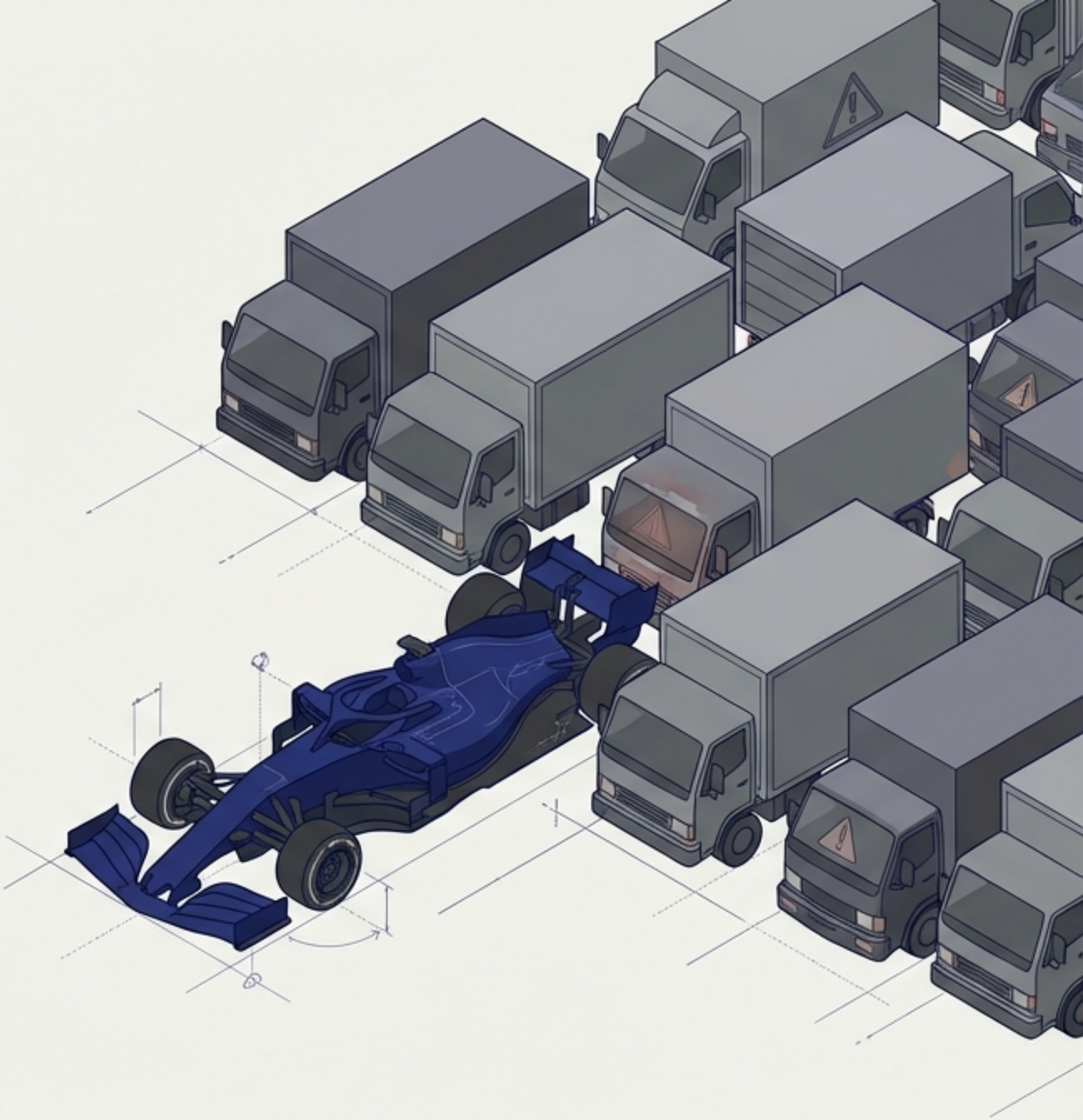
The Bottleneck and the Bypass

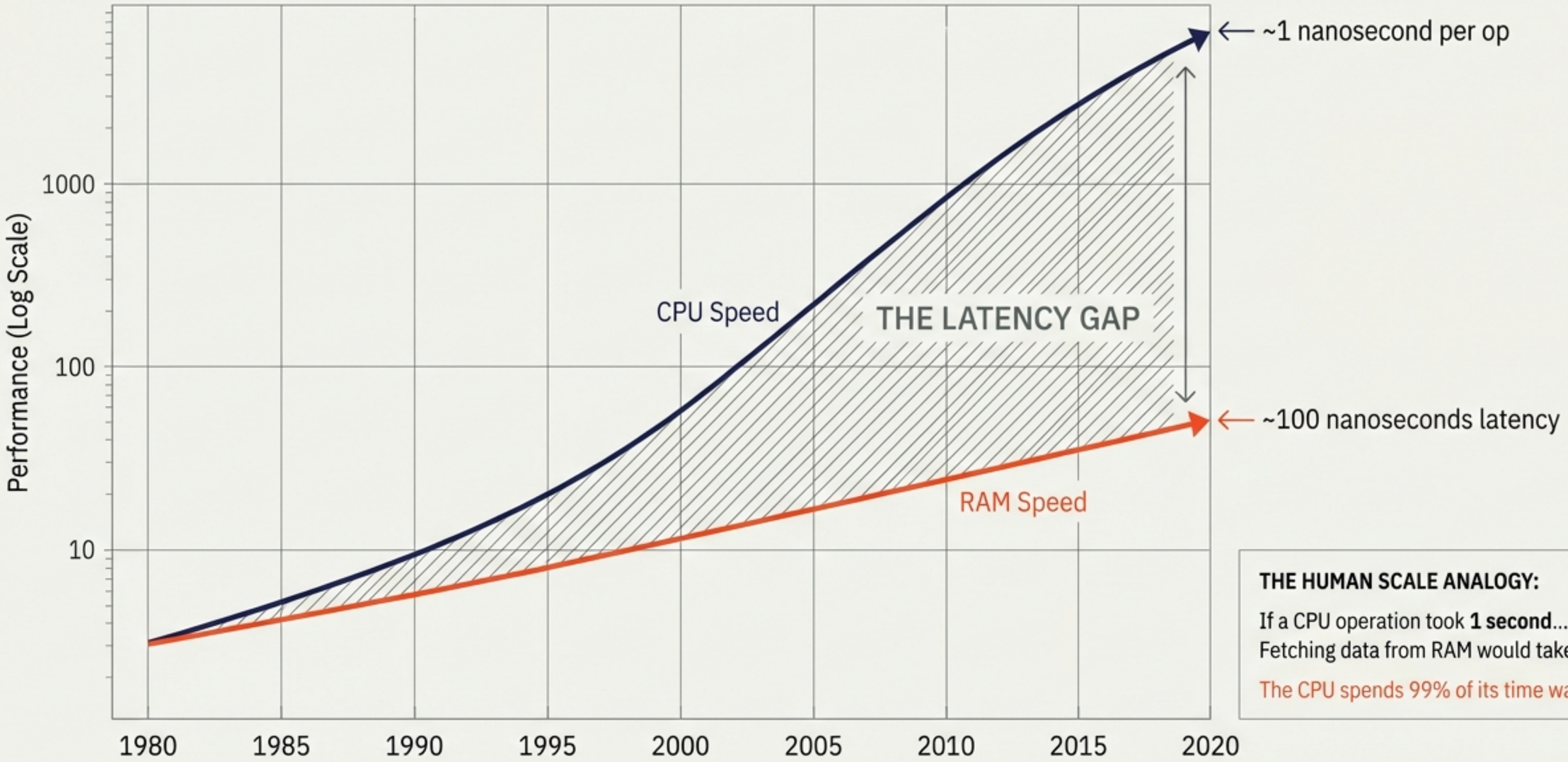
Computer Architecture & Memory Hierarchy: Why modern hardware is fast, but your code is slow.



Target: University CS/DS Students

Context: The Physics of NumPy Vectorisation

The Core Conflict: Computers Have a Split Personality

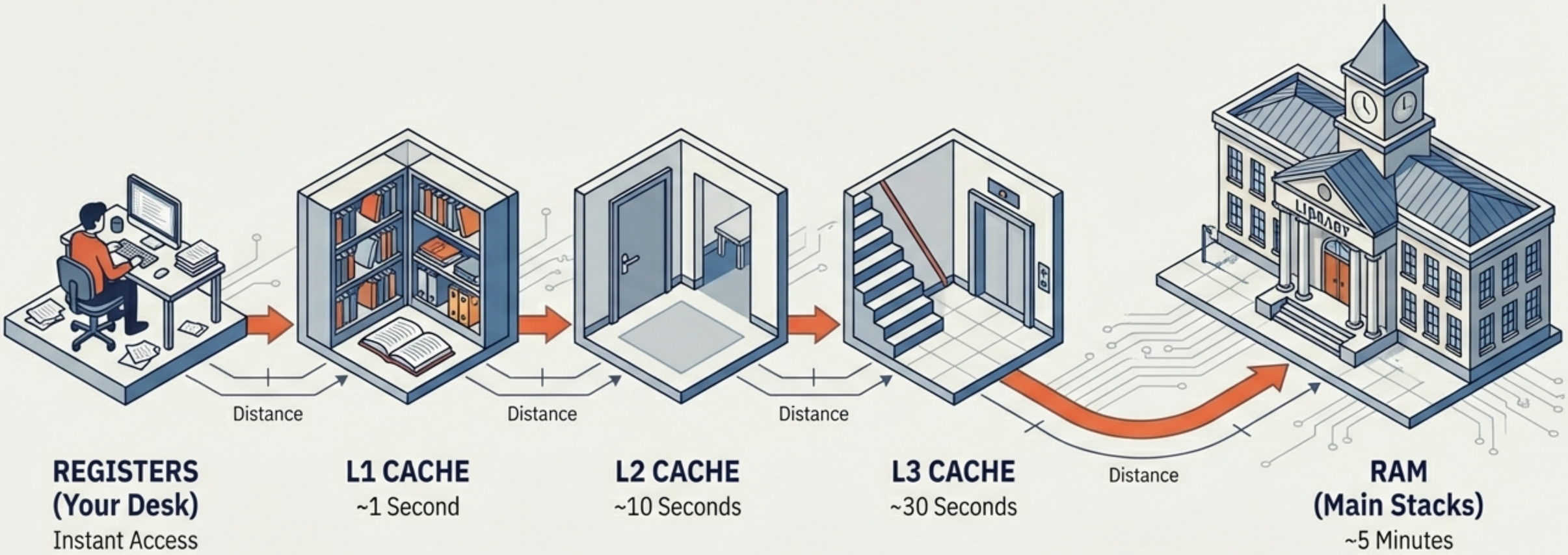


THE HUMAN SCALE ANALOGY:

If a CPU operation took **1 second**...
Fetching data from RAM would take **2 minutes.**

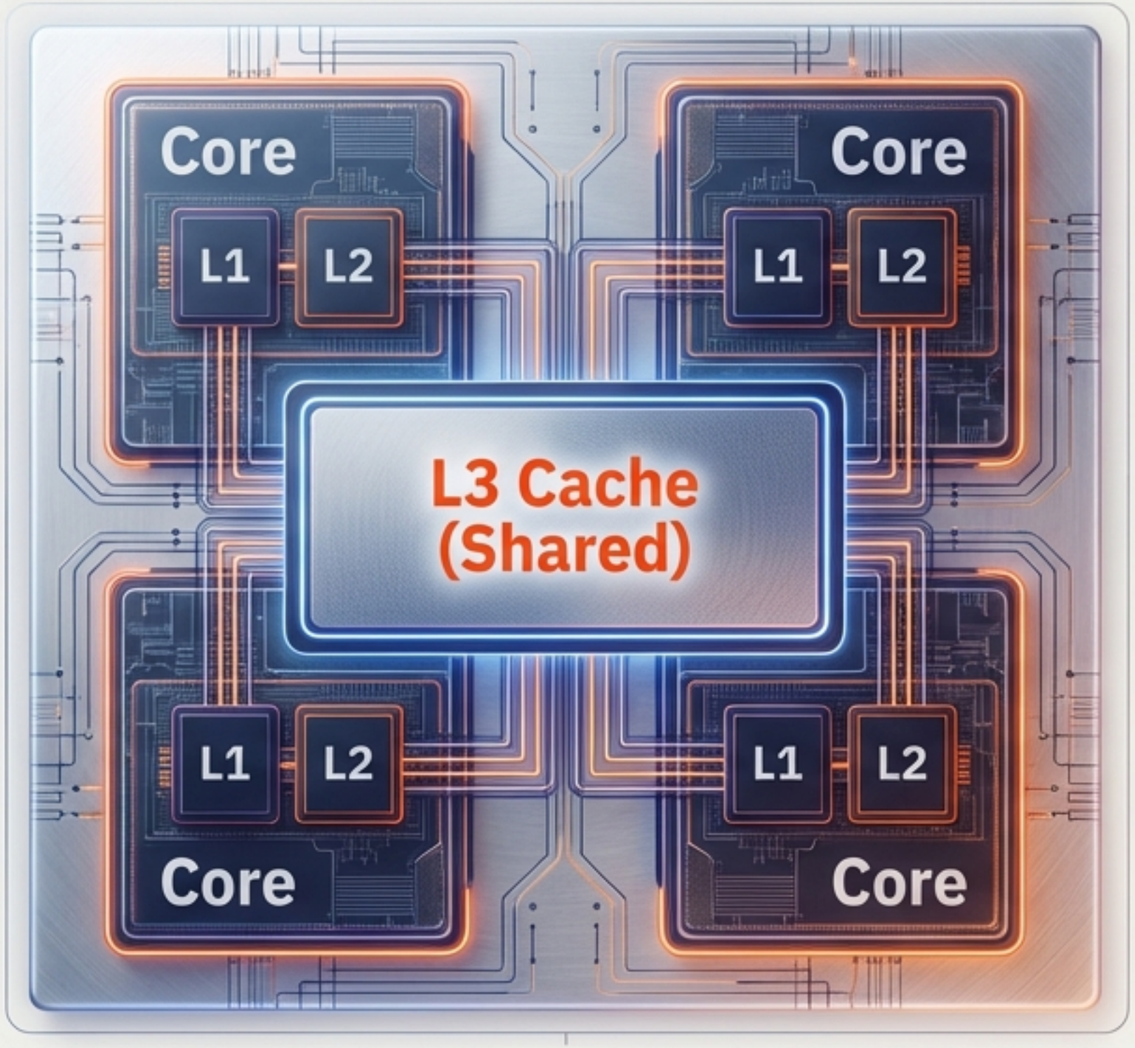
The CPU spends 99% of its time waiting.

The Library Analogy: Visualising the Cost of Distance



The goal of high-performance coding is to keep your work on the desk, avoiding the long walk to the main stacks.

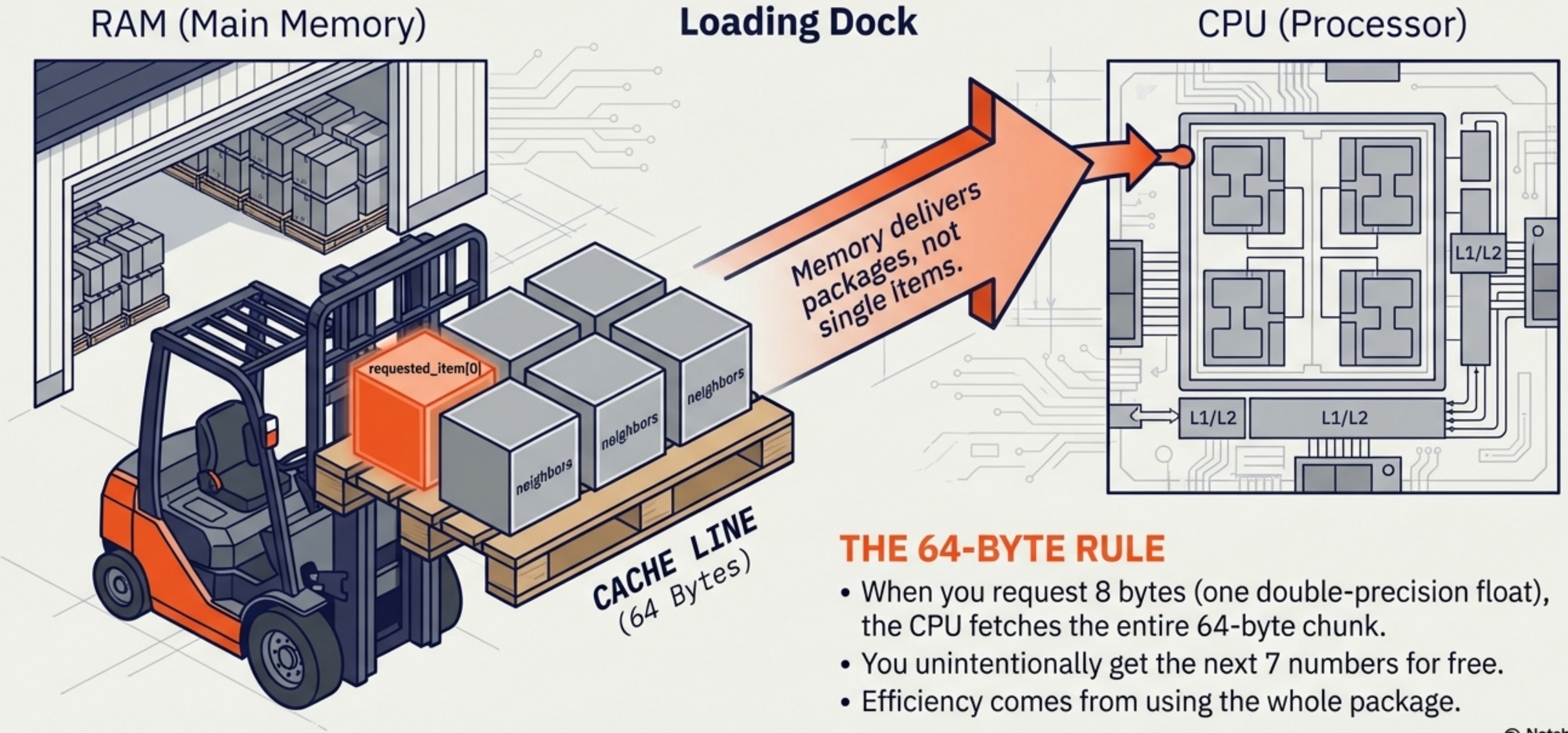
Inside the Processor: The Cache Hierarchy



COMPONENT	LOCATION	SIZE	LATENCY
L1 CACHE	On-Core	32-64 KB	1-2 ns
L2 CACHE	On-Core	256 KB - 1 MB	5-10 ns
L3 CACHE	Shared	8-32 MB	20-50 ns
RAM	Motherboard	16-64 GB	~100 ns

L1 and **L2** are private to the core (Fast/Small). L3 is the communal staging area before the slow trek to RAM.

The Unit of Transfer: Understanding Cache Lines

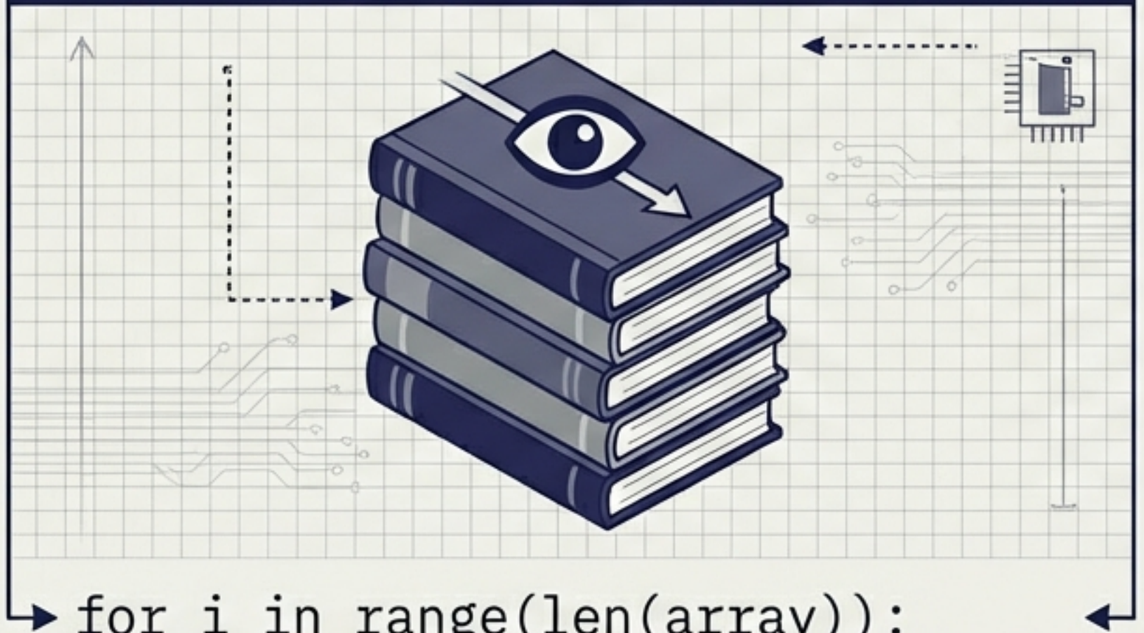


THE 64-BYTE RULE

- When you request 8 bytes (one double-precision float), the CPU fetches the entire 64-byte chunk.
- You unintentionally get the next 7 numbers for free.
- Efficiency comes from using the whole package.

Access Patterns Determine Speed

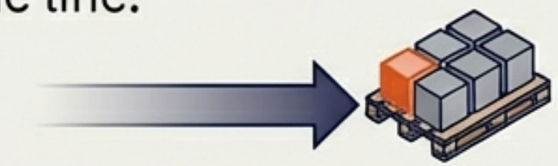
SEQUENTIAL ACCESS



```
for i in range(len(array)):
    process(array[i])
```

Reading page-by-page. The next data point is already in the cache line.

CACHE HIT

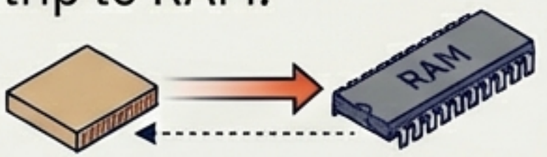


RANDOM ACCESS

```
for i in random_indices:
    process(array[i])
```

Reading random pages from random books. Every access requires a trip to RAM.

CACHE MISS

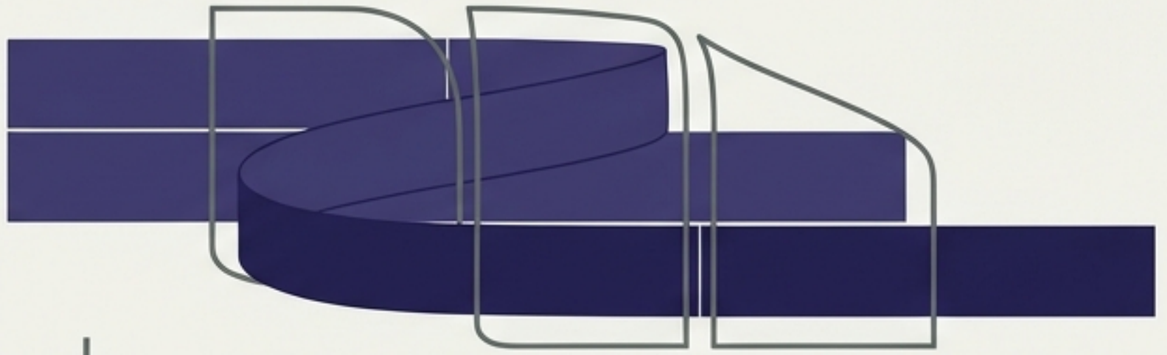


Data Layout: Row-Major vs. Column-Major

Step 1: A 2x3 Grid

[[1	2	3]]
[[4	5	6]]

Step 2: The grid “unrolling” like a carpet



Step 3: A single long strip of memory tape

NumPy (C-Style) Memory Layout

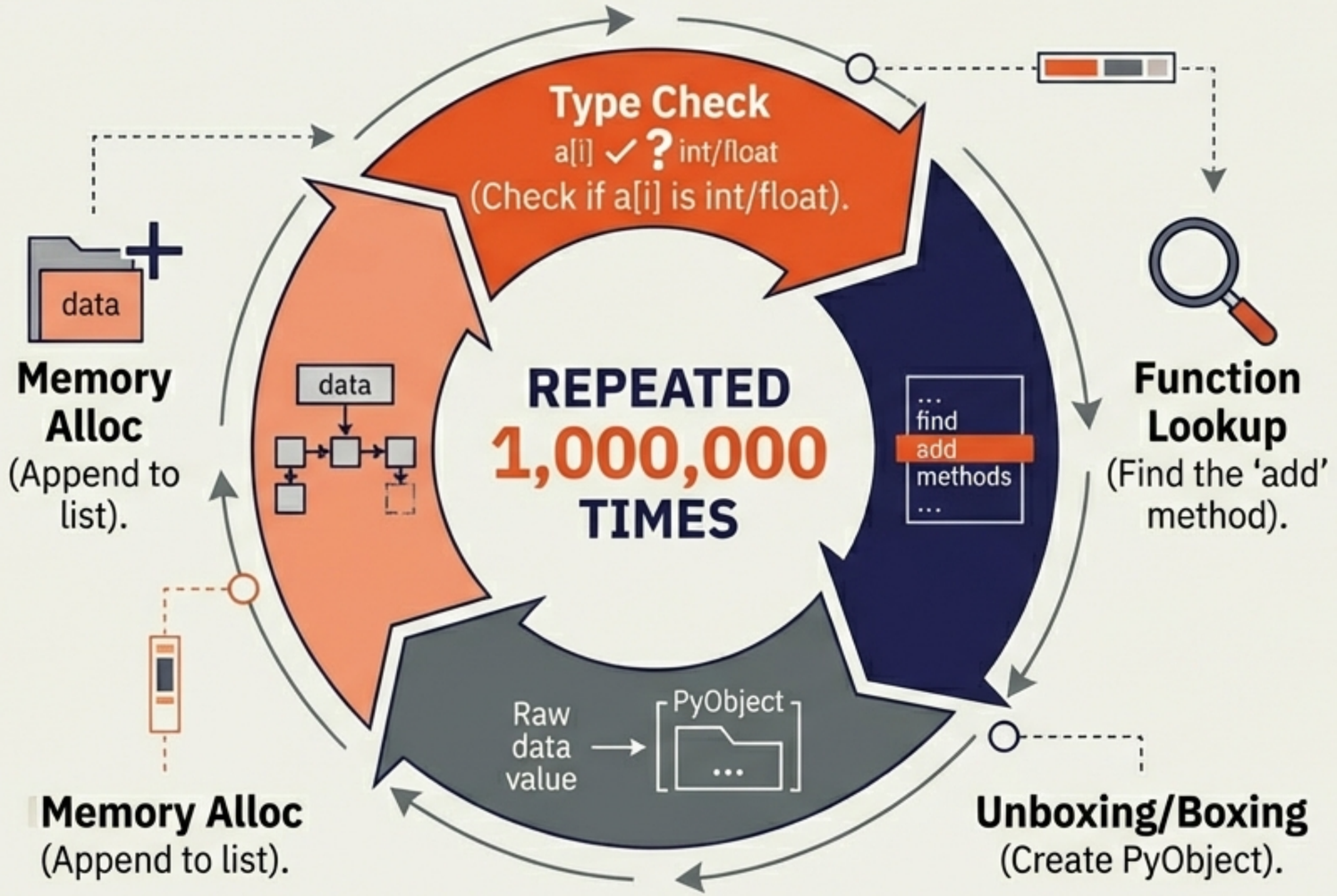
```
[1, 2, 3, 4, 5, 6]
```

Sequential in Memory = Row-by-Row.

Contrast Text: Iterating across rows exploits the `Cache Line`. Iterating down column, (`1 -> 4 -> 2`) fights against physics (`Strided Access`).

The Problem with Naive Python

Why standard loops are slow.



The Consequence:

- Standard Python treats every number as a unique object.

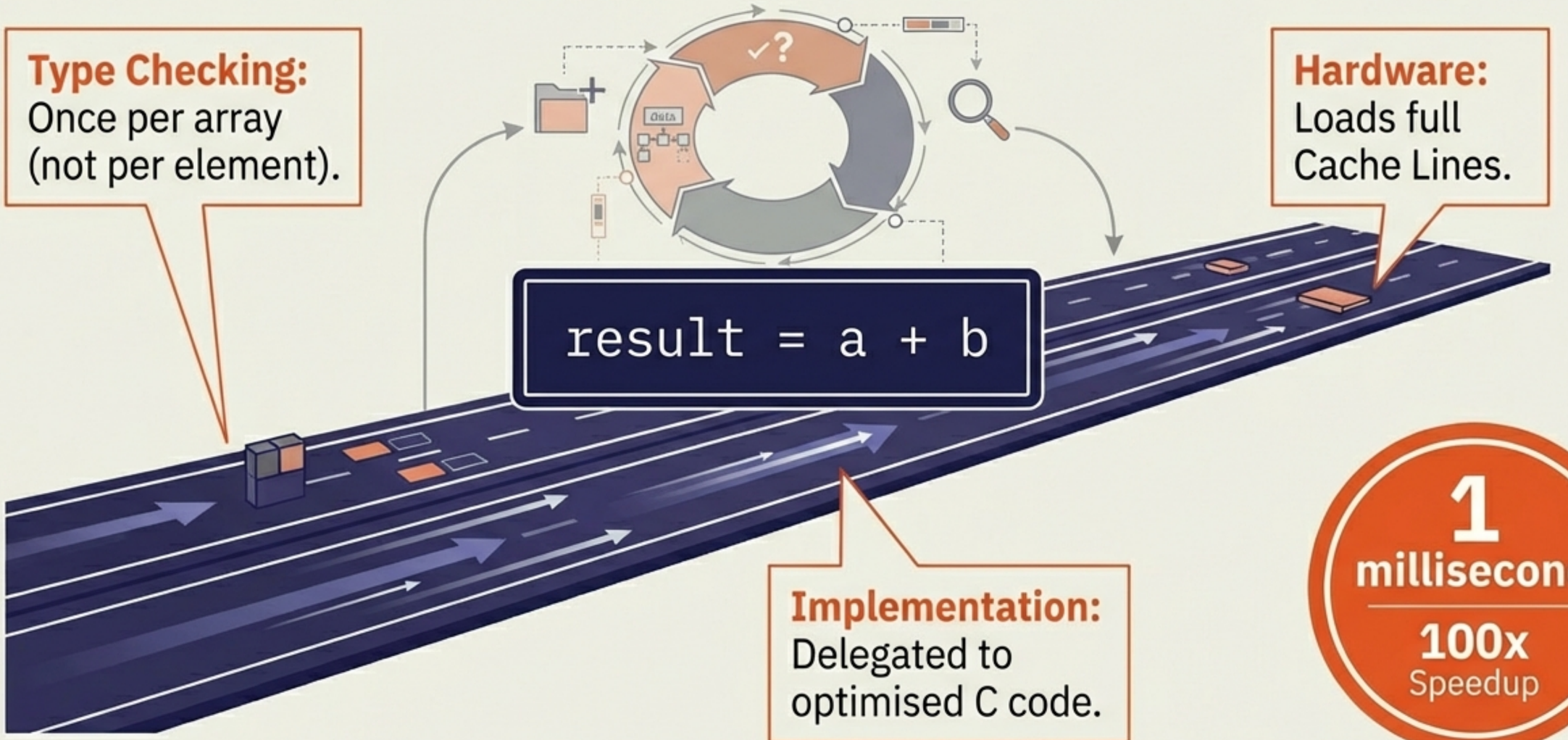
Total Time: ~100 milliseconds.

Analogy:

Buying 1000 apples, but paying for each one separately with a credit card.

The Solution: NumPy Vectorisation

Type Checking:
Once per array (not per element).



Hardware:
Loads full Cache Lines.

Implementation:
Delegated to optimised C code.

1 millisecond

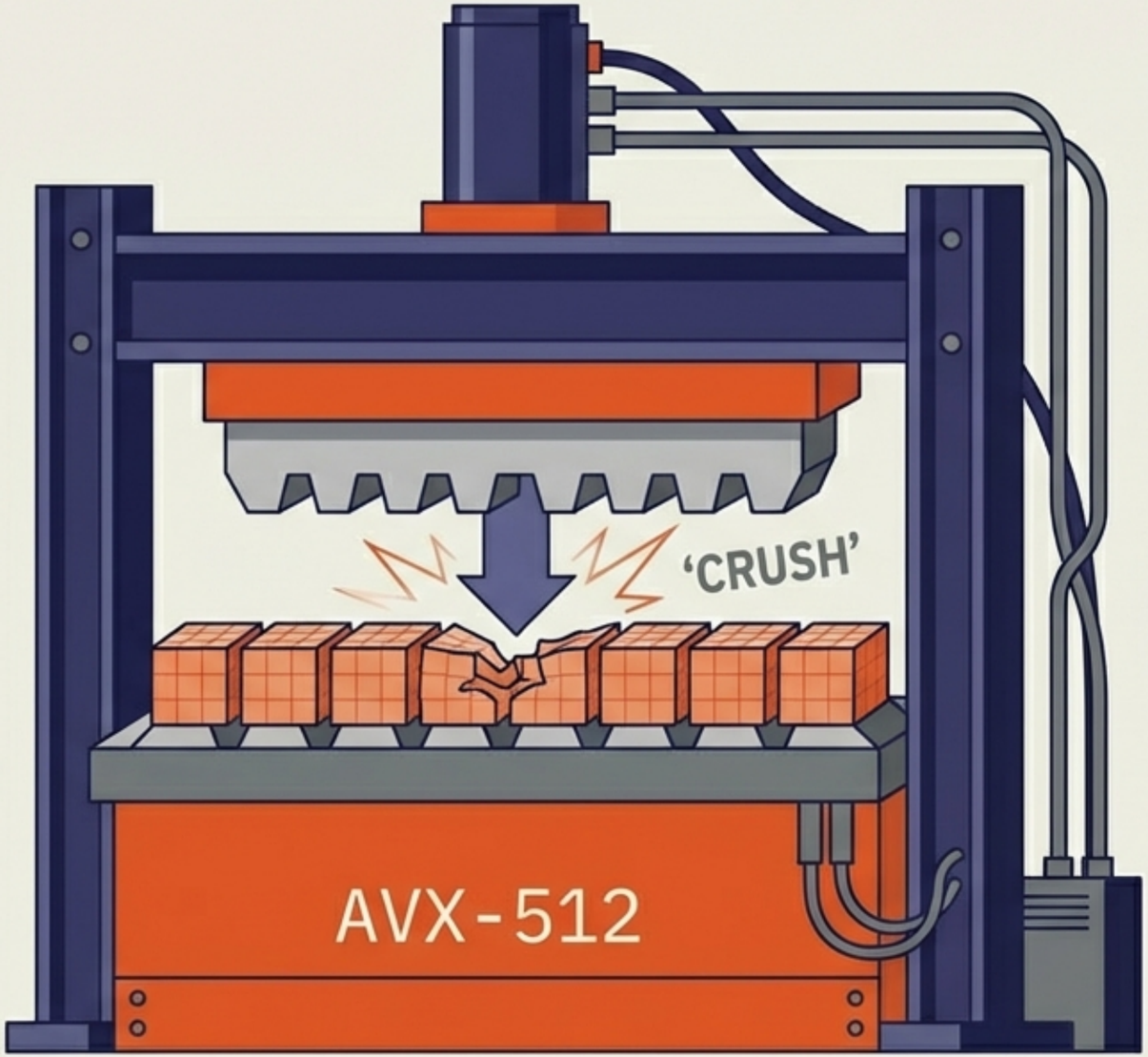
100x Speedup

Unlocking SIMD

Single Instruction, Multiple Data

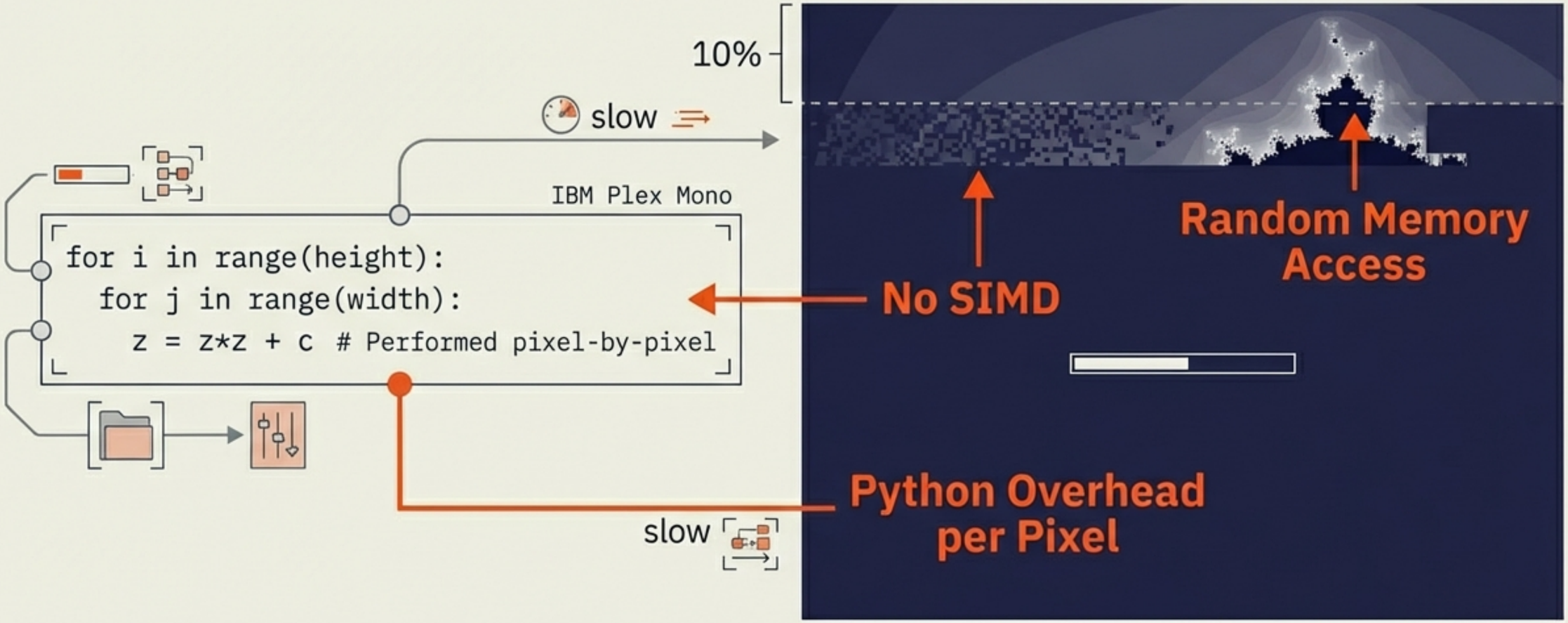
Scalar Processing
(Standard Python)

Vector Processing
(SIMD/NumPy)



- Modern CPUs have special registers (SSE, AVX) that process 4, 8, or 16 numbers in one clock cycle.
- Only **vectorised code** can trigger these hardware superpowers.

Case Study: The Mandelbrot Set (Naive Approach)



BENCHMARK: ~45 SECONDS

Case Study: The Mandelbrot Set (NumPy Approach)

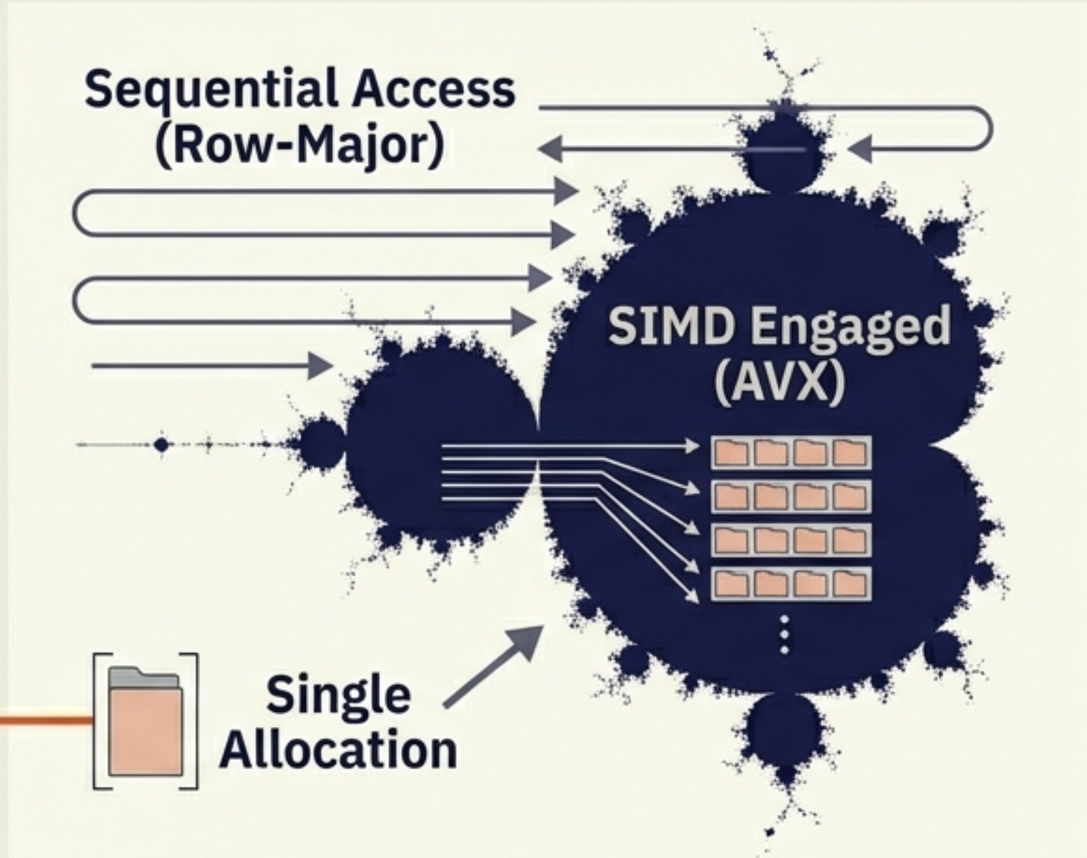
The Code

slow

IBM Plex Mono

```
X, Y = np.meshgrid(x, y)
Z[mask] = Z[mask]**2 + C[mask]
                    # Operates on grid
```

slow

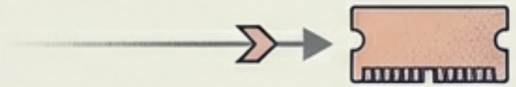


BENCHMARK: 2-3 SECONDS (20x Speedup)

Architecture-Aware Programming Checklist

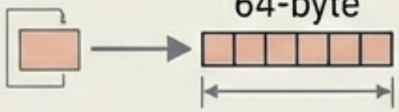
Memory is the Bottleneck

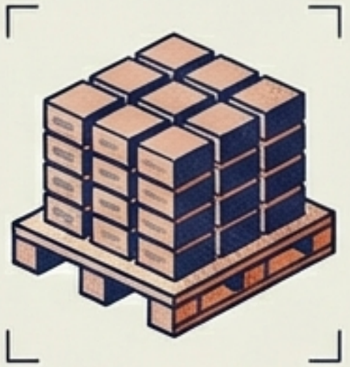
The CPU is fast. RAM is slow. Minimise trips to the library.



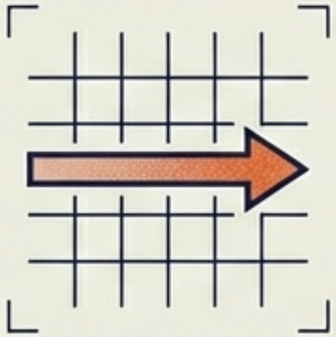
Respect the Cache Line

Data moves in 64-byte chunks. Use the whole chunk.



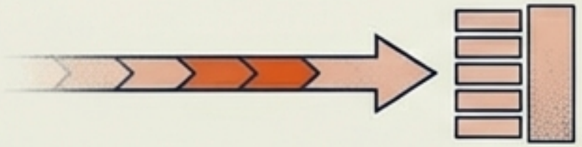


IBM Plex Mono



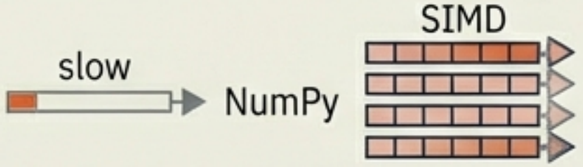
Predictability Wins

Sequential access patterns keep the cache fed.



Vectorise Everything

Use NumPy to unlock SIMD and bypass the Python interpreter.



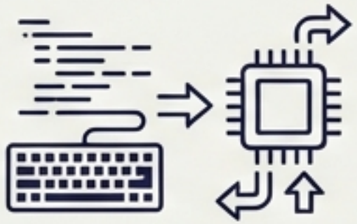
Studio Session: Visualise and Implement

1

Draw-Pair-Share

Sketch the memory hierarchy. Map the cost of distance.

2



Implementation

Convert the naive Mandelbrot loop into a vectorised NumPy grid.

3

Profiling

Measure cache hits and misses. Prove the theory.

GOAL: ACHIEVE A 20x SPEEDUP.