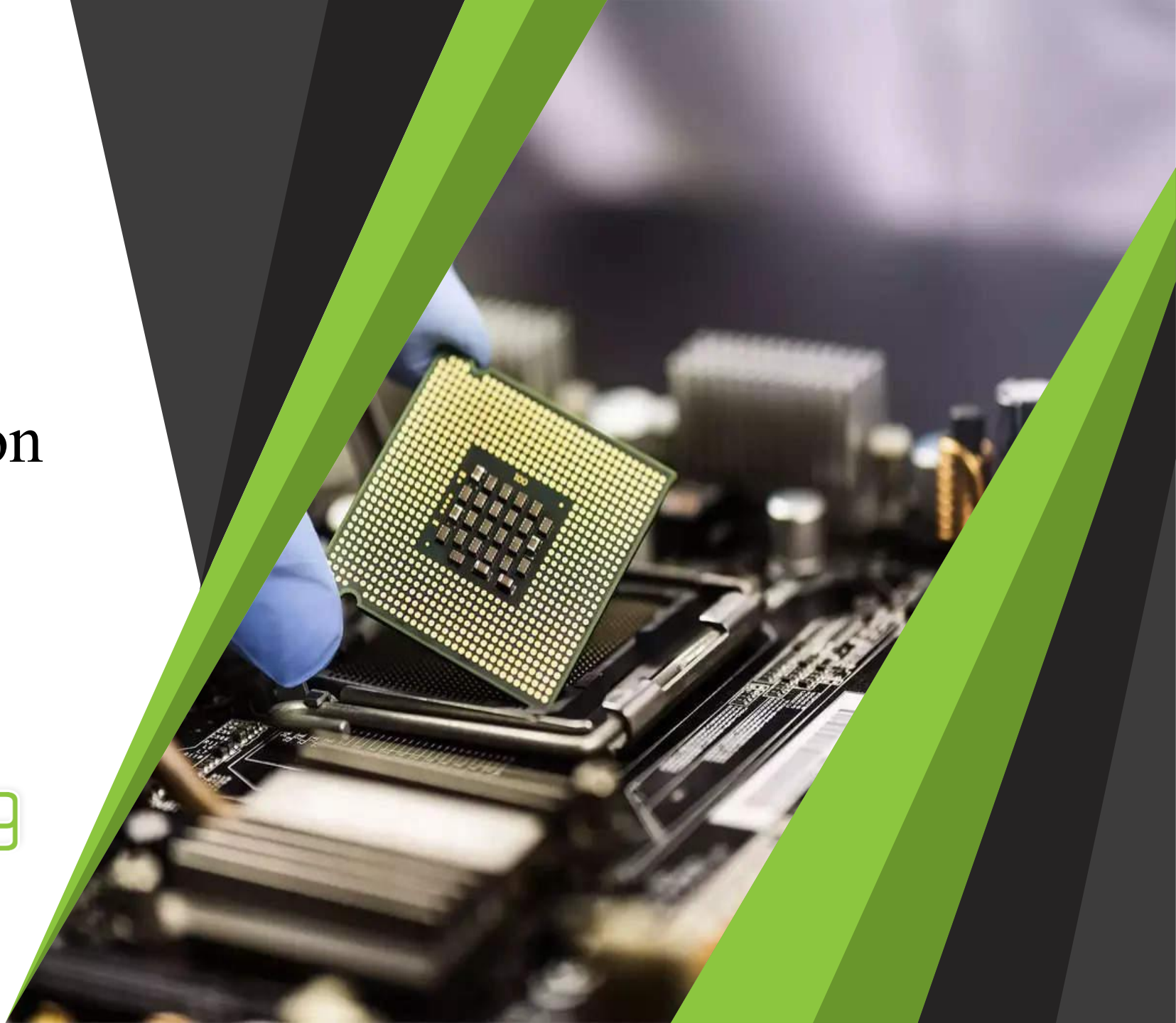


Scaling AI: Software to Silicon

By Khawaja Muhammad Mustafa
SW Eng. Tech Lead



Agenda

- Section 1: Setting the Stage
- Section 2: The AI Stack
- Section 3: Llama Case Study
- Section 4: Memory and Design Challenges

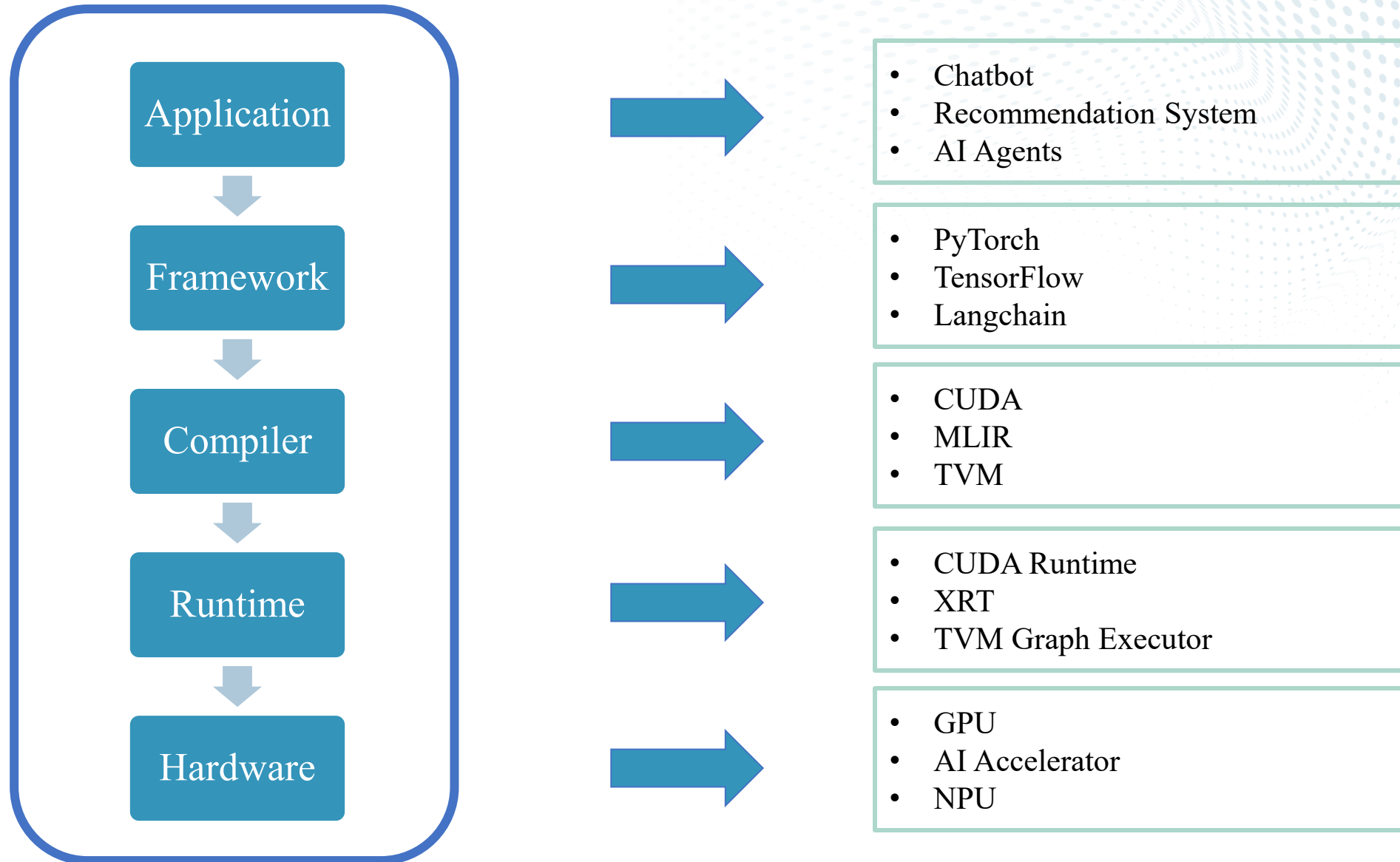
Section 1: Setting the Stage

Why AI Matters?

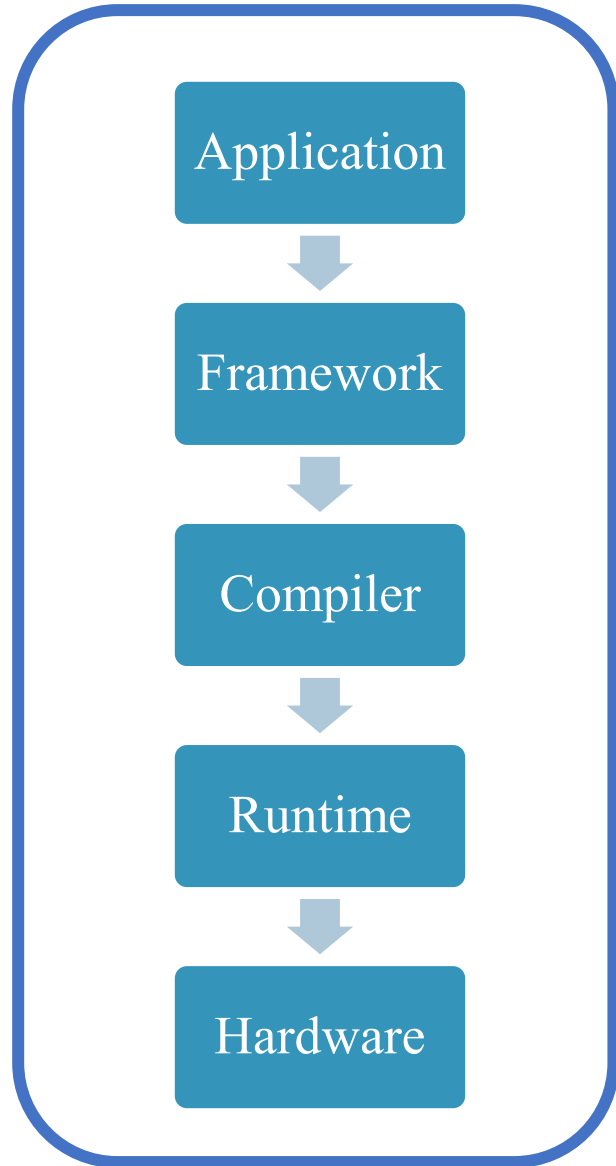
- **AI in everyday life**
 - Chatbots
 - Self-driving
 - Software/Hardware development support



The AI Stack



Analogy (Restaurant)



Menu



Recipes



Chefs



Kitchen Manager



Kitchen



Section 2: The AI Stack

Application Layer

'The visible face of AI'

AI Agents & Assistants

- Autonomous AI Agents (like AutoGPT, LangChain agents) – can plan tasks, call APIs, and act beyond just Q&A.
- Copilots (GitHub Copilot, MS Copilot) – domain-specific assistants for coding, office work, design.
- Customer Support Bots – integrated into banking, e-commerce, and healthcare.

Other User-Facing Apps

- Recommendation Systems – Amazon, YouTube, Spotify (technical enough, still real-world).
- Personalization Engines – news feeds (LinkedIn, Twitter/X), shopping suggestions.
- Creative Tools – image/video generation (Midjourney, Runway, Stable Diffusion).
- Productivity Tools – AI note summarizers, meeting assistants, transcription.

Frameworks

1

Data & Preprocessing Frameworks

- Pandas, RAPIDS, Spark MLlib
- Data Loader & preprocessing for ML/LLMs

2

Training & Fine-tuning Frameworks

- PyTorch / TensorFlow
- DeepSpeed /Hugging Face Transformers
- Tools for model training and optimization

3

Inference & Serving Frameworks

- llama.cpp, TensorRT, vLLM
- Ray Serve, Triton Inference Server
- ggml

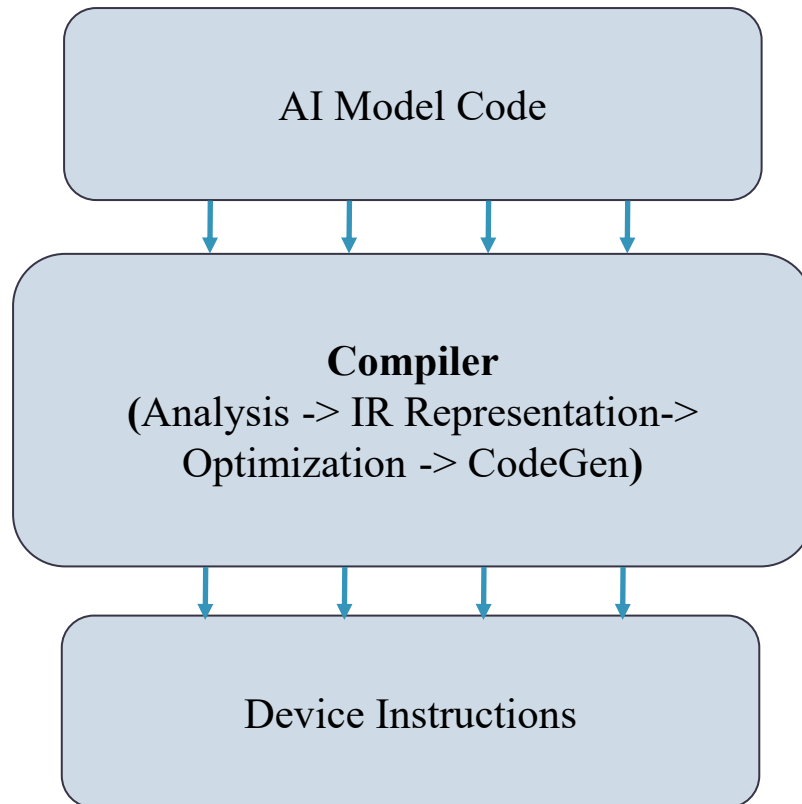
4

Orchestration & Agents Framework

- LangChain, LangGraph
- LlamaIndex
- Workflow + tool integration (API calling)

Compilers

Framework → Compiler → Runtime → Hardware



Role of Compilers

- Bridge between high-level ML frameworks and hardware execution.
- Convert models into optimized execution graphs.

Examples of AI Compilers

- CUDA, MLIR, TVM, Intel Gaudi Compiler

Key Functions

- Graph optimizations: operator fusion, quantization, pruning.
- Hardware-specific code generation (CPU, GPU, NPU).
- Memory mapping, Deterministic scheduling and optimization

Real-world Impact

- Faster inference & reduced latency.
- Efficient hardware utilization.

Runtime

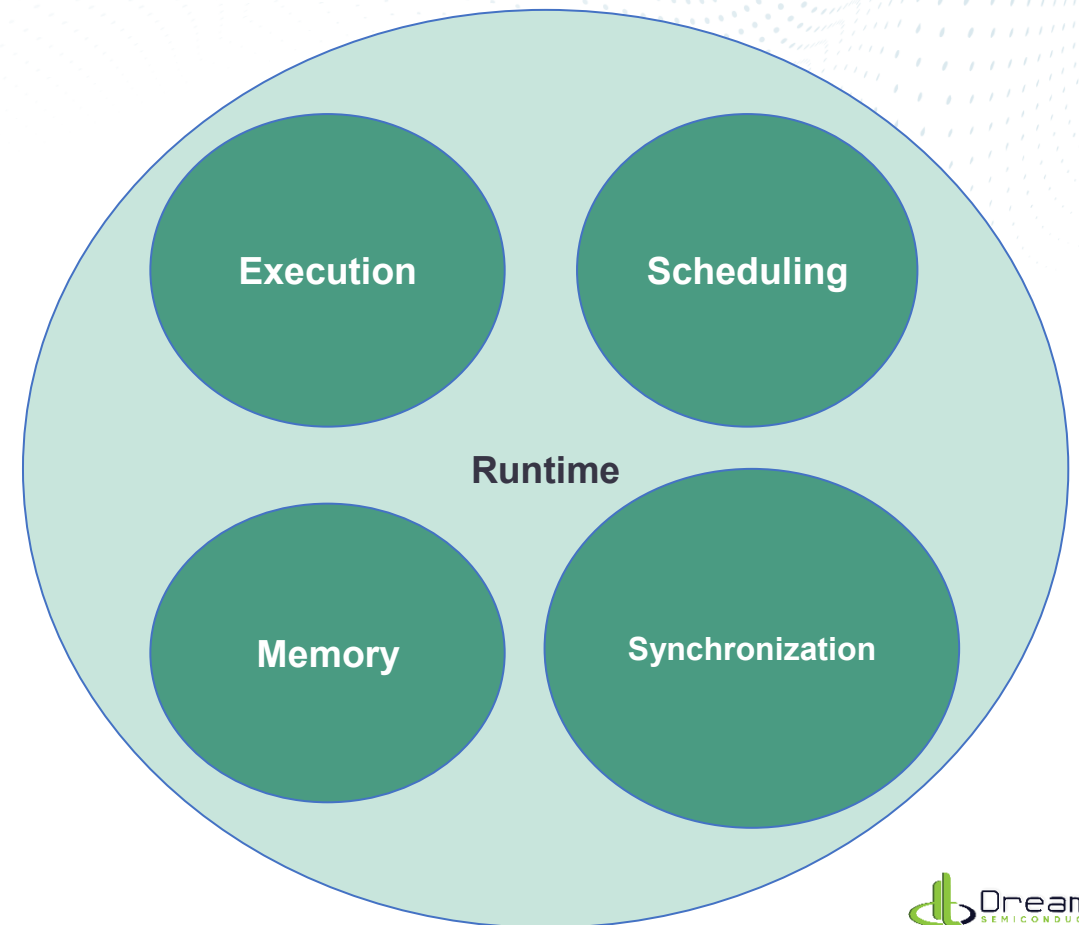
Role of Runtime

- Executes compiler-optimized code on target hardware via device driver
- Manages device memory (allocation, transfers, reuse)
- Dynamically schedules operators across CPU/GPU/Accelerators
- Provides portability across devices

Examples

- ONNX Runtime: cross-framework inference
- TensorRT: NVIDIA GPU inference runtime
- TVM Runtime: portable execution engine
- TorchScript Runtime: PyTorch models execution
























Framework → *Compiler* → ***Runtime*** → *Hardware*

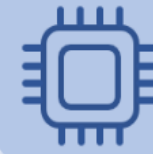


Hardware Layer

Role of Hardware

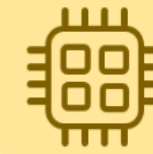
- Provides the compute resources for AI workloads
- Executes low-level instructions scheduled by runtime
- Handles parallelism, memory bandwidth, interconnects
- Trade-off between flexibility (CPU/GPU) vs. efficiency (ASIC/FPGA/TPU/NPU)

 CPU	 GPU	 TPU	 NPU
			
			
			
			
			
			



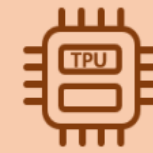
CPU

- Small models
- Small datasets
- Useful for design space exploration



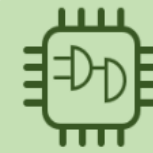
GPU

- Medium-to-large models, datasets
- Image, video processing
- Application on CUDA or OpenCL



TPU

- Matrix computations
- Dense vector processing
- No custom TensorFlow operations



FPGA

- Large datasets, models
- Compute intensive applications
- High performance, high perf./cost ratio

Section 3: Llama Case Study

Llama Architecture

Output Layer

Input: Context Input from Transformer Layer

Output: The next token in the sentence

Transformer Layer

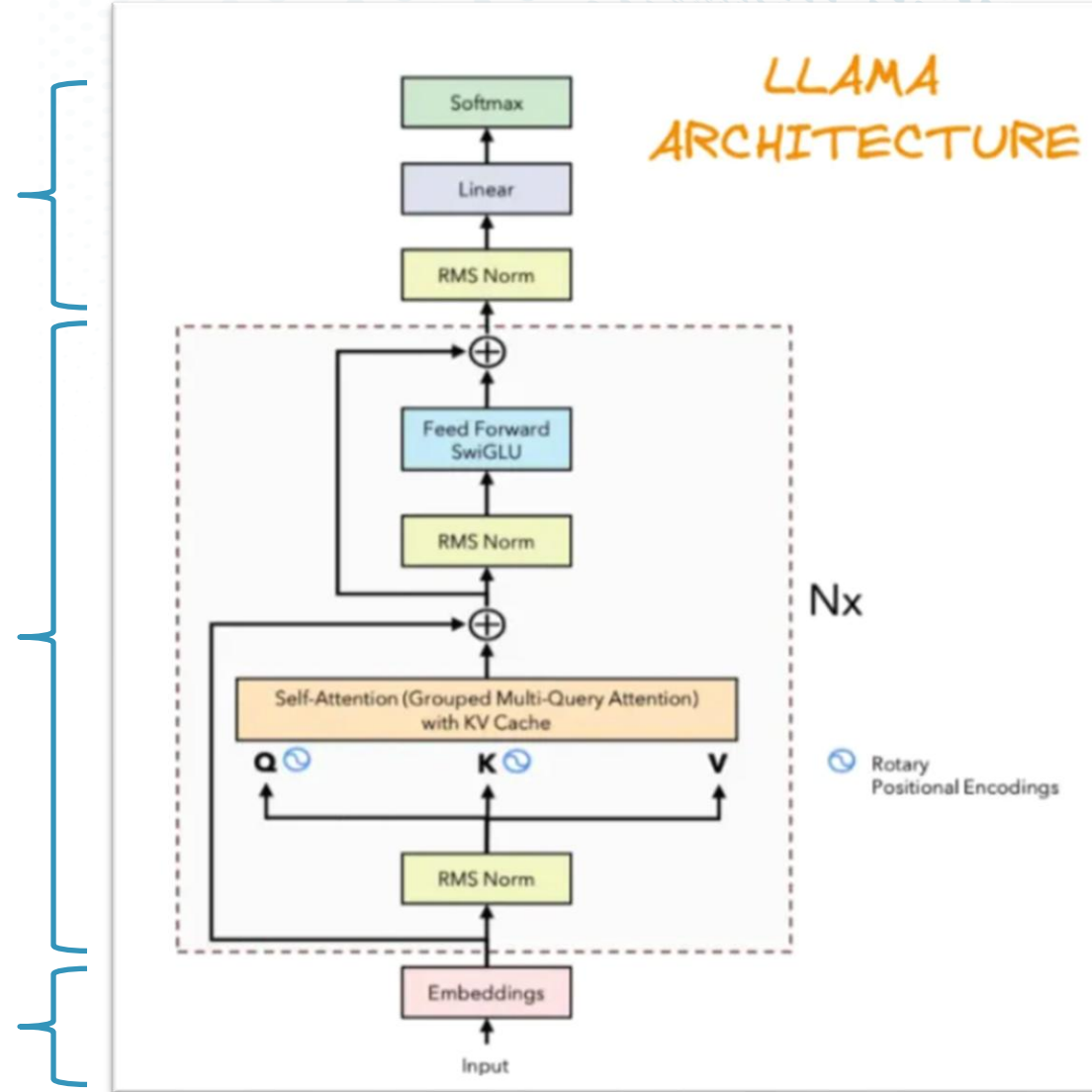
Input: The embeddings of each token in the prompt

Output: The complete context learning of the current sequence of tokens

Input Layer

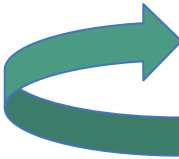
Input: English input sentence from the user (prompt)

Output: The embeddings of each token in the sentence



Inference Flow

Major Steps

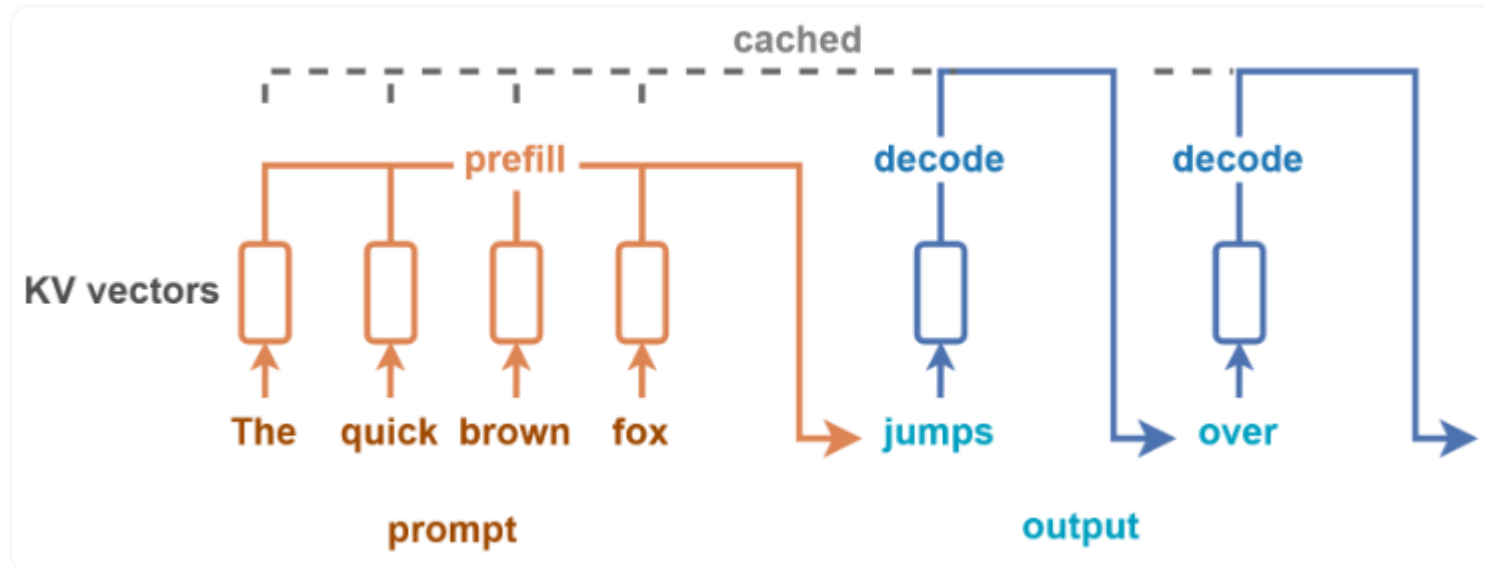
- 
- Step 1: Input token/tokens go through model layers.
 - Step 2: Activations and attention computed.
 - Step 3: Output logits → predicted token.
-
- Key distinction: Prefill vs Decode.

Prefill

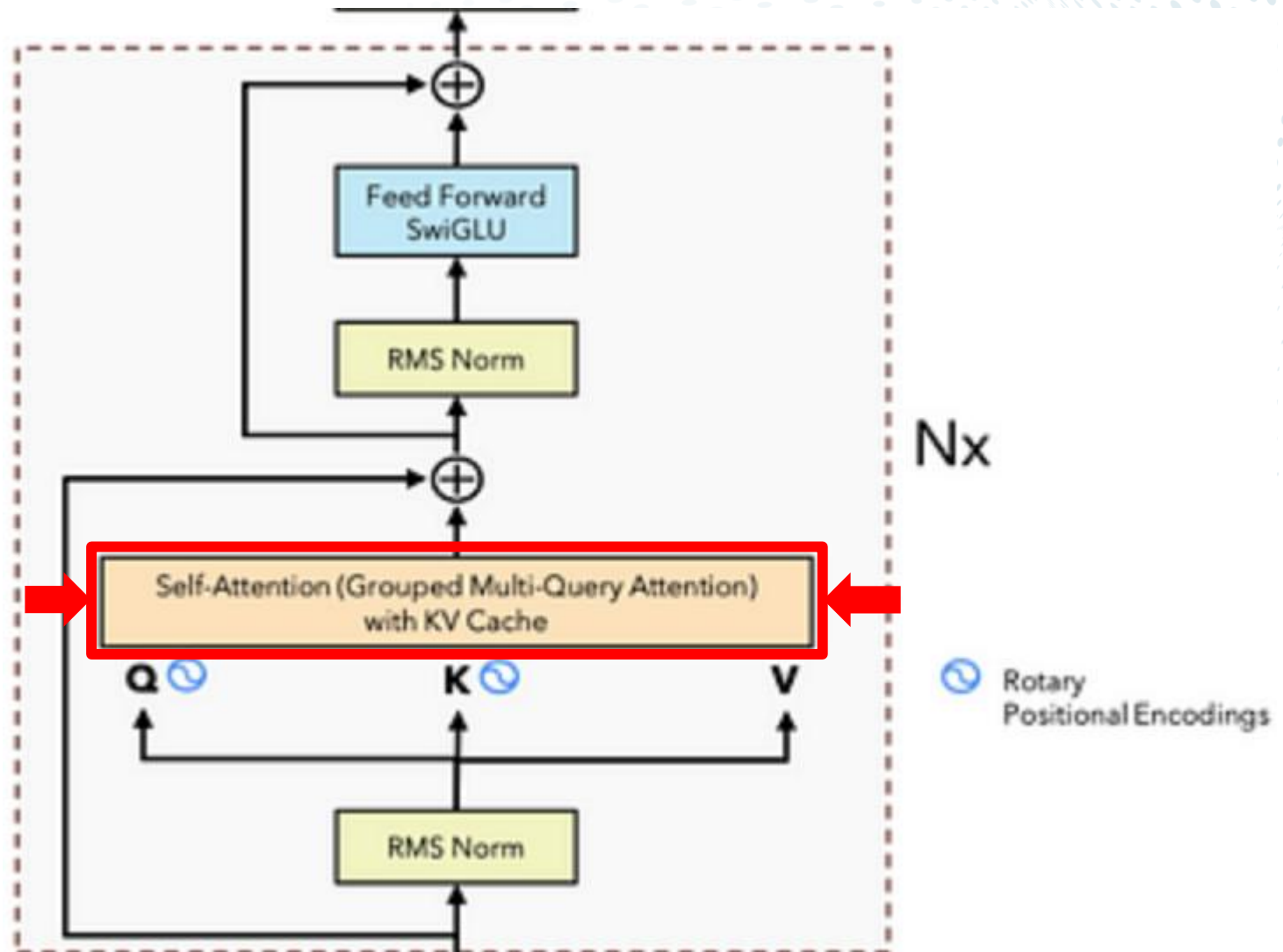
- Runs attention on entire input sequence.
- Heavy compute but parallelizable.

Decode

- One token at a time.
- Reuses KV-cache to save recomputation.

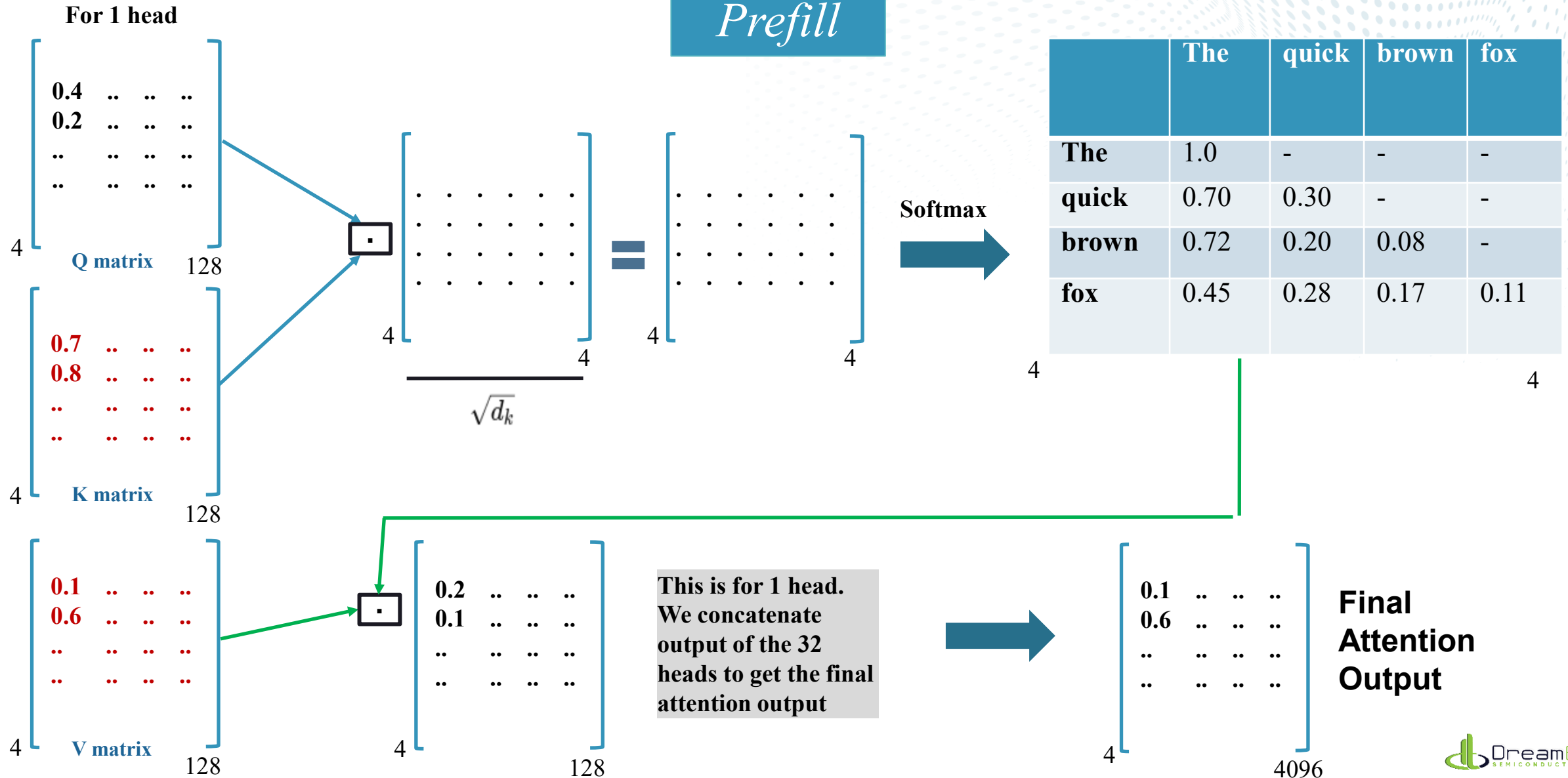


Self-Attention



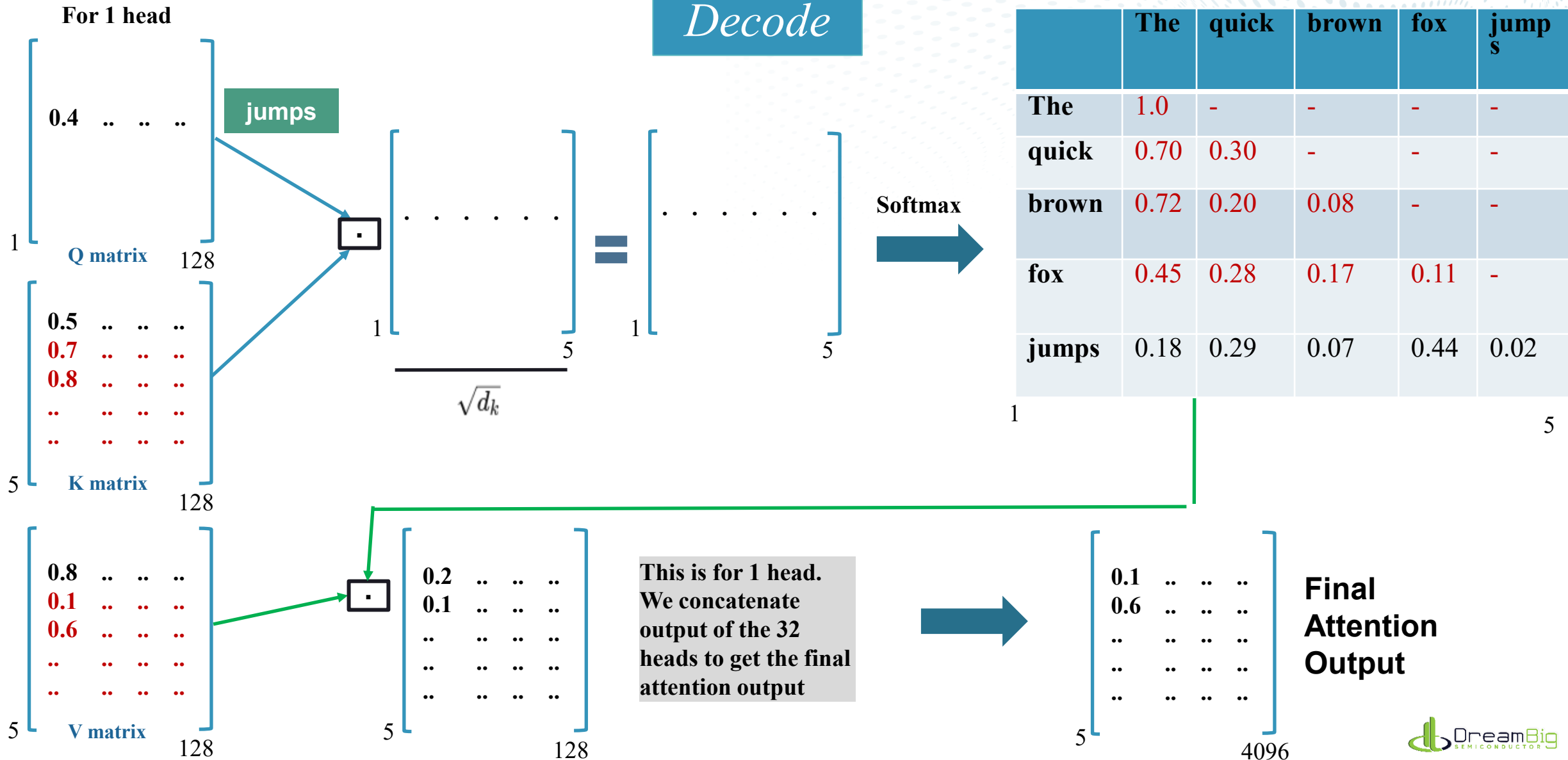
KV-Cache

$$\text{Attention Output} = \text{softmax} \left(\frac{Q \cdot K^T}{\sqrt{d_k}} \right) \cdot V$$



KV-Cache

$$\text{Attention Output} = \text{softmax} \left(\frac{Q \cdot K^T}{\sqrt{d_k}} \right) \cdot V$$



Example

- **Prompt** : The quick brown fox _____
- **Example Model**: Llama 3 - 8B

- **Features:**

Model Parameters: 8 Billion

Layers: 32

Embedding Size: 4096

Input Prompt Tokens: 4 (Space Tokenizer)

Vocabulary Size Tokens: 128,000

Number of heads: 32

KV Heads: 8

Group Size: 4

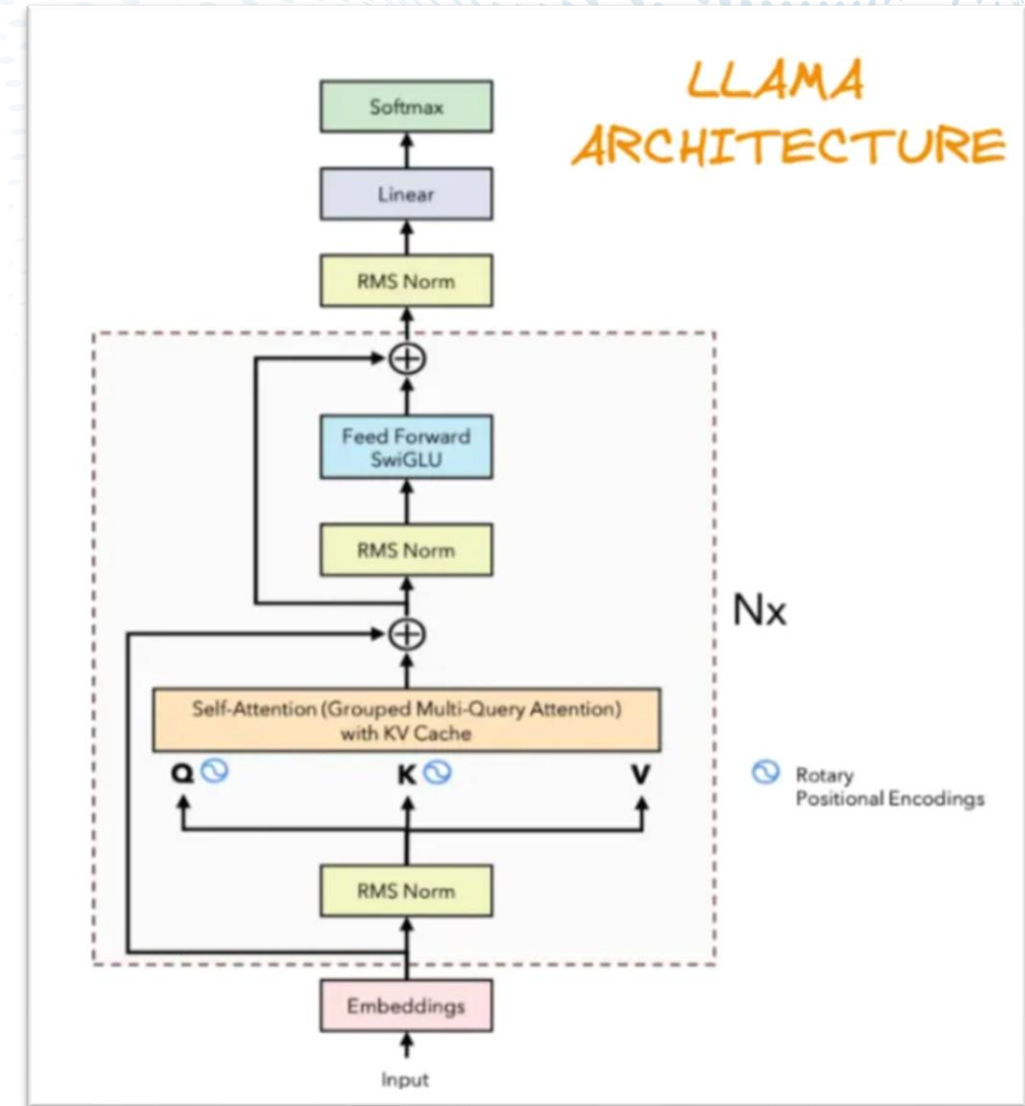
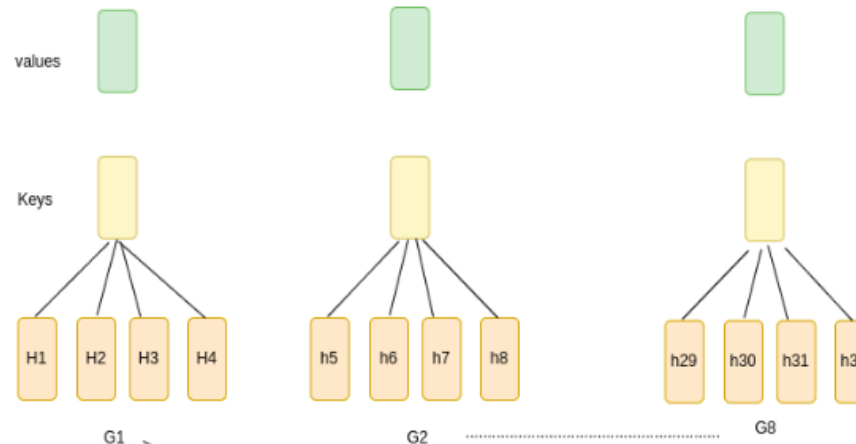
A 4-dimensional embedding

cat =>

mat =>

on =>

1.2	-0.1	4.3	3.2
0.4	2.5	-0.9	0.5
2.1	0.3	0.1	0.4



Why Memory Explodes

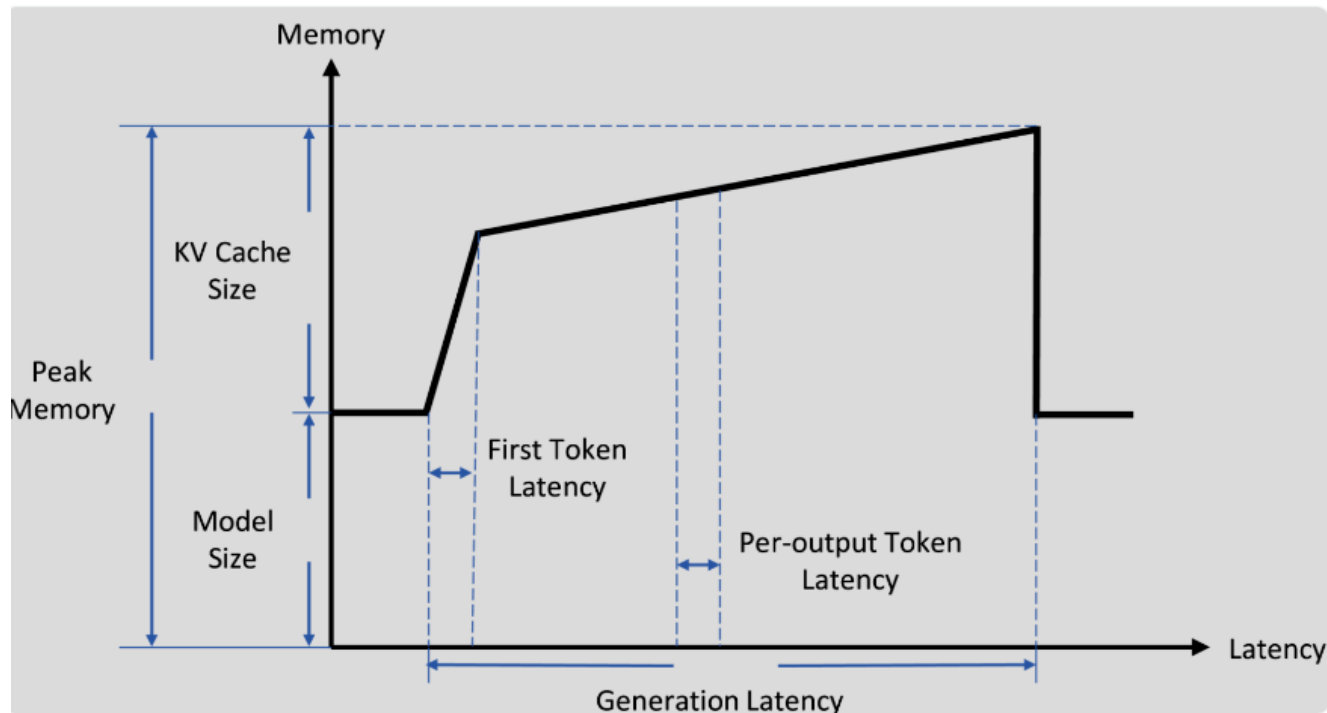
Weights

- Fixed Size
- Depends on models size eg. 7B, 70B etc.

KV-Cache

- Grows with sequence length
- Maximum size is according to the context length

Large memory footprint during long-context inference



Selected Model: meta-llama/Llama-3.1-8B-Instruct
Number of Key-Value Heads: 8

Head Size: 128

Data Type Size: 2 bytes

Total Elements: $2 \times 32 \times 128000 \times 8 \times 128 = 8388608000$

Total Bytes: $8388608000 \times 2 = 16777216000$ bytes

KV Cache Size: $16777216000 / (1024^3) \approx 15.6250$ GB

Section 4: Memory and Design Challenges

The Big Problem

Inference for LLMs requires holding both weights and growing KV-caches.

High Memory Demand



Compute units are fast, but memory access & bandwidth limit throughput

The “Memory Wall”



Even with powerful GPUs, memory becomes the bottleneck

Impact



Option 1 – Big Monolithic Chip

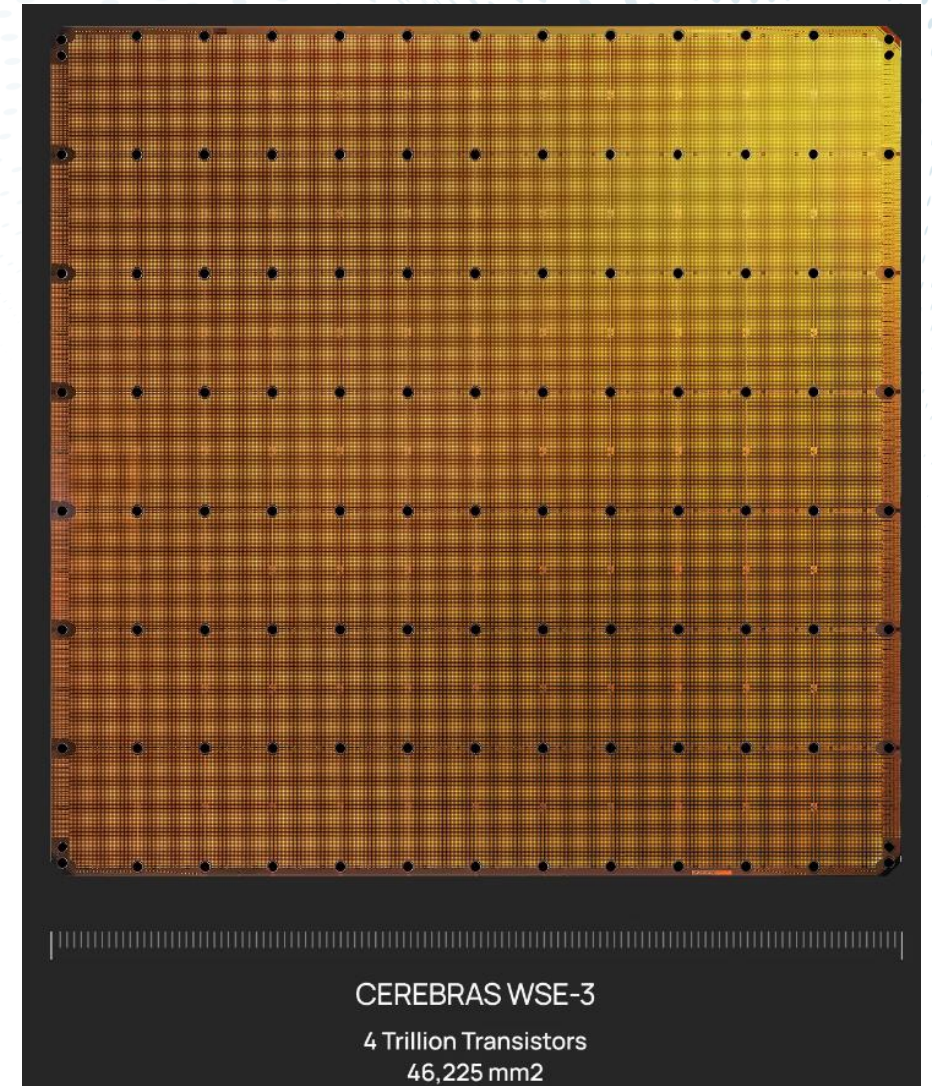
Approach: Put everything on one massive chip (e.g., wafer-scale).

Pros:

- Extremely low latency (no external hops).
- Simplified data movement.

Cons:

- Manufacturing yield issues (defects scale with die size).
- Expensive to produce and cool.
- Scaling impractical — cannot keep up with LLM growth.



Option 2 – Chiplet Based Design

Use HBM, DRAM, Chiplets to scale beyond a single chip.

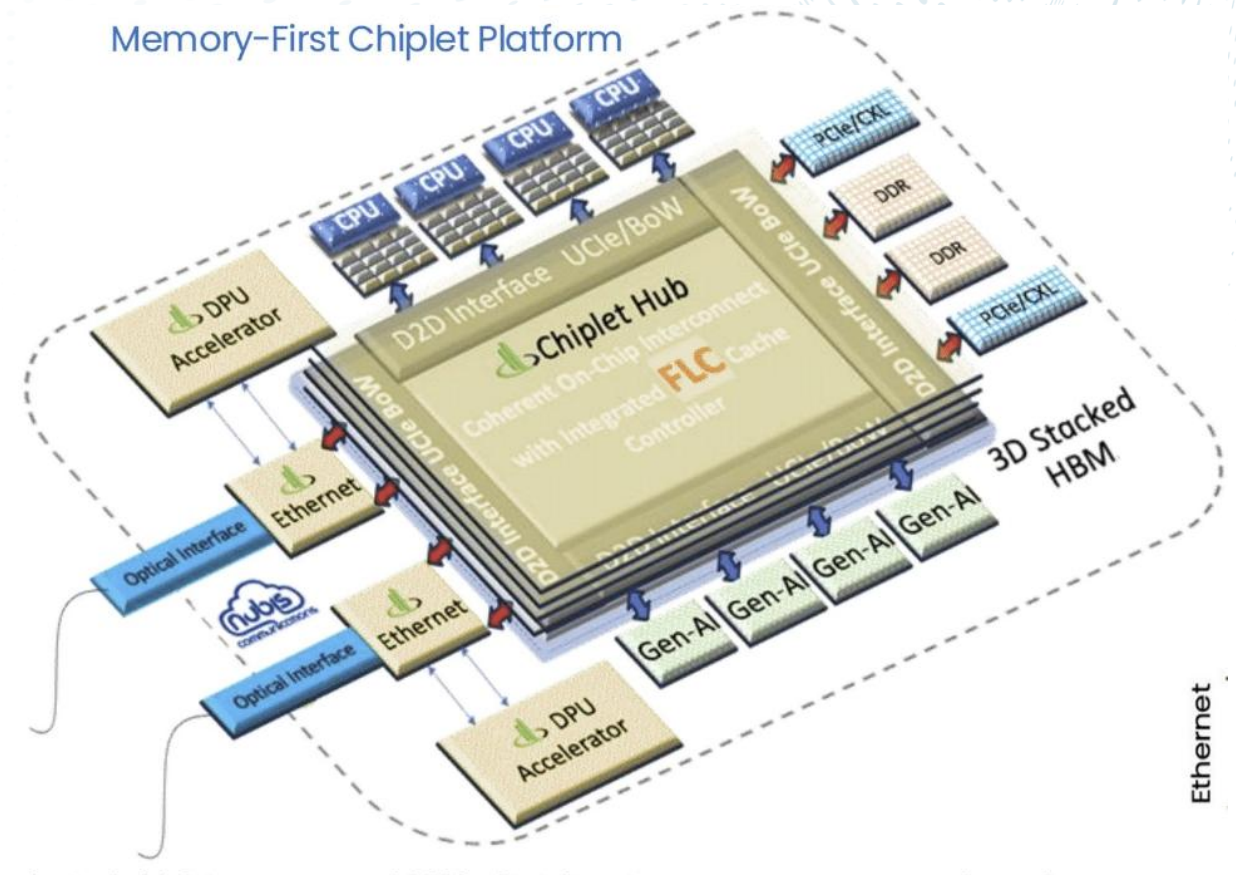
Pros:

- Scalable to large LLMs.
- Flexible designs (chiplets, stacking).

Cons:

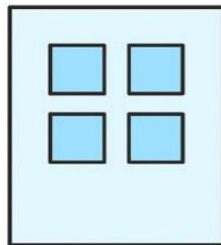
- Bandwidth limits: HBM is fast but expensive; DRAM is slower but cheaper.
- Latency overhead vs on-die memory.

Trade-off: More scalable, but bottleneck shifts to interconnects.



CPU vs GPU vs TPU

<u>Feature</u>	CPU	GPU	TPU
Architecture	Few powerful cores, large cache	Many smaller cores (SIMD/SIMT)	Systolic arrays (matrix-multiply units)
Optimization Target	Low latency, single-thread performance	High throughput for parallel workloads (e.g., graphics, AI inference/training)	Maximum efficiency for tensor/matrix math (deep learning)
ISA (Instruction Set Architecture)	General-purpose ISAs (e.g., x86, ARM)	General-purpose ISA + GPU-specific (CUDA/PTX, ROCm, etc.)	Domain-specific instructions (matrix multiply, systolic array ops)
Best For	control-heavy tasks, sequential logic	Training and inference of AI, graphics rendering, HPC	Large-scale AI inference/training



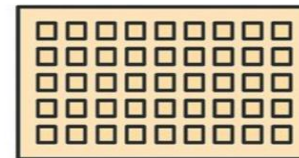
CPU

Few cores



GPU

Many cores



TPU

Many specialized
cores

Q&A