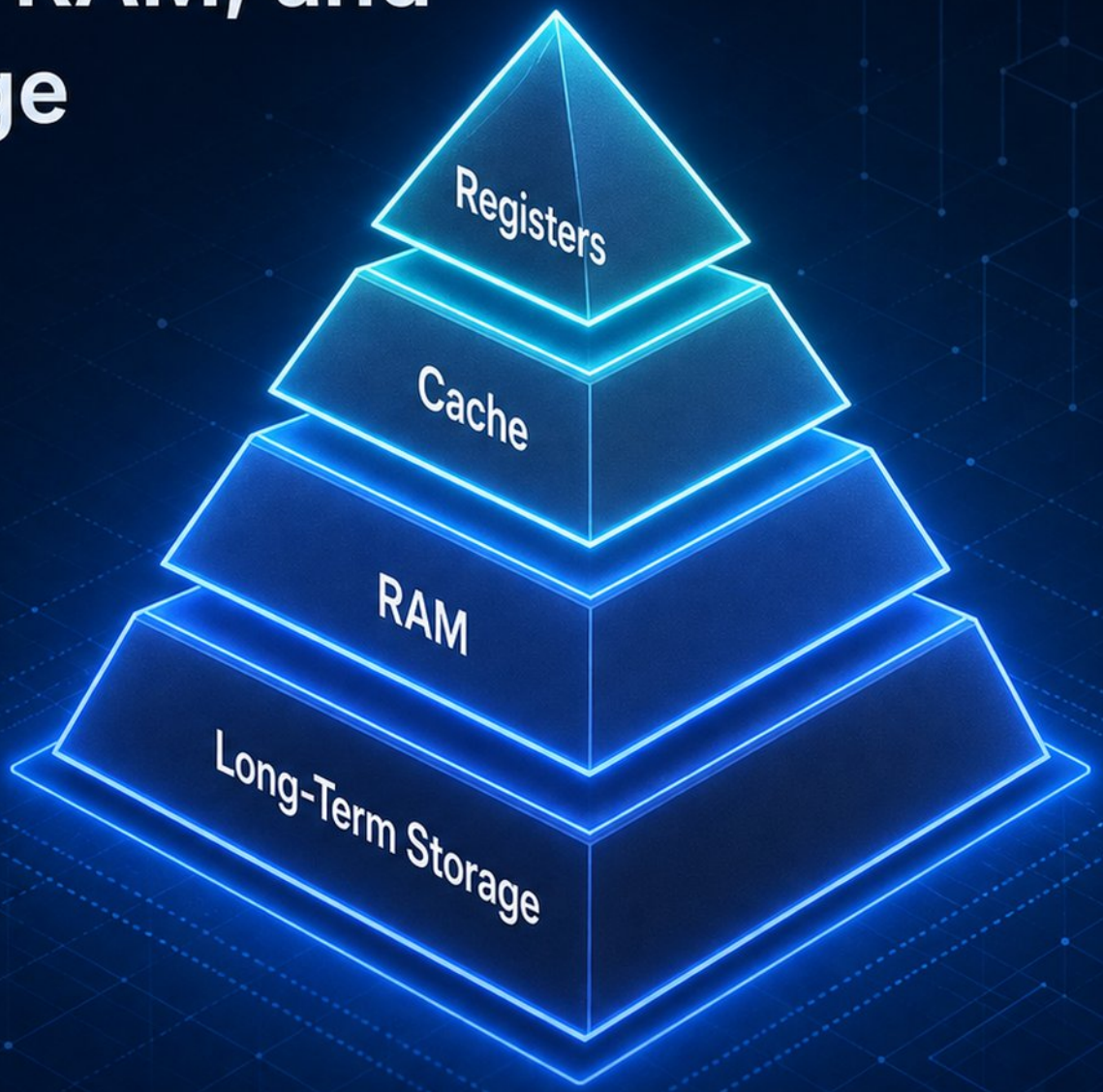
Introduction to Computer Memory: Registers, Cache, RAM, and Long-Term Storage

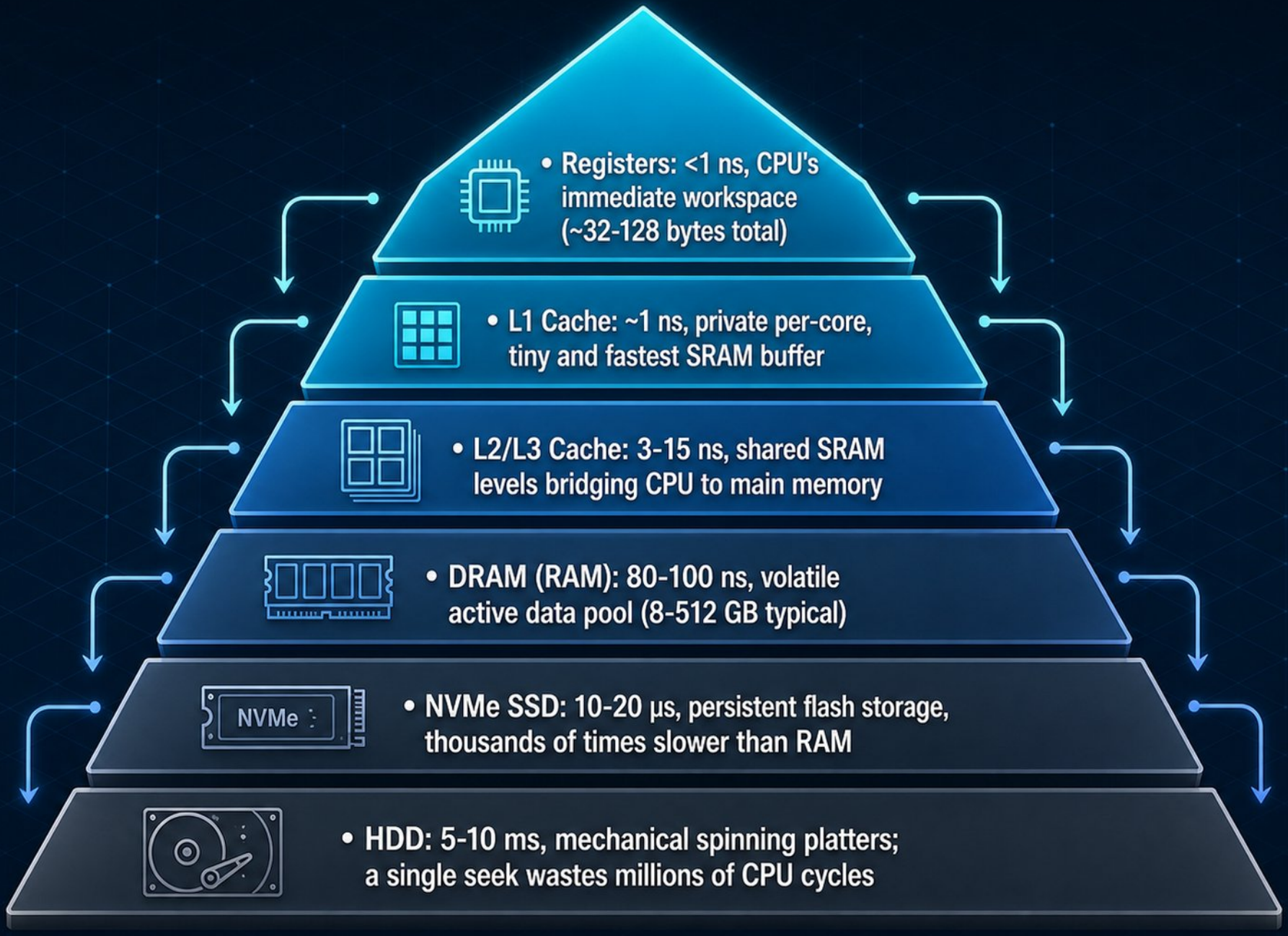
- Memory hierarchy: layered components storing data for processing and retrieval
- Registers -> Cache -> RAM -> Long-Term Storage: each tier trades speed for capacity and cost
- Goal: establish clear conceptual relationships between these four tiers
- Practical value: better system design, performance thinking, and bottleneck diagnosis



Why a Hierarchy Exists:
The Speed, Capacity, and Cost Trade-Off

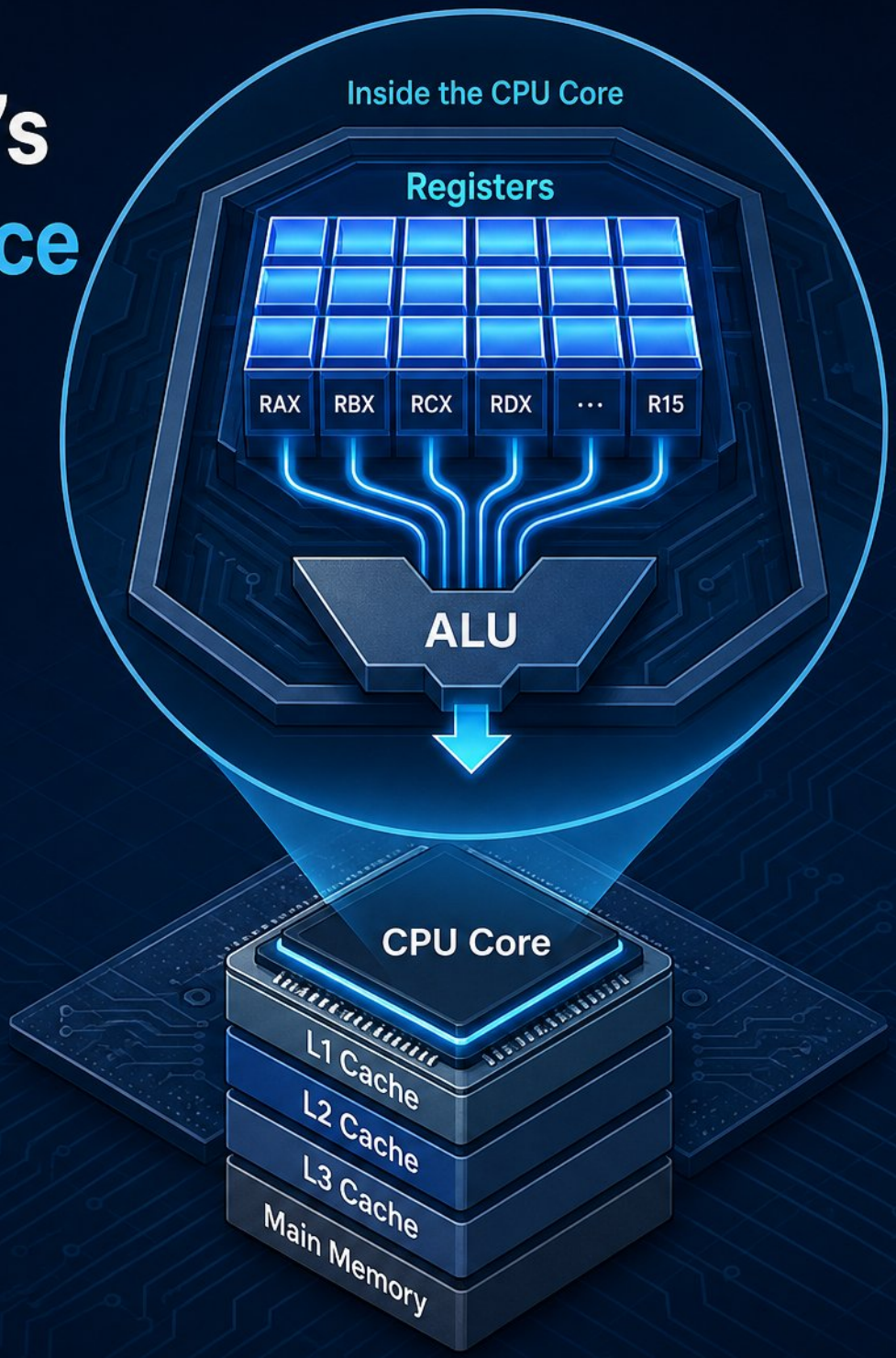
- No single tech is simultaneously fast, large, and cheap — a fundamental trade-off exists.
- Hierarchy layers fast/small/expensive (registers, SRAM cache) to slow/large/cheap (DRAM, SSD, HDD).
- Each tier is roughly ~100× slower and ~100× cheaper than the one above it.
- Temporal locality: recently accessed data will likely be accessed again soon.
- Spatial locality: data near a recent access will likely be accessed soon — exploits cache lines.

The Full Memory Pyramid: From Sub-Nanosecond to Milliseconds



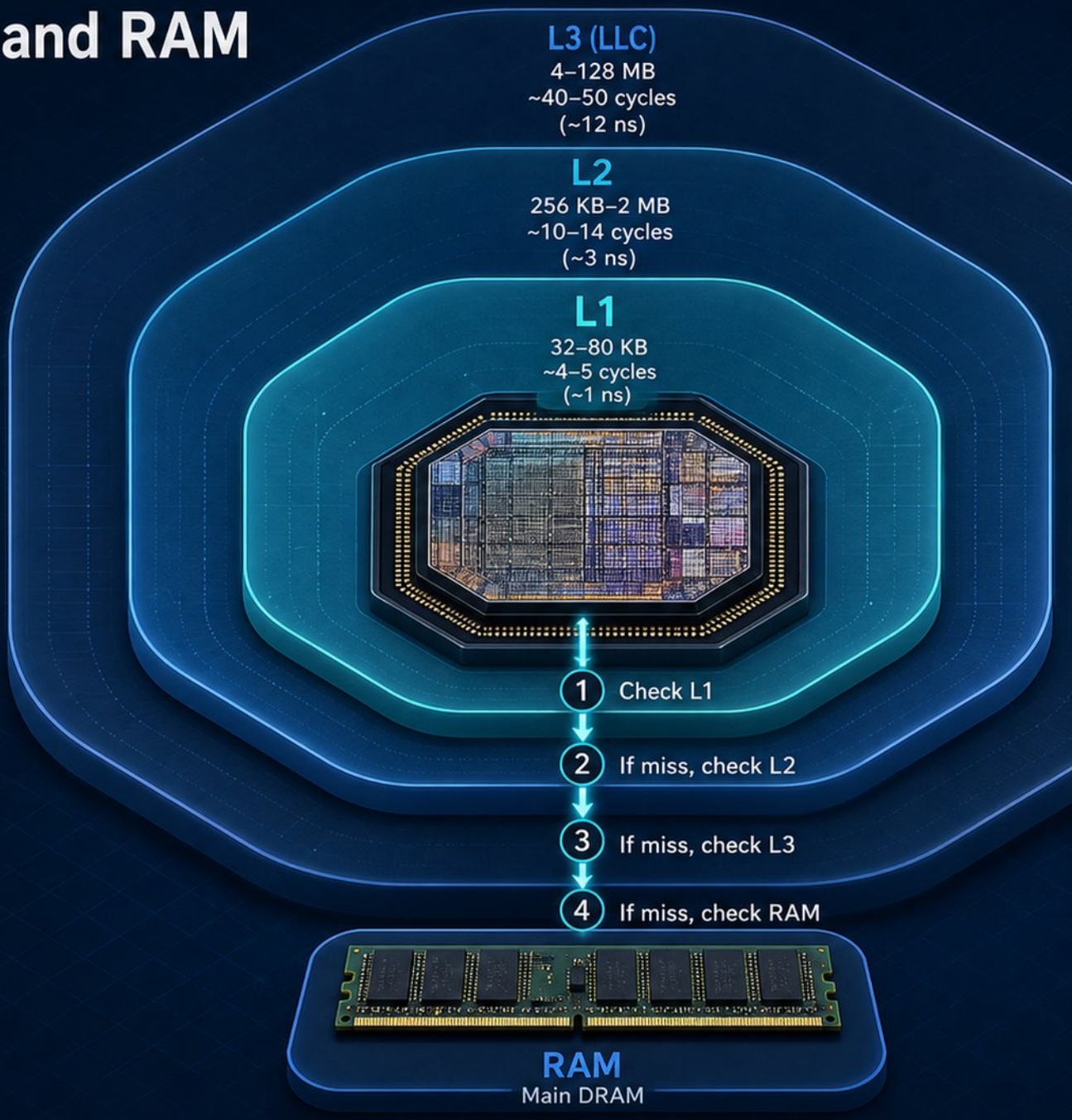
Registers: The CPU's Immediate Workspace

- Smallest, fastest storage inside the CPU core (0–1 cycle, ~0.3 ns)
- Hold operands, memory addresses, and intermediate results for the ALU
- Data must reach a register before any arithmetic or logic operation can run
- Modern x86-64: 16 general-purpose 64-bit registers, plus vector registers
- Managed by the compiler; not directly addressable in memory



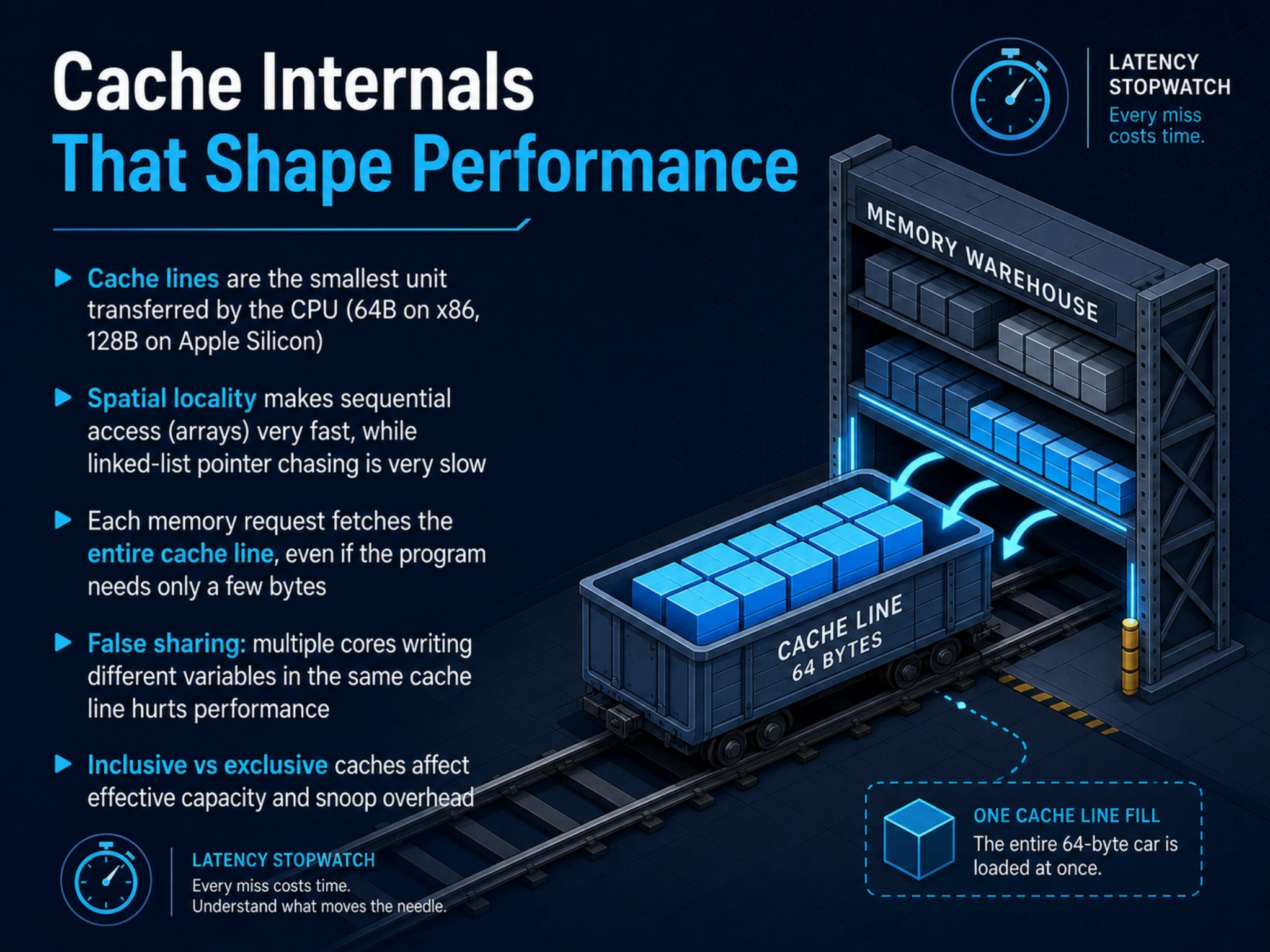
Cache Memory: The High-Speed Buffer Between CPU and RAM

- SRAM-based buffer between CPU registers and main DRAM
- L1: private per-core, 32–80 KB, ~4–5 cycles (~1 ns)
- L2: private per-core, 256 KB–2 MB, ~10–14 cycles (~3 ns)
- L3 (LLC): shared across cores, 4–128 MB, ~40–50 cycles (~12 ns)
- Hit vs. miss: CPU checks L1 → L2 → L3 → RAM



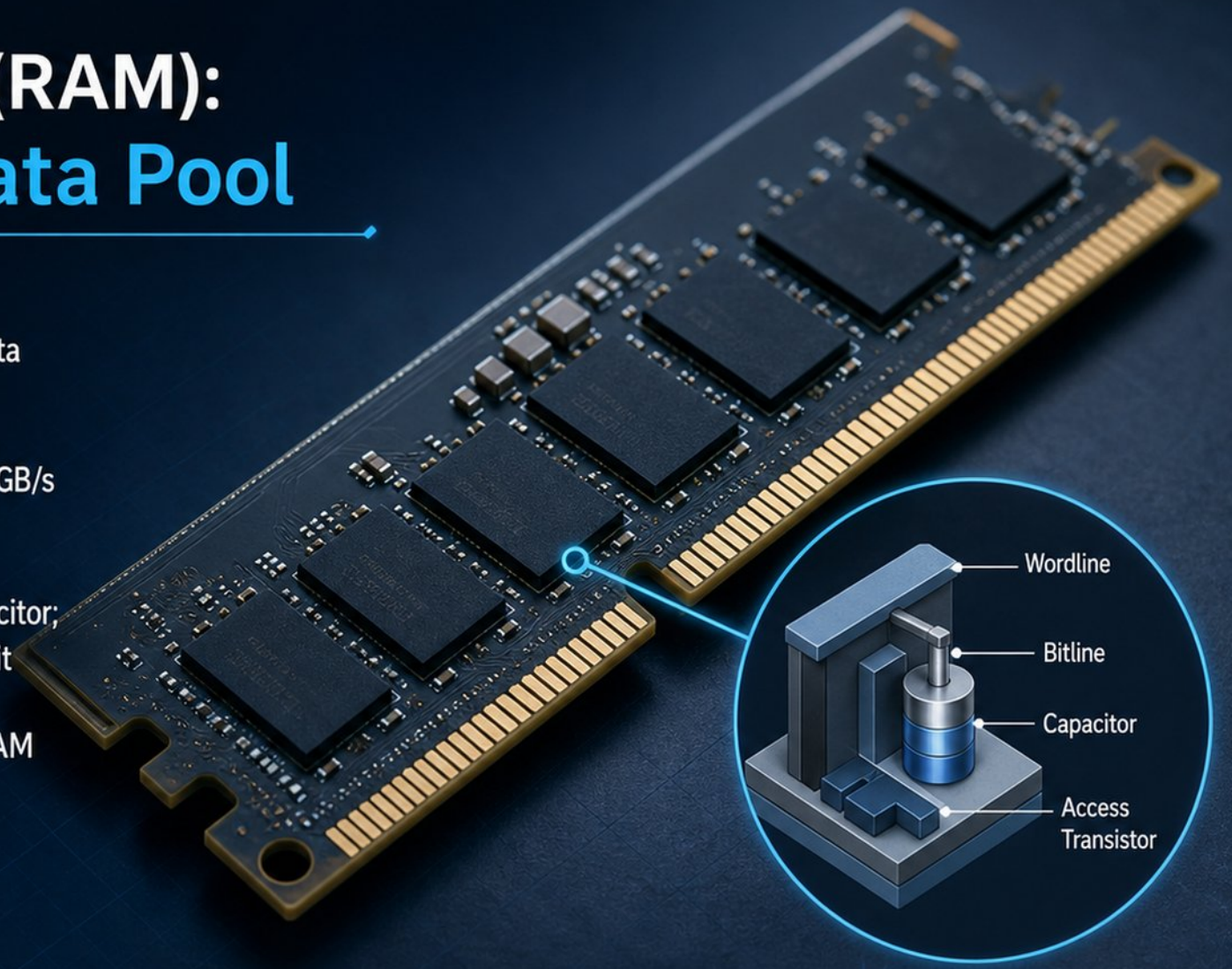
Cache Internals
That Shape Performance

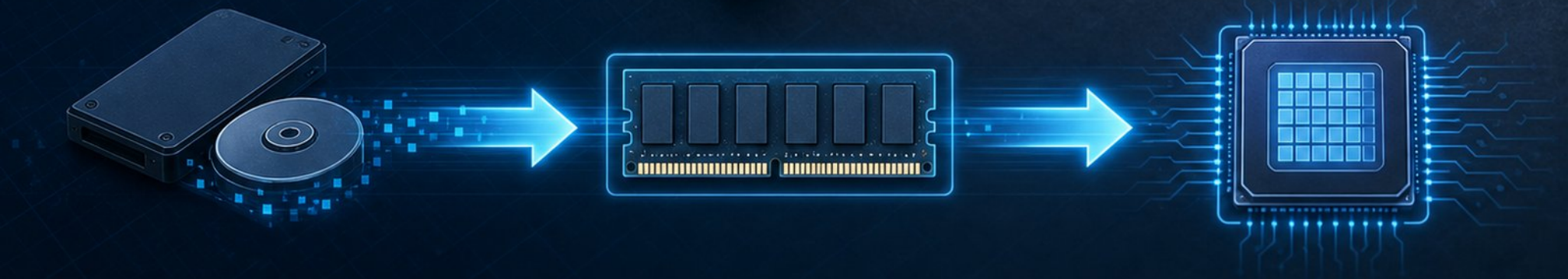
- **Cache lines** are the smallest unit transferred by the CPU (64B on x86, 128B on Apple Silicon)
- **Spatial locality** makes sequential access (arrays) very fast, while linked-list pointer chasing is very slow
- Each memory request fetches the **entire cache line**, even if the program needs only a few bytes
- **False sharing:** multiple cores writing different variables in the same cache line hurts performance
- **Inclusive vs exclusive** caches affect effective capacity and snoop overhead



Main Memory (RAM): The Active Data Pool

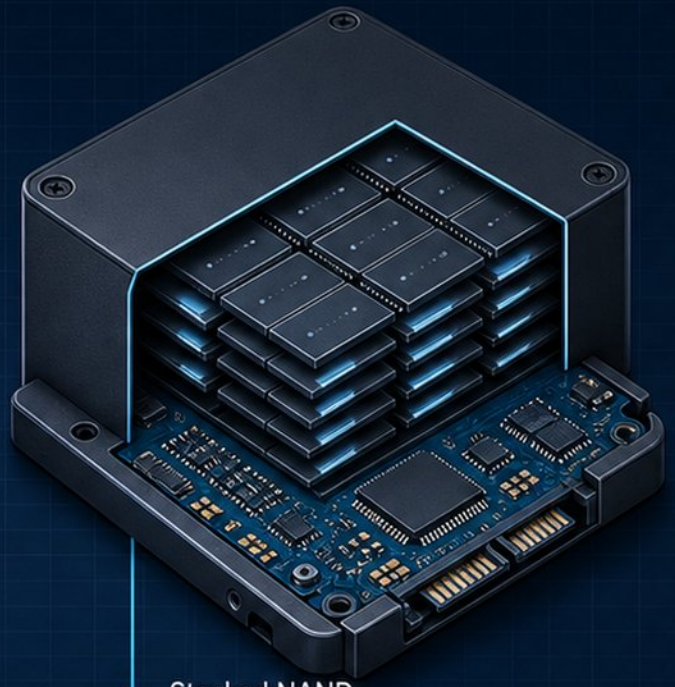
- Fast, volatile DRAM storage holding active OS, programs, and data
- Hierarchical sweet spot: 80–100 ns latency, 8–512 GB capacity, 50–300 GB/s bandwidth (DDR5)
- DRAM cell uses 1 transistor + 1 capacitor; SRAM cache uses 6 transistors per bit
- Cost and density trade-off defines RAM as denser, cheaper, but slower than SRAM cache





Long-Term Storage: Persistent Data at Scale

SSD



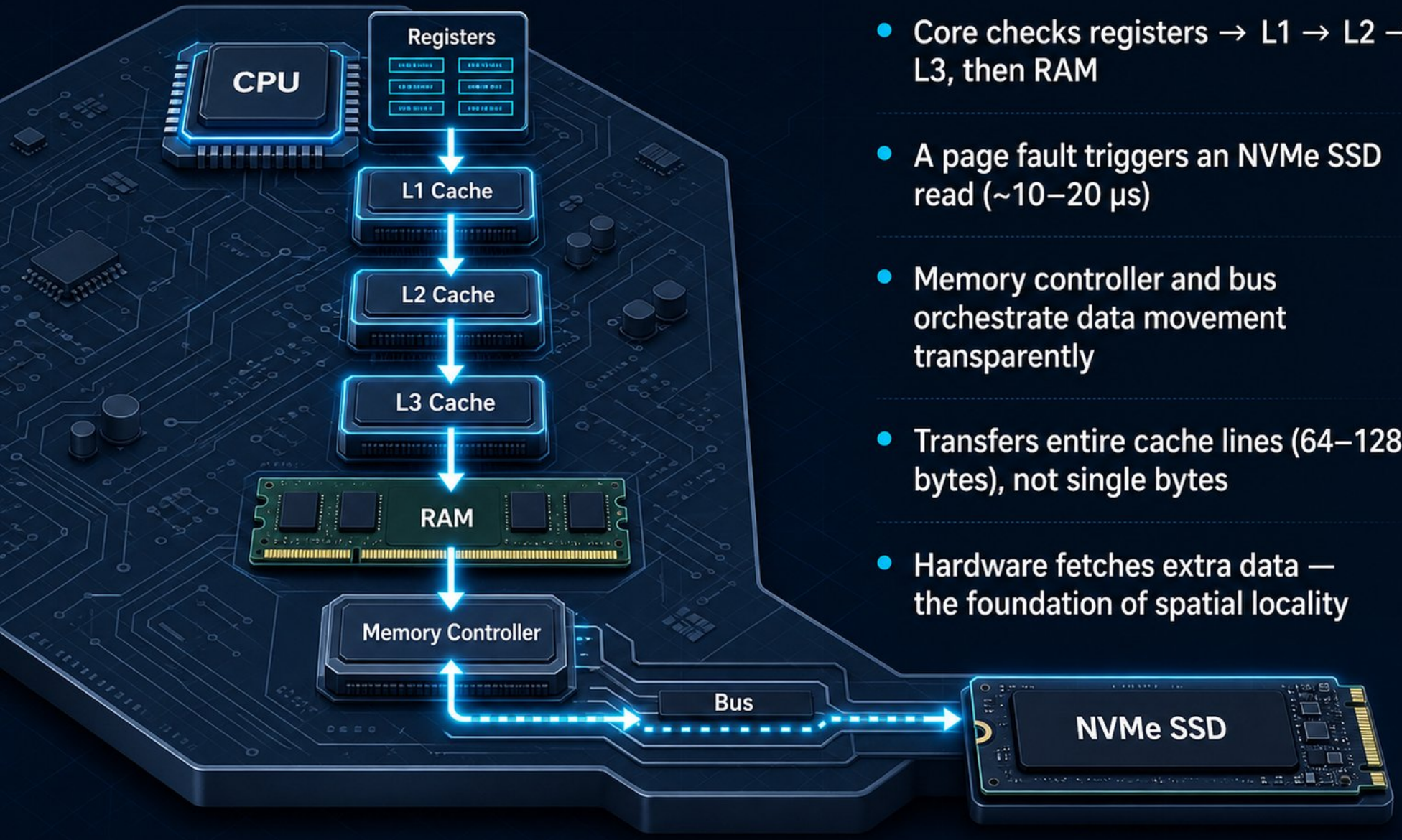
- Stacked NAND Flash Chips
- Controller Board
- Solid-State Housing

HDD

- Magnetic Platter
- Spindle Motor
- Actuator Arm

- Non-volatile memory: **SSDs** (NAND flash), **HDDs** (magnetic platters), and tape retain data without power
- **NVMe SSD** random read ~10–20 µs; **HDD** seek ~5–10 ms — 100,000× slower than RAM
- Throughput gap: **NVMe** ~14 GB/s vs. **HDD** ~200 MB/s; both dwarfed by **DDR5**'s 50–300 GB/s
- Virtual memory extends RAM onto **SSD/HDD** when physical memory is exhausted, causing severe slowdowns

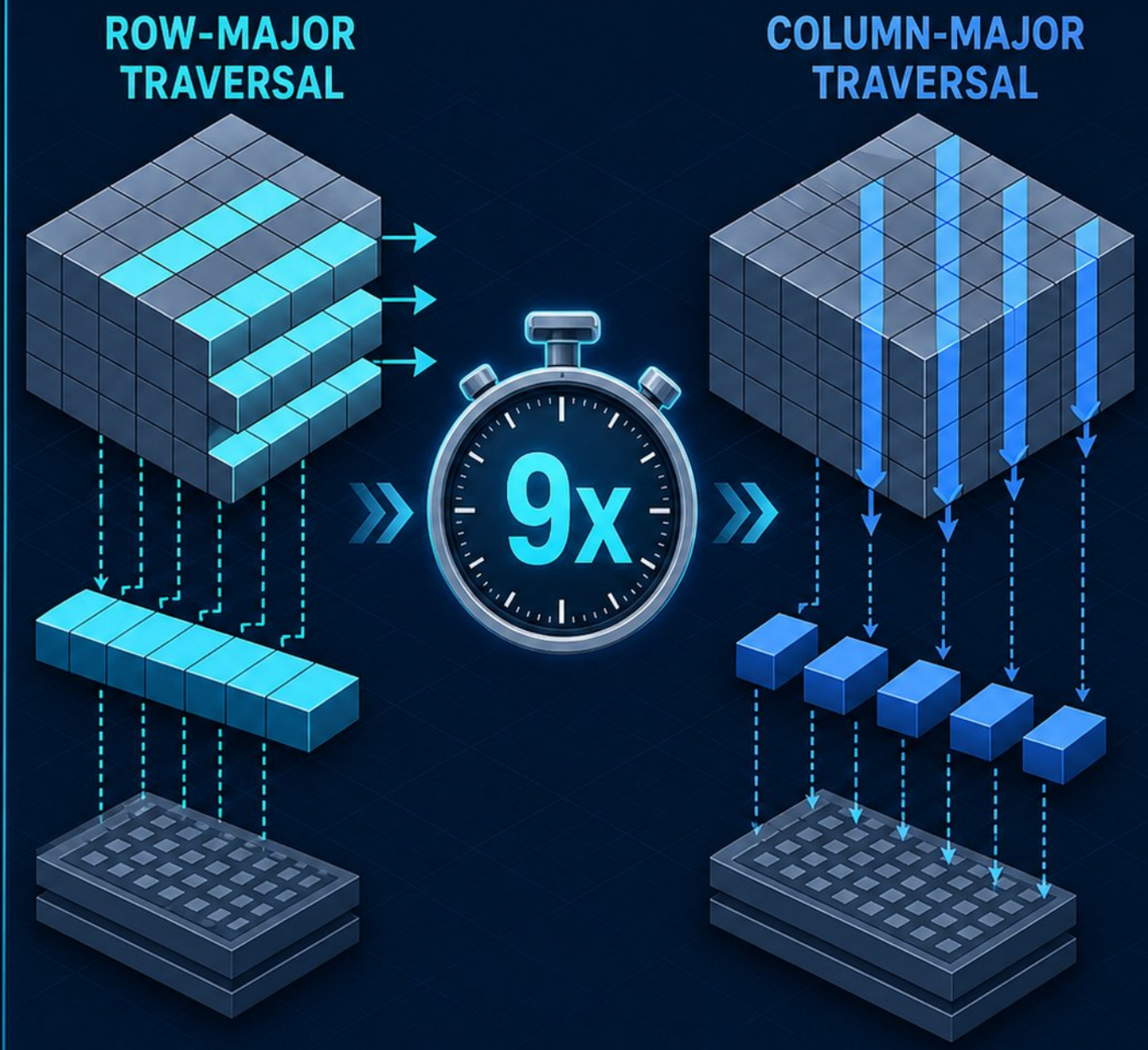
How Information Moves: Tracing a Data Read Through the Hierarchy



- Core checks registers → L1 → L2 → L3, then RAM
- A page fault triggers an NVMe SSD read (~10–20 µs)
- Memory controller and bus orchestrate data movement transparently
- Transfers entire cache lines (64–128 bytes), not single bytes
- Hardware fetches extra data — the foundation of spatial locality

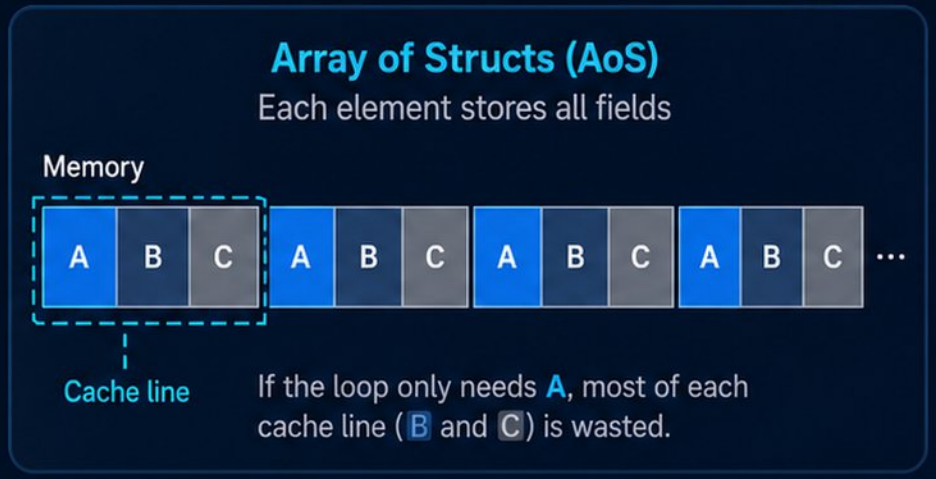
Practical Implications 1: How the Hierarchy Governs Real Performance

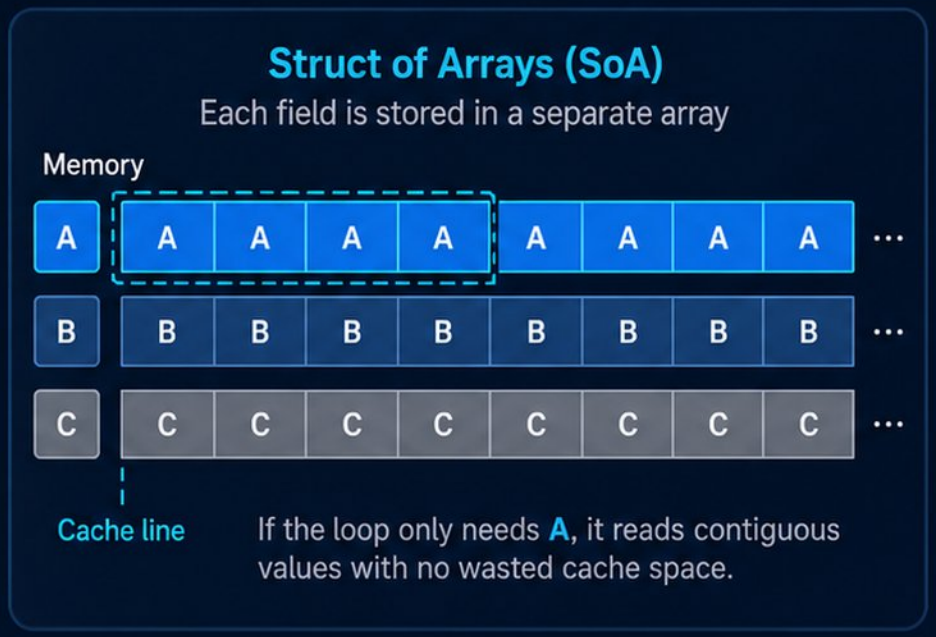
- A single DRAM miss wastes hundreds of CPU cycles and dominates execution time.
- Row-major traversal runs ~9× faster than column-major on large matrices because of cache-line reuse.
- std::vector iteration beats std::list by 10–17× — contiguous memory eliminates pointer-chasing cache misses.

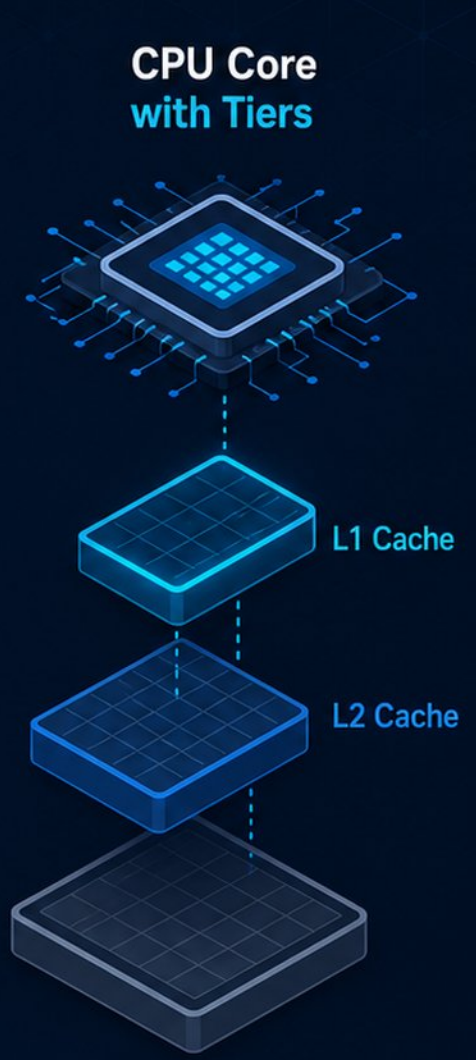


Practical Implications 2:
Writing Cache-Friendly Code

- **AoS vs. SoA:** store fields separately when loops access only a subset, preventing wasted cache lines.
- **Loop tiling:** break large loops into blocks fitting L1/L2 to reduce capacity misses.
- **False sharing:** align per-thread hot variables to 64-byte boundaries with alignas or padding.
- **Use std::vector** over std::list for hot iteration; prefer contiguous arrays over pointer-chasing structures.







Summary: The Complete Picture of Computer Memory

Tier	Managed by	Volatility	Key role
Registers	CPU	⚡	Fastest storage for current operations
L1	CPU	⚡	Very fast cache close to the CPU
L2	CPU	⚡	Faster cache backing L1 cache
L3	CPU	⚡	Last-level cache shared by cores
RAM	OS	⚡	Main memory for running programs
SSD	OS / hardware	🔒	Persistent storage for data and programs

- Registers → L1 → L2 → L3 → RAM → SSD: each tier backs the faster one above it
- Speed drops ~100×, capacity grows ~1000×, and cost per bit falls at each level
- Hardware and OS transparently move data, but the tiers define real performance limits

Key lesson: wall-clock speed depends on keeping your working set in the fastest tiers

PersonWise.

PersonWise • AI digital-human course platform

Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/54b8f571/public