OPTIMIZATION TIER

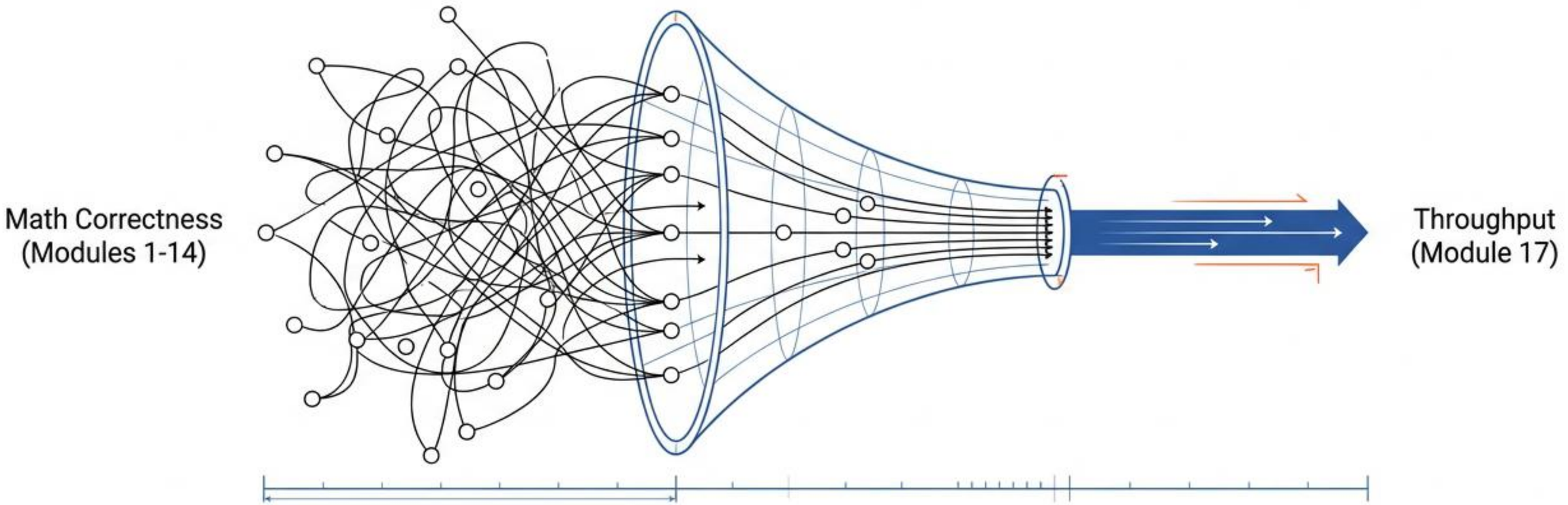
MODULE 17

Acceleration

Hardware and software techniques for deep learning speed

Module 17: Acceleration

Hardware-Aware Optimization: Vectorization, Fusion, and Tiling



****TIER: OPTIMIZATION**

Focus on runtime efficiency and hardware physics.

****PREREQUISITES**

Tensor Operations (Mod 01)
Profiling (Mod 14)

****GOAL**

Bridge the gap between abstract tensor math and silicon reality.

Engineering Editorial

The Scale of the Compute Problem

90% of training time is spent on matrix operations.

The Reality:

A single Transformer forward pass requires billions of FLOPs.

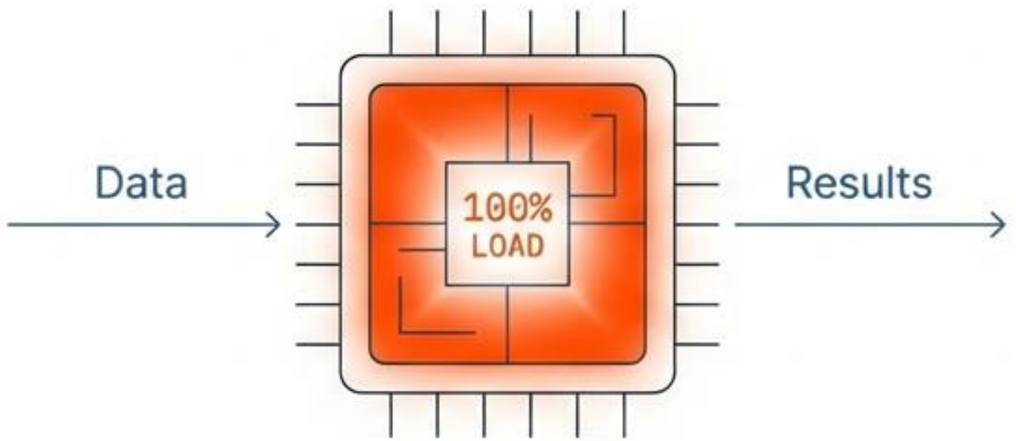
Acceleration is not an optional polish; it is the economic condition for Deep Learning viability.

Engineering Editorial

The Two Enemies: Compute vs. Memory

Optimization is the hunt for the bottleneck.

Enemy 1: Compute Bound



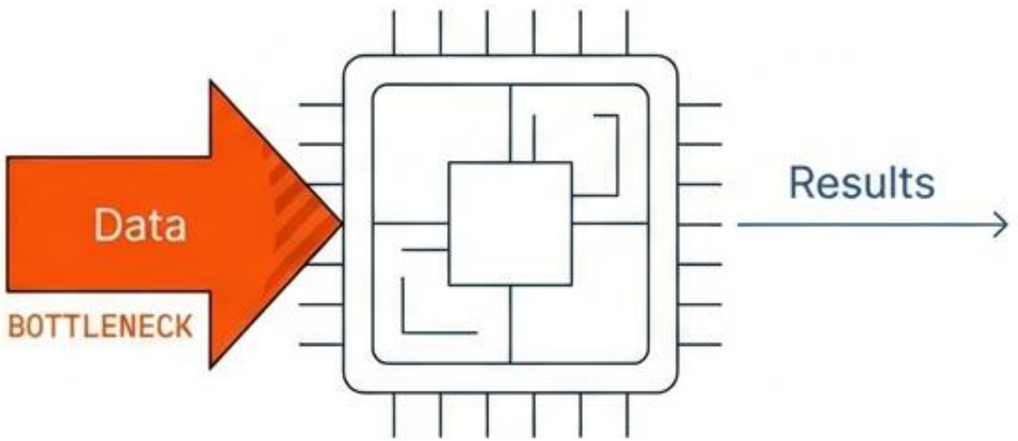
Constraint: Math Speed (FLOPs)

Symptom: Processor at 100% utilization.

Example: Matrix Multiplication (GEMM).

Solution: Vectorization.

Enemy 2: Memory Bound



Constraint: Bandwidth (GB/s)

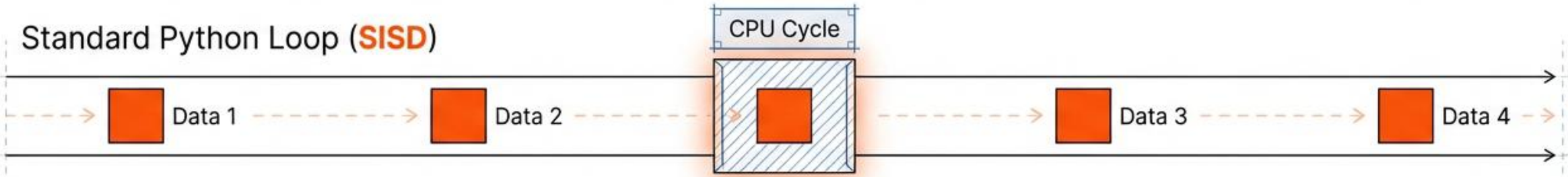
Symptom: Processor idle, waiting for data.

Example: Element-wise Activations (ReLU, GELU).

Solution: Kernel Fusion.

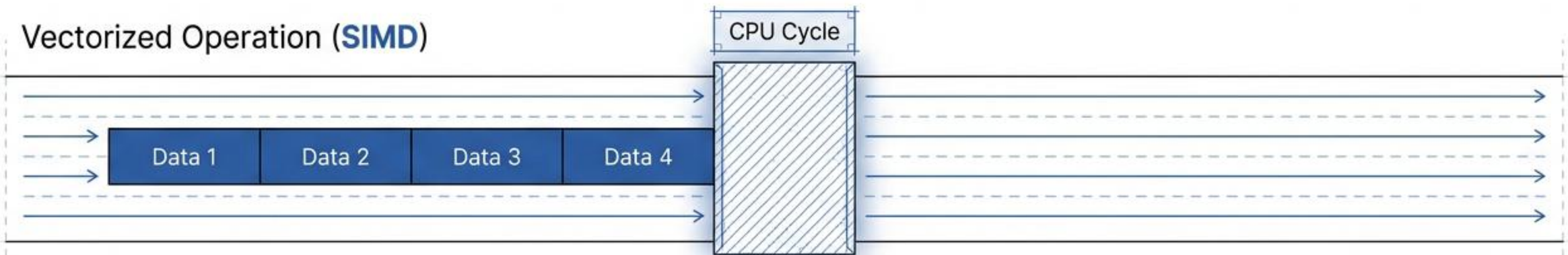
Breaking the Loop: SIMD Vectorization

Standard Python Loop (SISD)




```
Fetch A -> Fetch B -> Add -> Store (Repeats 4 times)
```

Vectorized Operation (SIMD)




```
Fetch 4 As -> Fetch 4 Bs -> Add All -> Store 4 (Happens once)
```

The Python Tax: Standard Python interprets code line-by-line, utilizing only a fraction of the CPU's capability. **SIMD (Single Instruction, Multiple Data)** leverages modern hardware lanes to process 4, 8, or 16 elements per clock cycle.

Implementation: Vectorized Matrix Multiplication

Mapping TinyTorch to NumPy/BLAS

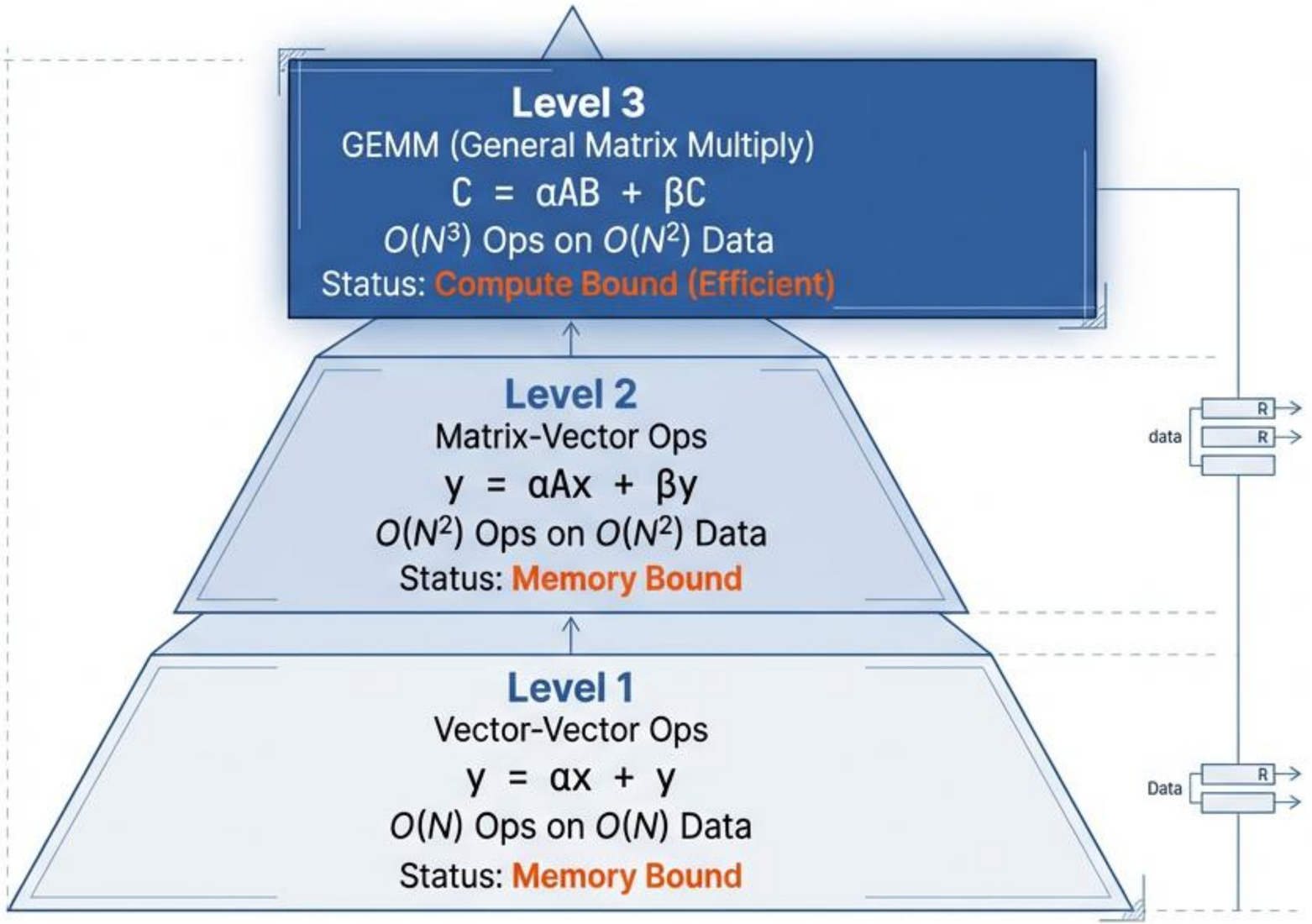
```python
# tinytorch/perf/acceleration.py

def vectorized_matmul(a: Tensor, b: Tensor) -> Tensor:
    """High-performance matrix multiplication via BLAS."""

    # Invariant: Inner dimensions must match
    if a.shape[-1] != b.shape[-2]:
        raise ValueError(f"Shape mismatch: {a.shape} @ {b.shape}")

    # Delegation to Optimized BLAS
    # Uses SIMD, Multi-threading, and Cache Blocking
    result_data = np.matmul(a.data, b.data)

    return Tensor(result_data)
```

The Invariant
Rigid shape alignment is required. Inner dimensions (Columns of A, Rows of B) must allow dot products.

The Heavy Lifting
We do not write loops. We delegate to `np.matmul`, which invokes compiled C/Fortran libraries (BLAS).

Result: 10-100x speedup over native Python loops.

Under the Hood: The Power of GEMM



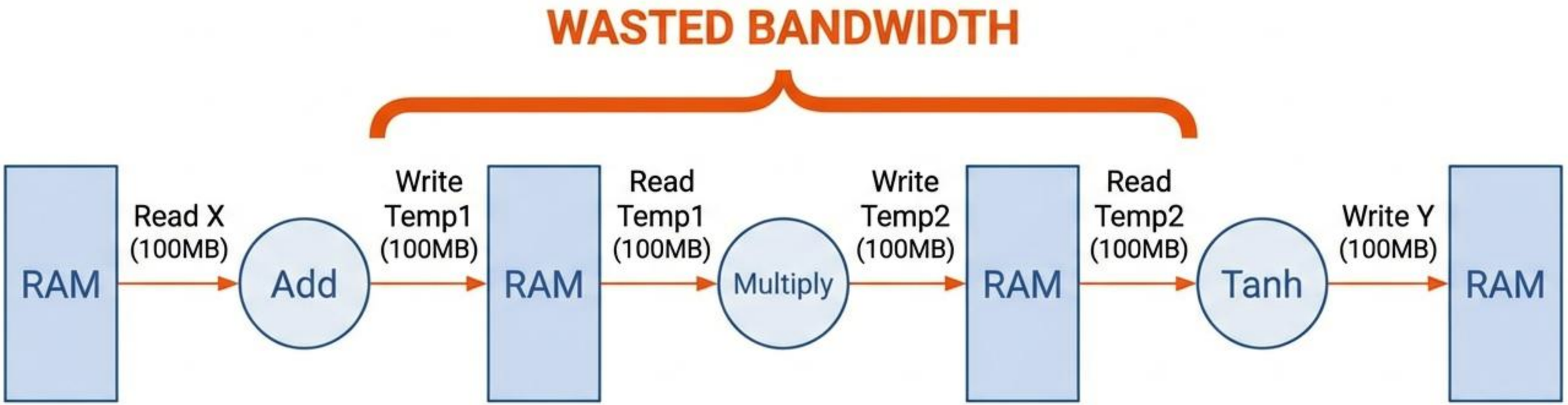
Why GEMM?

Matrix Multiplication allows high **Arithmetic Intensity**. It performs significantly more math operations per byte of memory loaded, allowing the processor to stay busy rather than waiting for RAM.

Engineering Editorial

The Memory Tax

The hidden cost of element-wise operations.



- **The Bottleneck:** Moving data takes orders of magnitude longer than the math itself.
- **The Cost:** Intermediate arrays flood the memory bus, leaving the CPU idle.

Engineering Editorial

The Unfused Baseline (Bad Code)

```
def unfused_gelu(x: Tensor) -> Tensor:
    """Creates 7 intermediate arrays."""
    sqrt_2_over_pi = np.sqrt(2.0 / np.pi)

    # Each line allocates new memory
    temp1 = Tensor(x.data**3)                  # WRITE -> RAM
    temp2 = Tensor(0.044 * temp1.data)     # READ -> WRITE
    temp3 = Tensor(x.data + temp2.data)    # READ -> WRITE
    temp4 = Tensor(sqrt_2_over_pi * temp3) # READ -> WRITE
    temp5 = Tensor(np.tanh(temp4.data))    # READ -> WRITE
    temp6 = Tensor(1.0 + temp5.data)       # READ -> WRITE
    result = Tensor(0.5 * x.data * temp6)  # READ -> WRITE

    return result
```

Analysis:

- **7 Operations**
- **7 Reads + 7 Writes** to Main Memory
- **Traffic:** For a 4MB tensor, this code moves ~64MB of data.
- **Status:** CPU Idle.

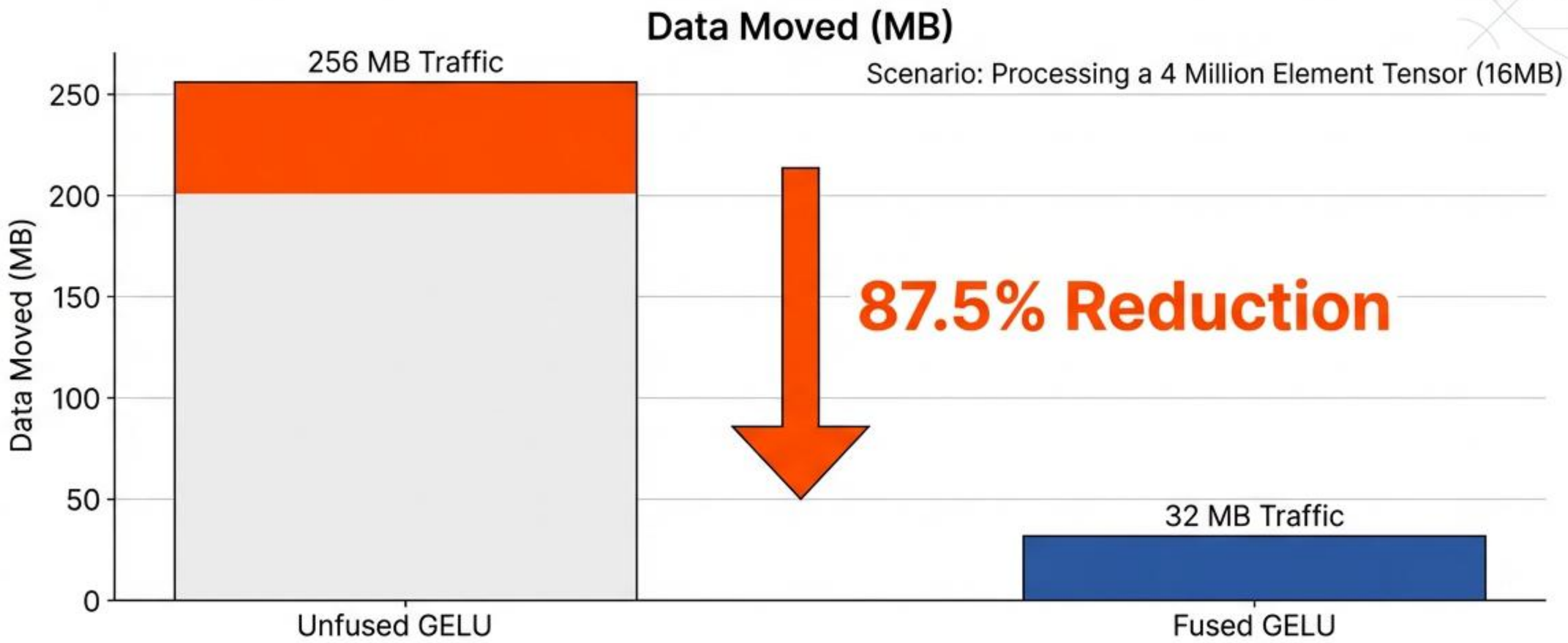
Implementation: Kernel Fusion (Good Code)

```python
# tinytorch/perf/acceleration.py

def fused_gelu(x: Tensor) -> Tensor:
    """Combines all ops into a single kernel."""
    sqrt_2_over_pi = np.sqrt(2.0 / np.pi)

    # Single expression: 1 Read, 1 Write
    # Intermediate values stay in CPU Registers/L1 Cache
    result_data = 0.5 * x.data * (
        1.0 + np.tanh(sqrt_2_over_pi * (x.data + 0.044715 * x.data**3))
    )

    return Tensor(result_data)
```

Analysis:

- **1 Compound Operation**
- 1 Read (Input) + 1 Write (Output)
- **Traffic:** For a 4MB tensor, this code moves 8MB of data.
- **Status:** 8x Reduction in Memory Traffic.

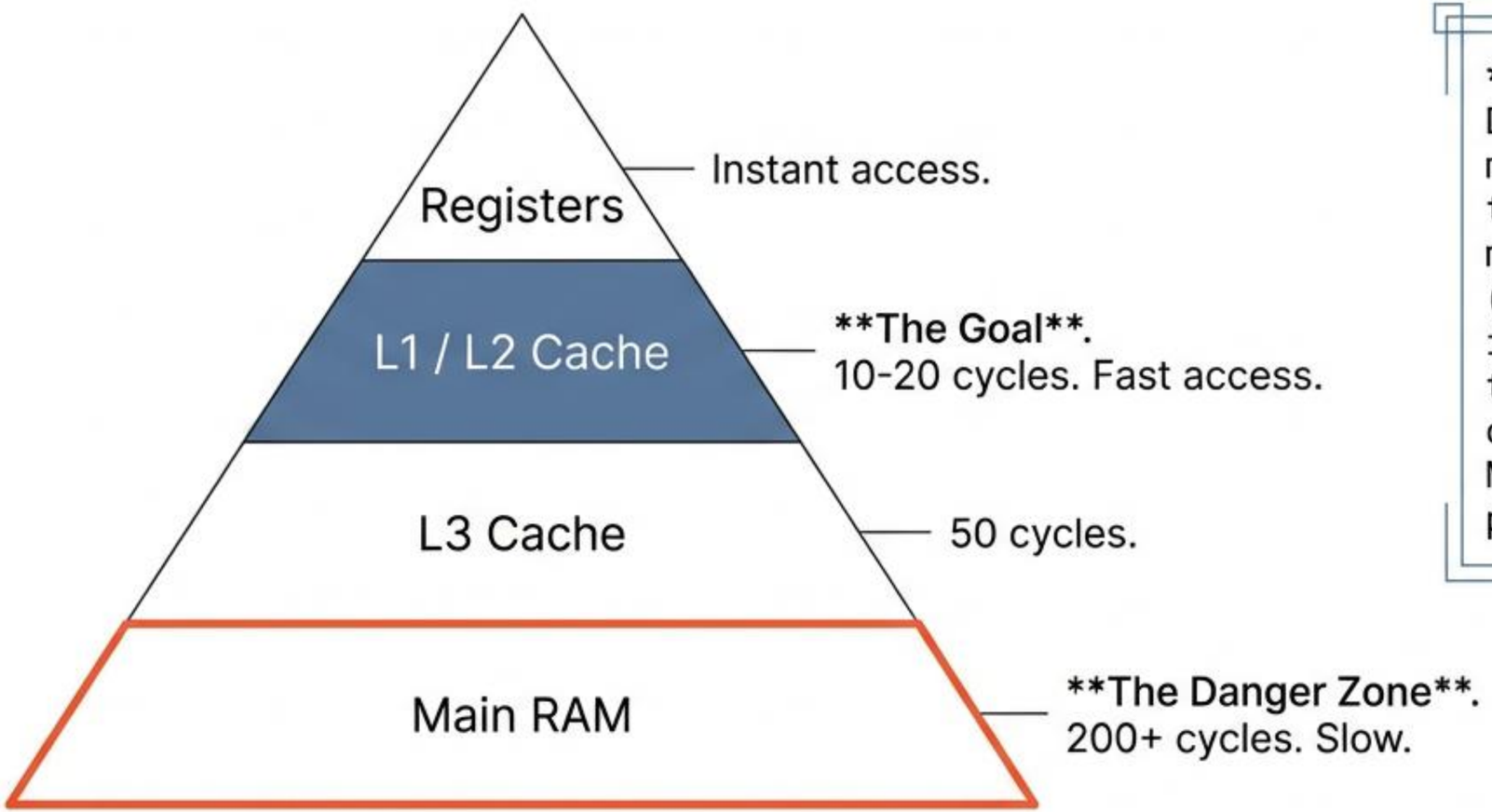
Quantifying Fusion Gains



Performance Impact:

Performance Impact: Even with identical mathoperations, the fused version runs **2-3x faster** purely because the CPU isn't waiting for memory.

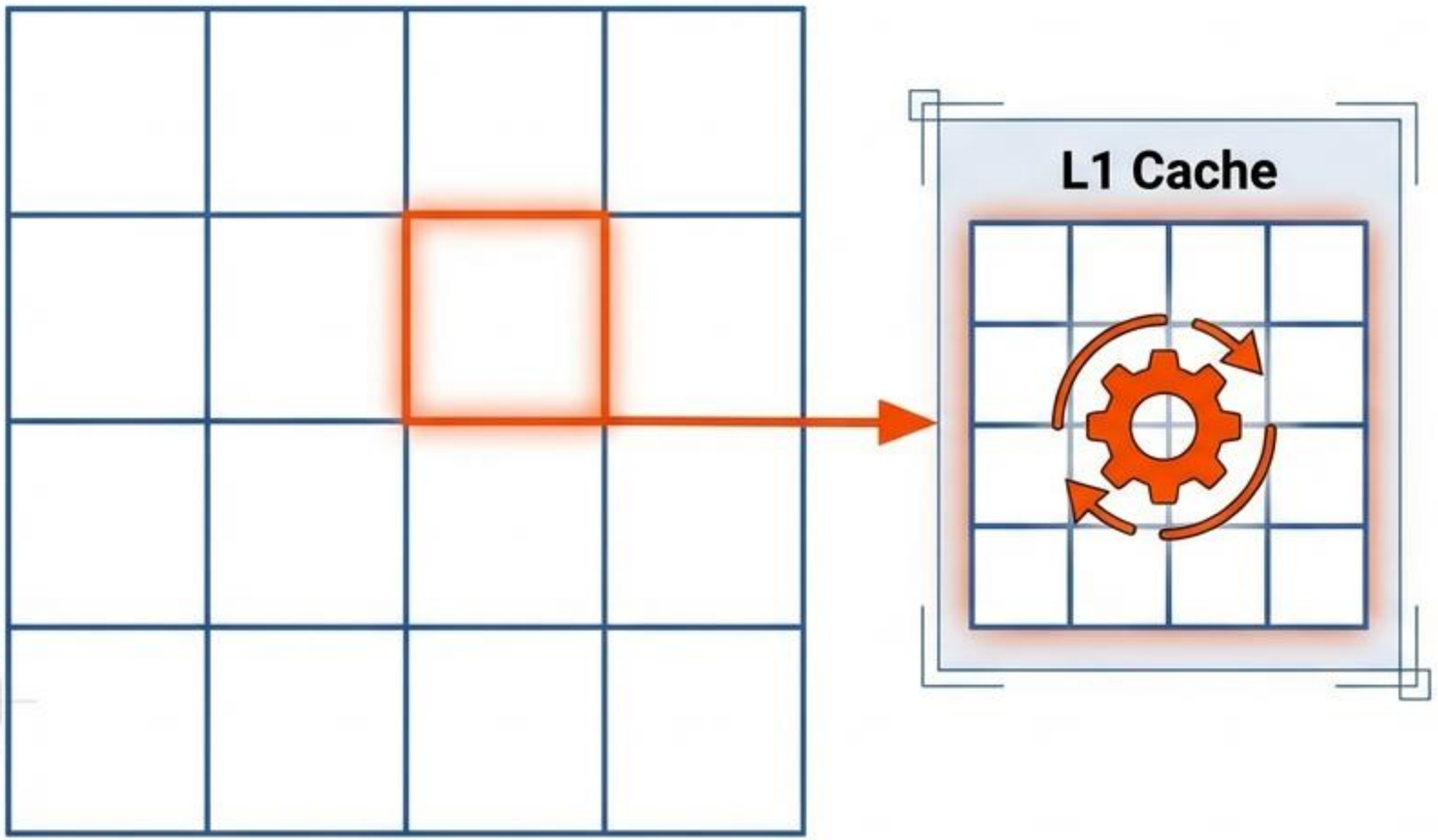
The Latency Gap: Registers vs. RAM



Memory Hierarchy Pyramid

Locality Principle: Data used together must be stored together. If a matrix is too large (e.g., 2048x2048), it spills out of the Cache into RAM, causing "Cache Misses" and processor stalls.

Strategy: Cache-Aware Tiling



Tiling (Blocking):

1. Divide large matrices into small sub-matrices (“tiles”).
2. Load a tile into L1 Cache.
3. Perform ALL math involving that tile.
4. Evict.

Result: **Maximizes reuse** of loaded data; **minimizes expensive RAM** access.

Implementation: Tiling Logic

```python
def tiled_matmul(a: Tensor, b: Tensor, tile_size: int = 64) -> Tensor:
    """Conceptual implementation of cache-aware tiling."""

    # In practice, BLAS libraries implement this automatically.
    # Visualizing the logic:
    # -----------------------------------------------------
    # for i in range(0, M, tile_size):
    #     for j in range(0, N, tile_size):
    #         # Compute block interactions in L1 Cache...
    #         # Only load these specific blocks from RAM
    # -----------------------------------------------------

    result_data = np.matmul(a.data, b.data)
    return Tensor(result_data)
```

****Tile Size Matters****
The `tile_size` (e.g., 64, 128) is **tuned** to the specific hardware's cache line size.
Too small = overhead.
Too big = cache thrashing.

The Unifying Metric: Arithmetic Intensity

$$AI = \frac{\text{FLOPs (Math Operations)}}{\text{Bytes (Data Movement)}}$$

Low AI (< 1)

Memory Bound

- Examples: Activation functions, Element-wise Addition
- Optimization: **Kernel Fusion**

High AI (> 10)

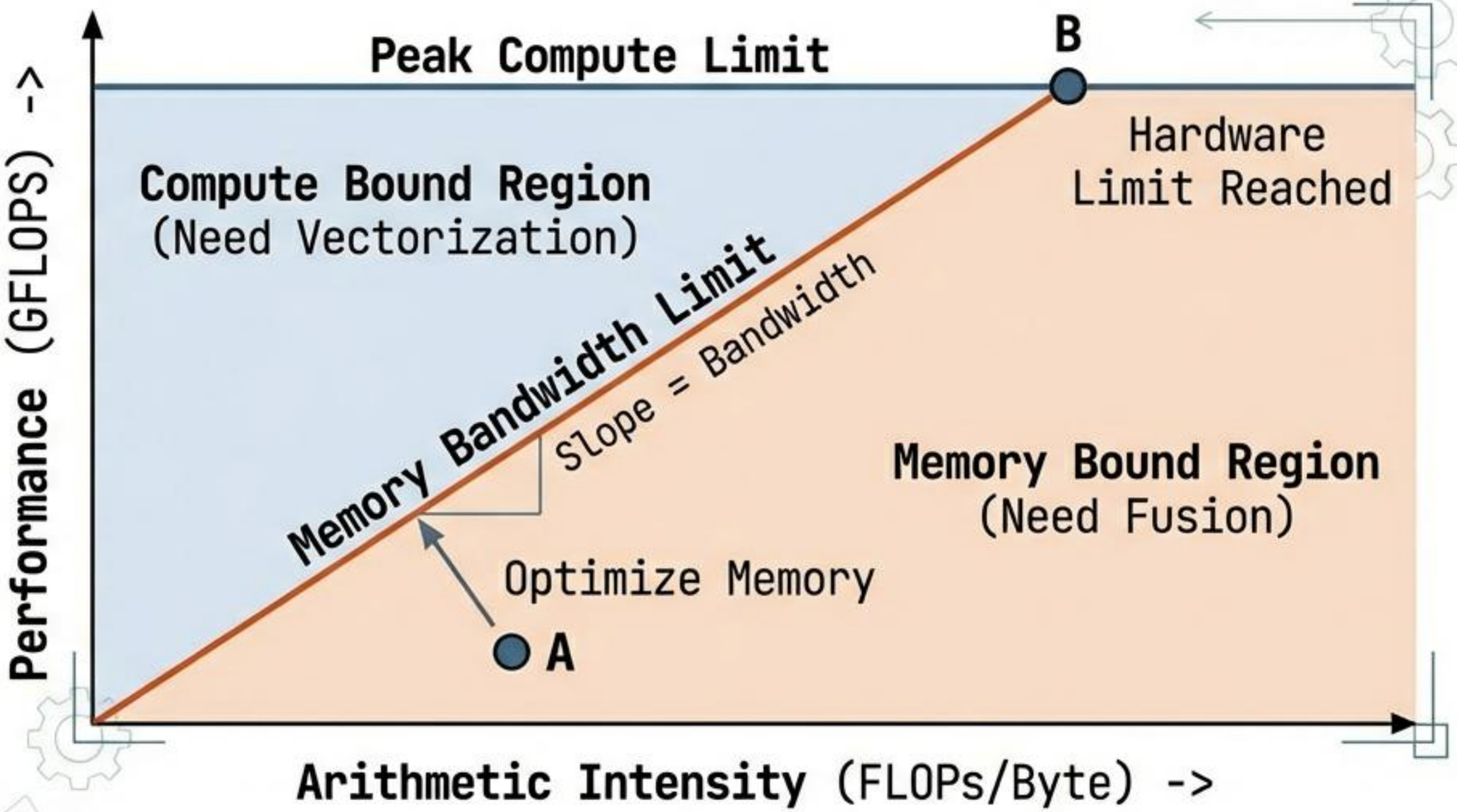
Compute Bound

- Examples: Matrix Multiplication, Convolutions
- Optimization: **Vectorization** & **Tiling**

"You cannot optimize what you do not measure."

The Roofling Model

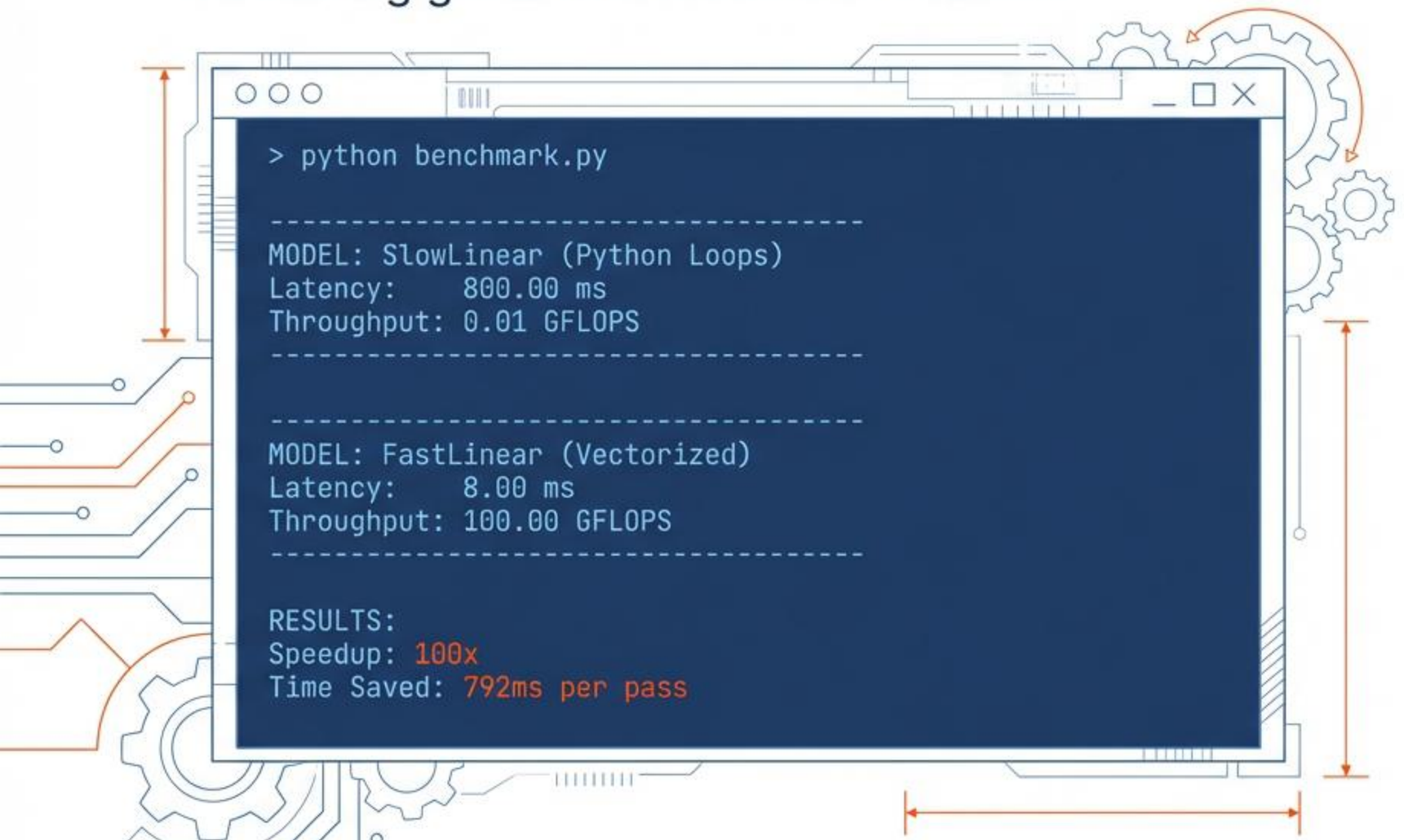
Mapping performance limits.



Engineering Editorial

Proving the Speedup

Measuring gains with Module 14 Profiler.



Insight:

We achieved a **100x speedup** without changing the mathematical output. Correctness is not enough; throughput defines usability.

Engineering Editorial

Common Acceleration Errors

Three tiny of 3-points checklist.

1. Shape Mismatches

```
ValueError: shapes (128, 256) and (128, 512) not aligned
```

Fix: Always validate inner dimensions (`a.shape[-1] == b.shape[-2]`).

2. False Dependencies

Creating temporary arrays inside loops prevents parallelization.

Fix: Pre-allocate output arrays before the loop.

3. Cache Thrashing

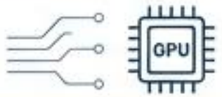
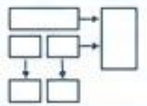
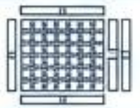
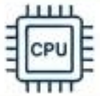
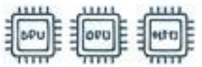
Working set exceeds L2 cache size, causing performance to drop off a cliff.

Fix: Tune tile size or reduce batch size.

Engineering Editorial

Production Context: TinyTorch vs. PyTorch

Measuring gains with Module 14 Profiler.

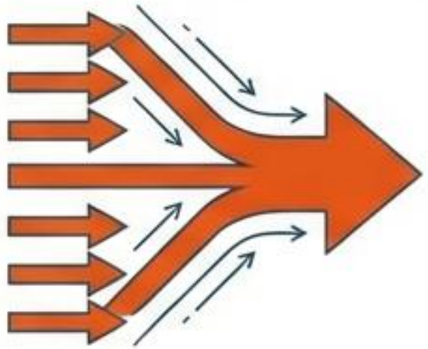
Feature	TinyTorch (Edu)	PyTorch (Prod)
Vectorization	NumPy (CPU BLAS) 	cuBLAS (GPU) 
Fusion	Manual (`fused_gelu`) 	JIT / `torch.compile` 
Tiling	Implicit (via BLAS) 	Manual (Triton/CUDA) 
Hardware	CPU Only 	CPU, GPU, TPU, NPU 

The Physics are Universal: PyTorch's `torch.compile` is just an automated way of writing `fused_gelu`. The constraints of **bandwidth** and **latency** apply to every processor.

Engineering Editorial

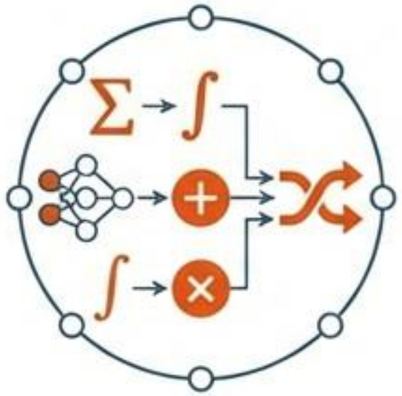
Module 17 Summary

Vectorization



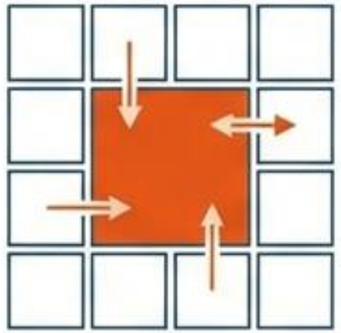
Use for: Compute Bound Ops (Matrix Mul).
Why: Leverages SIMD lanes.

Fusion



Use for: Memory Bound Ops (Activations).
Why: Reduces memory bandwidth tax.

Tiling

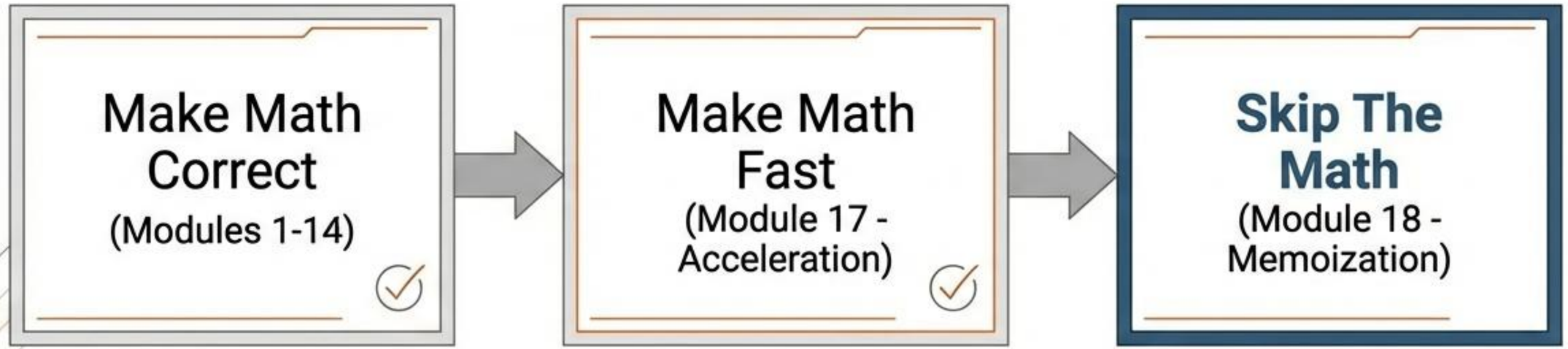


Use for: Large Matrices.
Why: Keeps data ‘hot’ in Cache.

Use the **Roofline Model** to decide which tool to apply.

Engineering Editorial

What's Next?



Coming Up: Memoization & KV-Cache

Now that our computation is optimized, we learn the ultimate speedup: avoiding the calculation entirely.