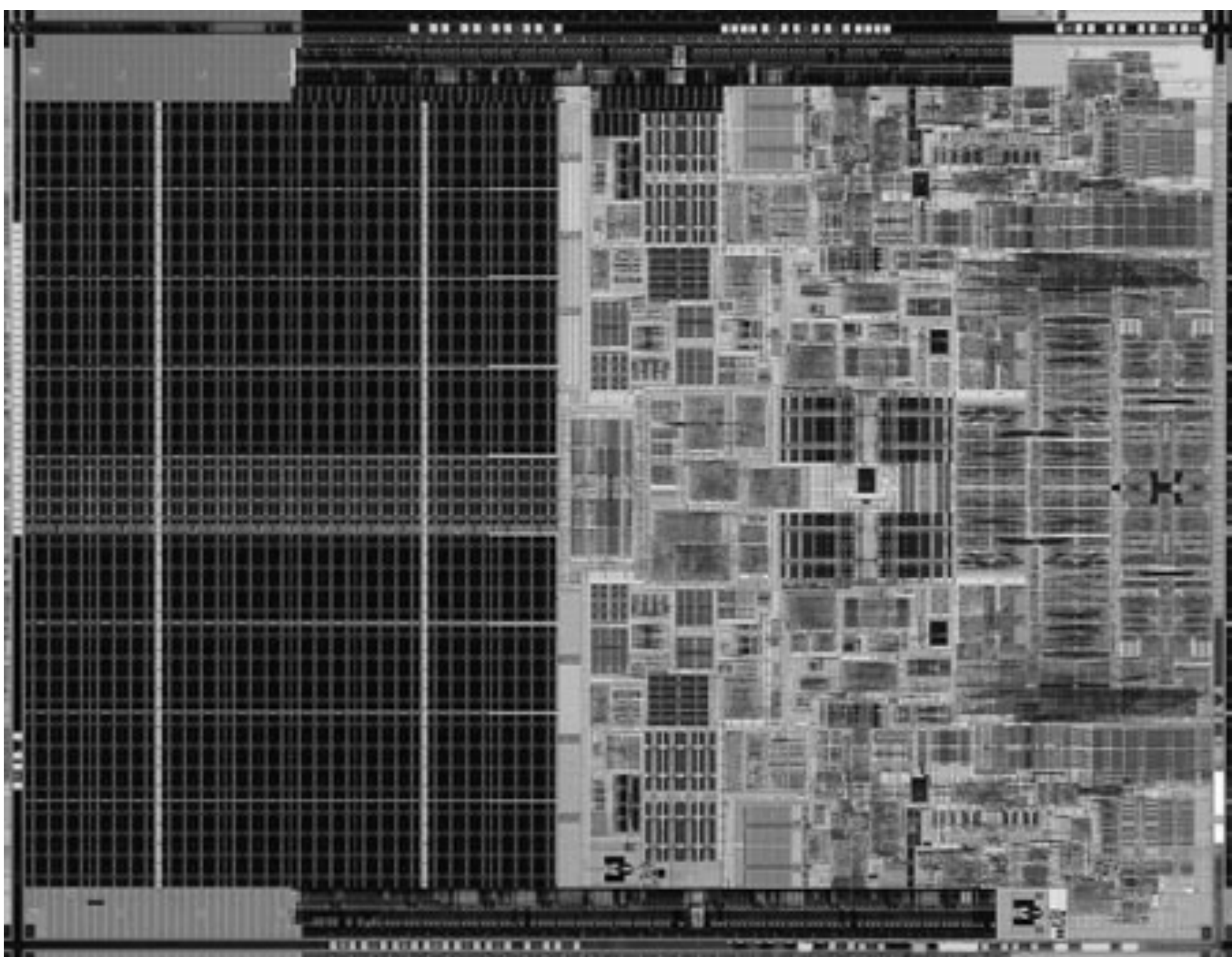
Stage	Equation	Computed data count	Memory
Parameters	$12Nd^2 + Vd$	1,310,359,552 parameters (1.31B)	FP16 weights: 2.44 GiB Full Adam state: 19.53 GiB
Forward activations	$9BLd + 2BhL^2$ per layer	10,066,329,600 FP16 values	18.75 GiB
Backward peak	$\sim 2 - 3\times$ activations + gradients + optimizer states	20.13B to 30.20B FP16 activation values + 1.31B gradients	57.03 to 75.78 GiB including full Adam state

Memory uses GiB = 2³⁰ bytes. Full Adam state assumes 16 bytes per parameter: FP16 weights and gradients, FP32 master weights, and two FP32 Adam moments. The backward range is a peak-memory estimate because frameworks free and recompute tensors differently.

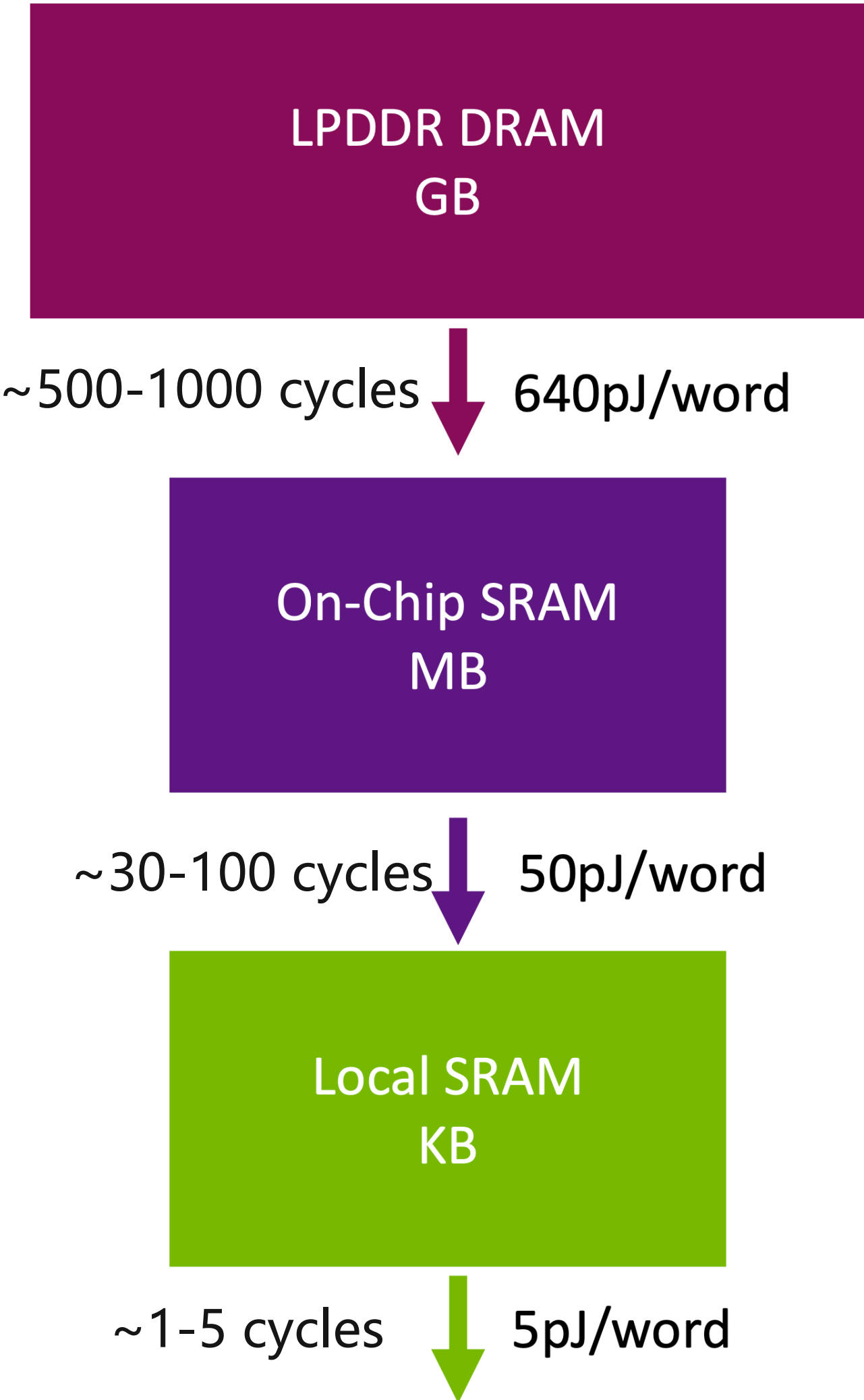
At this sequence length, saved activations already exceed the FP16 parameter weights by nearly 8x.

Lesson 3: Data processing is cheap, data storage is slow

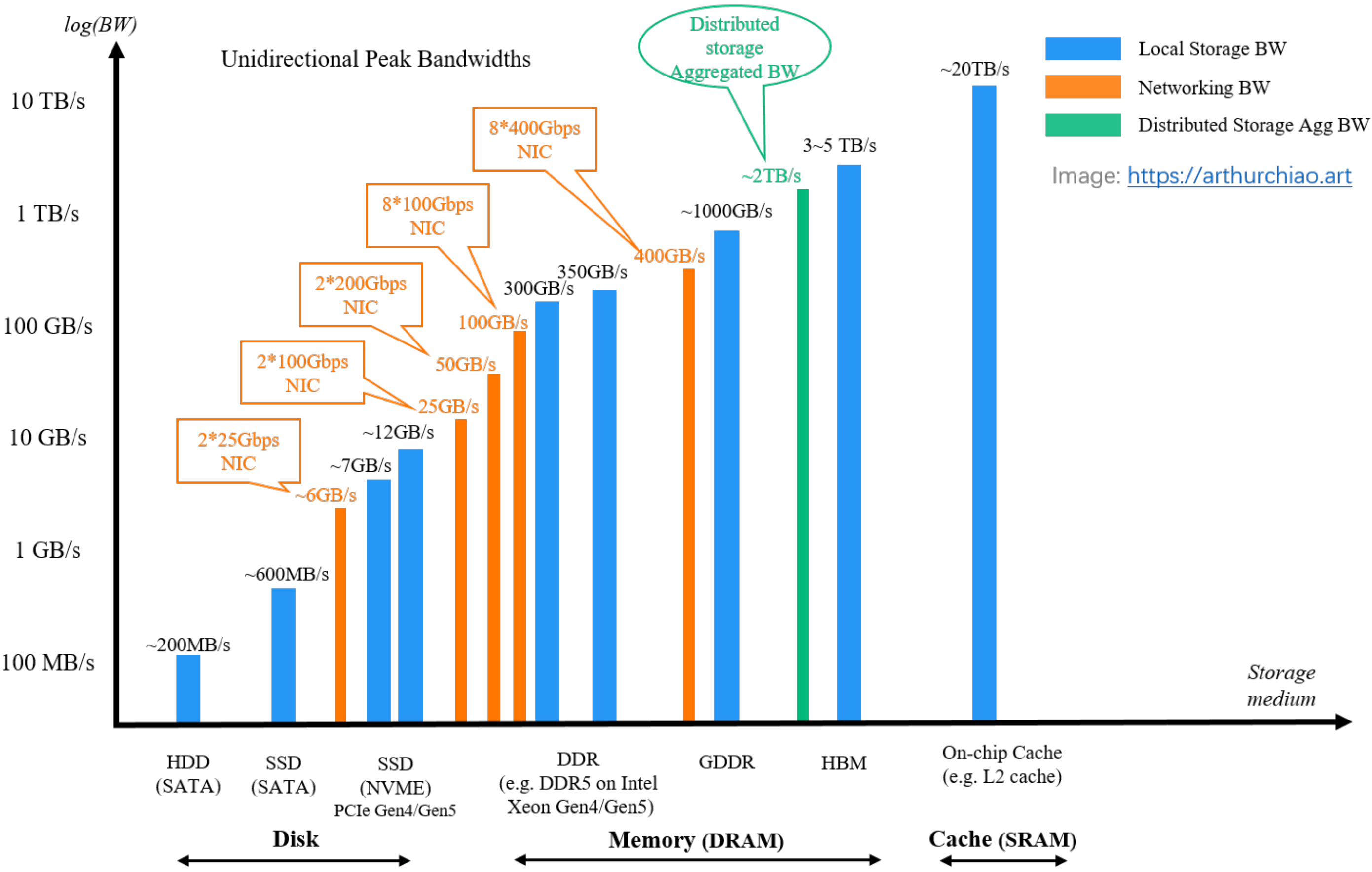
–Tom Jerry



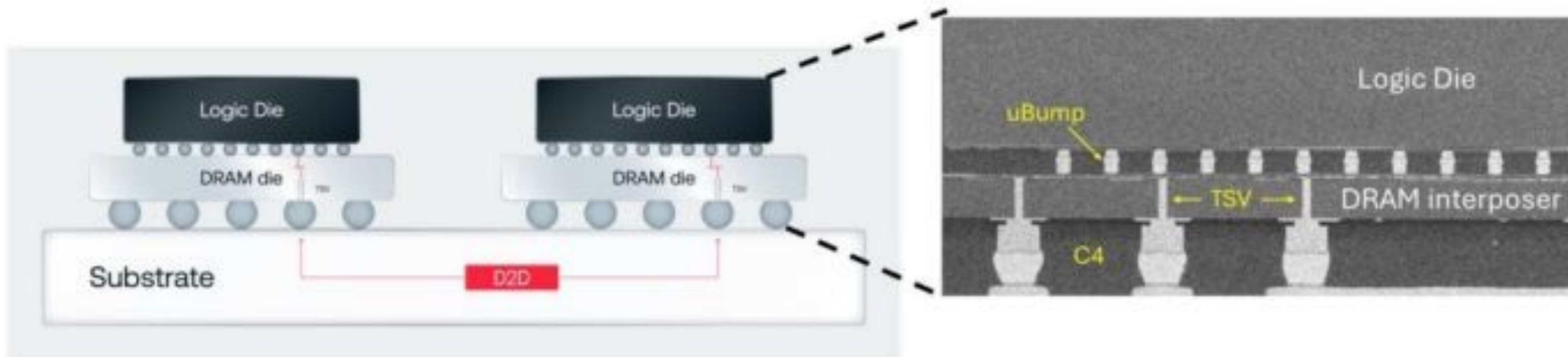
The Importance of Staying Local



The energy to fetch a word from DRAM can equal 100–1000 MAC operations!



Raptor 3D-DRAM



- Top die: TSMC N4 logic
- Bottom die: 3D DRAM
- Integration: 36 μ Face to Face (F2F) stacking
- Proven low cost, high volume, high yield process

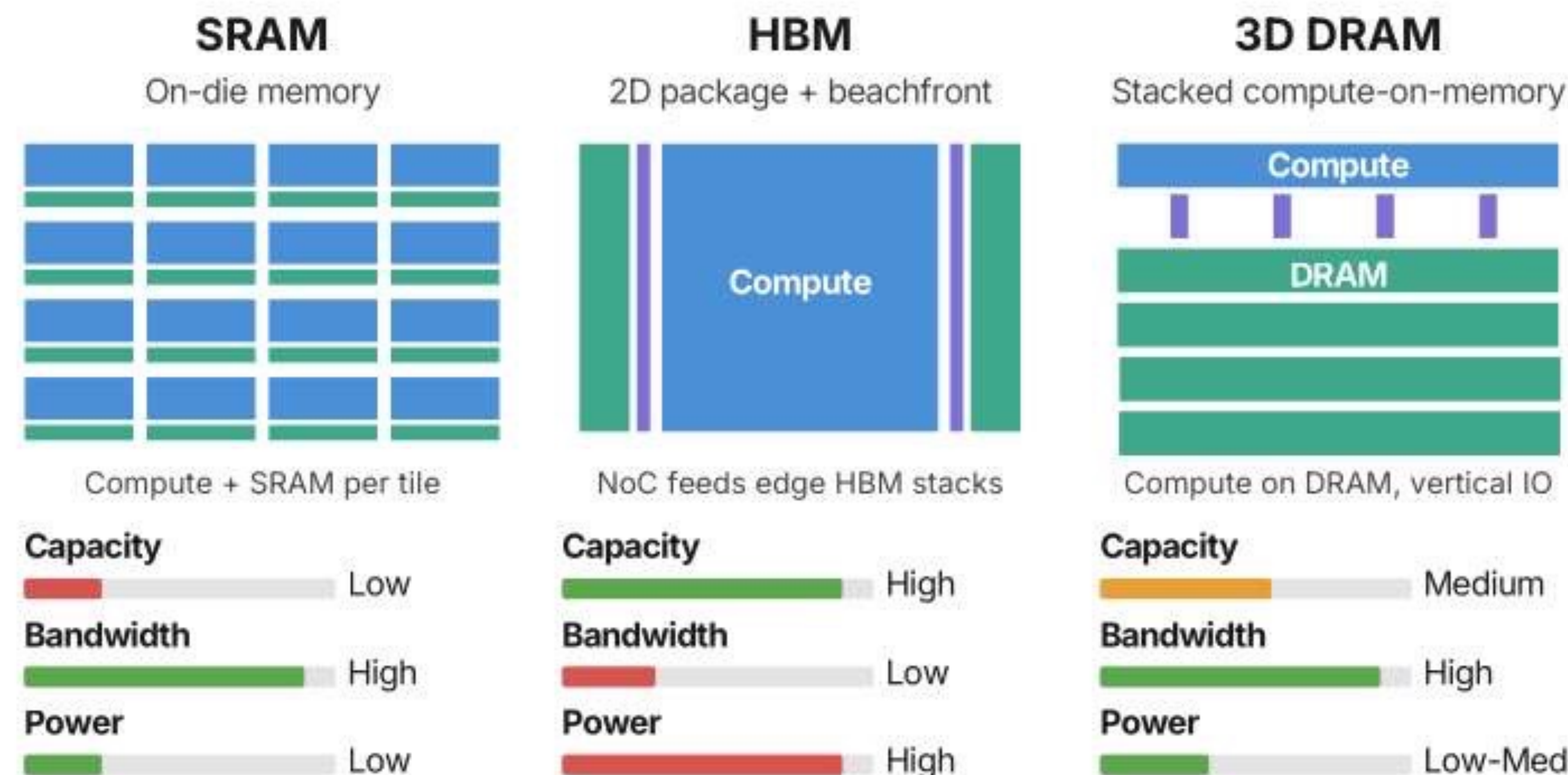
	HBM4	3D-DRAM
Topology	2.5D (beside)	F2F (below)
IO energy	2-3 pJ/bit	0.37 pJ/bit
BW / card	18 TB/s	100+ TB/s
Capacity	192 GB	32 GB

5.6x
higher BW / card

5-8x
lower energy / bit

7x
denser IO

Why 3D-DRAM?

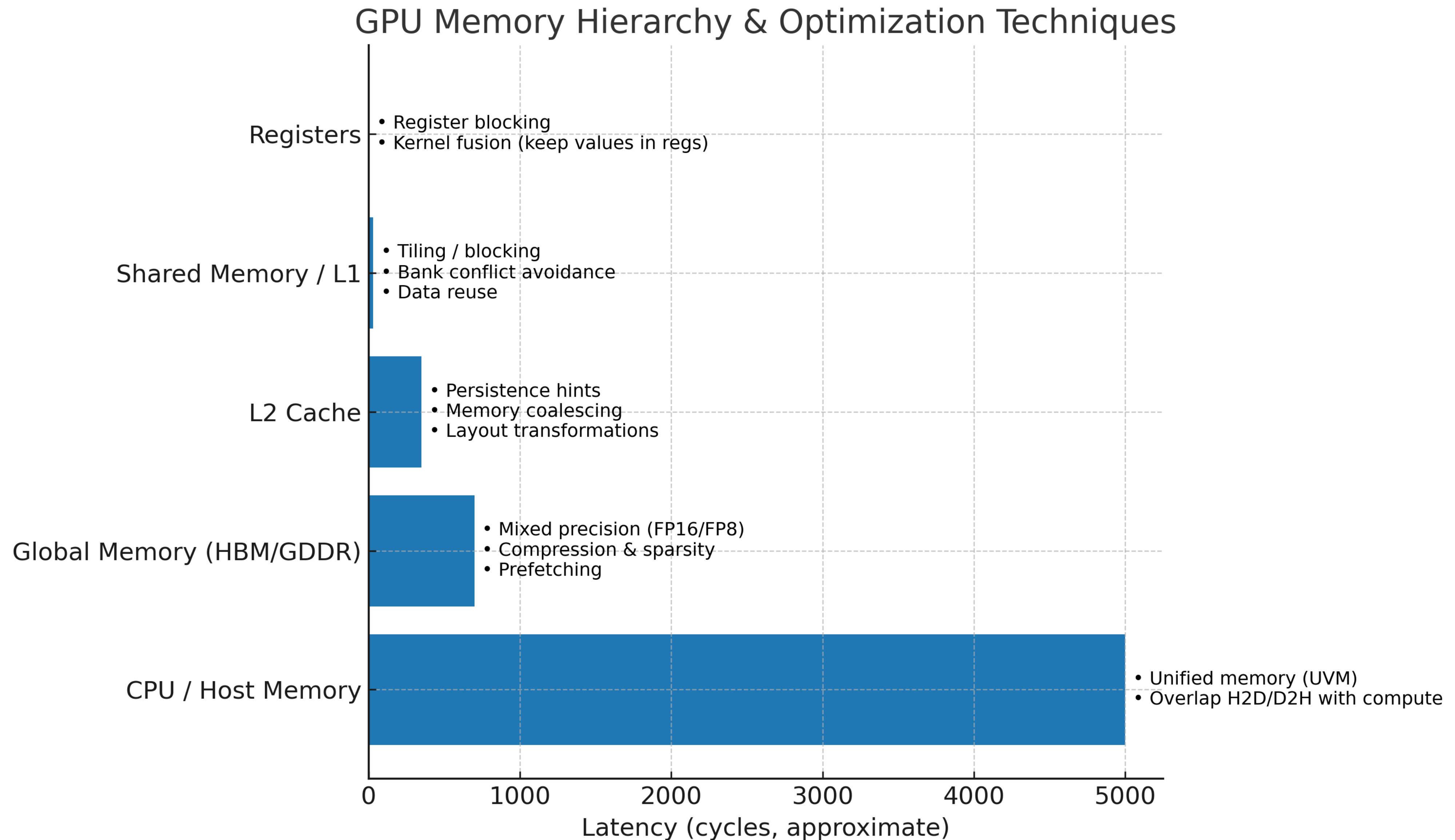


The energy ladder

Memories & interconnect	Energy / bit
SRAM (on-die)	~50 fJ
On-chip wire	~35 fJ/mm
3D vertical IO	0.3–0.4 pJ
Interposer trace	~500 fJ/mm
2.5D HBM4 (system)	2.5pJ + 3pJ (on chip)
3D IO lands ~10× below HBM — PHY-less, mm-scale vertical path instead of cm-scale interposer + PHY	

3D DRAM larger die (≤4 layers vs. 12–16 HBM): Better yield

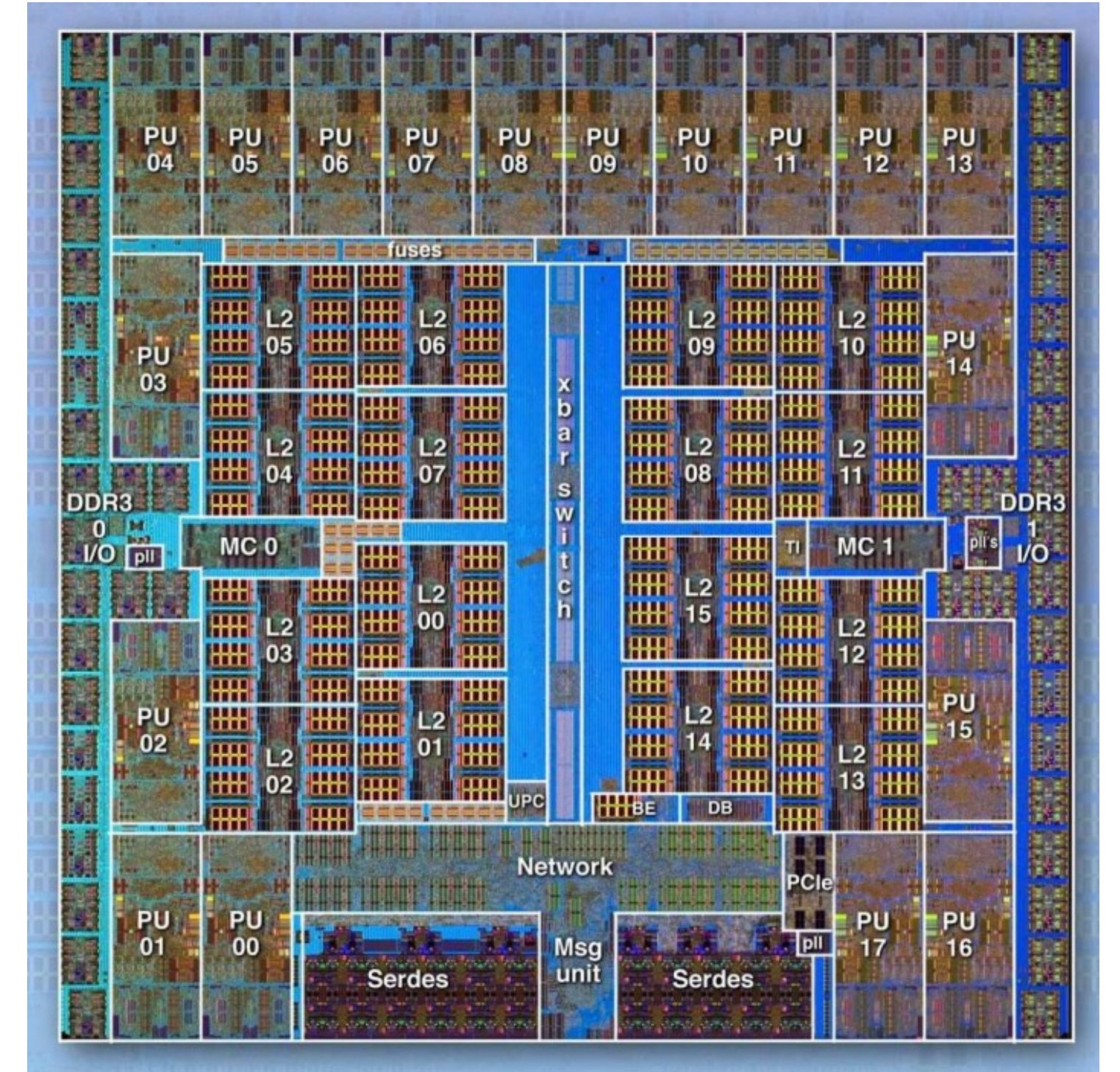
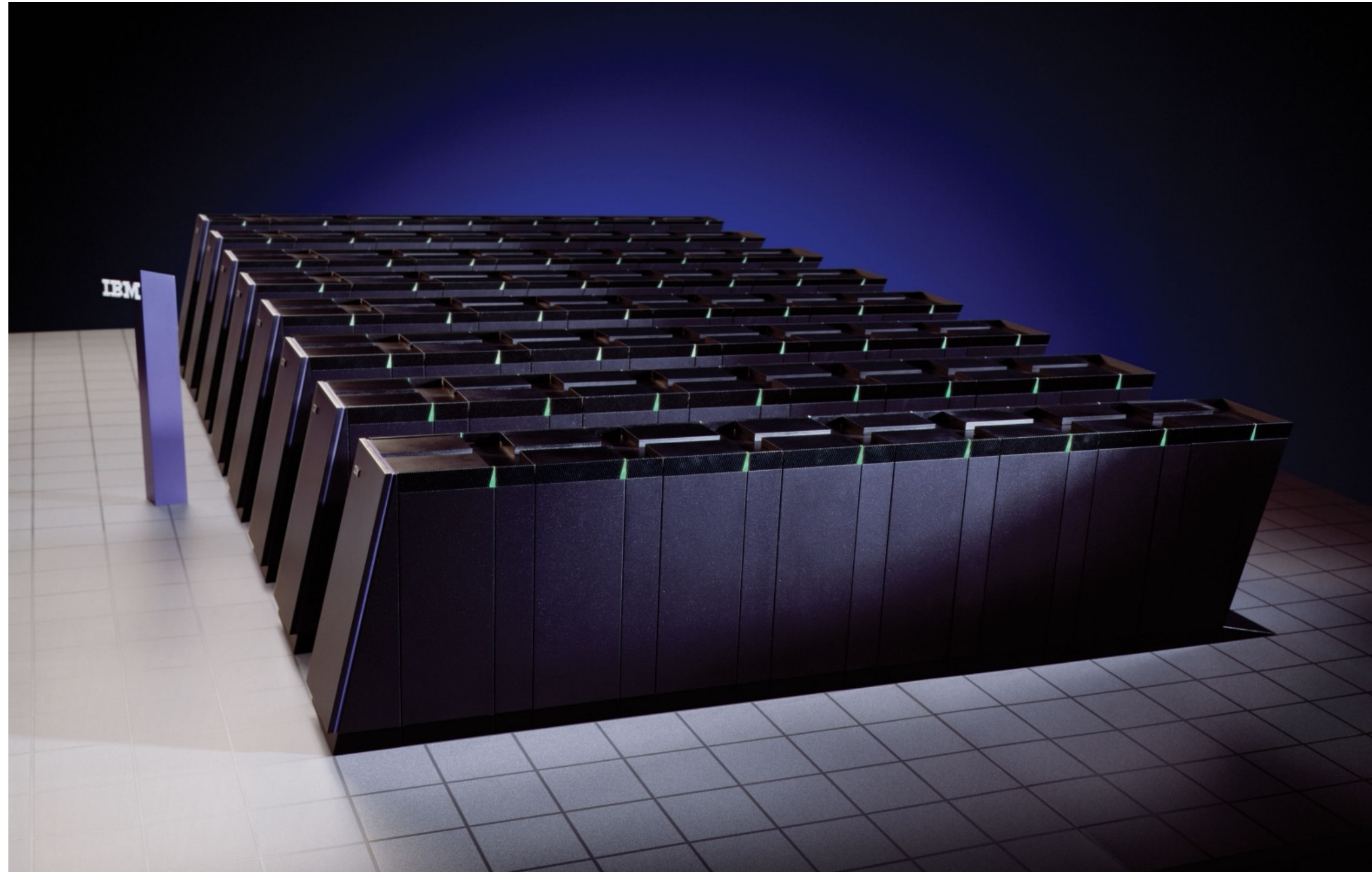
SRAM-class bandwidth at ~1/10th HBM energy — short 3D IO, no sideways data movement, no PHY, no beachfront limit



The GPU software stack aggressively utilizes locality, reuse, fusion, quantization, and overlap to minimize global memory traffic.

*“A supercomputer is a device for turning compute-bound problems
into I/O-bound problems.”*

Ken Batchers*



Supernode

NVIDIA DGX Vera Rubin NVL72

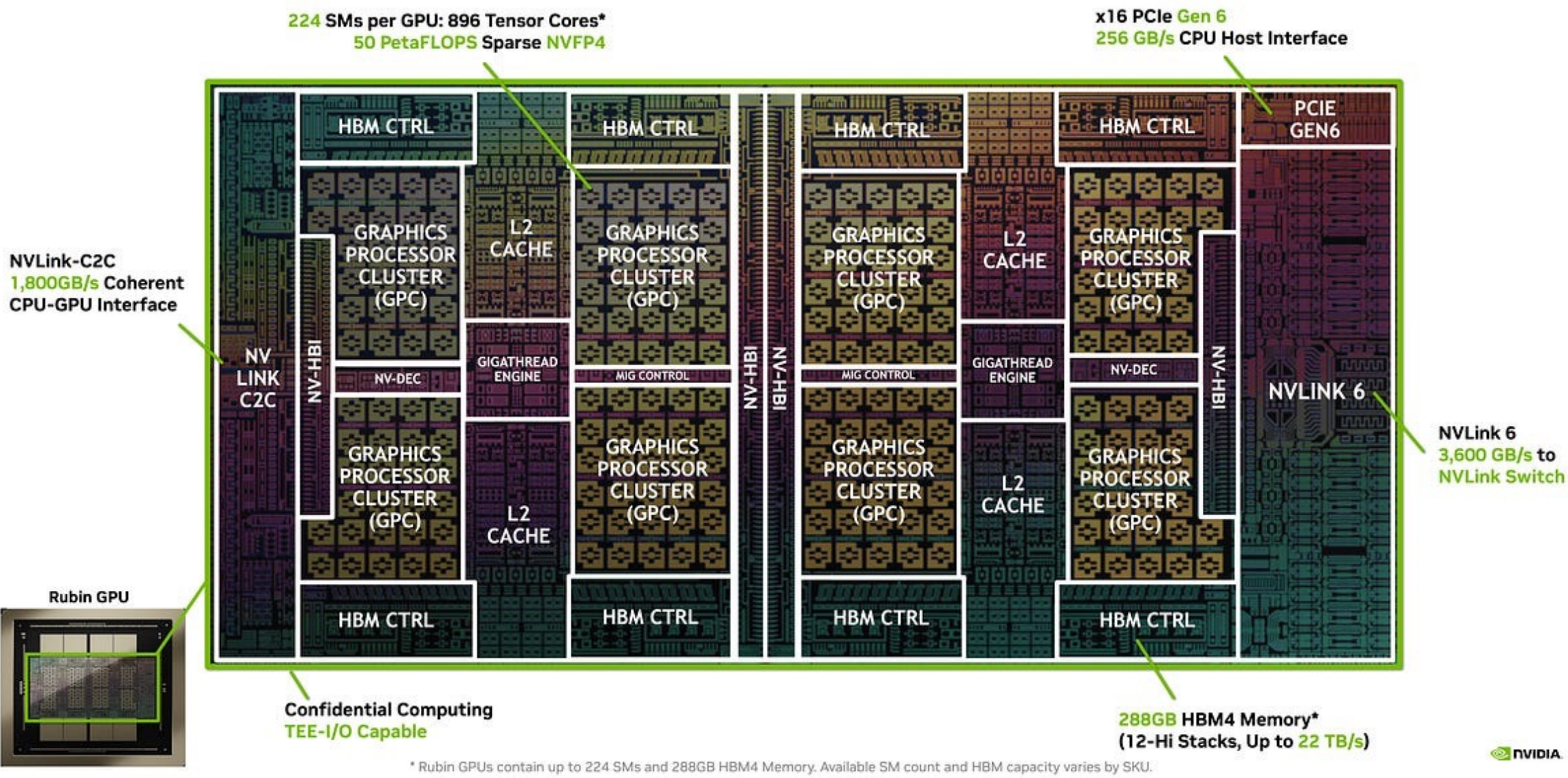


NVIDIA Vera Rubin NVL72

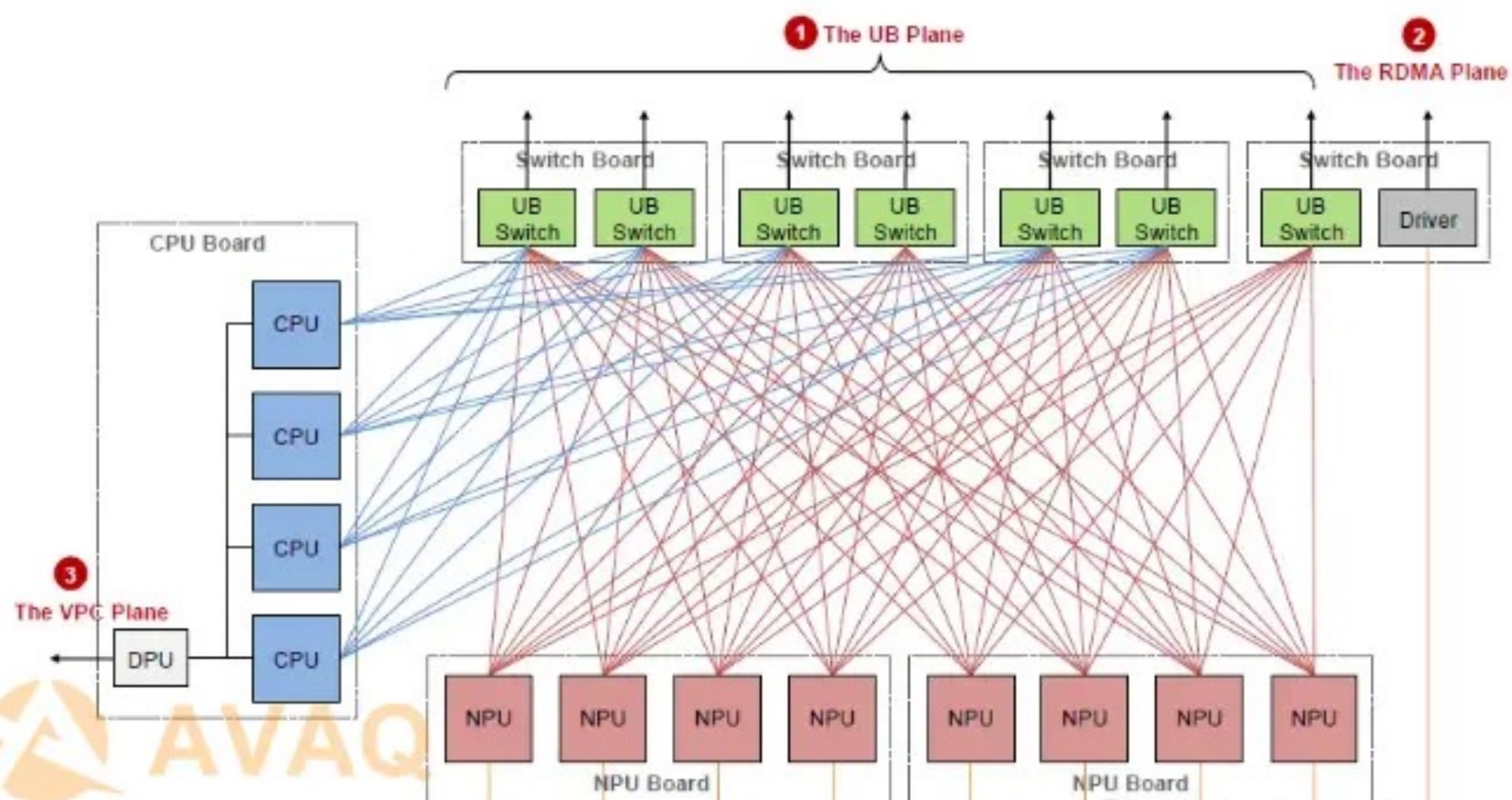
AI supercomputer for the next generation of AI

NVFP4 Inference	3.6 EFLOPS	5x Blackwell
NVFP4 Training	2.5 EFLOPS	3.5x
LPDDR5X Capacity	54 TB	2.5x
HBM4 Capacity	20.7 TB	1.5x
HBM4 Bandwidth	1.6 PB/s	2.8x
Scale-Up Bandwidth	260 TB/s	2x

NVIDIA Rubin GPU Architecture



Supernode



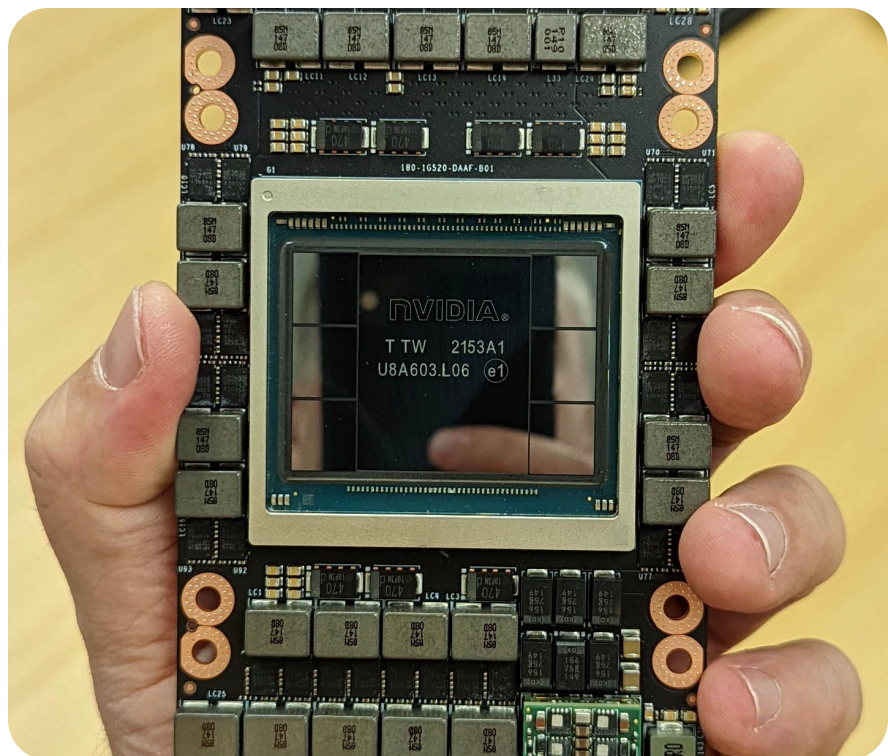
Key Capabilities - Huawei Ascend 910C Cloud Matrix 384 vs Nvidia GB200 NVL72

Chip and Package Level				
	Unit	GB200	Ascend 910C	Huawei vs Nvidia
BF16 dense TFLOPS	TFLOPS	2,500	780	0.3x
HBM capacity	GB	192	128	0.7x
HBM bandwidth	TB/s	8.0	3.2	0.4x
Scale Up Bandwidth	Gb/s uni-di	7,200	2,800	0.4x
Scale Out Bandwidth	Gb/s uni-di	400	400	1.0x
System Level				
	Unit	Nvidia GB200 NVL72	Cloud Matrix CM384	Huawei vs Nvidia
BF16 dense PFLOPS	PFLOPS	180	300	1.7x
HBM capacity	TB	13.8	49.2	3.6x
HBM bandwidth	TB/s	576	1,229	2.1x
Scale Up Bandwidth	Gb/s uni-di	518,400	1,075,200	2.1x
Scale Up Domain Size	GPUs	72	384	5.3x
Scale Out Bandwidth	Gb/s uni-di	28,800	153,600	5.3x
All-In System Power ₁	W	145,000	599,821	4.1x
All-in Power per BF16 dense FLOP	W/TFLOP	0.81	2.00	2.5x
All-in Power per memory bandwidth	W per TB/s	251.7	488.1	1.9x
All-in Power per memory capacity	kW/TB	10.5	12.2	1.2x

1. All-in System Power is total cluster power including scale-out networking, storage, etc.

How GPU data travels through a VCSEL optical link

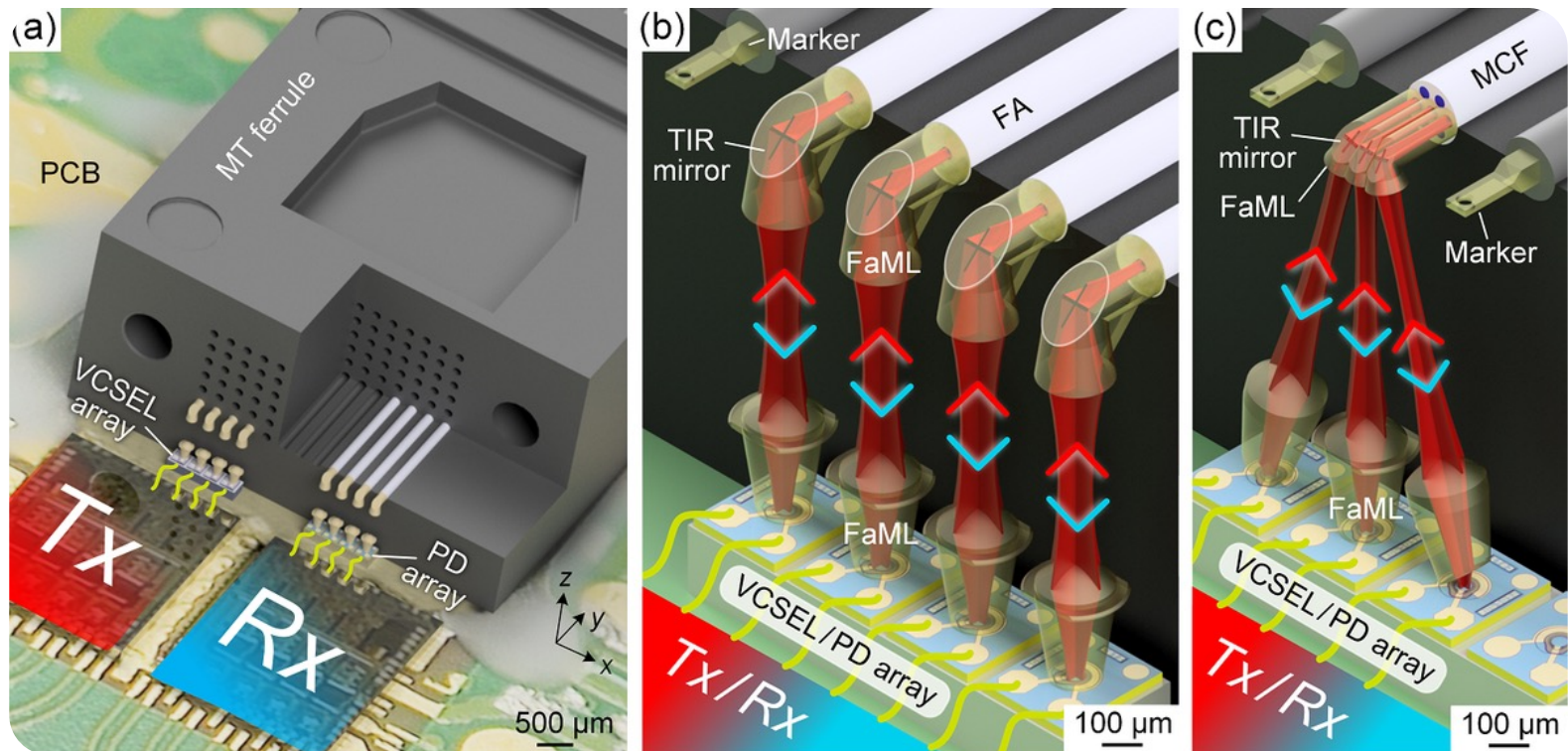
A vertical-cavity surface-emitting laser (VCSEL) link carries data as changing brightness levels in an optical fiber.



GPU + local memory



VCSEL array



Microlens and multimode fiber



Photodiode array

1 GPU and transmitter
The network chip turns GPU data into a very fast stream of electrical symbols.

2 VCSEL driver
The driver changes the laser current to match each data symbol.

3 VCSEL laser
The laser turns the current levels into matching brightness levels of 850 nm light.

4 Microlens and fiber
Microlenses focus each light lane into a multimode fiber.

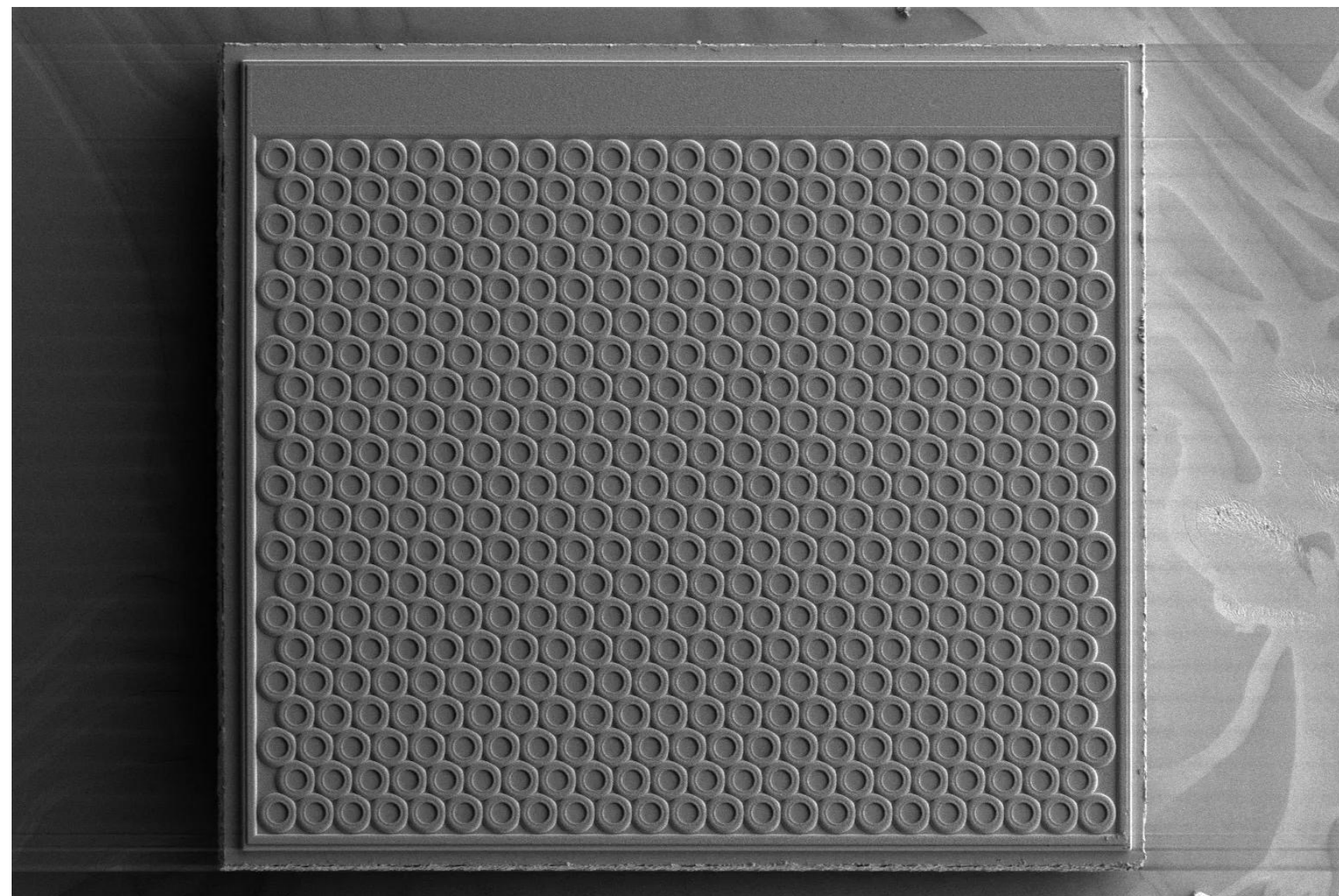
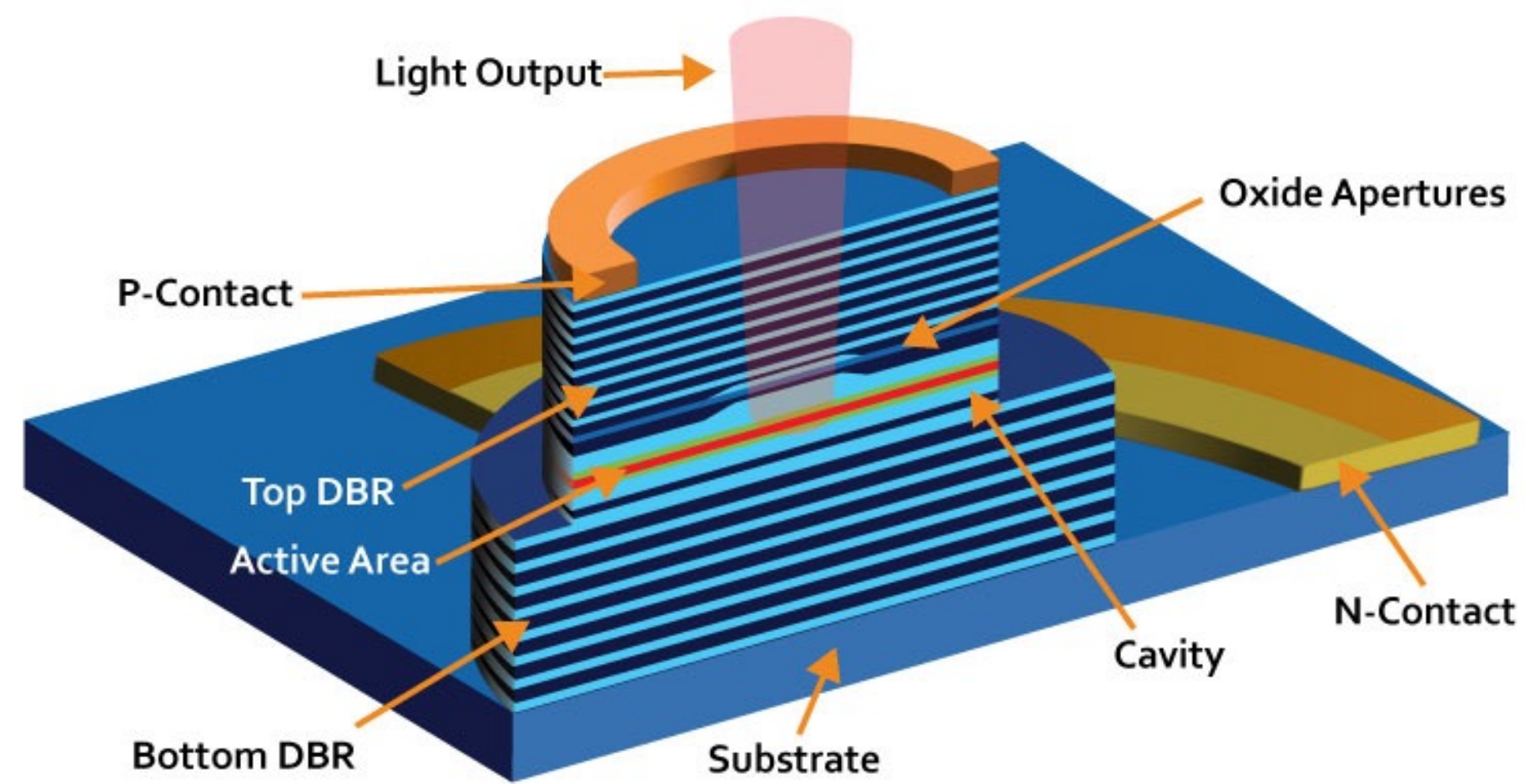
5 Photodiode
The detector turns the received light into a tiny electrical current.

6 Receiver amplifier
The transimpedance amplifier makes the tiny current usable. The receiver identifies the symbols and rebuilds the data.

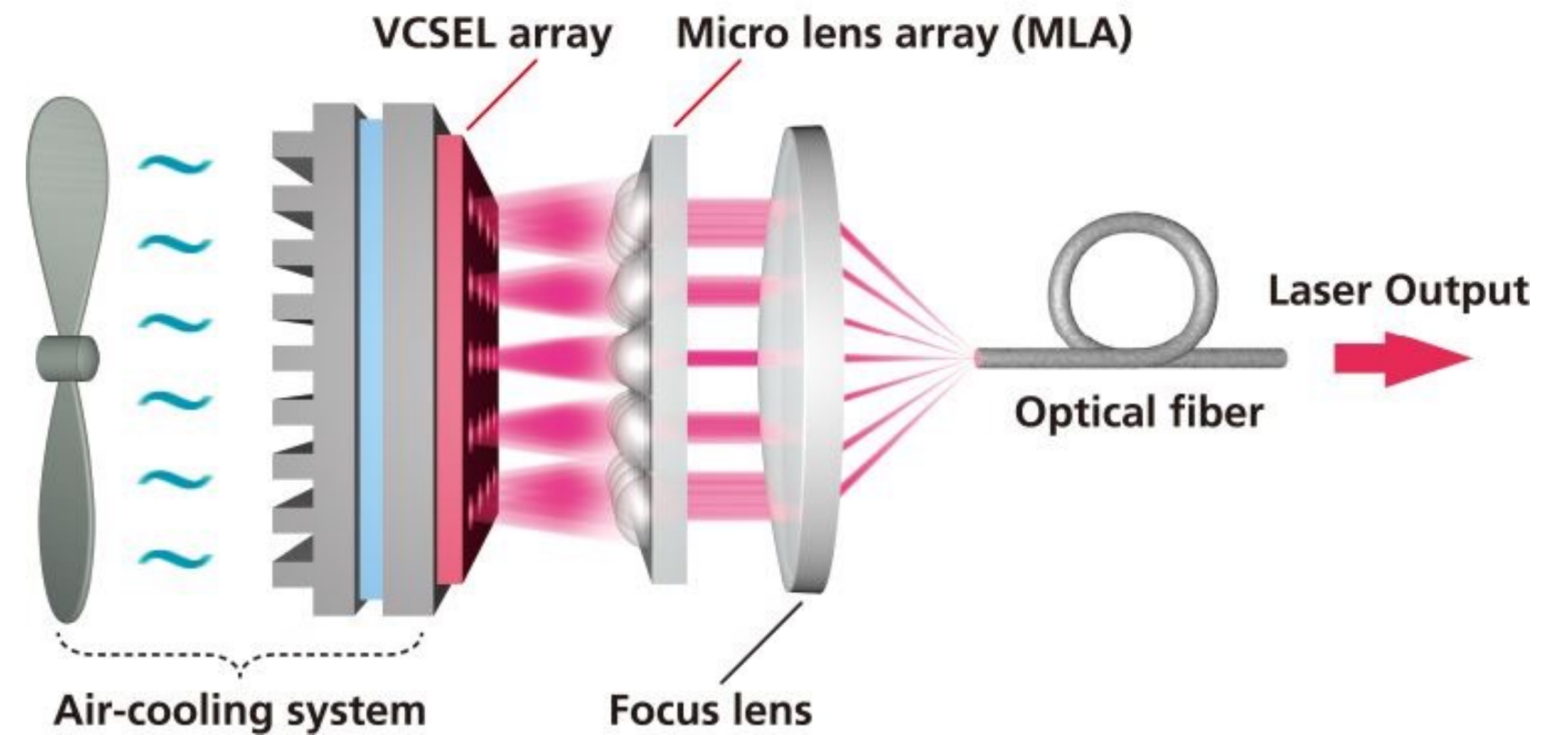
Core idea: the transmitter places data onto light by changing its brightness. The receiver measures those brightness levels and rebuilds the original bits.

NRZ uses two brightness levels. PAM4 uses four brightness levels and carries two bits in each symbol.

This is a typical short-range VCSEL link. Some GPU systems use silicon photonics or copper instead, depending on the product and distance.



High-Power VCSEL Module



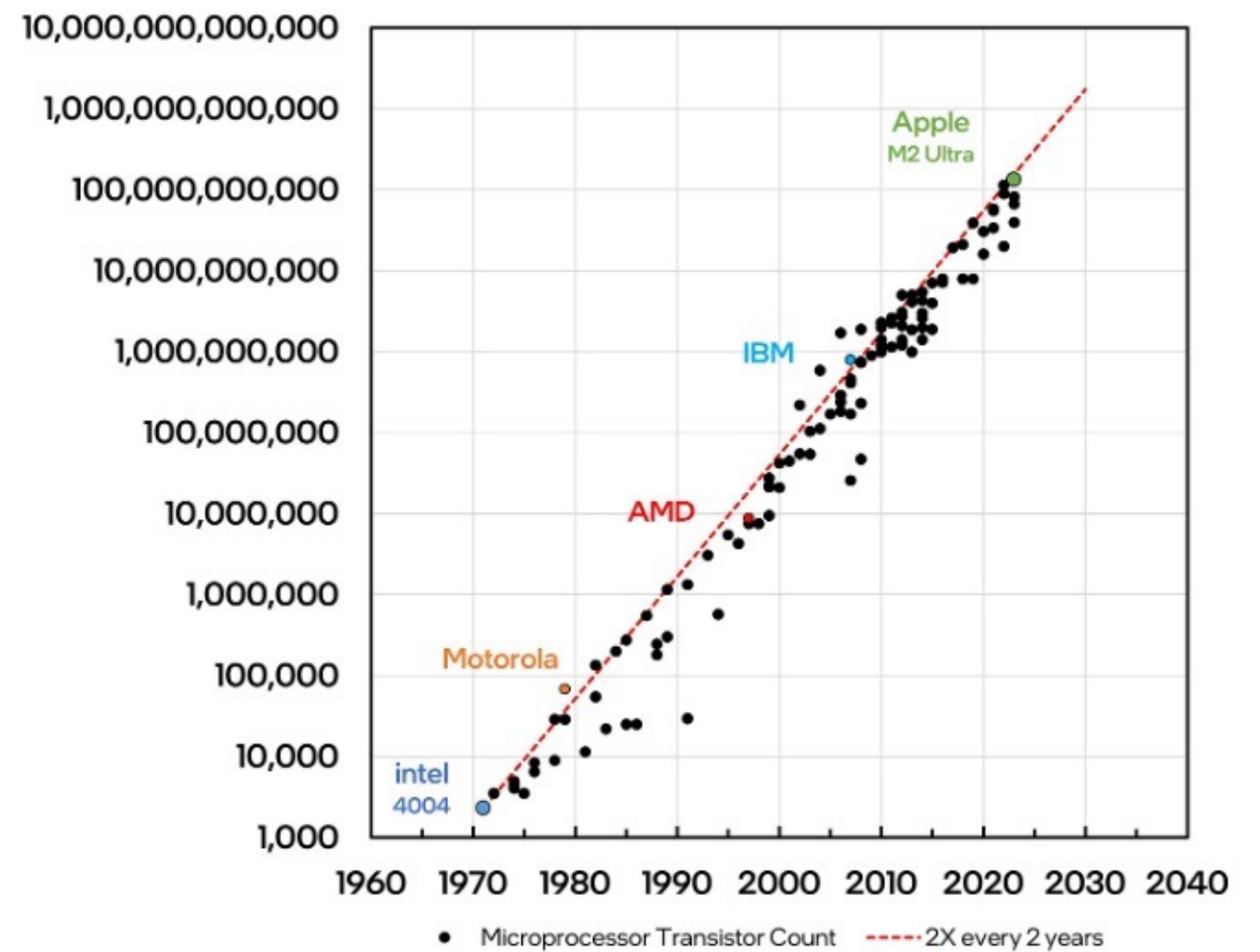
vertical-cavity surface-emitting laser
(VCSEL)

Lesson 4: Bending the laws of physics, is that possible?

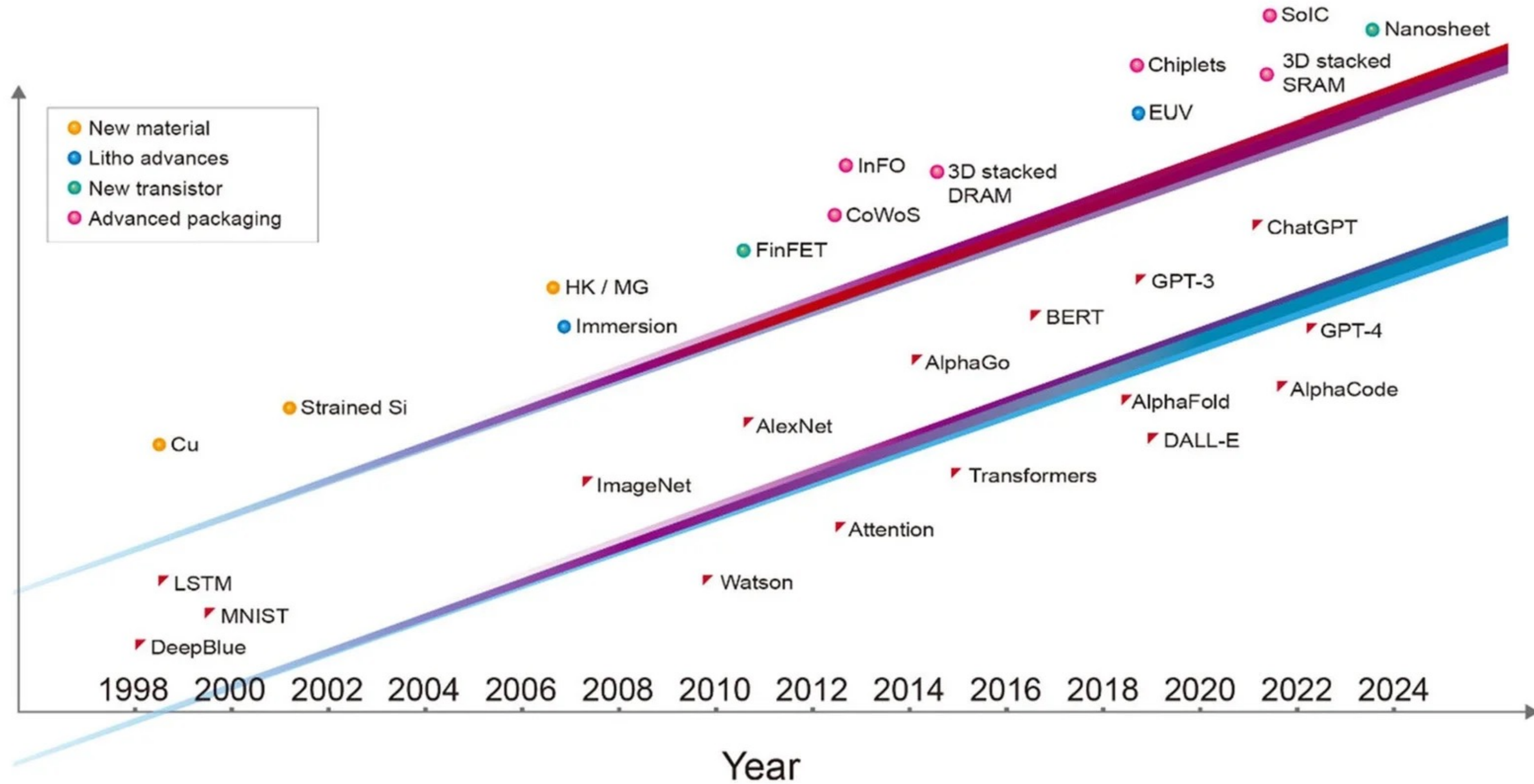
–Tom Jerry

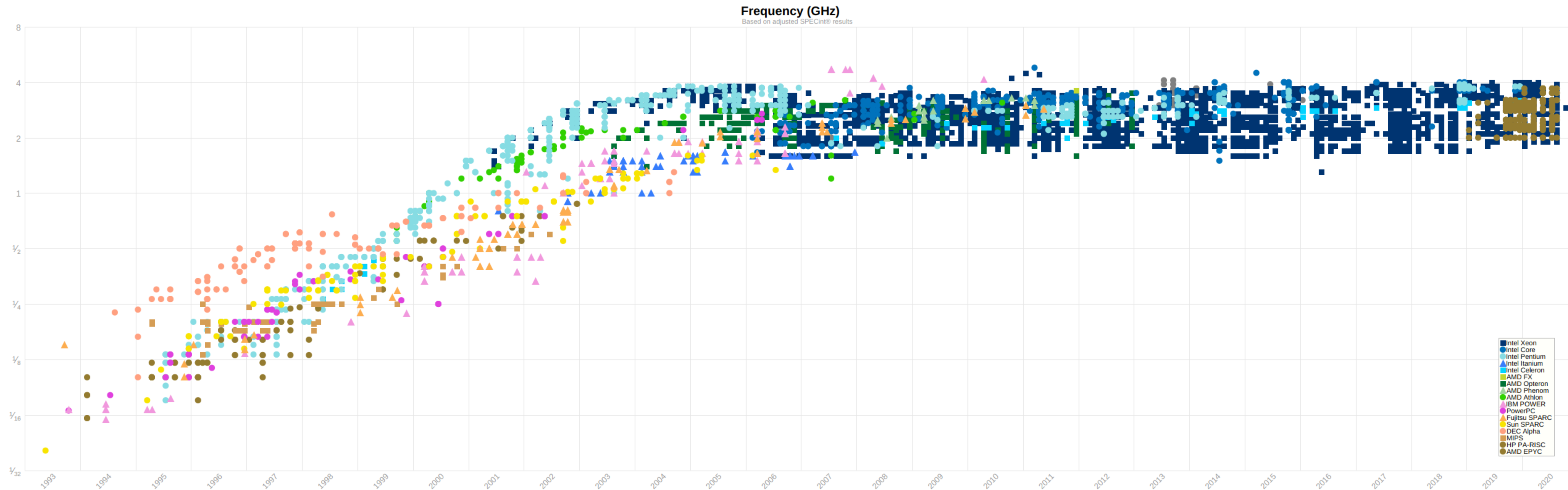
Modern Computing

- Transistors shrink, allowing more to fit on a silicon wafer, increasing computing power.
- Moore's Law predicts that microprocessor capabilities will double every 18-24 months.
- The trend has led to microprocessors with up to 100 billion transistors.



Modern Computing



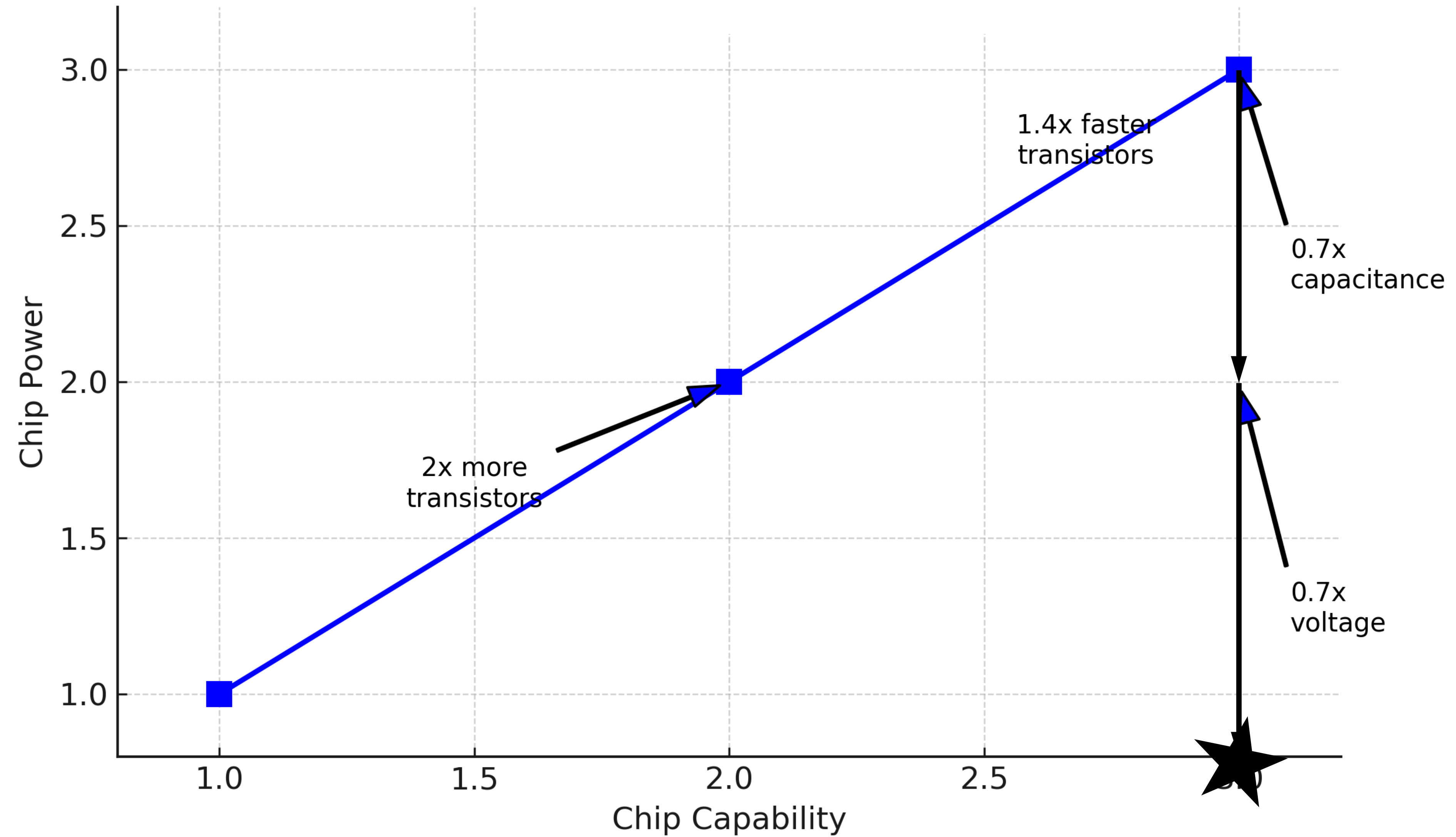


Transistors got exponentially faster until the moment they didn't.

How does Dennard scaling work?

- The total power used by a chip for a given area stays the same [↗](#)
- The number of transistors doubles at a specific rate [↗](#)
- The frequency increases by a certain percentage every two years [↗](#)
- Both voltage and current scale down with length [↗](#)

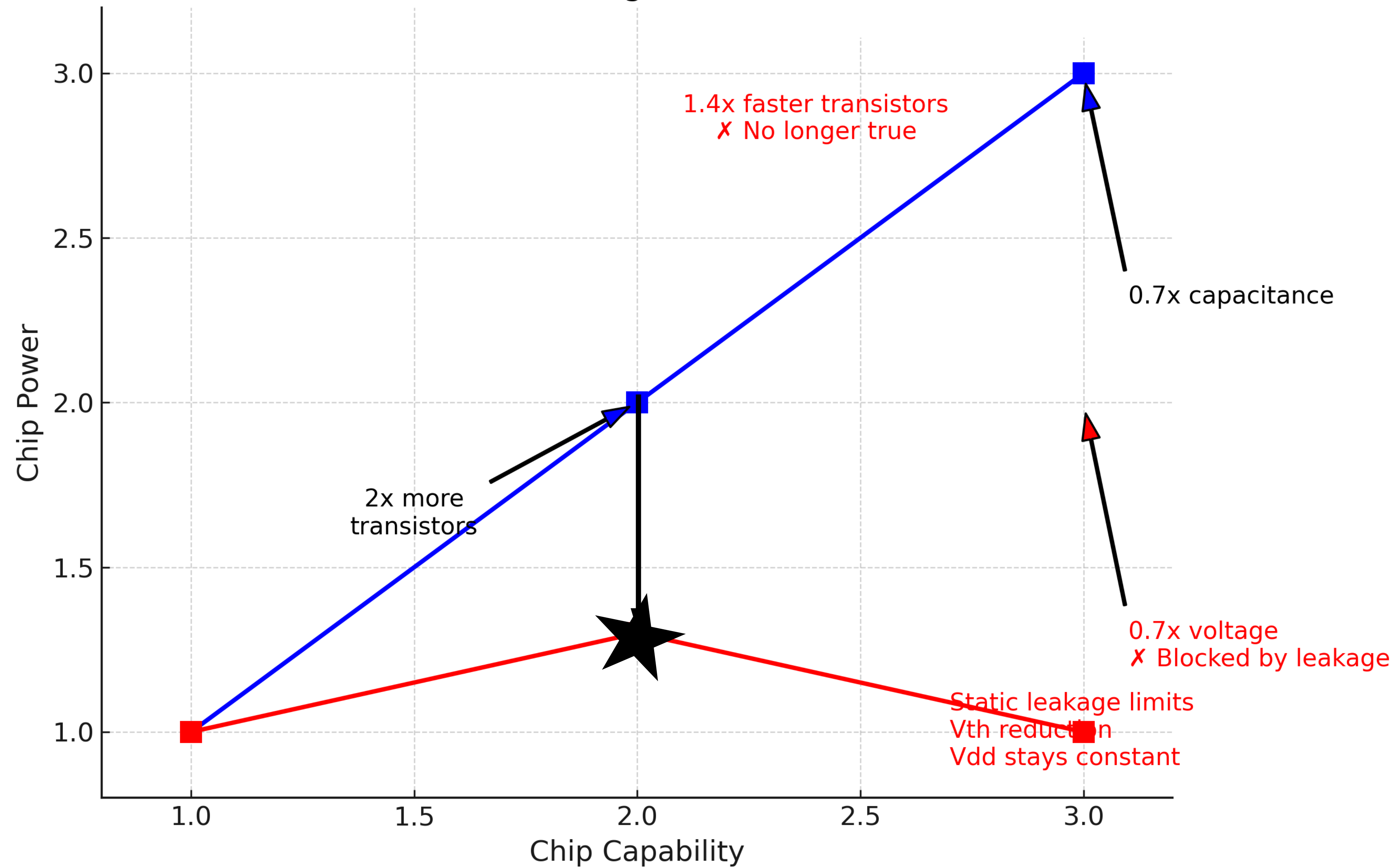
Dennard Scaling



Challenges

- However, Dennard scaling ignored "leakage current" and "threshold voltage", which establish a baseline of power per transistor. 
- Leakage power became problematic around 2005. 

Post-Dennard Scaling

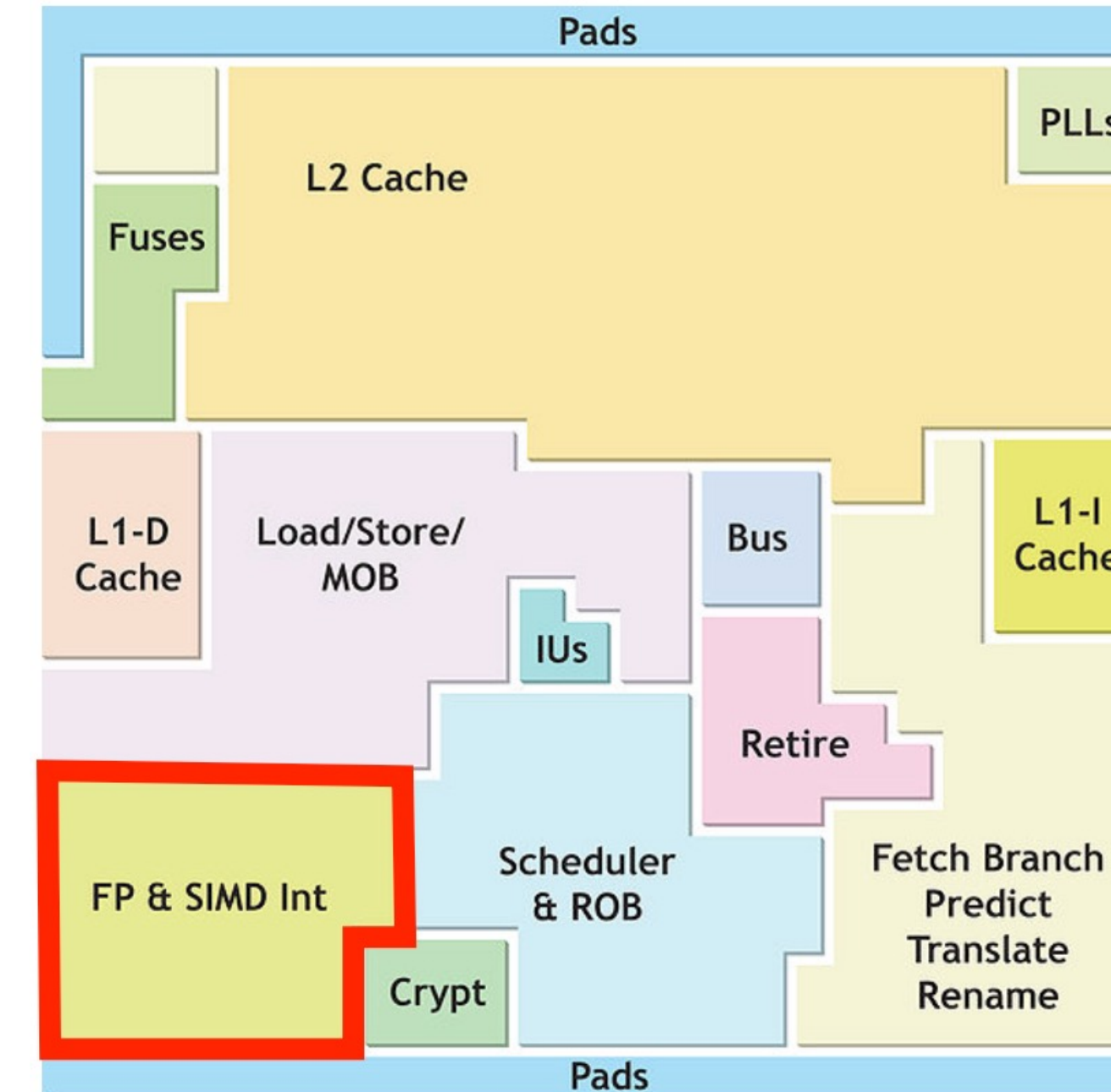
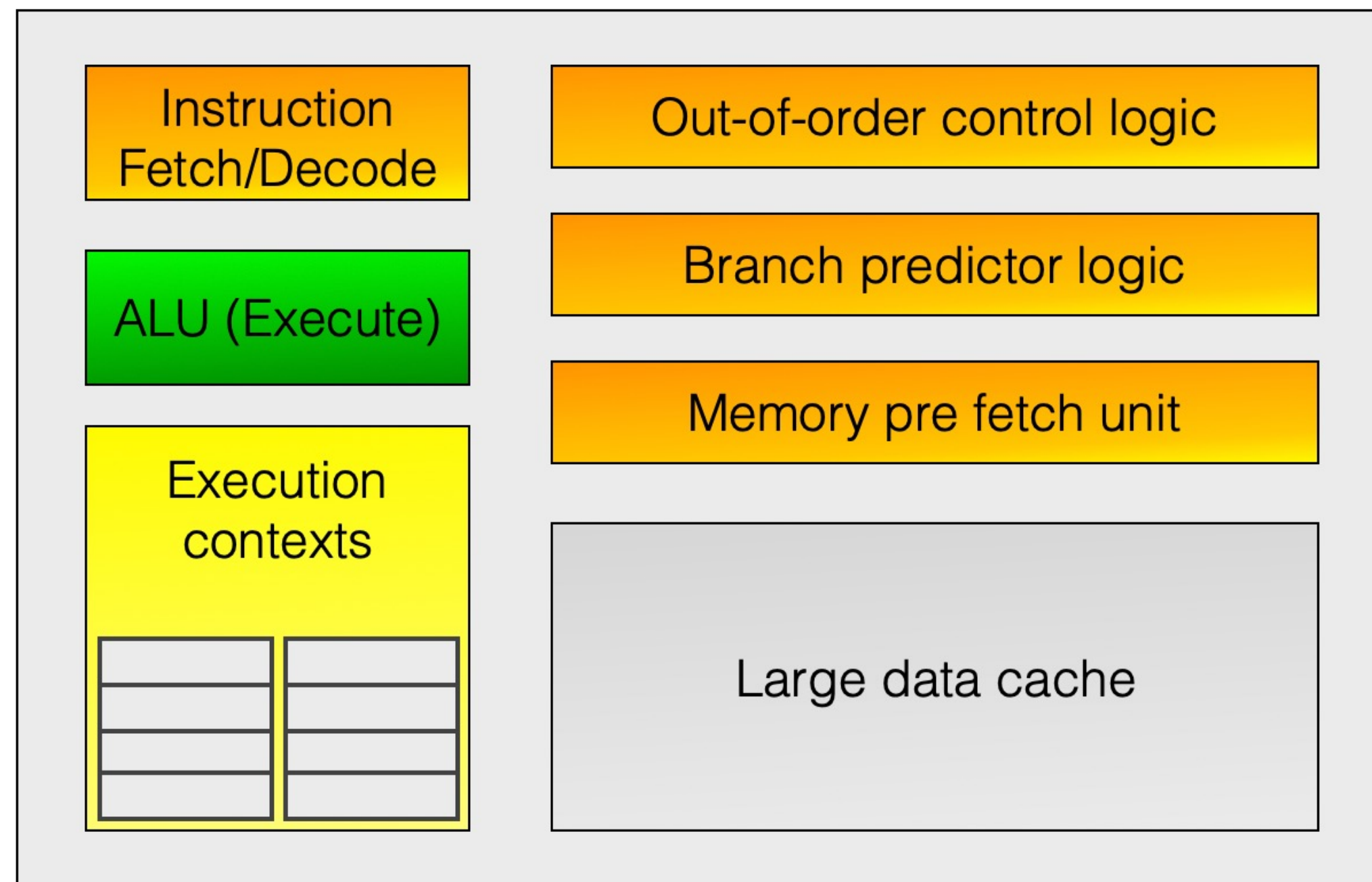


The alternative path is massive parallelism with many cores, doubles every two years, which however introduces the challenge to software development.

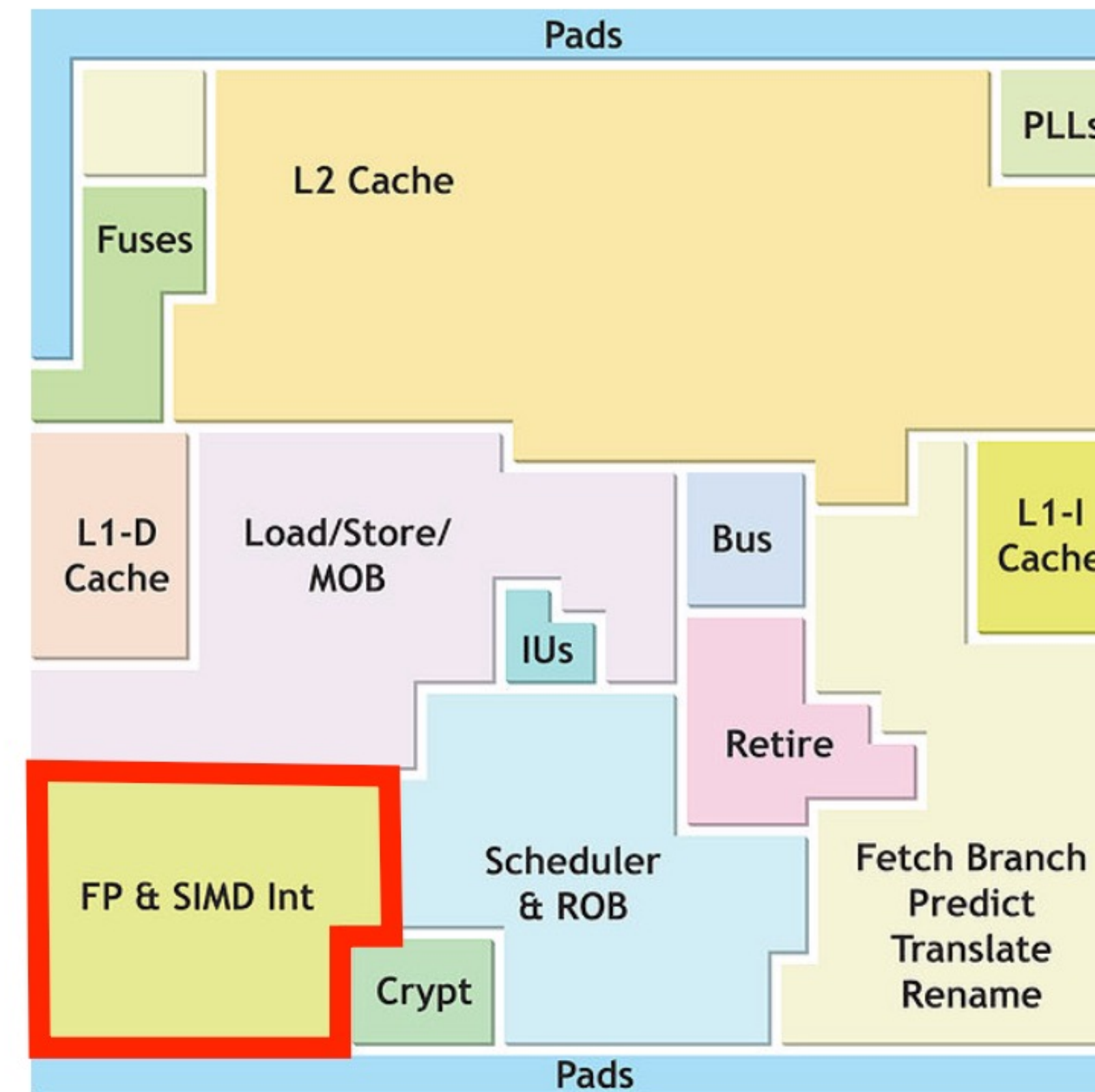
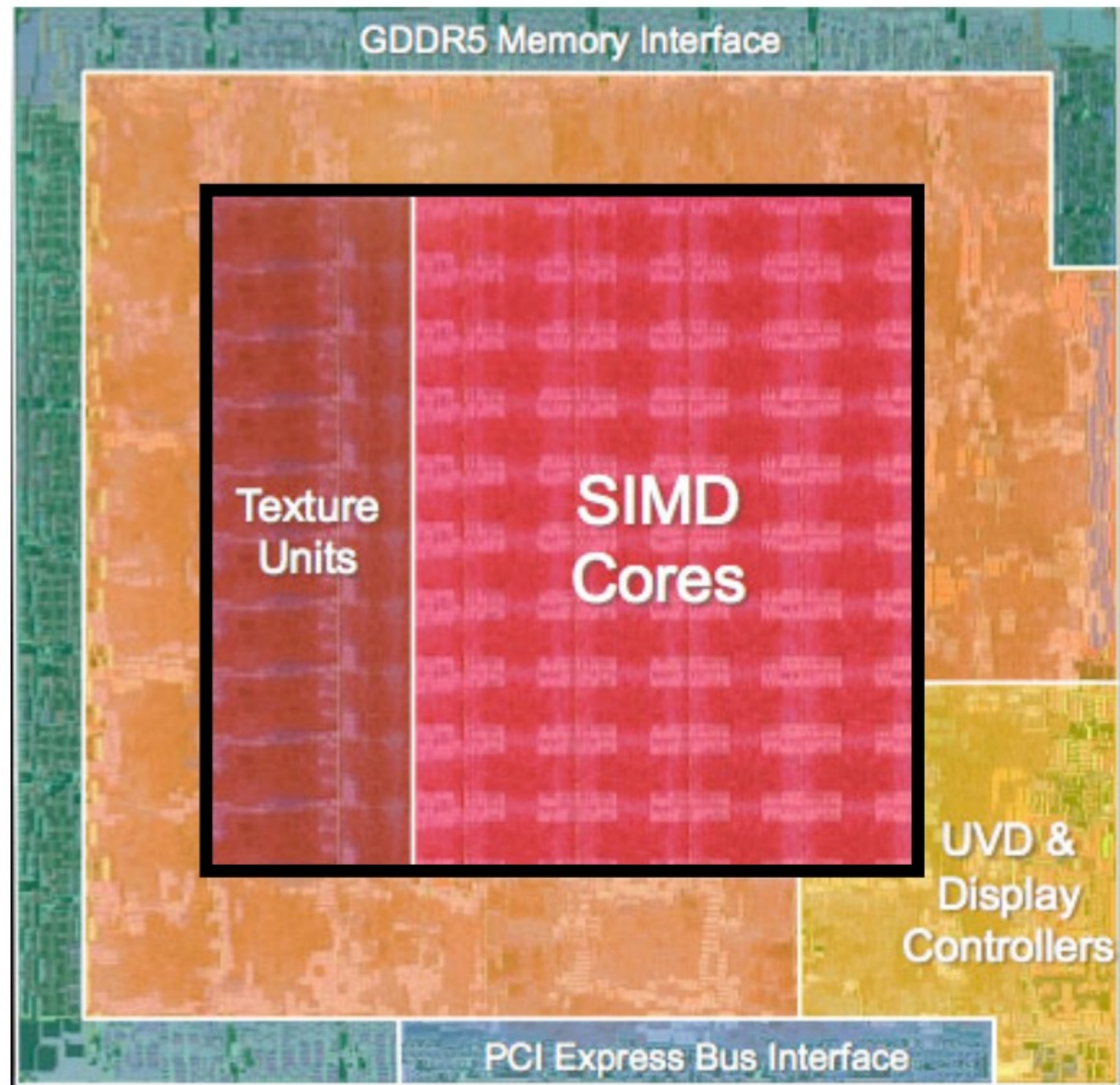
Lesson 5: Who sells memory bandwidth better, CPU or GPU?

–Tom Jerry

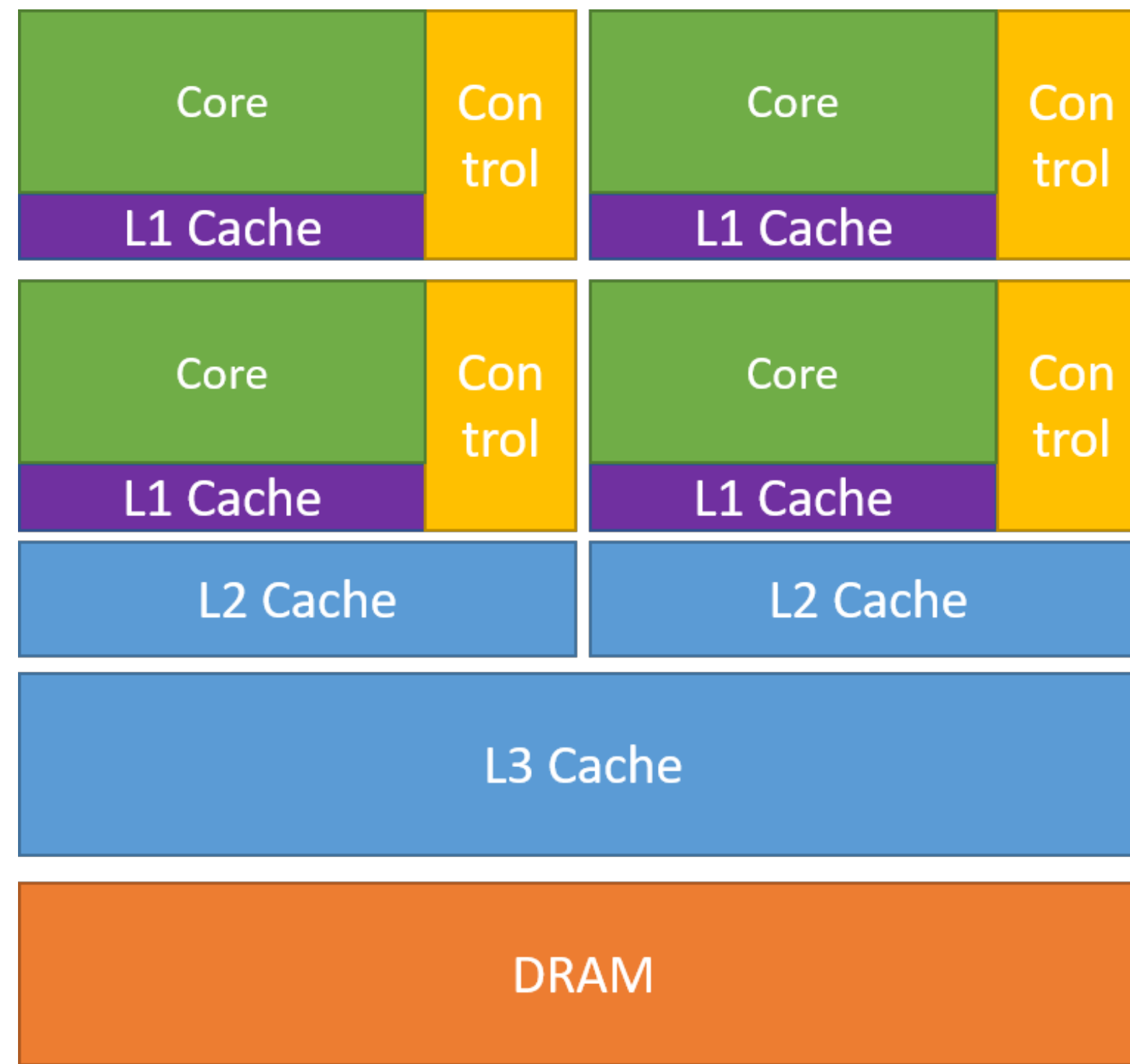
CPU: Battling for Single-Thread Performance in the Face of the Memory Wall



GPU: Defying Moore's Law through massive parallelism



CPU vs. GPU



CPU

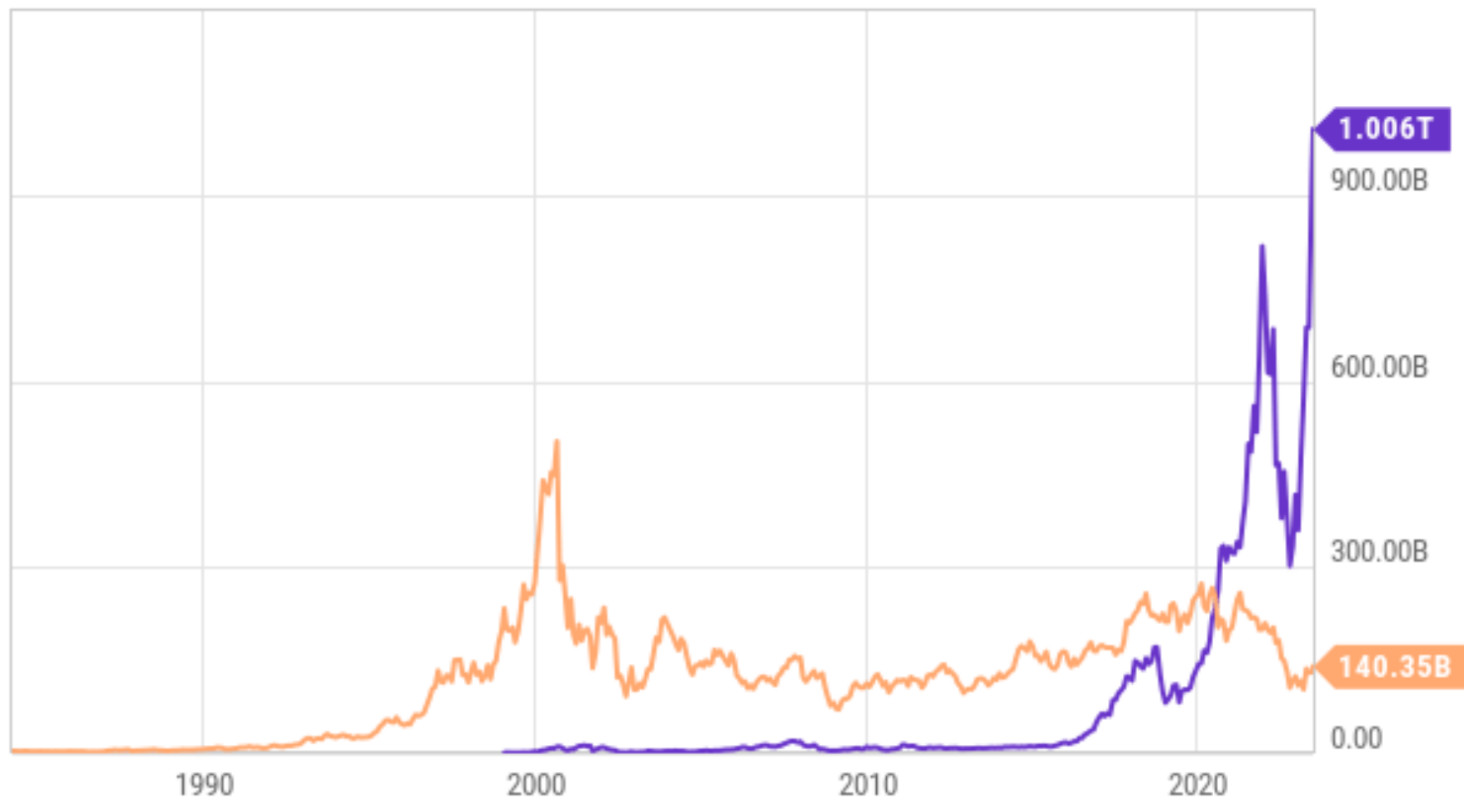
Less than 20 cores
1-2 threads per core
Latency is hidden by large cache



GPU

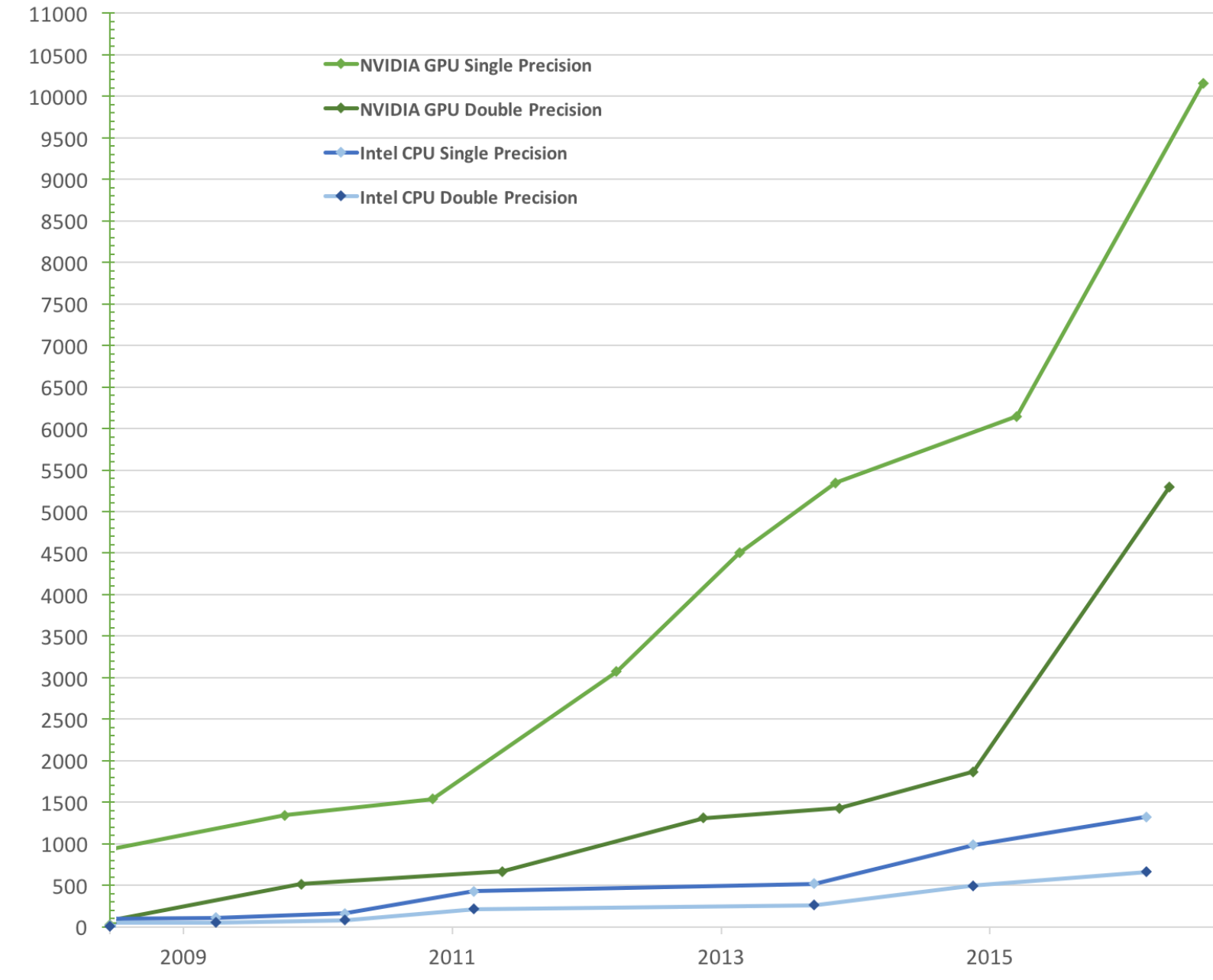
512 cores
10s to 100s of threads per core
Latency is hidden by fast context switching

● NVIDIA Corp (NVDA) Market Cap
● Intel Corp (INTC) Market Cap

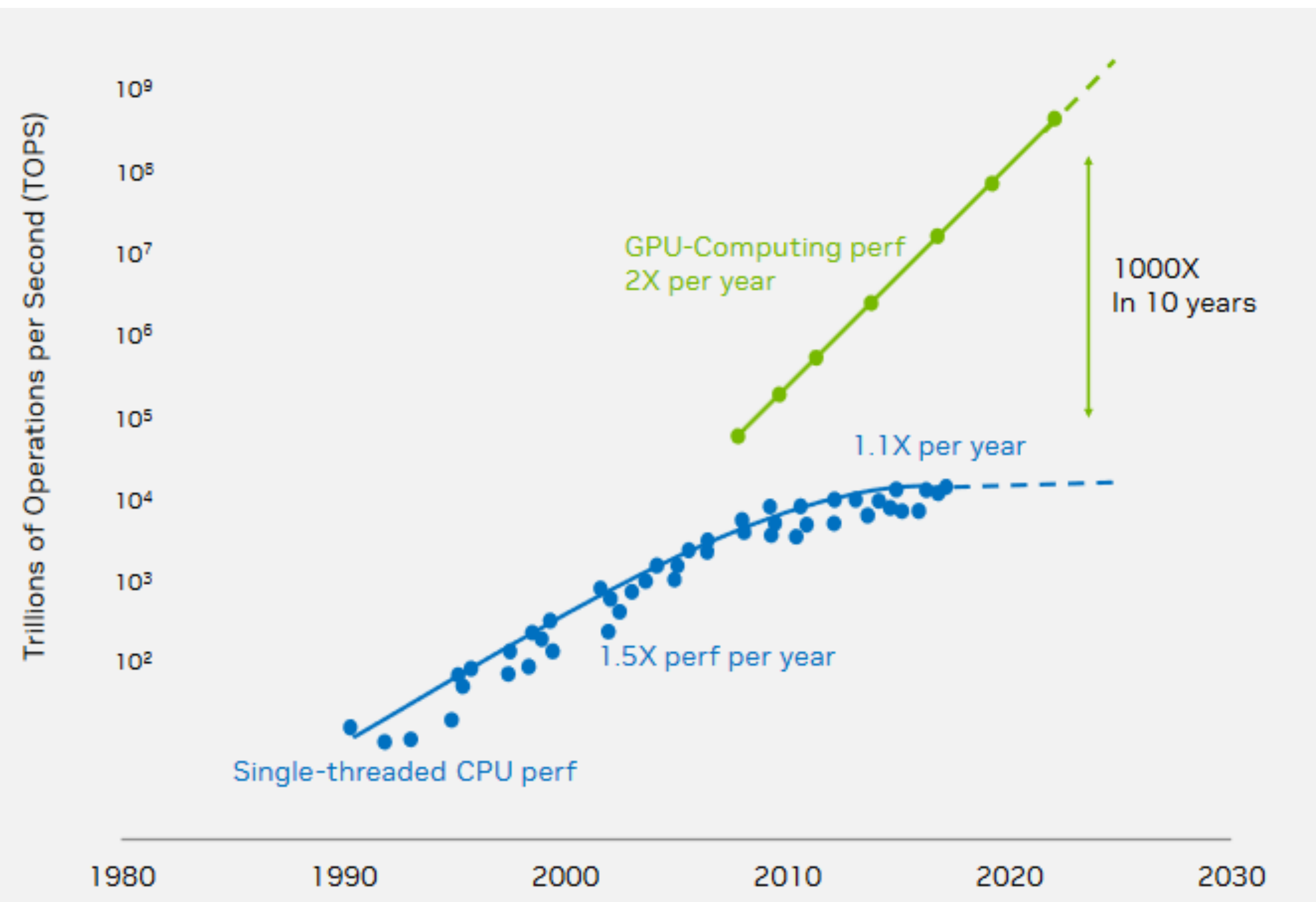


Seeking Alpha^α
Jun 13 2023, 2:05PM EDT. Powered by YCHARTS

Theoretical GFLOP/s at base clock



Defying Moore's Law



What is “Machine Learning Systems”

–Tom Jerry

Three Pillars of ML

Decades long algorithmic research, big data, and scalable computing

SVM ConvNet
Backprop GBM

ML Research

1958



IMAGENET

Data

2000

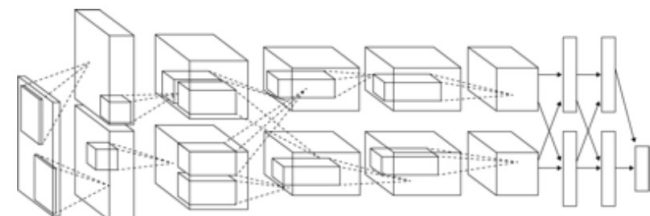


Public
cloud

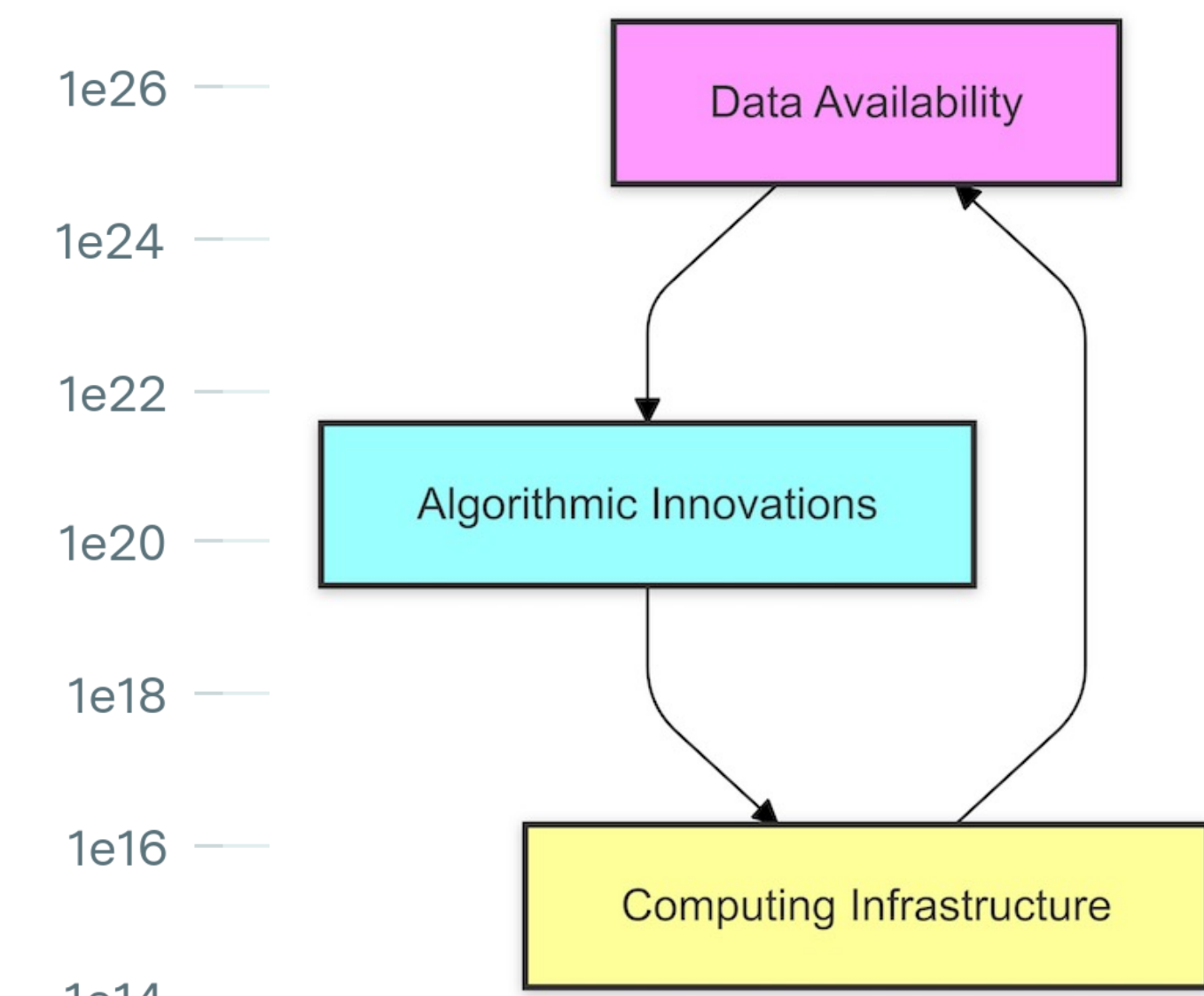


Compute

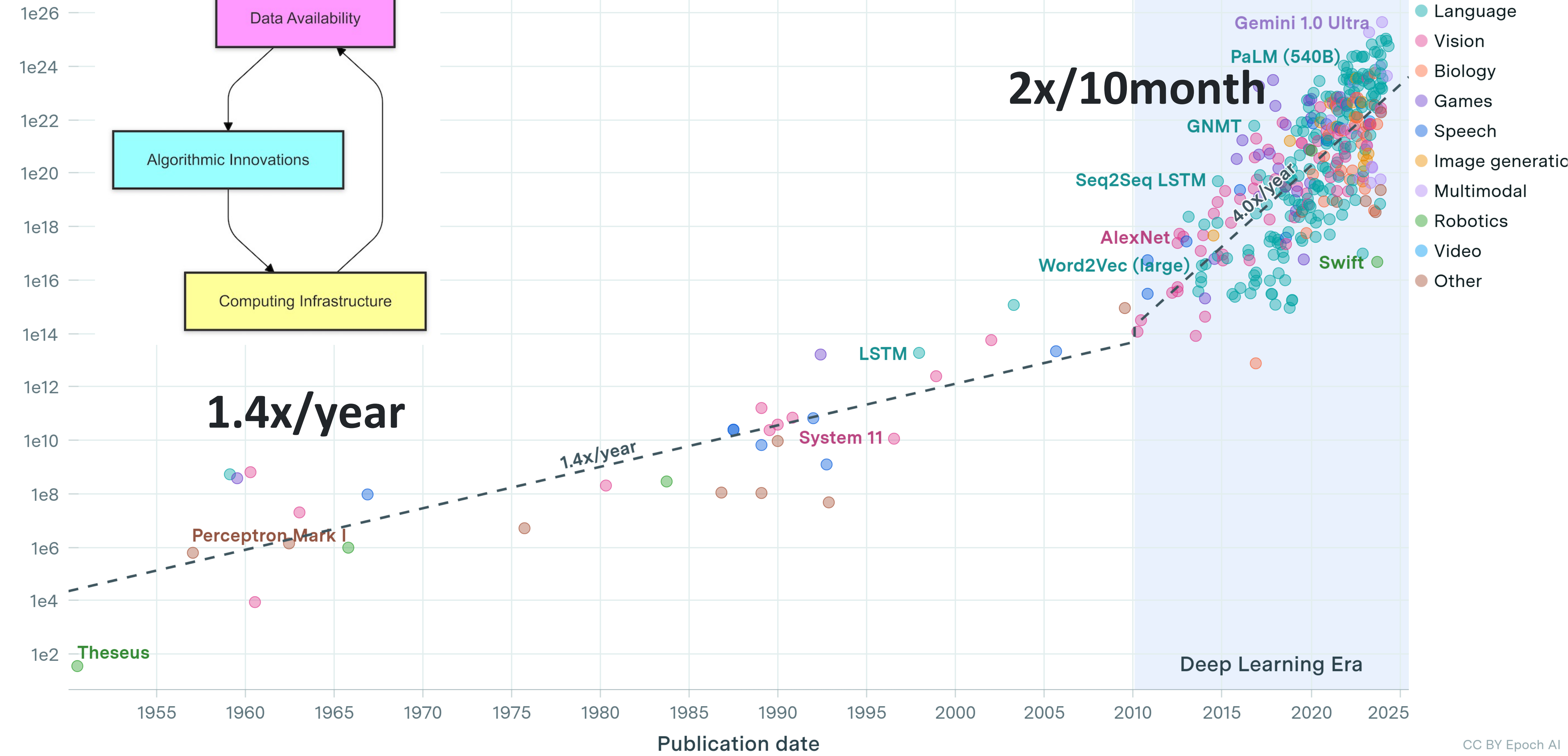
2007



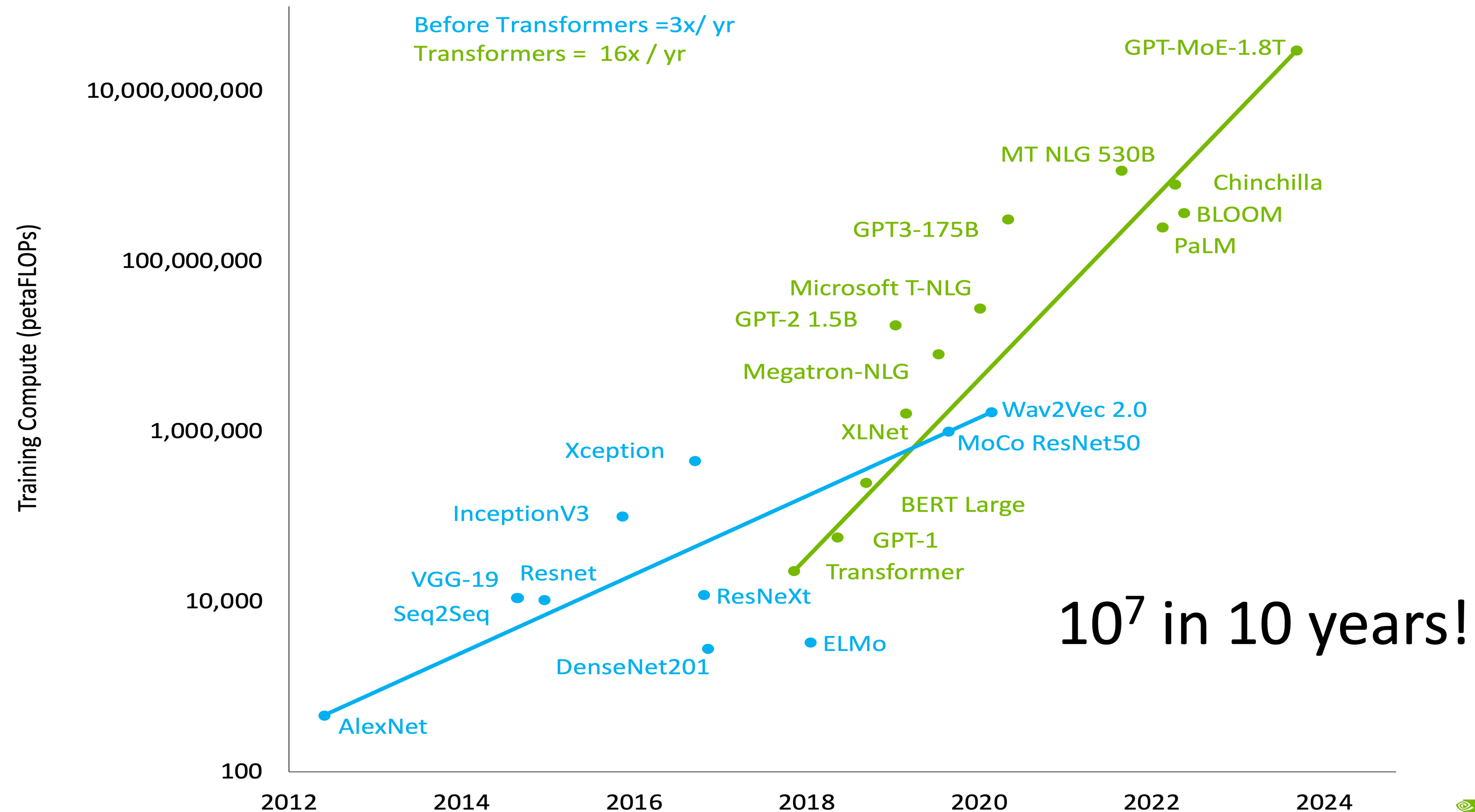
Training compute (FLOP)



ML to DL



Transformer: 10⁷ demand

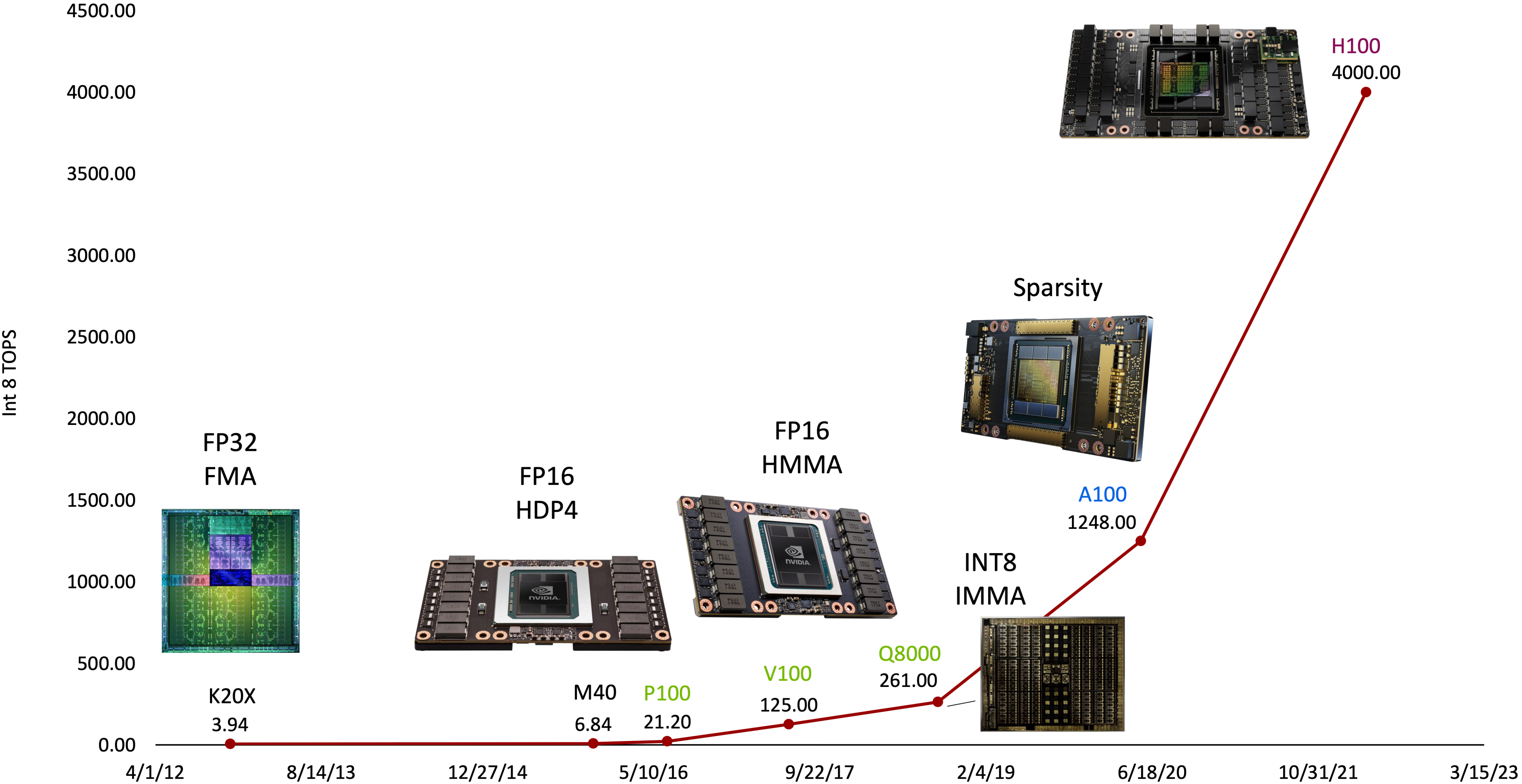


Computer Architect's View

The computer architect's role is to bridge the gap between hardware constraints and the computational needs of modern AI workloads by developing architectures that maximize efficiency, scalability, and performance.

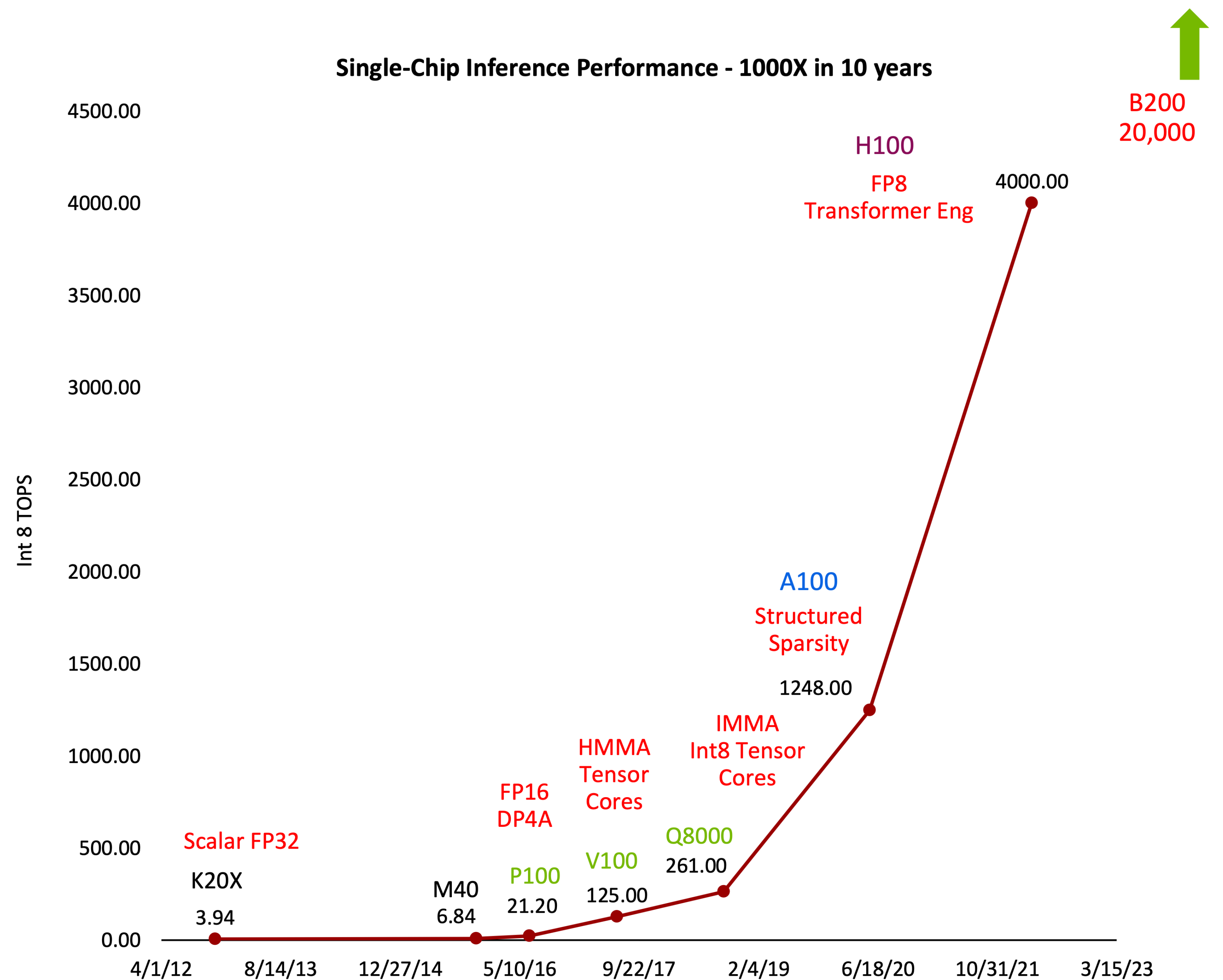
- **Specialized AI accelerators:** Optimizing GPUs, TPUs, NPUs for ML computation.
- **Memory hierarchy design:** Improving HBM, cache optimizations, SRAM/DRAM bandwidth to handle the massive data movement required for ML training.
- **Interconnect innovations:** Enhancing NVLink, PCIe, InfiniBand, NVSwitch to scale large distributed training workloads across thousands of GPUs.
- **Power and efficiency trade-offs:** Designing low-power AI chips while maintaining high throughput for inference and training.

Single-Chip Inference Performance - 1000X in 10 years

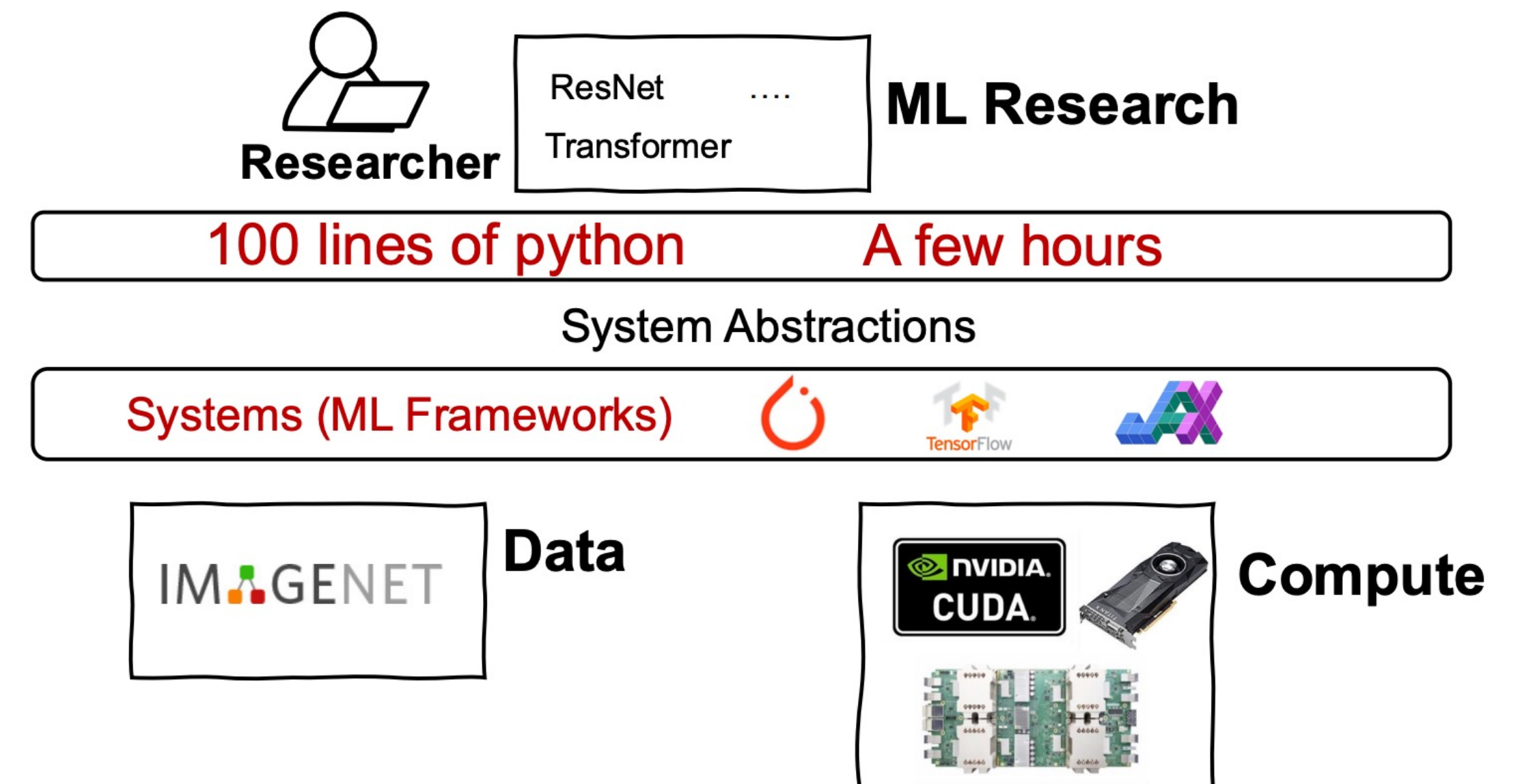
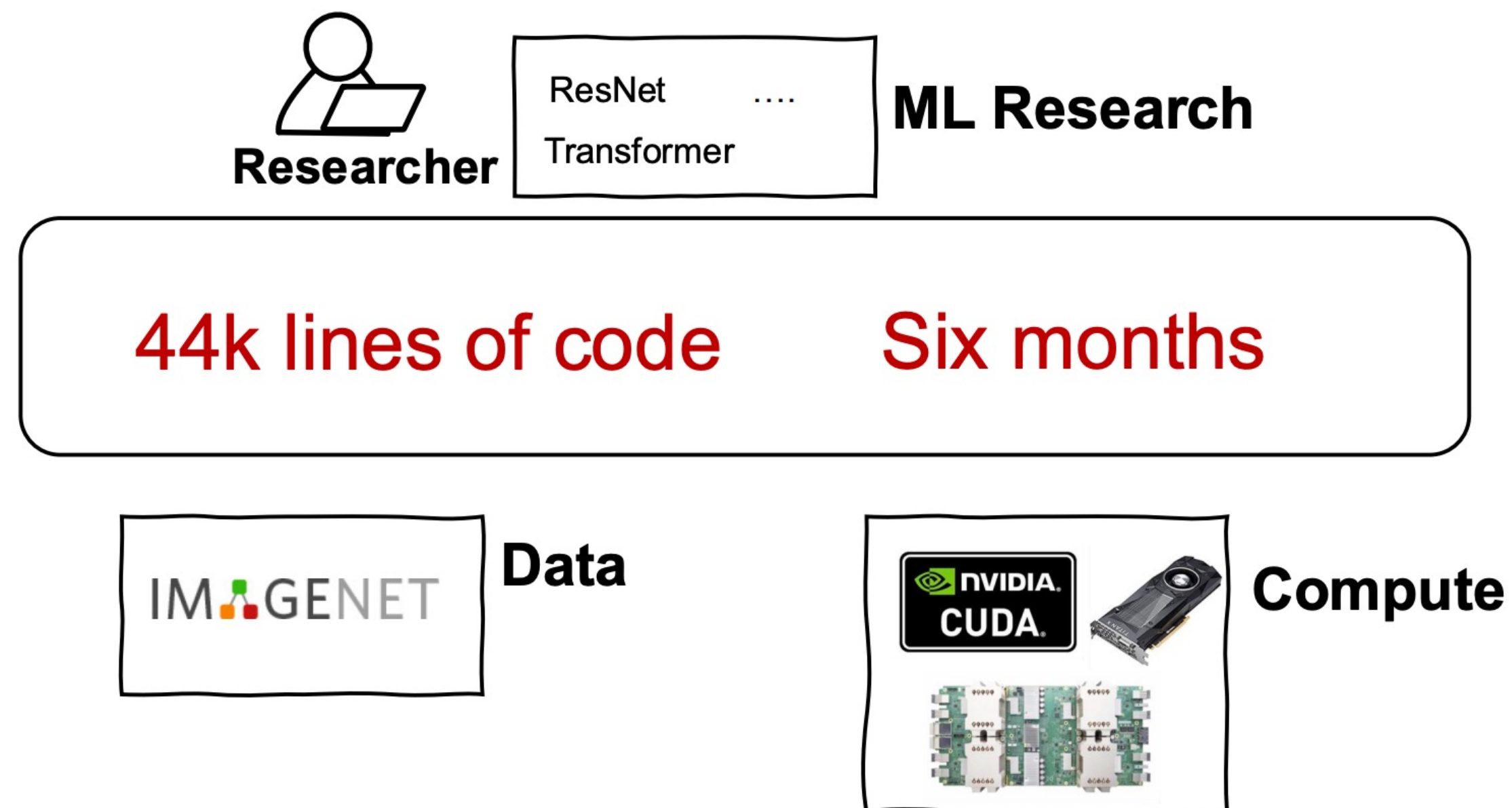


Gains from

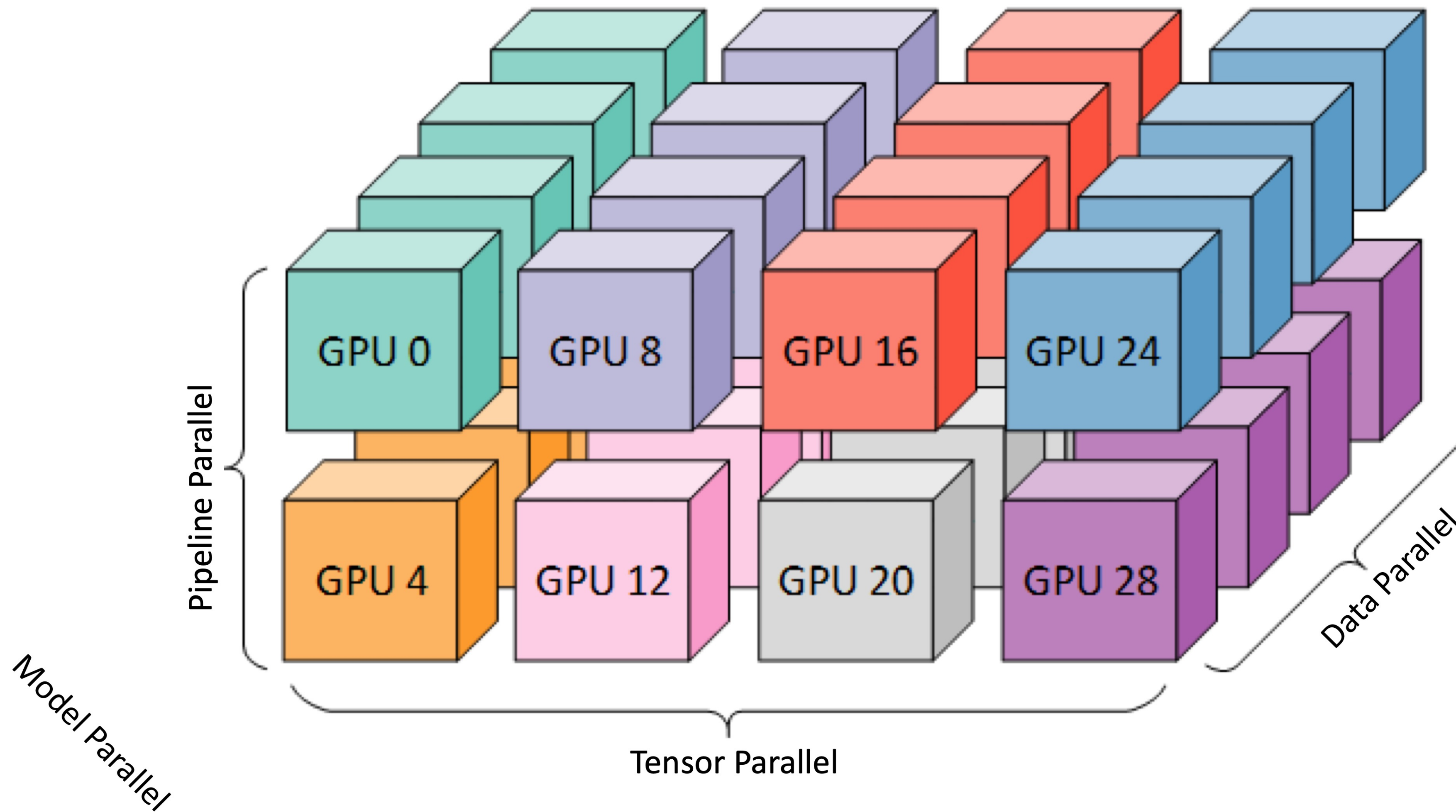
- Number Representation
 - FP32, FP16, Int8, **FP4**
 - (TF32, BF16)
 - ~16x, **32x**
- Complex Instructions
 - DP4, HMMA, IMMA
 - ~12.5x
- Process
 - 28nm, **16nm**, **7nm**, **5nm**, **4nm**
 - ~2.5x, **3x**
- Sparsity ~2x
- Die Size **2x**
- Model efficiency has also improved – overall gain > 1000x



Model Developer's View



The other 1000x



It takes 20 GPUs to hold one copy of GPT4 model parameters

Algorithm Designer's View

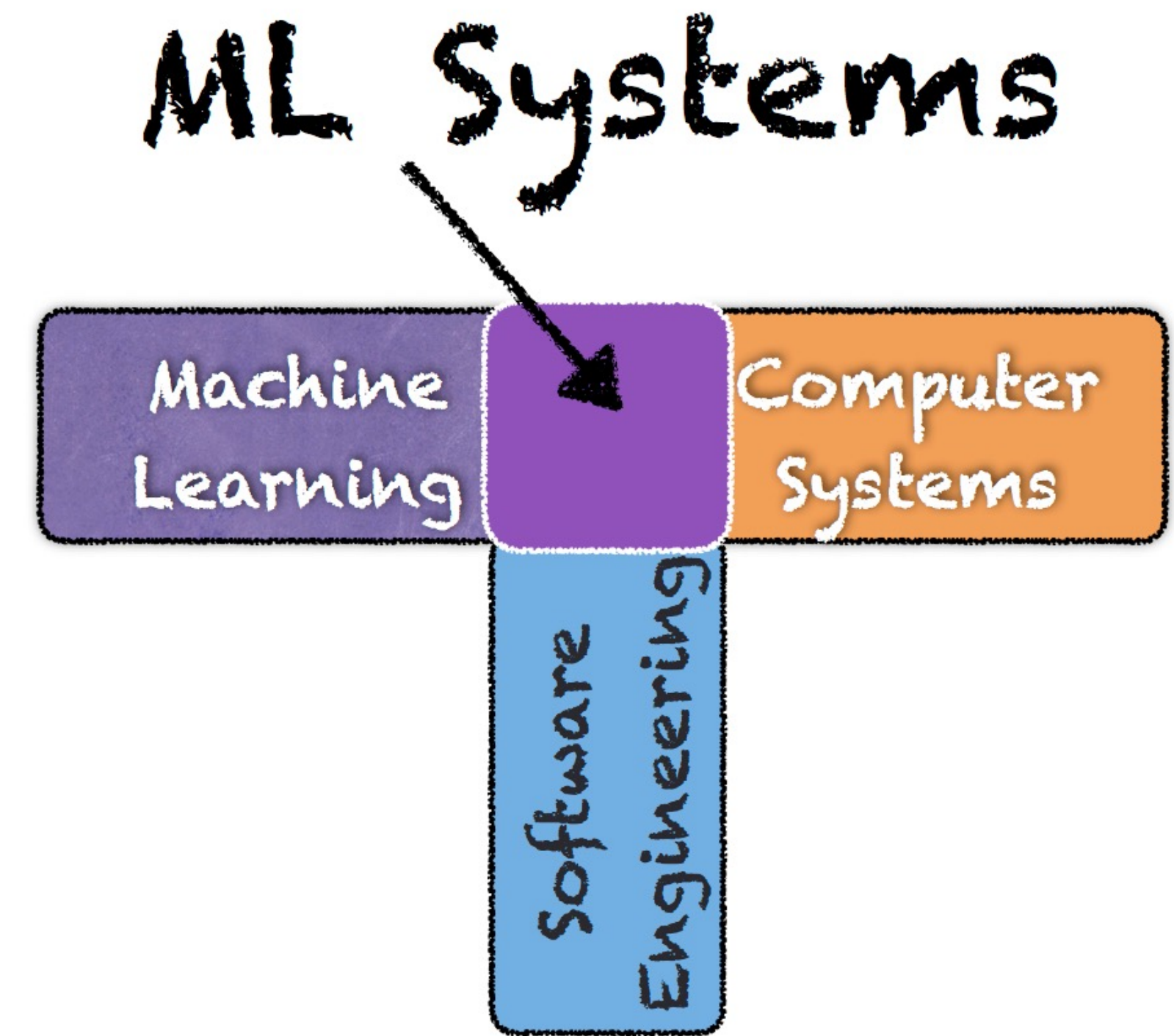
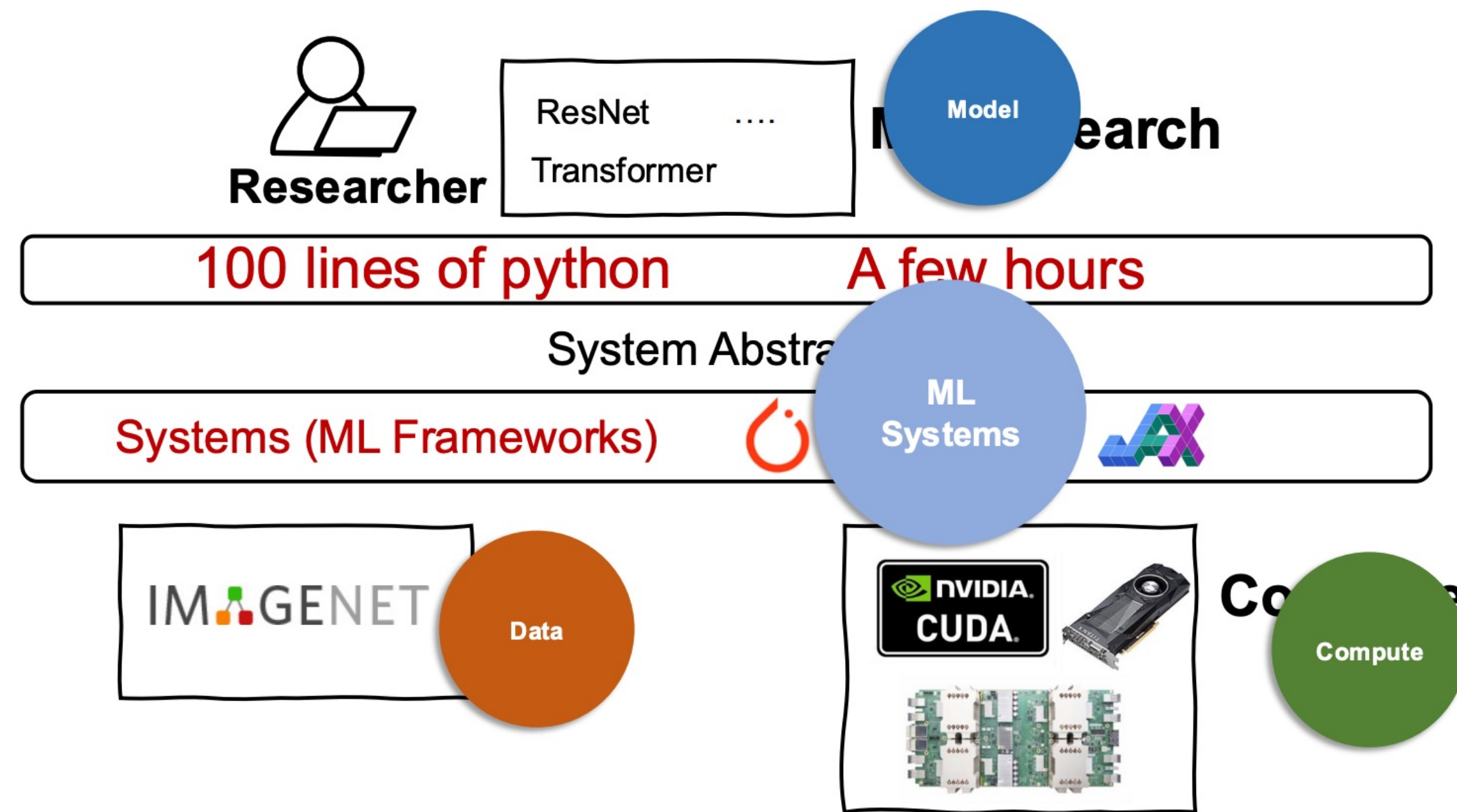
Machine Learning Algorithm Designer's View: **How to** design ML algorithms **that are not only** intelligent **but also** computationally efficient.

This includes:

- **Model efficiency techniques:** Quantization, pruning, knowledge distillation to reduce compute/memory footprint.
- **Data pipeline:** Techniques that cleans and tokenizes text, packs and shards it into optimized formats, streams it reliably, and monitors throughput to avoid wasted compute.
- **Training and inference optimization:** Techniques like parallel training, mixed-precision training, and batch scheduling to improve execution on real hardware.
- **Hardware-aware algorithm design:** Creating ML models that can efficiently run on modern architectures without excessive resource consumption.

A Holistic View of Machine Learning Systems

Bridging the theoretical principles and practical implementation



Instead of treating ML as separate layers (algorithms, software, and hardware), we must view it as a vertically integrated stack.

Three views of AI infrastructure scalability

Hardware, systems software, and model structure impose different limits

01 Hardware architecture

Which physical resources limit throughput and efficiency?

- Accelerators and numerical units
- Memory capacity and bandwidth
- Scale-up and scale-out networks
- Power and thermal limits

02 Systems software architecture

How should computation and data be mapped and coordinated?

- Kernels, compilers, and runtimes
- Parallel and distributed execution
- Communication and resource scheduling
- Reliability and observability

03 Model scalability

Which model structures determine the resources needed to scale?

- Numerical precision
- Latent dimension and usable capacity
- Context and memory management
- Sparse and conditional computation

Scalability emerges from co-design

Model requirements define the workload. Systems map it to hardware. Hardware sets the physical envelope.

Machine Learning Systems

ML Sys. is the discipline of designing, implementing, and operating artificially intelligent systems across computing scales—from resource-constrained embedded devices to warehouse-scale computers.

It focuses on algorithm-software-hardware co-design to create systems that are efficient, scalable, and reliable for their deployment context.

It encompasses the complete lifecycle of AI applications: from requirements engineering and data collection through model development, system integration, deployment, monitoring, and maintenance.

Machine Learning Systems

Through lectures and hackathons, students will build, profile, debug, and optimize modern AI systems using quantitative and reproducible methods. They will also explore agentic workflows for profiling, kernel optimization, experiment management, and training and inference infrastructure, with an emphasis on verifying correctness, performance, and reliability.

Industry partners such as Huawei, Tencent, and ByteDance may contribute project opportunities. Final partners, topics, awards, and internship or research opportunities will be announced on the course website.

Not every student will become a great AI infrastructure architect, but a great architect can emerge from any student.

Survey

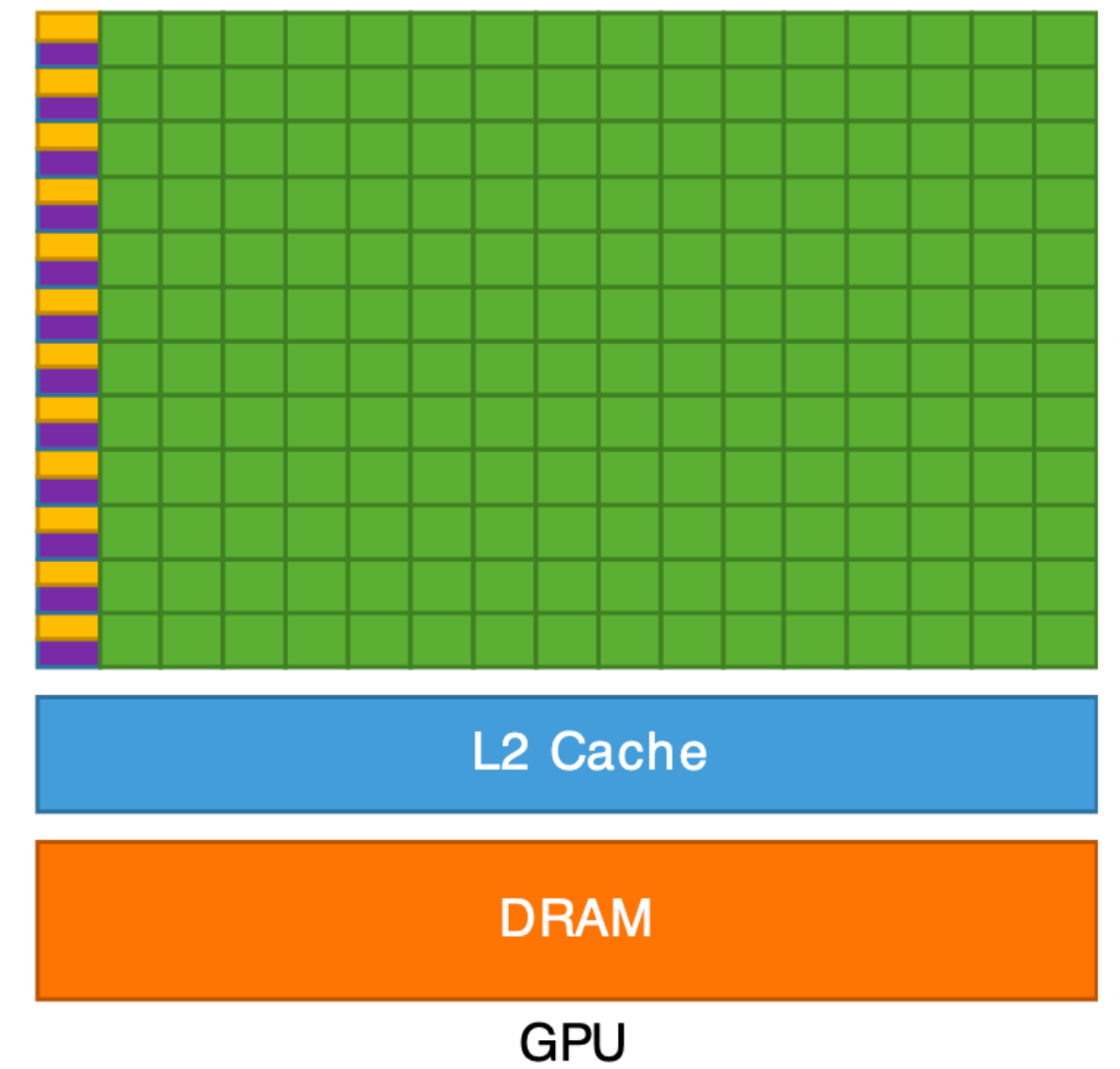
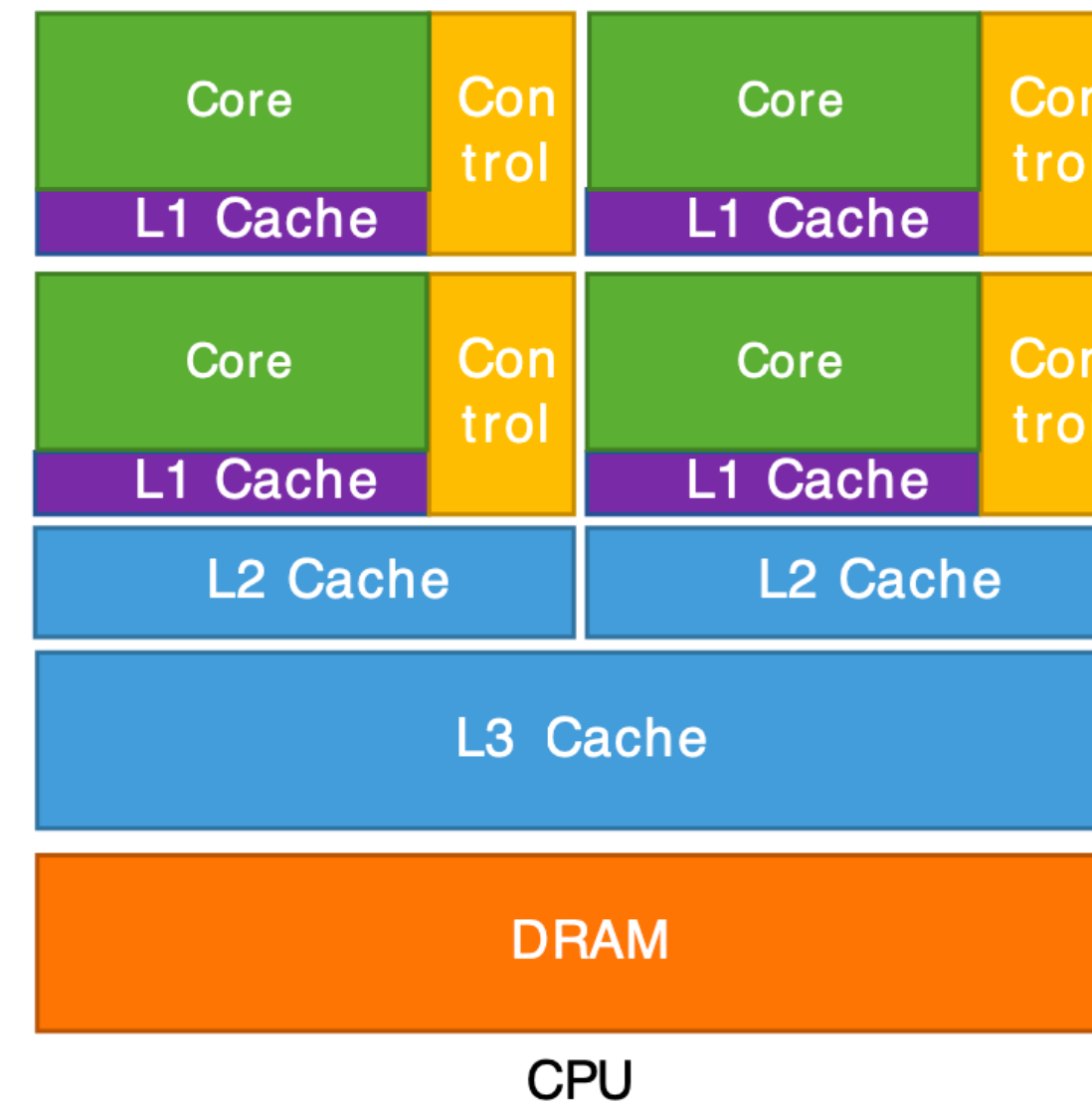
- Research interest: LLM, embodied AI/vision, AI Infra
- Academic background in ML: CUDA, ML compiler, LM training & inference

Tentative Schedule

Week	Theme	Format	Topic
1	Introduction	Lecture	The Intellectual Map of AI Infrastructure
2	GPU/NPU Architecture	Lecture	GPU Architecture
3	GPU/NPU Architecture	Lecture	NPU Architecture
4	CUDA Programming	Lecture	CUDA Programming Through the Lens of GPU Architecture
5	CUDA Programming	Lecture	CUDA Programming as Hardware–Software Co-design
6	GPU Kernel Development	In-class Hackathon	Agentic CUDA Kernel Development and Optimization
7	NPU Kernel Development	In-class Hackathon	Agentic NPU Kernel Development and Optimization
8	NPU Kernel Development	In-class Hackathon	Agentic FlashAttention Development and Optimization on NPUs
9	LLM Training	Lecture	LLM Training Systems A–Z: Data, Tensor, Pipeline, and Expert Parallelism
10	LLM Training	Lecture	Advanced Topics in Large-scale LLM Training Systems
11	LLM Training	In-class Hackathon	Agentic Communication-kernel Development and Optimization
12	LLM Training	In-class Hackathon	Agentic Parallel-training Infrastructure Development and Optimization
13	LLM Inference	Lecture	LLM Inference Systems A–Z: Performance Optimization
14	LLM Inference	In-class Hackathon	Agentic LLM-inference Infrastructure Development and Optimization

GPU Architecture

- GPU: Architectural fundamentals
- Modern GPU architecture and deep learning
- Optimization Strategies in GPU Computing
- Advanced GPU Features and Future Directions



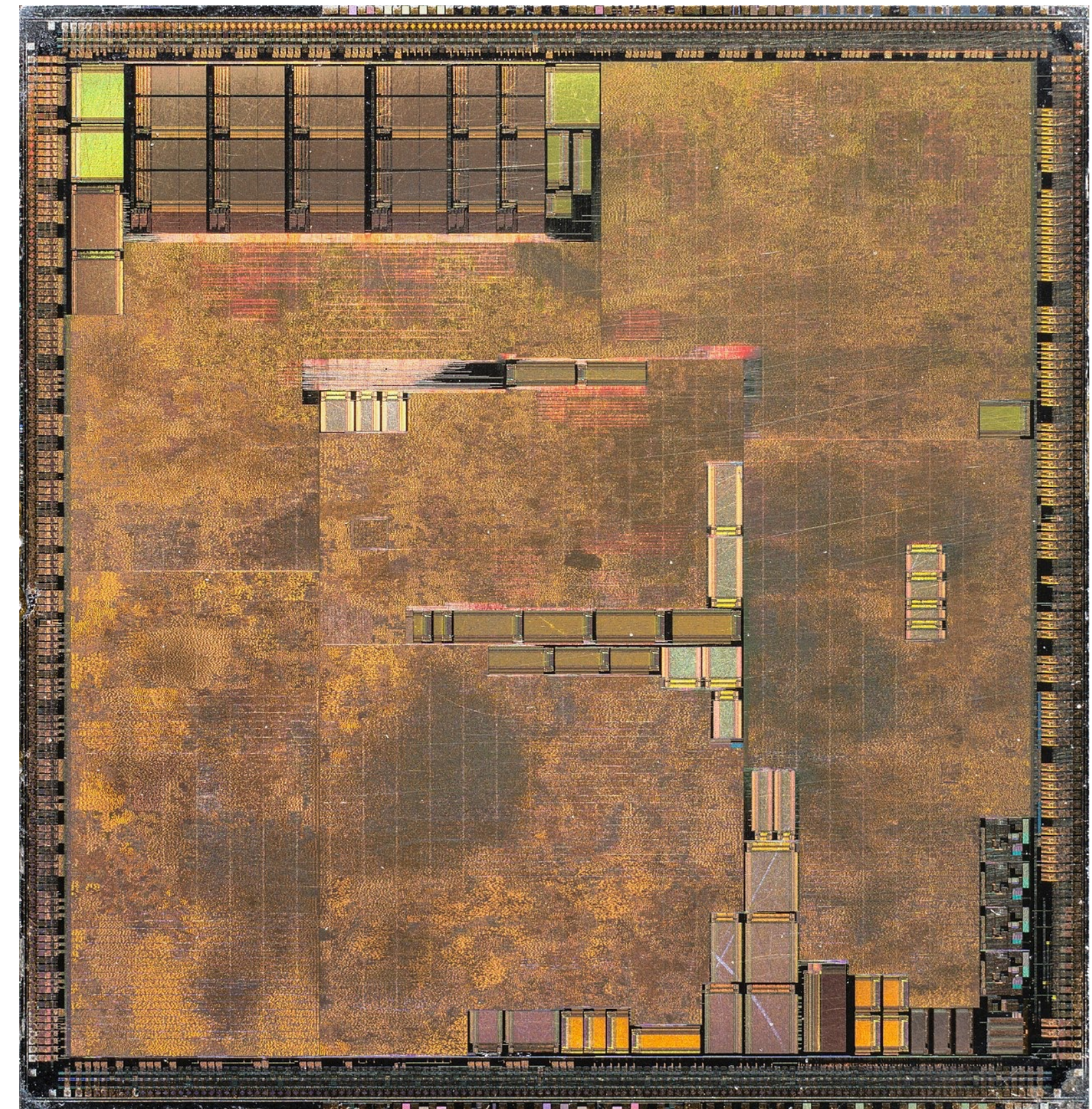
GPU: Introduction and Historical Overview

- GPU evolution: From fixed-function accelerator to programmable processor.
- Early adoption in machine learning: The massive parallel nature suitable for linear algebra.
- GPU in the deep learning : Architecture evolution driven by rapid growth of deep learning.

Phase	Era	Key Features	Examples
1 Hardwired GPU (Fixed-Function Graphics)	1980s – Early 2000s	- Specialized fixed-function pipelines for rendering - No programmability, only hardware-based transformations & rasterization	- 1981: IBM CGA (First consumer graphics card) - 1999: NVIDIA GeForce 256 (First GPU)
2 Programmable GPU (Shader-Based Graphics Processing)	Early 2000s – 2010s	- Vertex & Pixel Shaders introduced for custom effects - Unified Shader Architecture allows software-defined rendering	- 2001: NVIDIA GeForce 3 (First programmable vertex shader) - 2006: NVIDIA Tesla (First Unified Shader)
3 GPGPU for AI/ML (General-Purpose GPU Computing)	2010s – Present	- CUDA/OpenCL enable parallel computing for AI, HPC - Tensor Cores & AI Accelerators introduced	- 2007: NVIDIA CUDA (GPGPU programming) - 2017: NVIDIA Volta (First Tensor Cores for ML)

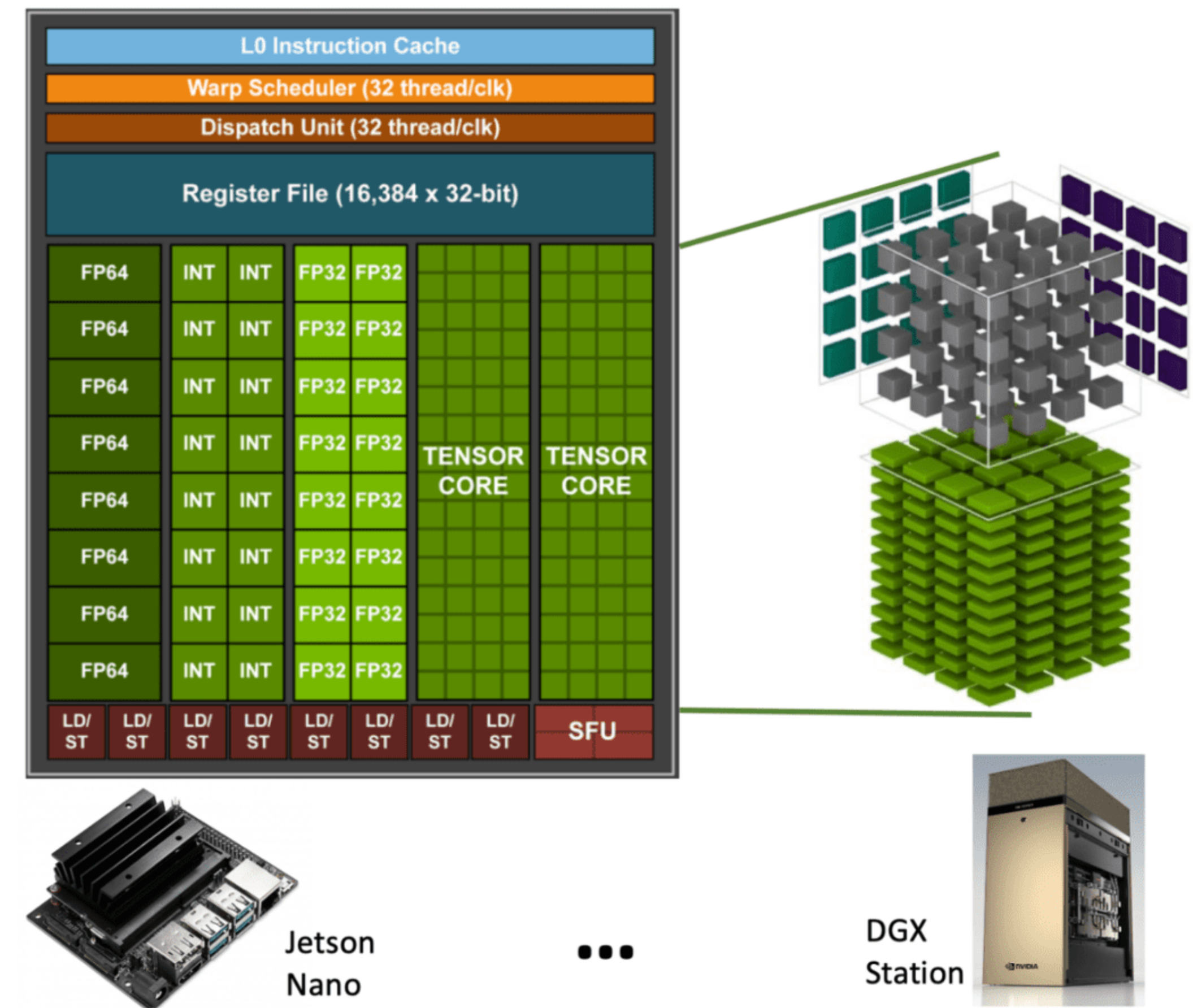
Architectural Fundamentals for Machine Learning

- GPU many-core computing
 - Unified graphics and parallel computing
 - CUDA cores, tensor cores, SFU
 - Warp Scheduler & control logic
 - Interconnect Network
- GPU memory hierarchy and bandwidth
 - Global Memory, Shared Memory, Local/Private Memory, Register File
 - L1, L2 Cache, Texture and Constant Memory
- GPU programming model basics



Modern GPU Architectures and Deep Learning

- From CUDA core to Tensor cores
- TensorFloat-32 (TF32) & Bfloat16 (BF16)
- Fine-grained structured sparsity
- Transformer engine
- Memory and interconnect advances
 - High-bandwidth memory (HBM)
 - NVLink and NVSwitch
 - Infiniband
- Multi-instance GPU (MIG)
- Specialized hardware vs GPUs

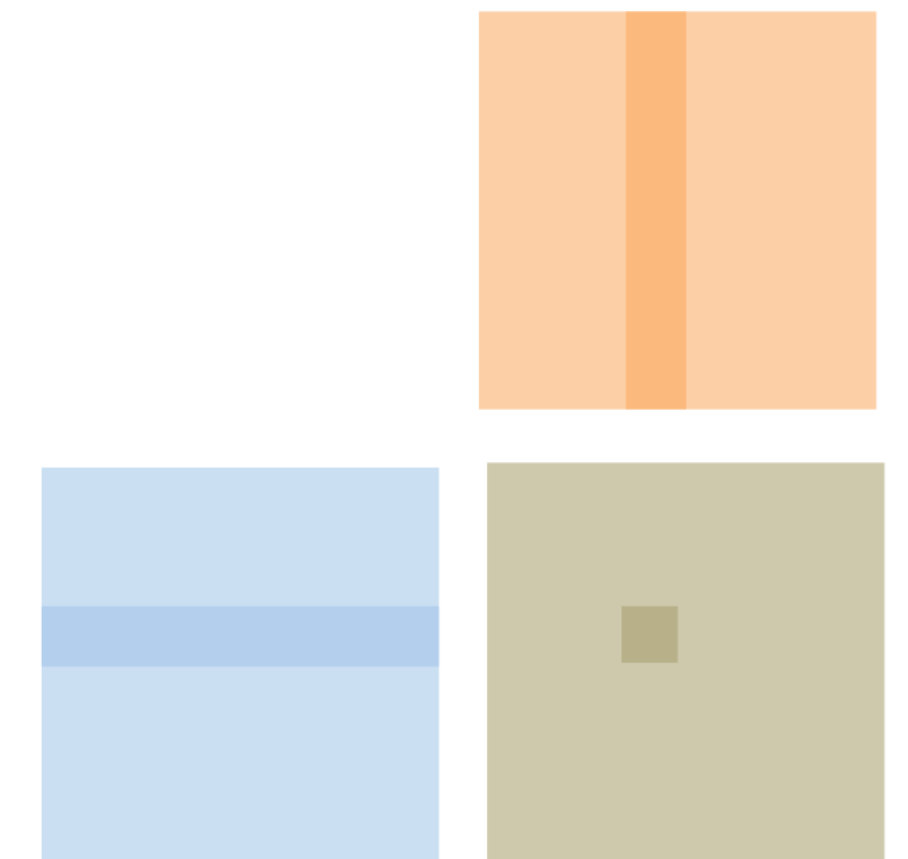


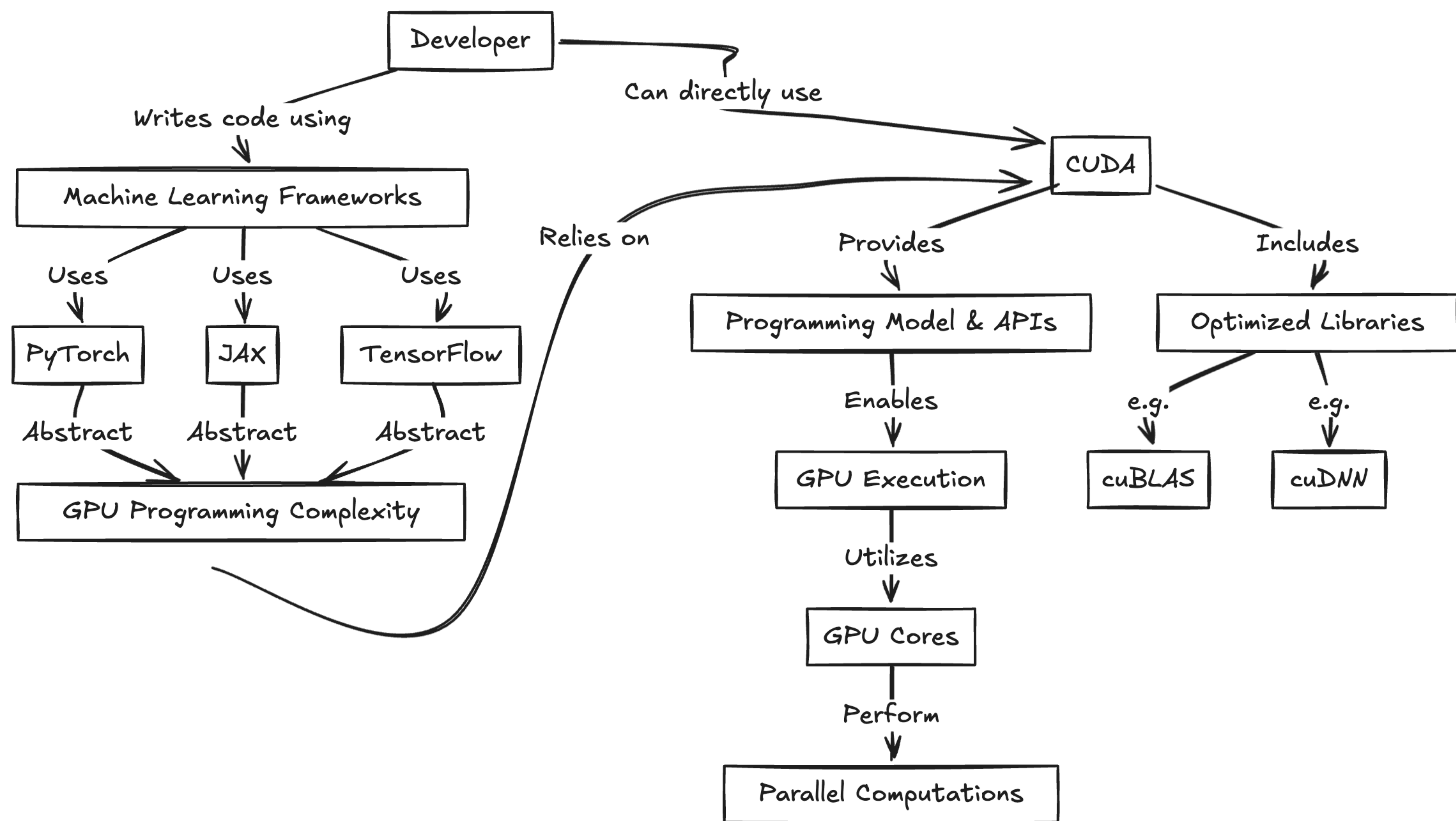
Optimization Strategies in GPU Computing

- Understand the performance bottleneck
- Efficient utilization of GPU resources
 - Maximizing parallel workload
 - Memory access patterns
 - Algorithmic optimizations: mixed precision training, overlapping computation and data transfers
- Case study: Matrix Multiplication Optimization
 - Blocking/tiling for efficient memory access
 - Arithmetic intensity considerations

```
void matrixMultiplication ( float* A, float* B, float* C, int WIDTH)
{
    for (i → 0 : WIDTH)
        for (j → 0 : WIDTH)
            for (k → 0 : WIDTH)
                a = Ai;
                b = Bj;
                sum += a * b;

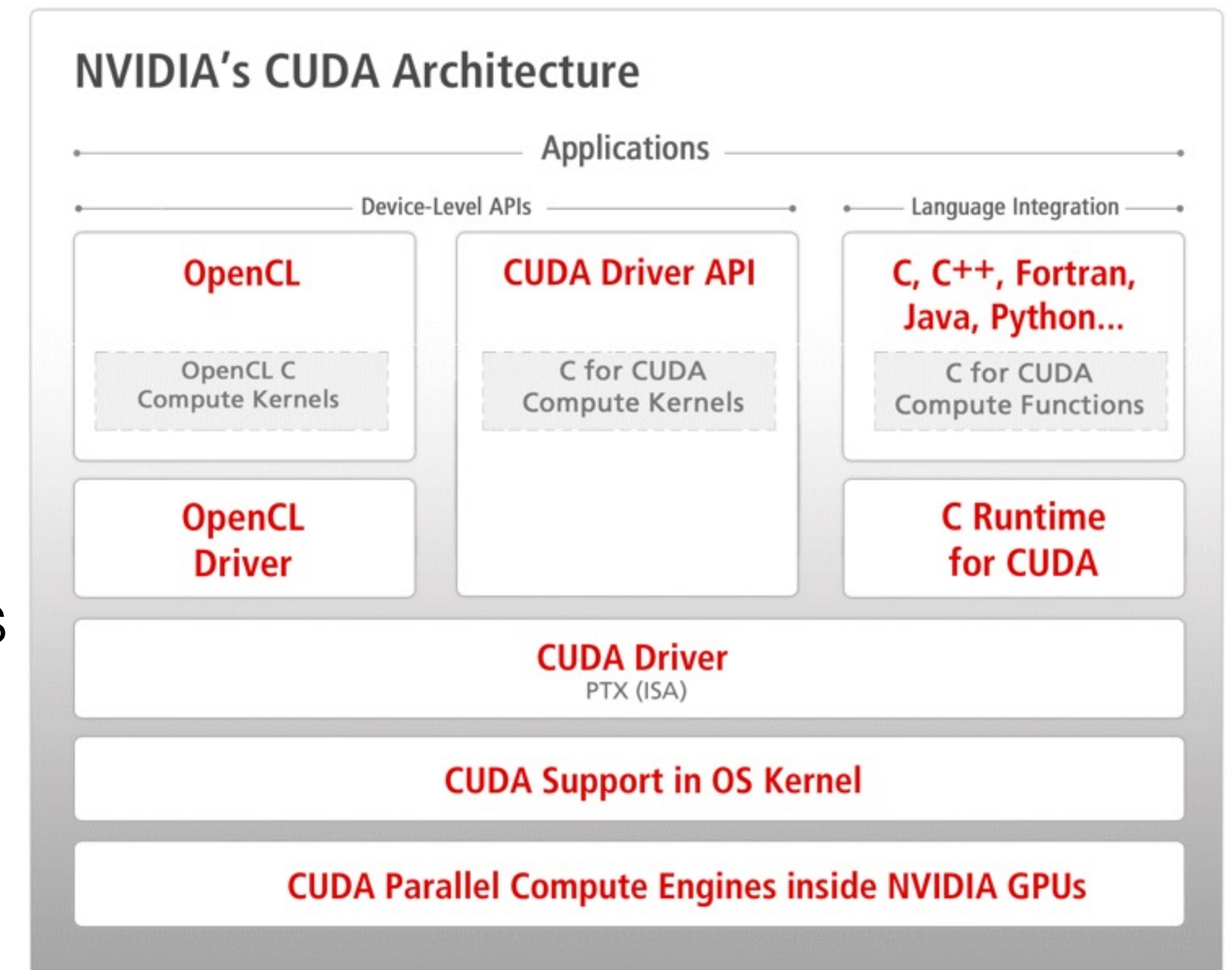
    Cij = sum;
}
```





CUDA Programming

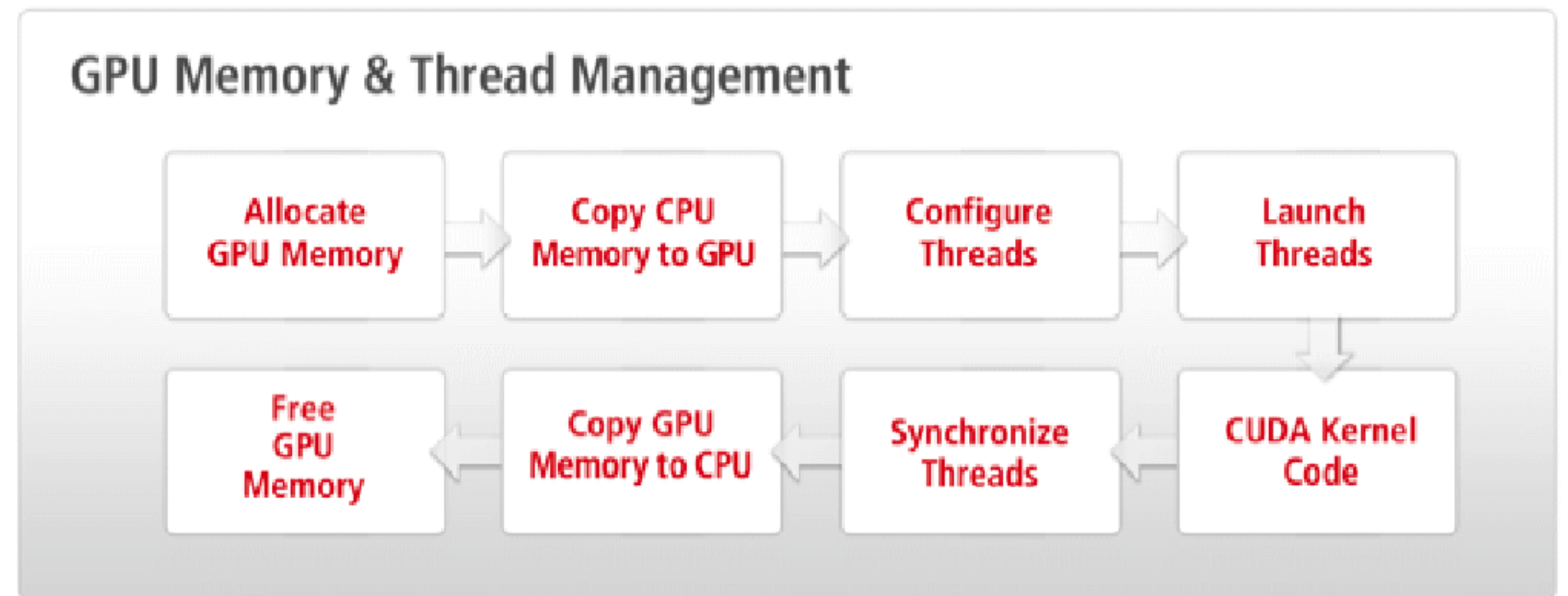
- Part 1: CUDA fundamentals and architecture
- Part 2: Hands-on CUDA program development
- Part 3: Memory management and optimization
- Part 4: Debugging tools and advanced resources
- Part 5: Recap and Q&A



CUDALink is an application of the NVIDIA architecture stack.

CUDA Programming

- Part 1: CUDA fundamentals and architecture
- Part 2: Hands-on CUDA program development
- Part 3: Memory management and optimization
- Part 4: Debugging tools and advanced resources
- Part 5: Recap and Q&A



Typical CUDA program cycle.

Deep Learning Compiler: The old days

- The MxN problems for model deployment
 - Increasing models and hardware backends
 - DNN programs struggle with memory footprints/execution efficiency
- Milestones for deep learning compilers
 - Halide: compute-schedule separation, but in graphics
 - TVM: the first demo of deep learning compilers
 - Ansor: the first demo of auto-schedulers for kernel optimization
 - Roller & Welder: tile-based operator fusion and code generation



Deep Learning Compiler: The old days

TVM: The First Demo of Deep Learning Compilers: TVM, an end-to-end deep learning compiler stack, marked its debut as a pioneering framework for optimizing deep learning models across diverse hardware (e.g., CPUs, GPUs, accelerators) with its first demonstration in 2018. It introduced graph-level and operator-level optimizations, such as high-level operator fusion, memory latency hiding, and hardware-specific code generation, showcasing its ability to outperform frameworks like TensorFlow by automating performance portability for deep learning workloads, starting with its initial release and demonstrations on server-class GPUs, embedded devices, and FPGAs.

Deep Learning Compiler for LLM

Triton, an open-source Python-like language by OpenAI, simplifies GPU programming, enabling efficient, high-performance kernel development for AI/ML workloads with minimal expertise, often outperforming PyTorch equivalents by 2x.

OpenAI's GPU programming framework (Python DSL)

Write NumPy-like code, compile to high-performance GPU kernels

Why Triton?

- CUDA is powerful but complex; Triton makes it simple
- Optimized for deep learning primitives (matmul, softmax, layer norm)
- Integrated into PyTorch 2.0 via TorchInductor

Deep Learning Compiler for LLM

CuTile, Triton, and TileLang are all high-level, Python-based programming models designed to simplify GPU kernel development by abstracting away low-level thread-level operations in favor of "tile-based" operations. While they share a similar mental model—dividing data into multidimensional blocks (tiles) and processing them in parallel—they differ significantly in their level of hardware control, compiler backends, and optimization strategies.

Feature 	CuTile	Triton	TileLang
Primary Goal	Simplified, portable CUDA programming.	High-level abstraction for ML kernels.	Fine-grained control with high productivity.
Memory Management	Automatic memory movement and synchronization.	Implicit memory and parallelism management.	Explicit control over memory hierarchy (SRAM, registers).
Hardware Targets	Strictly NVIDIA (requires CUDA 13.1+).	Multi-vendor (NVIDIA, AMD, Intel).	Multi-vendor (NVIDIA, AMD).
Optimization	Fully automated by Tile IR compiler.	Expert parameterization required for peak performance.	Decouples dataflow from scheduling via primitives.

Deep Learning Compiler for LLM

CUDA-L2: Surpassing cuBLAS Performance for Matrix Multiplication through Reinforcement Learning

Songqiao Su, Xiaofei Sun, Xiaoya Li, Albert Wang, Jiwei Li and Chris Shum

DeepReinforce Team

 github.com/deepreinforce-ai/CUDA-L2

CUDA-L2 demonstrates that LLM-guided reinforcement learning can outperform even NVIDIA's heavily engineered libraries (cuBLAS and cuBLASLt) on half-precision GEMM (HGEMM) kernels across 1000 different (M, N, K) configurations. Its core finding is that an LLM, when trained with execution-time rewards, profiling signals, and contrastive feedback, can autonomously discover hardware-optimized strategies that human experts rarely explore due to the vastness of the search space.

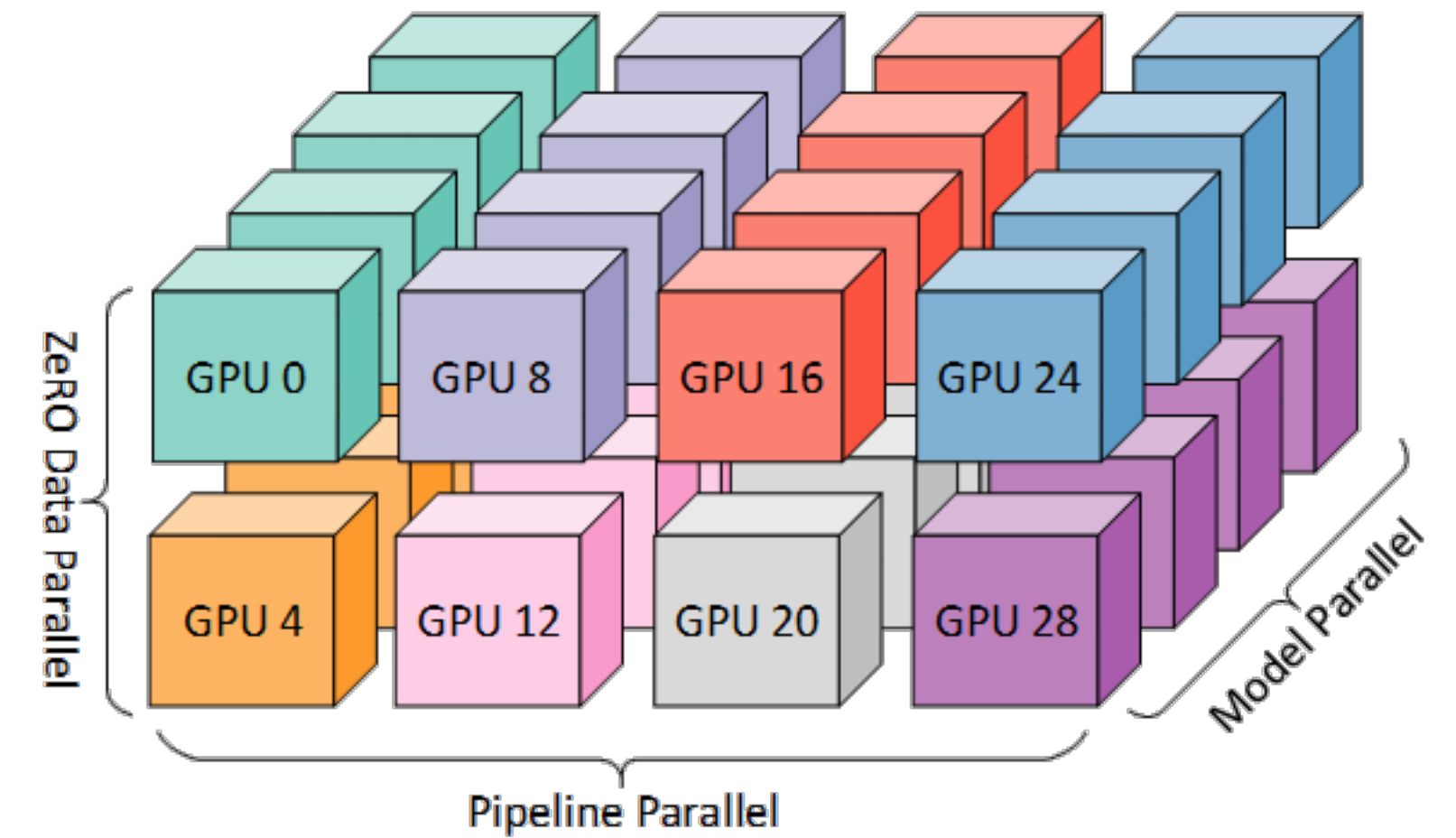
Parallel training

Fundamentals of Large Model Training

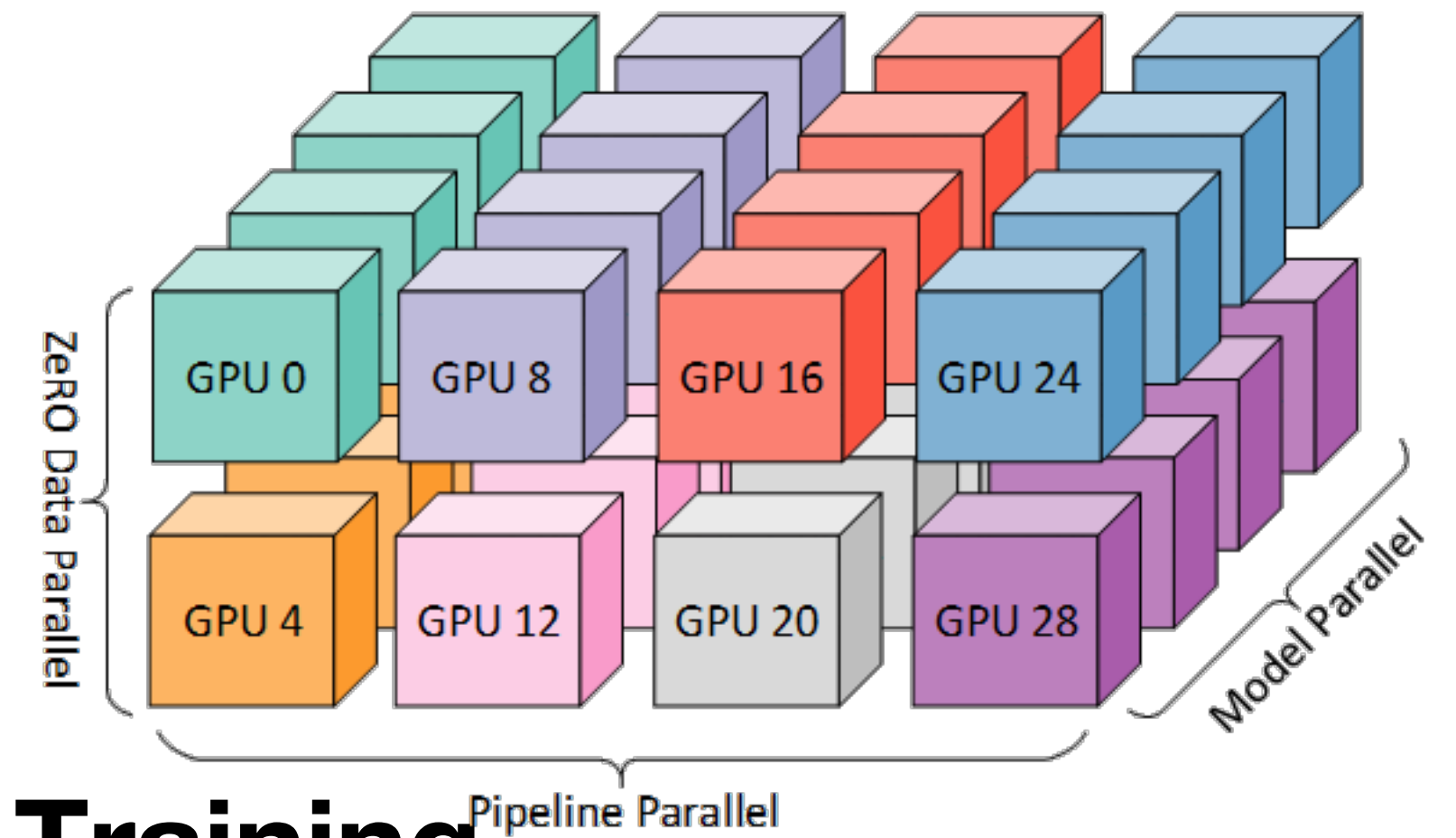
- Review of large-scale neural network training processes
- Introduction to Transformer model training
- Overview of software systems for large model training

Collective Communication for Large Model Training

- Basics of collective communication
- Important collective communication primitives
- Optimization of reduction (Reduce) and all-to-all collective communication



Parallel training



Multi-Dimensional Parallelization in Large Model Training

- Data parallelism, tensor parallelism, pipeline parallelism
- Expert parallelism, sequence parallelism, context parallelism

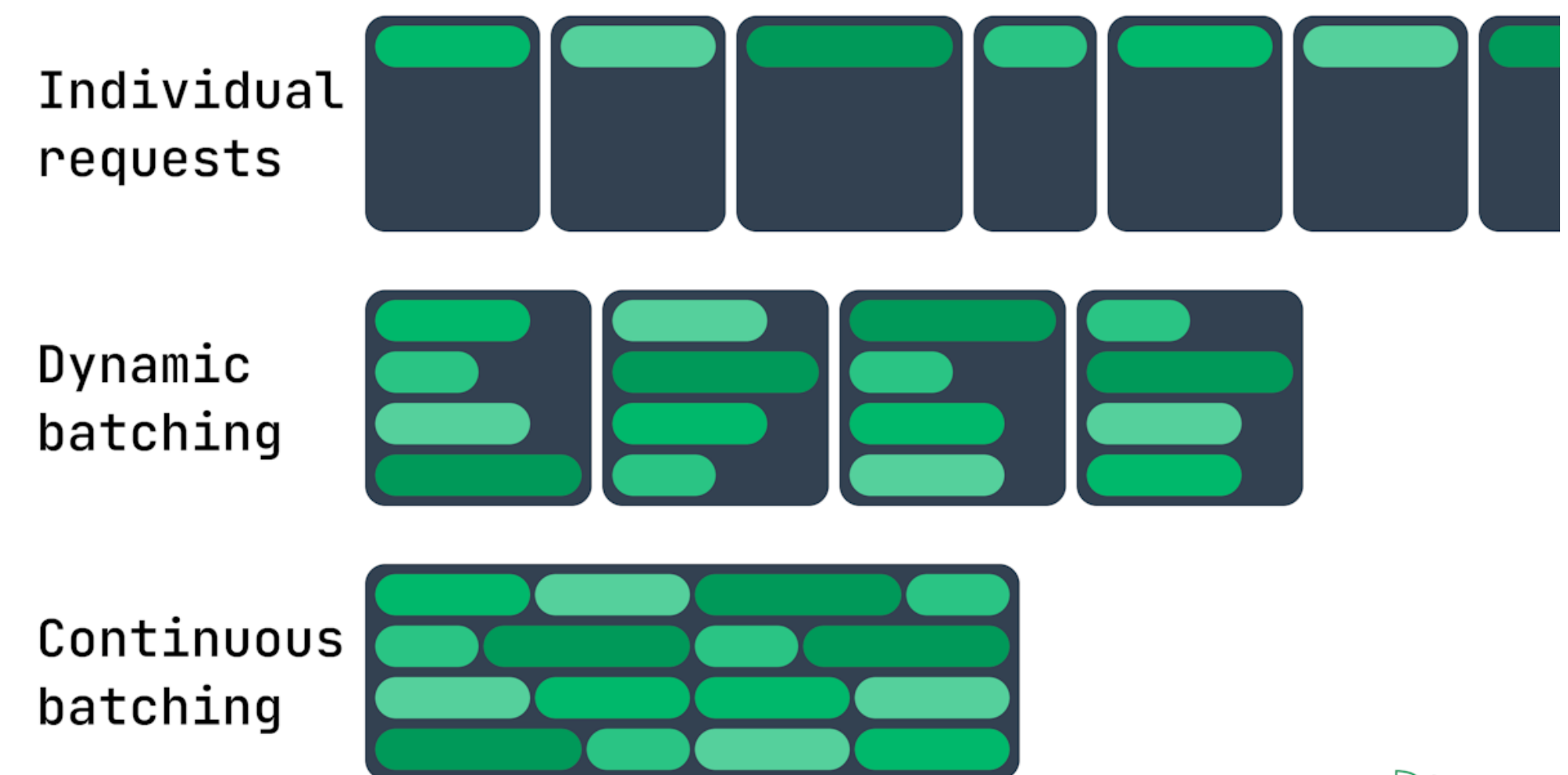
Memory Optimization and Acceleration in Large Model Training

- Re-computation methods
- Memory optimization techniques for data parallelism
- Typical methods for overlapping computation and communication

LLM Inference

- Challenges
 - Generative Model: Prefill and Decode, KV Cache
 - Resource: Computation-bound and Memory-bound
 - Service Level Objectives
- Modeling
 - Architect: MoE (Mixture-of-experts) and its variants.
- Distributed System
 - Parallism: Tensor-Pipeline-Expert Parallelism
 - Concurrency: Batching, Caching

Batching strategies for LLM inference

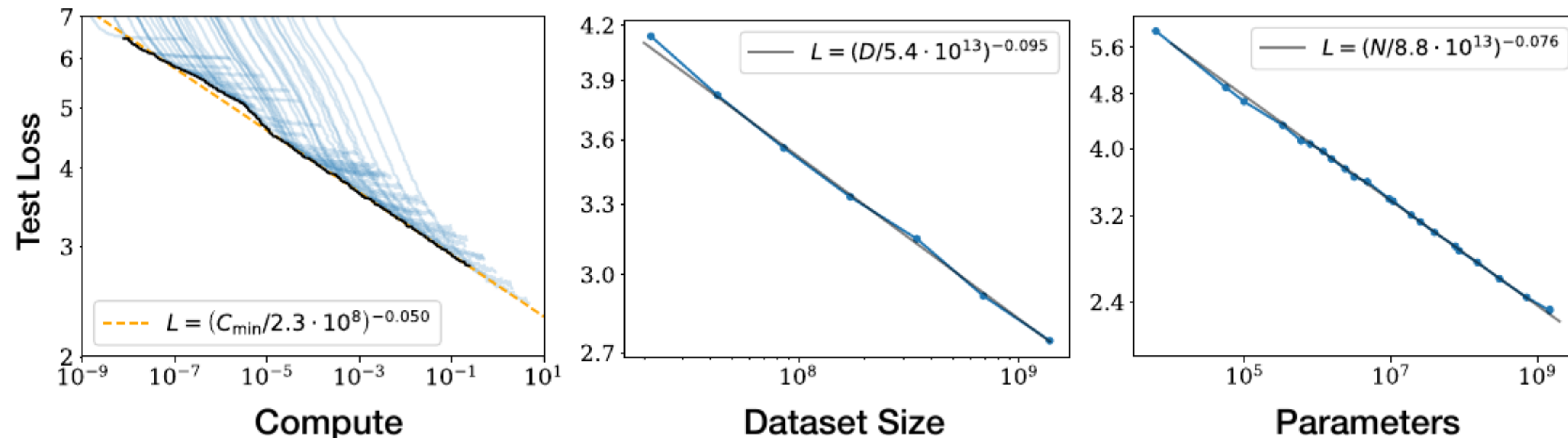


The challenges ahead

–Tom Jerry

Scalability: The first principle of LLMs

- Large language models begin with a deliberately weak prior: the meaning of a word is determined by its context.
- The opportunity is empirical. The infrastructure question is how far we can extend it before cost or information loss becomes the bottleneck.



Kaplan et al. (2020), Fig. 1 — language-model test loss as model size, data, and training compute vary.

As LLMs scale further, the design freedom of AI infrastructure is whatever still improves performance and cost at larger model, context, and deployment scale.

This talk examines three limits that constrain that freedom, i.e., data resolution, latent dimension, and KV cache, and highlights the opportunities and challenges towards scalable AI Infra structure design.

Three design freedoms, and three resource walls

SCALING AXIS	VALUE	COST	
Precision, <i>b</i>	Preserve feature numerical accuracy.	Represent values with less distortion.	Storage, traffic, and arithmetic scale with bits per value.
Latent width, <i>d</i>	Give features more space to separate.	Reduce geometric interference.	Dense $d \times d$ transformations use parameters and compute proportional to d^2 .
Context window, <i>T</i>	Keep relent context accessible.	Support retrieval, synthesis, and long-range reasoning.	KV state grows with T ; dense prefill work grows approximately with T^2 .

Verifiable freedom names the preserved capability, saved resource, tested scale, and failure boundary.

Precision *b*: The low-hanging fruit, so far

Numerical representation became a major source of hardware efficiency.

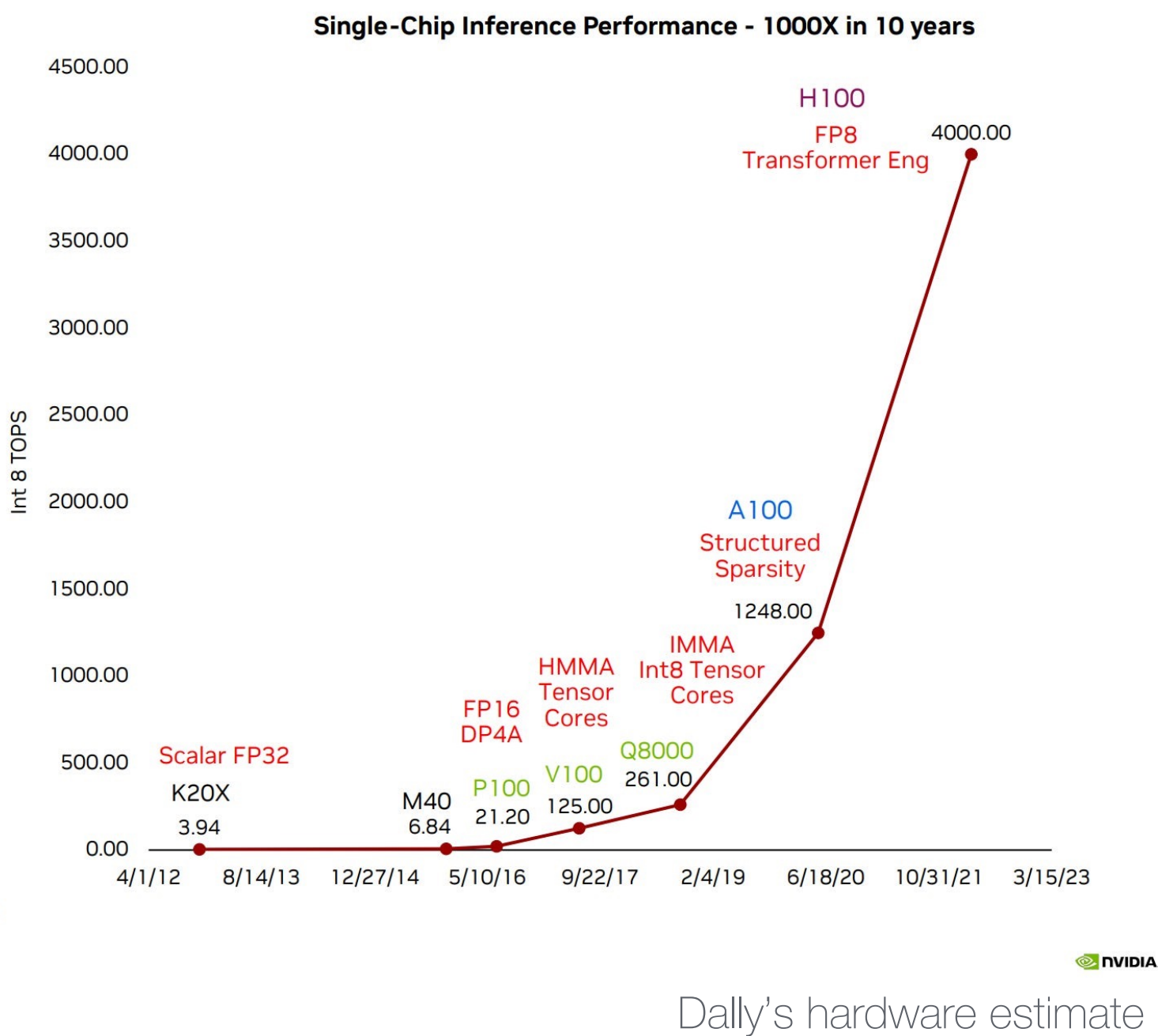
Across a decade of accelerator development, single-chip inference throughput rose by roughly 1,000x through a combination of lower precision, tensor instructions, process scaling, sparsity, and model efficiency.

~32x lower arithmetic energy, FP32 → FP4

Fewer bits reduce the cost of arithmetic, and more importantly, the bytes that must move through the memory hierarchy.

Gains from

- Number Representation
 - FP32, FP16, Int8
 - (TF32, BF16)
 - ~16x
- Complex Instructions
 - DP4, HMMA, IMMA
 - ~12.5x
- Process
 - 28nm, 16nm, 7nm, 5nm
 - ~2.5x
- Sparsity
 - ~2x
- Model efficiency has also improved – overall gain > 1000x

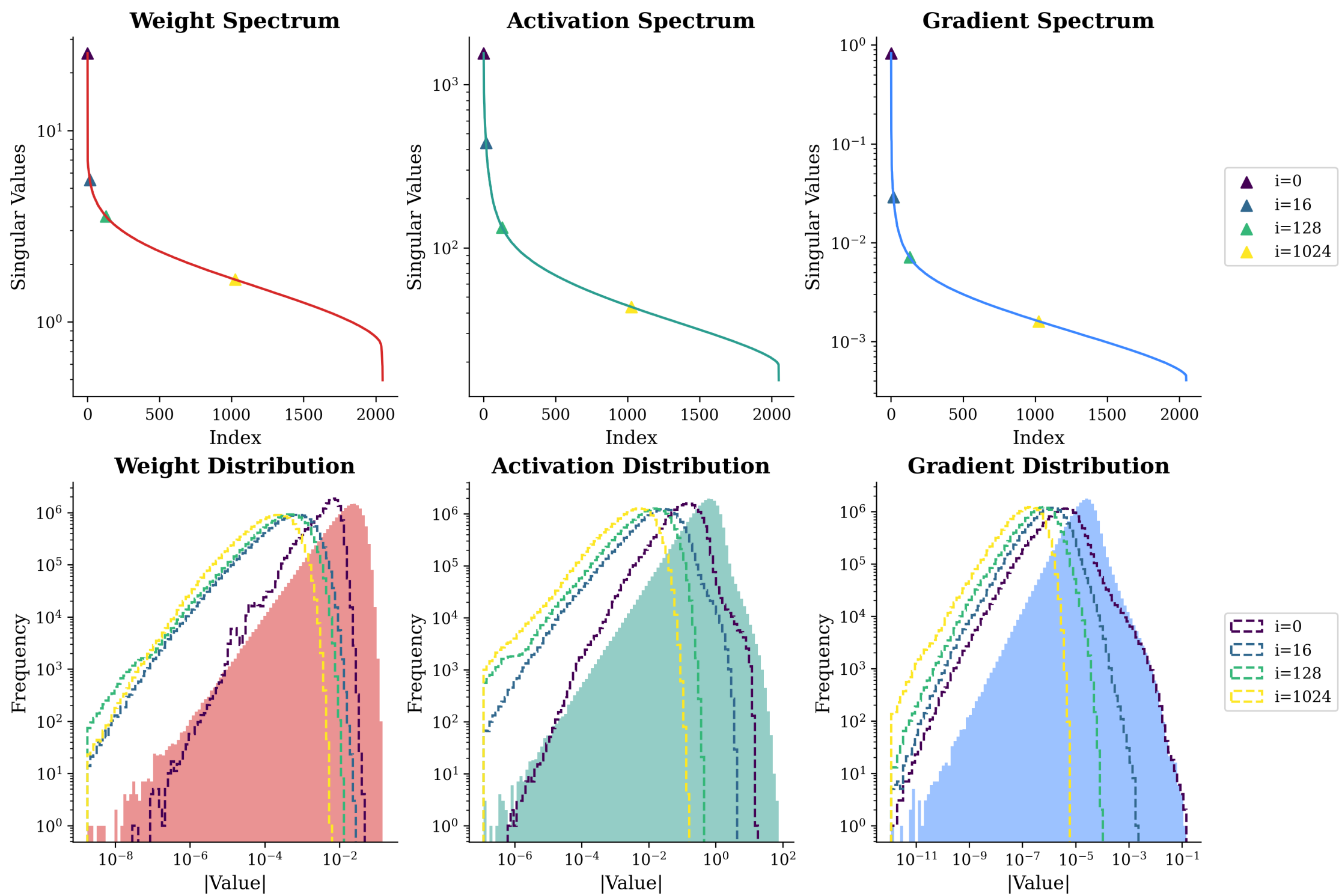


The frontier is no longer whether low precision saves cost. It is whether the signal still survives.

FP4: Low-precision SOTA

10¹⁰+

weights · activations · gradients



2015–2020 Learn around quantization noise

Straight-through estimation · quantization-aware training · stochastic rounding

2021–2023 Protect salient weights and outliers

GPTQ reconstructs weights · AWQ protects salient channels
SmoothQuant shifts activation outliers into weights

2024–NOW End-to-end LM pre-training with FP4

Metis: End-to-end FP4 LM pre-training by isolating dominant spectral components and quantizing the narrower residual.

outlier sets one block scale; small values lose codes, or round to zero

How can we present 1010+ data distribution with a few bits?

Precision scalability towards 1 bit: High dimension averages error

A representation uses the geometry of many coordinates; no single coordinate must be exact.

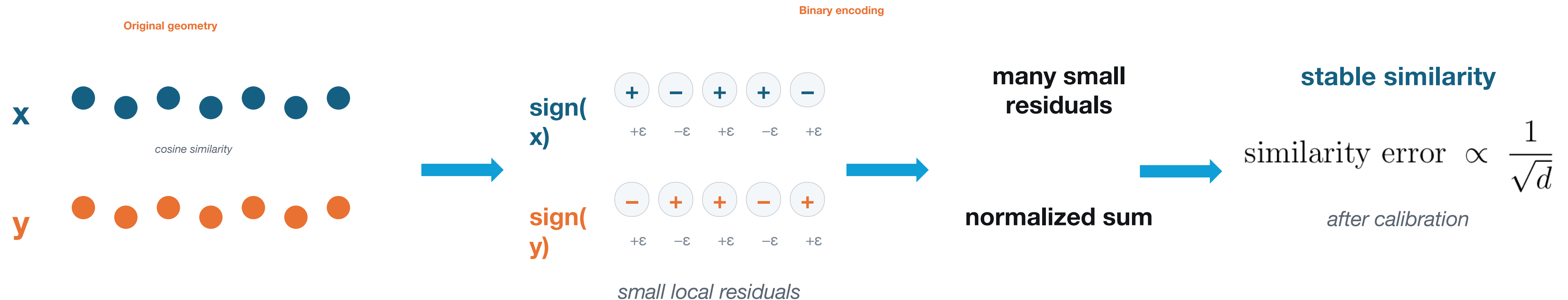
$$\mathcal{B}^m = \{-1, +1\}^m$$

$$\cos(a, b) = \frac{a^\top b}{m} = 1 - \frac{2H(a, b)}{m}$$

One-bit coordinates $\rightarrow$ fine-grained population geometry

For unrelated random bipolar directions, normalized similarity concentrates as dimension grows.

$$S_m = \frac{1}{m} a^\top b, \quad \mathbb{E}[S_m] = 0, \quad \text{Std}(S_m) = \frac{1}{\sqrt{m}}$$



HYPOTHESIS If representation is sufficiently high-dimensional, individual coordinates may not need high numerical precision.

The curse of latent width d

An anisotropic covariance spectrum can leave a wide representation with only a small effective dimension—and therefore weak feature separability.

THEOREM 1 — ISOTROPY IS OPTIMAL

$$r_{\text{eff}} \leq D, \quad \mathcal{I} \geq \frac{1}{D}$$

THEOREM 2 — EFFECTIVE DIMENSION

$$r_{\text{eff}}(\Sigma) = \frac{(\text{tr } \Sigma)^2}{\text{tr}(\Sigma^2)}$$
$$\mathcal{I} = \frac{\text{tr}(\Sigma^2)}{(\text{tr } \Sigma)^2} = \frac{1}{r_{\text{eff}}}, \quad C_{\text{sep}} = r_{\text{eff}}$$

THEOREM 3 — ZIPF COLLAPSE

$$\lambda_i = \frac{c}{i} \implies r_{\text{eff}} \sim \frac{6}{\pi^2} (\log D)^2$$

KEY OBSERVATION

Nominal width is not usable separability capacity. If the spectrum remains Zipf-like, widening can become sharply inefficient even while dense compute continues to grow quadratically.

Separability capacity equals effective dimension

Two independent features interfere according to how unevenly variance is distributed across latent directions.

NORMALIZED SQUARED OVERLAP

$$\mathcal{I} = \frac{\mathbb{E}[(x^\top y)^2]}{\mathbb{E}\|x\|^2 \mathbb{E}\|y\|^2}$$

For independent, zero-mean vectors with covariance Σ

$$\mathbb{E}[(x^\top y)^2] = \text{tr}(\Sigma^2), \quad \mathbb{E}\|x\|^2 = \text{tr}(\Sigma)$$

SEPARABILITY CAPACITY

$$C_{\text{sep}} = \frac{1}{\mathcal{I}} = r_{\text{eff}} = \frac{\text{tr}(\Sigma)^2}{\text{tr}(\Sigma^2)} = \frac{(\sum_i \lambda_i)^2}{\sum_i \lambda_i^2}$$

The eigenvalues λ_i measure variance along the principal directions of Σ .

A width-D representation behaves geometrically like an isotropic space with only r_{eff} usable dimensions.

Isotropy uses latent width most efficiently

For fixed width D , spreading variance equally across directions minimizes expected feature overlap.

THE BOUND

$$r_{\text{eff}} \leq D, \quad \mathcal{I} \geq \frac{1}{D}$$

No covariance spectrum can use more effective dimensions than the representation physically contains.

WHY THE BOUND HOLDS

Cauchy–Schwarz applied to the covariance eigenvalues gives

$$\left(\sum_{i=1}^D \lambda_i \right)^2 \leq D \sum_{i=1}^D \lambda_i^2$$

Dividing by $\sum_i \lambda_i^2$ produces $r_{\text{eff}} \leq D$.

EQUALITY CONDITION

$$\lambda_1 = \cdots = \lambda_D \quad \Longleftrightarrow \quad r_{\text{eff}} = D, \quad \mathcal{I} = \frac{1}{D}$$

Spectral concentration wastes nominal width: anisotropy makes r_{eff} strictly smaller than D .

A Zipf spectrum collapses usable width

Assume the covariance eigenvalues decay as $1/i$. Most added coordinates then carry progressively less variance.

ASSUMPTION

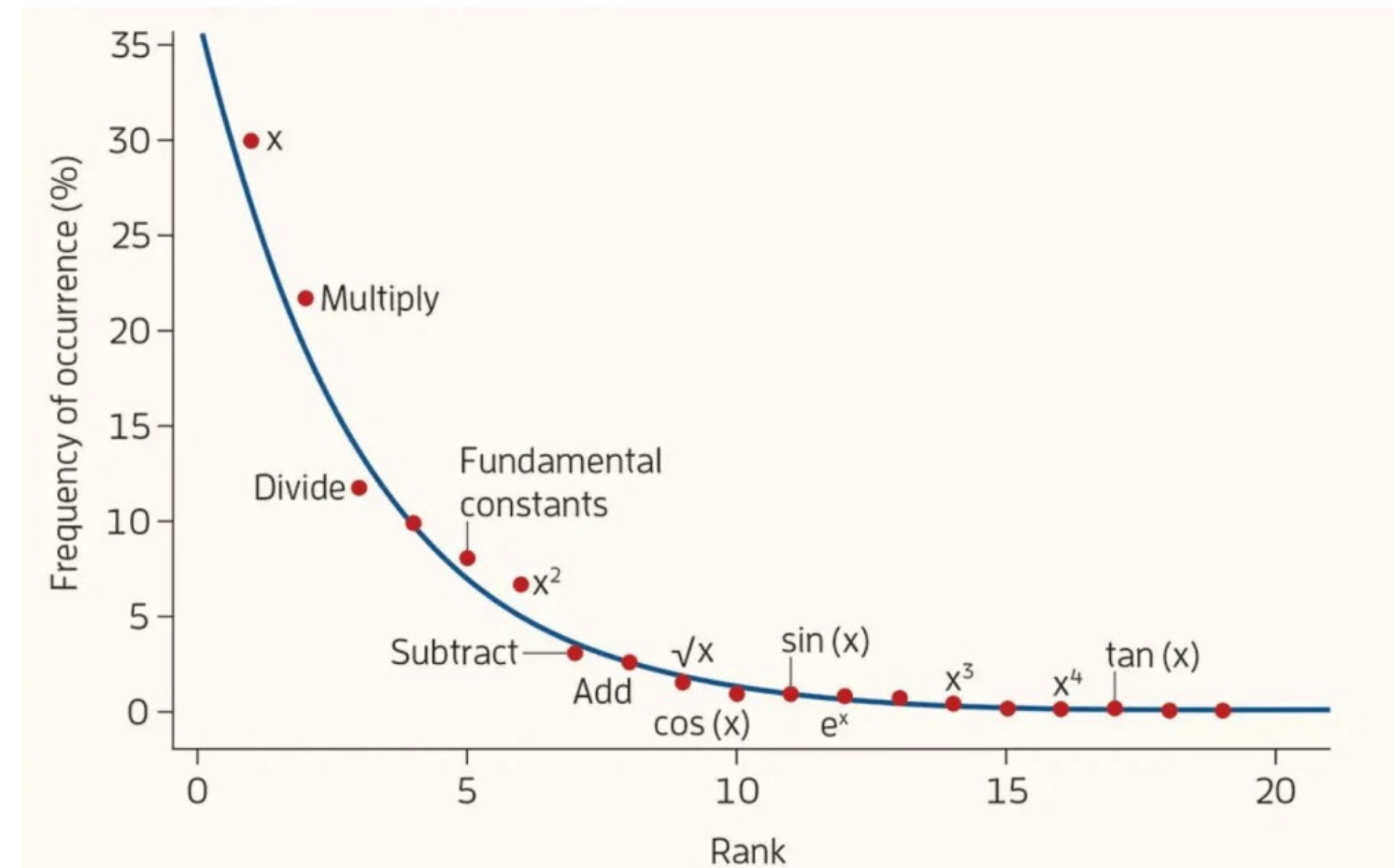
$$\lambda_i = \frac{c}{i}$$

RESULT

$$C_{\text{sep}} = r_{\text{eff}} \sim \frac{6}{\pi^2} (\log D)^2$$

DOUBLING WIDTH

$$\frac{C_{\text{sep}}(2D)}{C_{\text{sep}}(D)} \approx \left(\frac{\log(2D)}{\log D} \right)^2 \rightarrow 1$$

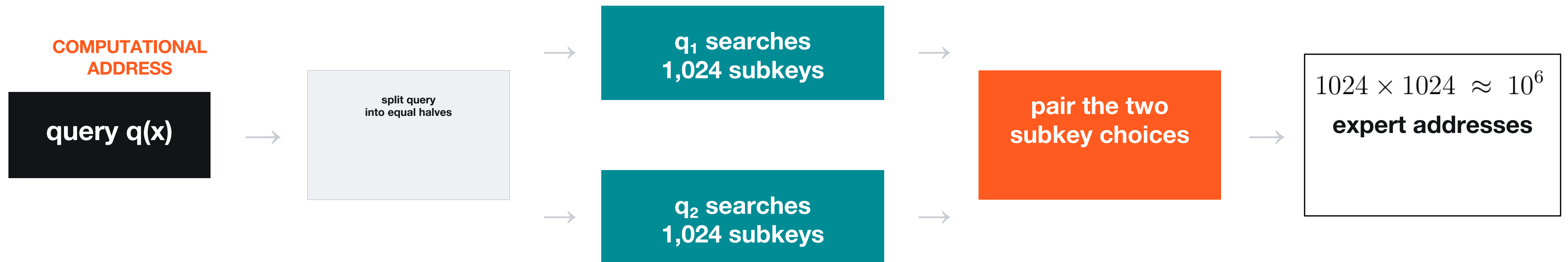


Nominal width grows as D , but separability capacity grows only as $(\log D)^2$.

If the Zipf shape persists, doubling a very wide representation yields almost no multiplicative capacity gain, while dense compute can still grow as D^2 .

A million expert addresses do not create a million useful experts

Product-key routing makes roughly one million addresses searchable. Useful capacity still requires representation geometry, routing, specialization, and load balance to align.



Address space → representation geometry → expert specialization → stable load balance

The open problem is not addressability. It is learning separations that remain meaningful as the model changes

Long context *T*: KV-state wall

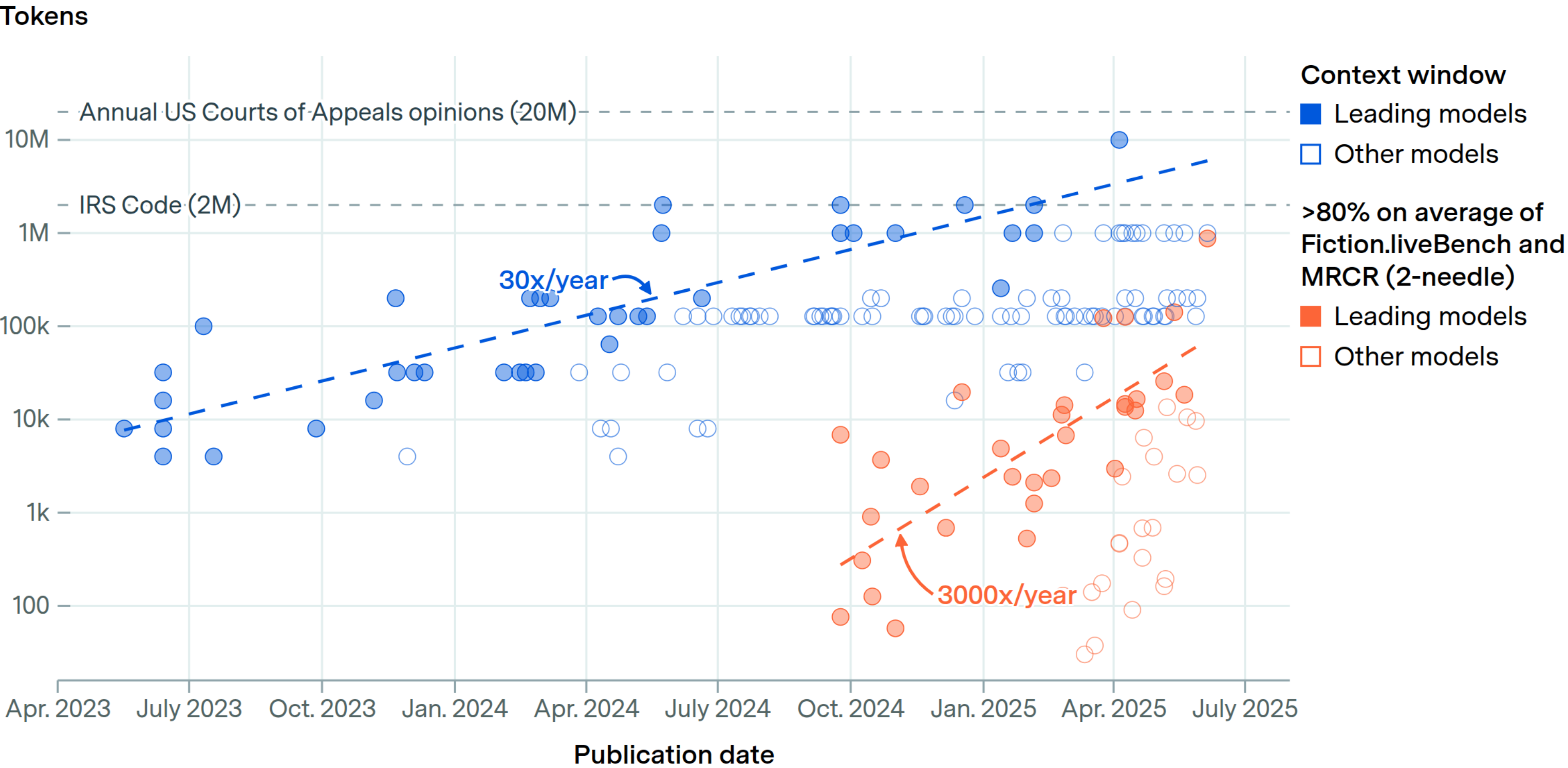
Cite



By Greg Burnham and Tom Adamczewski

Today’s LLMs can only ingest a limited number of tokens per query, often called the context window. Since mid-2023, the longest LLM context windows have grown by about 30x per year. Their ability to use that input effectively is improving even faster: on two long-context benchmarks, the input length where top models reach 80% accuracy has risen by over 250x in the past 9 months.

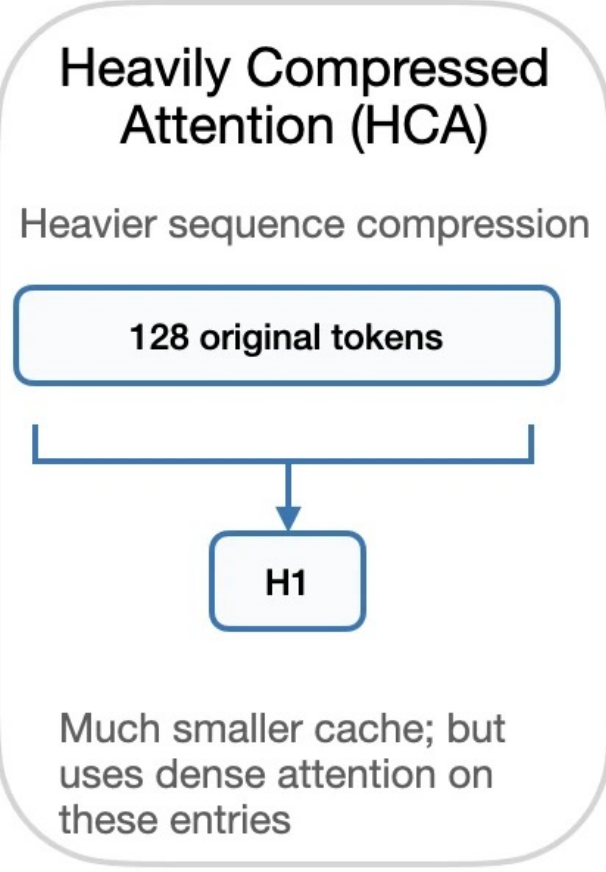
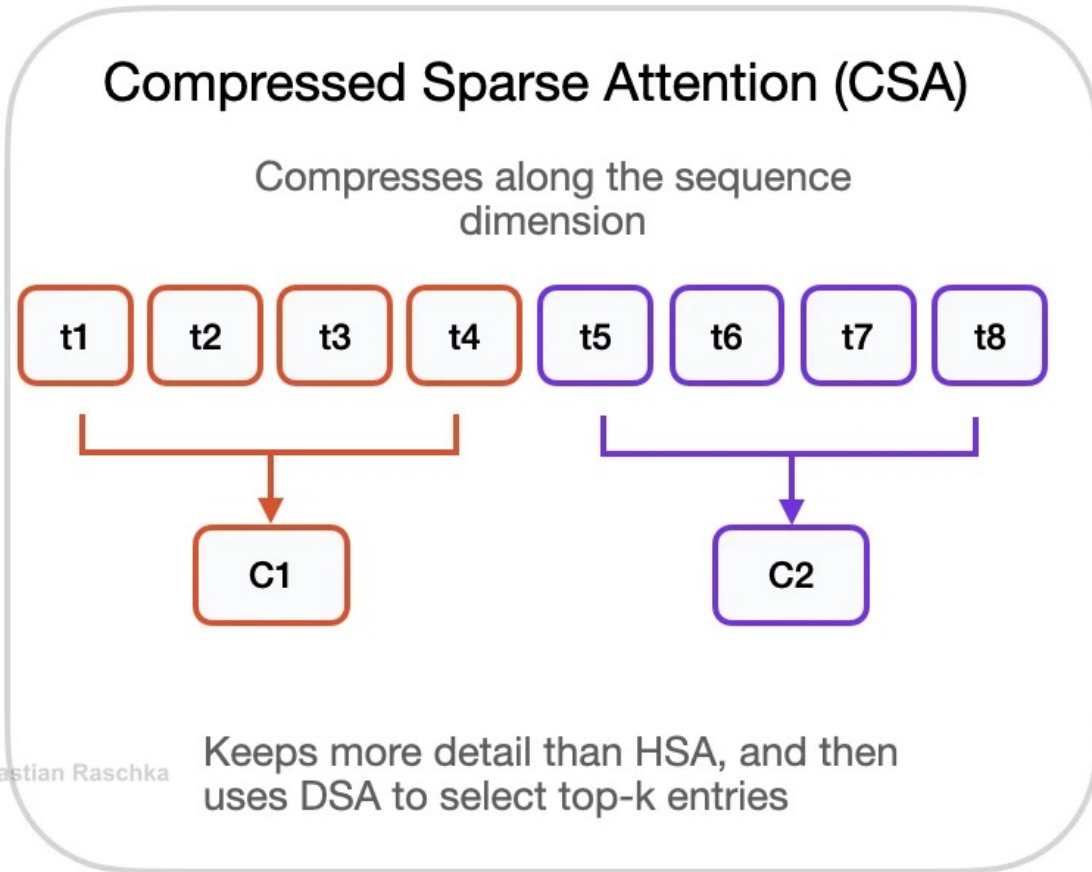
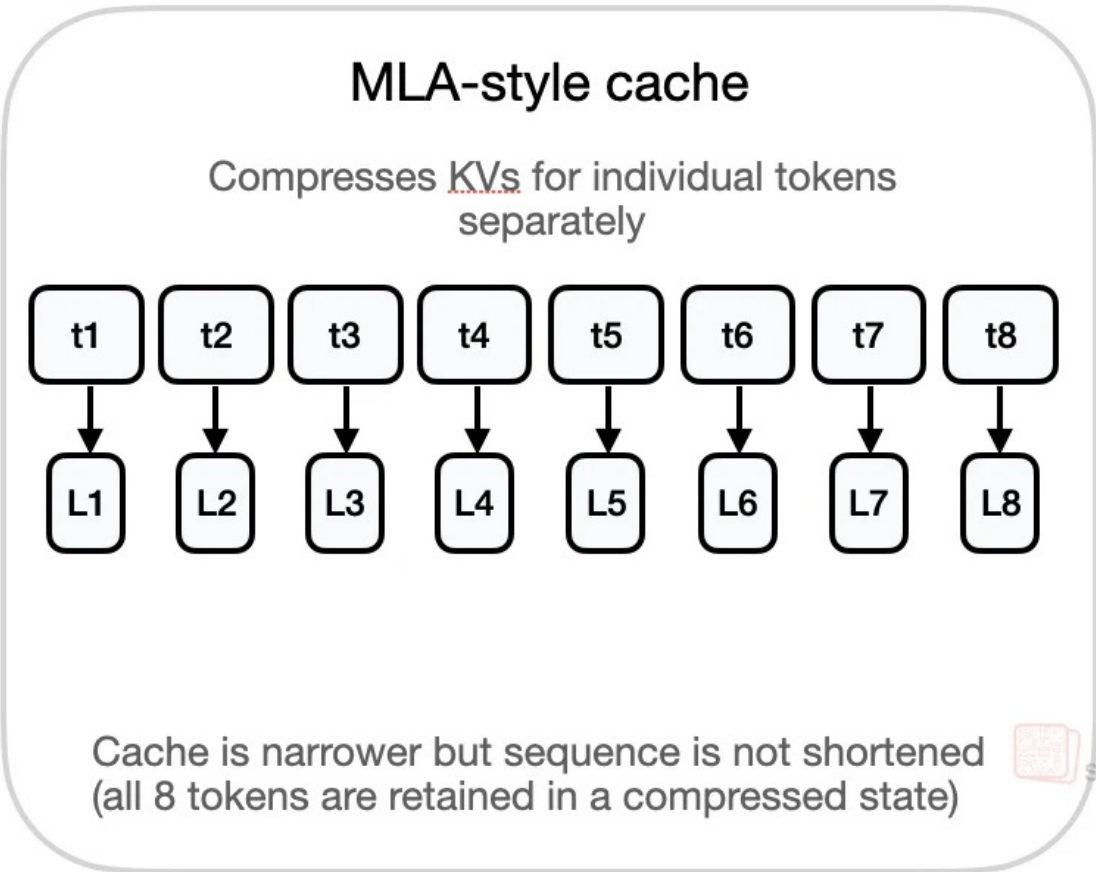
Frontier LLMs accept increasingly long inputs, and their usable context length has grown even faster



Compression extends the single-context regime

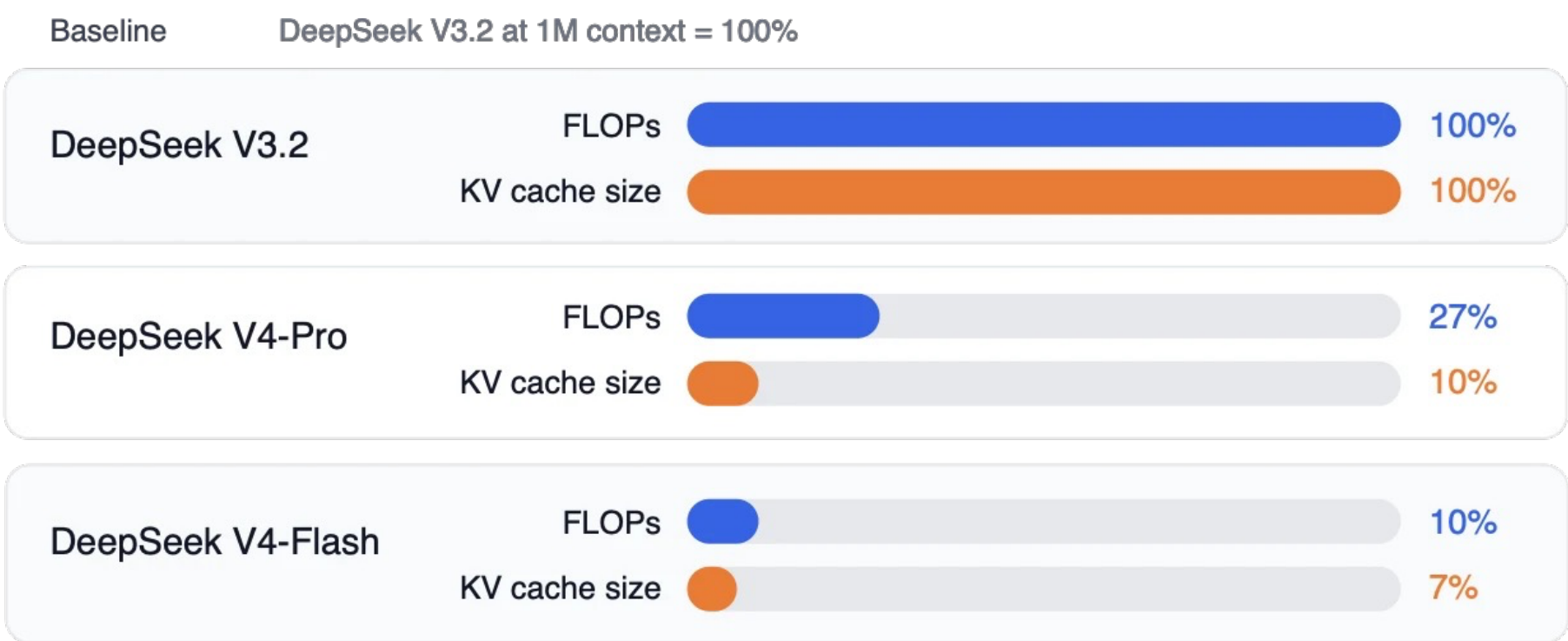
CSA Compressed Sparse Attention

Compresses nearby KV tokens modestly, then uses an indexer to retrieve only the few relevant compressed blocks. It preserves specific long-range facts efficiently.



HCA Heavily Compressed Attention

compresses the distant history much more aggressively, then attends densely over that small summary. It preserves the global context and broad structure cheaply.



Values are relative to DeepSeek V3.2 in the DeepSeek V4 paper, not absolute FLOP or cache values.

Agent fleets turn context into shared systems state

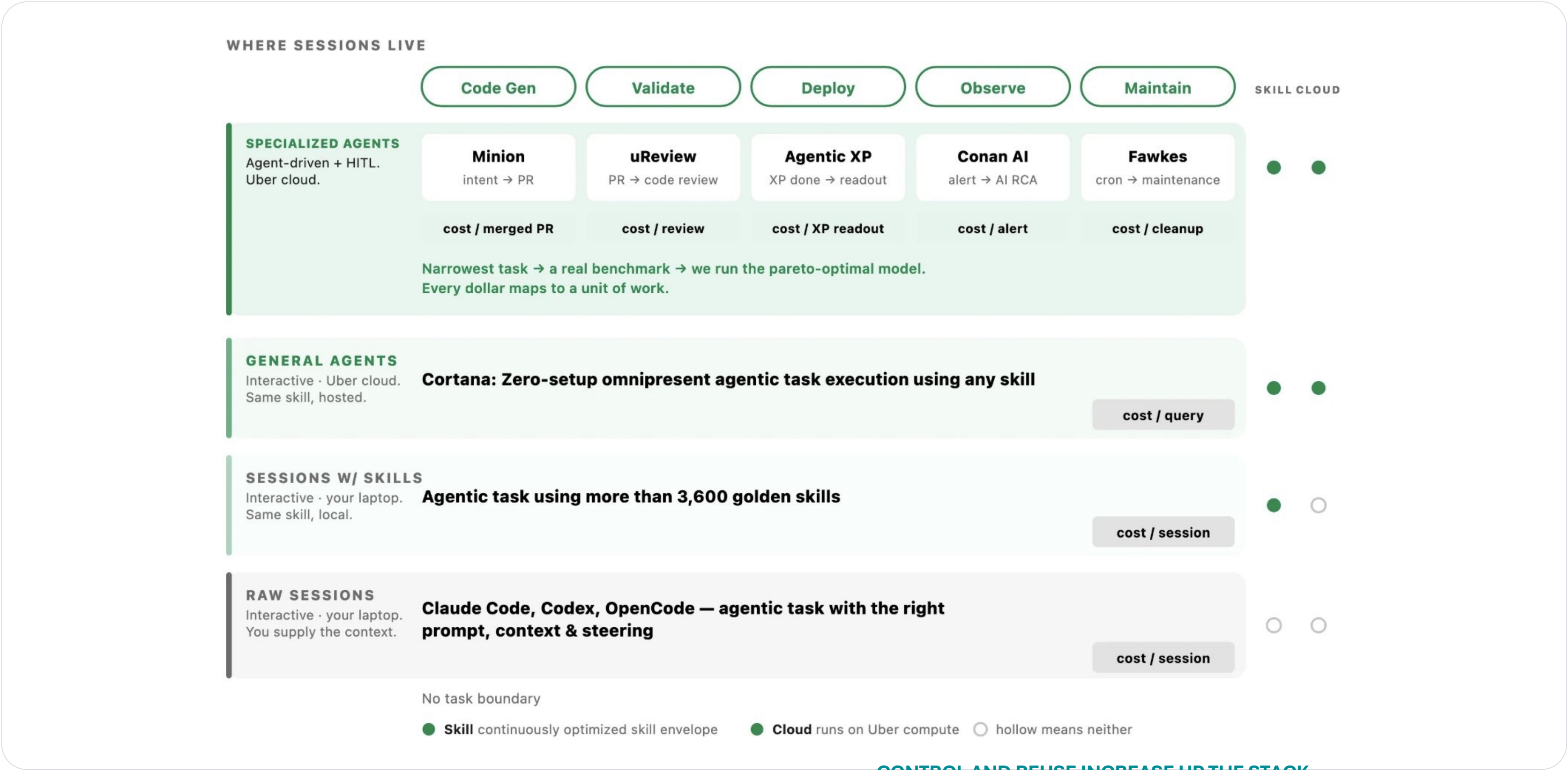
At fleet scale, the reusable unit is no longer one prompt or one session—it is project state shared across agents and time.

30K+
skill executions per day

3,600
agent skills

>70%
of pull requests attributed
to local or cloud agents

Project state is repeatedly
retrieved, carried,
transformed, and
recomputed across agents
and turns.



CONTROL AND REUSE INCREASE UP THE STACK

Source: Uber Engineering; figures are Uber-specific workload evidence

Scaling AI at Uber

7x weekly active users since Feb

9.4x weekly agent requests since Feb

What makes an agentic workload different?

One task contains many model calls, tool executions, and verification steps.

A coding agent does not finish after one model response. It reads the repository, calls the model, runs tools, observes the result, and tries again. A task may run for hours, and many agents may run at the same time.

Problem: repeated model calls

Each call carries instructions, repository state, tool results, and earlier decisions. Repeated long prefixes consume inference compute and KV-cache memory.

Problem: tool interruptions

Between model calls, CPUs run compilers, tests, browsers, search, storage, and APIs. These tools may run much longer than the model call.

Success: verified outcome

The task succeeds only when it produces a merged change, completed review, repaired failure, or measured optimization that passes verification.

Schedule the complete task. Separate request scheduling loses its dependencies.

What has industry already demonstrated?

Agent fleets are real. Their complete runtime policies are still in its infancy.

Uber: production workload

Uber reports more than 3,600 reusable agent skills and more than 30,000 skill executions per day. Local or cloud agents are attributed to more than 70% of pull requests. Managed agents perform review, CI repair, debugging, and maintenance.

OpenAI: agents and sandboxes

At the 99th percentile, highly active internal users generated more than 60 hours of parallel agent turns per day. The Agents SDK provides isolated sandboxes, parallel containers, externalized state, and snapshot recovery.

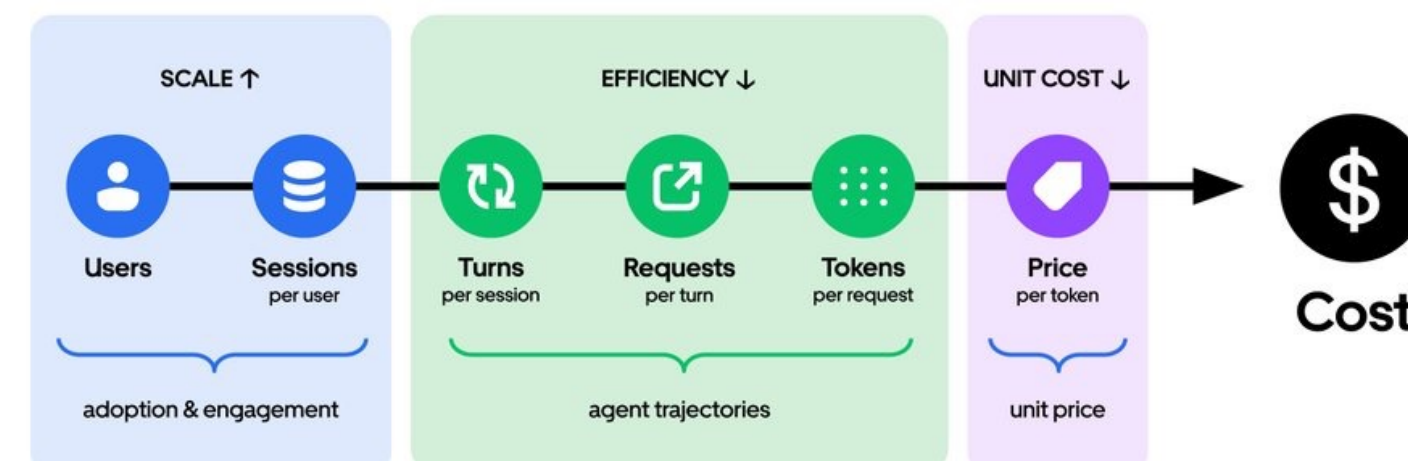
GitHub: cloud execution

Copilot coding-agent tasks run asynchronously in GitHub Actions environments. Teams can launch parallel tasks across repositories, execute builds and tests, and receive proposed changes as pull requests.

Anthropic: parallel experiment

Sixteen agents used nearly 2,000 Claude Code sessions and about \$20,000 in API cost to build a roughly 100,000-line Rust C compiler. This was an experiment, not a continuous production service.

The
Cost
Machine



What must an agent-farm runtime manage?

It must coordinate model work, tool work, state, and verification until the task finishes.

An agent alternates between two kinds of work. The GPU runs the language model. CPUs and external systems run compilers, tests, browsers, search, storage, and APIs.

Model phase

The model reads the current context, reasons about the task, and produces text or a tool call. Its attention state may remain in GPU high-bandwidth memory as a KV cache.

Tool phase

The agent waits for a build, test, browser, or external service. The GPU is idle for this agent, but its KV cache and sandbox still occupy resources that other tasks need.

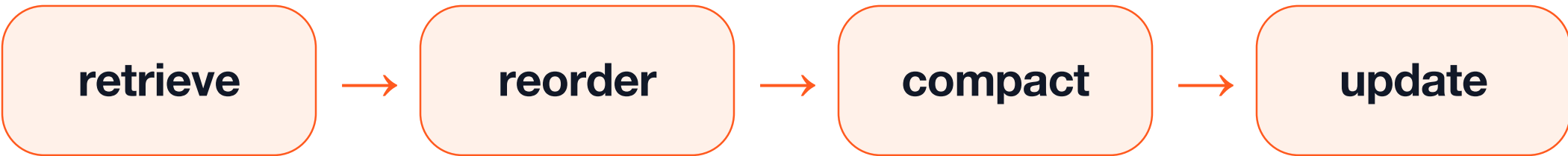
The runtime must keep the task moving while controlling resources held by waiting agents.

Semantic freshness conflicts with prefix-cache reuse

Context engineering changes evidence; prefix caching rewards an unchanged token prefix and compatible positions.

CONTEXT ENGINEERING

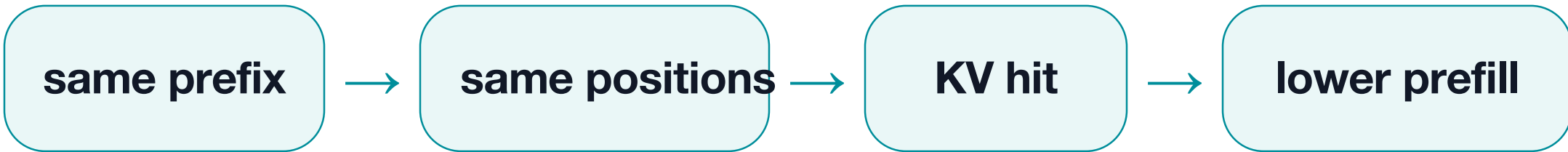
Goal: maximize useful evidence under a finite attention budget.



The token sequence changes as the task evolves.

PROMPT / KV CACHING

Goal: reuse previously computed attention state.



Reuse survives through the unchanged prefix, then falls after the first incompatible change.

Fundamental challenge

Update what the agent knows while preserving every region of computation that is still valid.

Why is request-level scheduling insufficient?

Consider one agent that edits a repository and then runs a ten-minute test.

- 1 The model reads the repository and builds a long context. The GPU stores the resulting KV state so later turns can reuse the earlier computation.
- 2 The agent launches the test. CPU resources and a sandbox remain active, but this agent no longer needs GPU computation.
- 3 Other agents arrive while the test runs. The scheduler must decide whether the waiting agent should continue occupying scarce GPU memory.
- 4 If the scheduler evicts the KV state, the agent may need to rebuild its context when the test completes. If it keeps the state, fewer agents can be batched.
- 5 After the next model turn, the proposed change enters a verification queue. The task is complete only when the result passes the required checks.

The model server sees separate requests. The runtime must see one dependent task.

Should a waiting agent keep its KV cache?

Keeping the cache saves future computation. Evicting it frees HBM for other agents.

Option 1: keep it in HBM

The next model call starts quickly because the previous attention state is already available. The cost is less memory for batching other agents.

Option 2: evict or offload it

Other agents gain GPU memory immediately. When the original agent returns, the system may pay a large transfer or prefill cost to rebuild the removed state.

Current solution

Most systems use least-recently-used eviction, a fixed time-to-live, or prefix matching. These policies see cache history, not task meaning. They do not know how long the tool will run, whether the task is urgent, or whether the next model call will still need the same context.

Open challenge: predict the cache's future value before the agent returns.

Why schedule the full pipeline together?

The GPU, tool, and verification queues constrain one another.

The GPU may generate work faster than the rest of the system can process it. More model throughput creates more builds, tests, browser sessions, and candidate patches. When downstream queues are full, extra inference increases waiting time without completing more tasks.

Problem: tool bottleneck

Tool duration ranges from milliseconds to many minutes. Waiting and running sandboxes also consume CPU, memory, storage, and network capacity.

Problem: verification bottleneck

Agents may create candidates faster than numerical checks, security analysis, regression tests, or human reviewers can validate them.

Current solution: backpressure

Delay new model work until downstream capacity becomes available. This prevents queues from growing without increasing completed work.

System objective: maximize verified tasks per hour, not GPU utilization.

What other problems must the runtime solve?

Long-running agents share infrastructure, preserve state, and modify external systems.

Admission and fairness

Before starting a task, the runtime must estimate whether the farm has enough model, tool, and verification capacity. It must also prevent one long workflow from starving smaller or higher-priority tasks.

Shared memory and isolation

Agents should reuse useful code indexes, summaries, and prior results. The runtime must prevent stale, private, or task-specific state from leaking into another workflow.

Sandbox lifecycle

Keeping every environment warm wastes memory. Recreating every environment adds delay. The runtime chooses when to reuse, restore, create, or release each sandbox.

Failure recovery

A checkpoint must preserve completed work without repeating external effects. Re-running a test is usually safe. Repeating a deployment, payment, or message may not be safe.

Open challenge: preserve reuse and progress without leaking state or repeating side effects.

What do current systems solve?

Each system manages one part of the runtime. No public system coordinates the entire task.

Workflow scheduling

Parrot exposes dependencies among model calls. Agentix schedules the complete agent program rather than treating every model request independently.

Inference and KV management

ThunderAgent pauses programs at tool boundaries. TokenCake, PBKV, CacheScout, and TOPAS use workflow state or predictions to improve cache placement and task progress.

Sandboxes and execution

OpenAI supports isolated sandboxes, parallel containers, and snapshot restoration. GitHub Actions supplies autoscaled execution environments. SpecBox predicts sandbox demand.

Durable state and recovery

Artifact stores and environment snapshots preserve work outside the model. Crab coordinates checkpoint and restore with agent-turn boundaries.

Open challenge

No public system describes one runtime that jointly controls admission, GPU scheduling, KV placement, sandboxes, verification pressure, fairness, and recovery at large production scale.

One controller must coordinate decisions across all four layers.

Can a small model manage the agent farm proactively?

The model predicts near-term demand. Deterministic code enforces safety and resource limits.

The small model observes task stage, tool type, elapsed tool time, context size, KV footprint, queue pressure, and remaining verification. It predicts resource demand before the request arrives.

Prediction task

Estimate when the agent will return, which resource it will request next, whether its cached context will remain useful, and whether the task is likely to branch or retry.

Deterministic runtime action

Use the prediction to retain or offload KV state, create or release a sandbox, admit or delay work, choose a model and GPU, and start verification. Hard limits enforce security, fairness, and permissions.

Two research questions

1. Does model-guided scheduling improve completed tasks per dollar and tail task latency?
2. Can a small model predict well enough to beat LRU and fixed tool-aware heuristics without reducing task success?

POC: run 8–16 concurrent agents through one inference endpoint and one verification queue.

AI Infrastructure: Cambrian Explosion



Data-center AI

DEFINING CHARACTERISTIC

Centralized accelerator clusters train and serve large models at very high throughput.

DOMINANT CHALLENGE

Keep processors fed and utilized while controlling data movement, power, cooling, and failures.



Wearables

DEFINING CHARACTERISTIC

Always-on personal intelligence must fit within tiny power, heat, and memory budgets.

DOMINANT CHALLENGE

Combine noisy sensors, protect private data, and respond quickly when cloud access is weak.



Humanoid robots

DEFINING CHARACTERISTIC

Perception, reasoning, and joint control run as one continuous physical feedback loop.

DOMINANT CHALLENGE

Deliver predictable real-time behavior, physical safety, robustness, and useful battery life.



Autonomous fleets

DEFINING CHARACTERISTIC

Each vehicle decides locally while sharing maps, learning, and software through the cloud.

DOMINANT CHALLENGE

Handle rare road events, prove safety, maintain redundancy and cybersecurity, and update fleets safely.

Long live the infrastructure

–Tom Jerry

