# MachineArchitectures and Parallel Computing

## Week 4

Amittai Aviram – 29 September 2020

## The Memory Cache

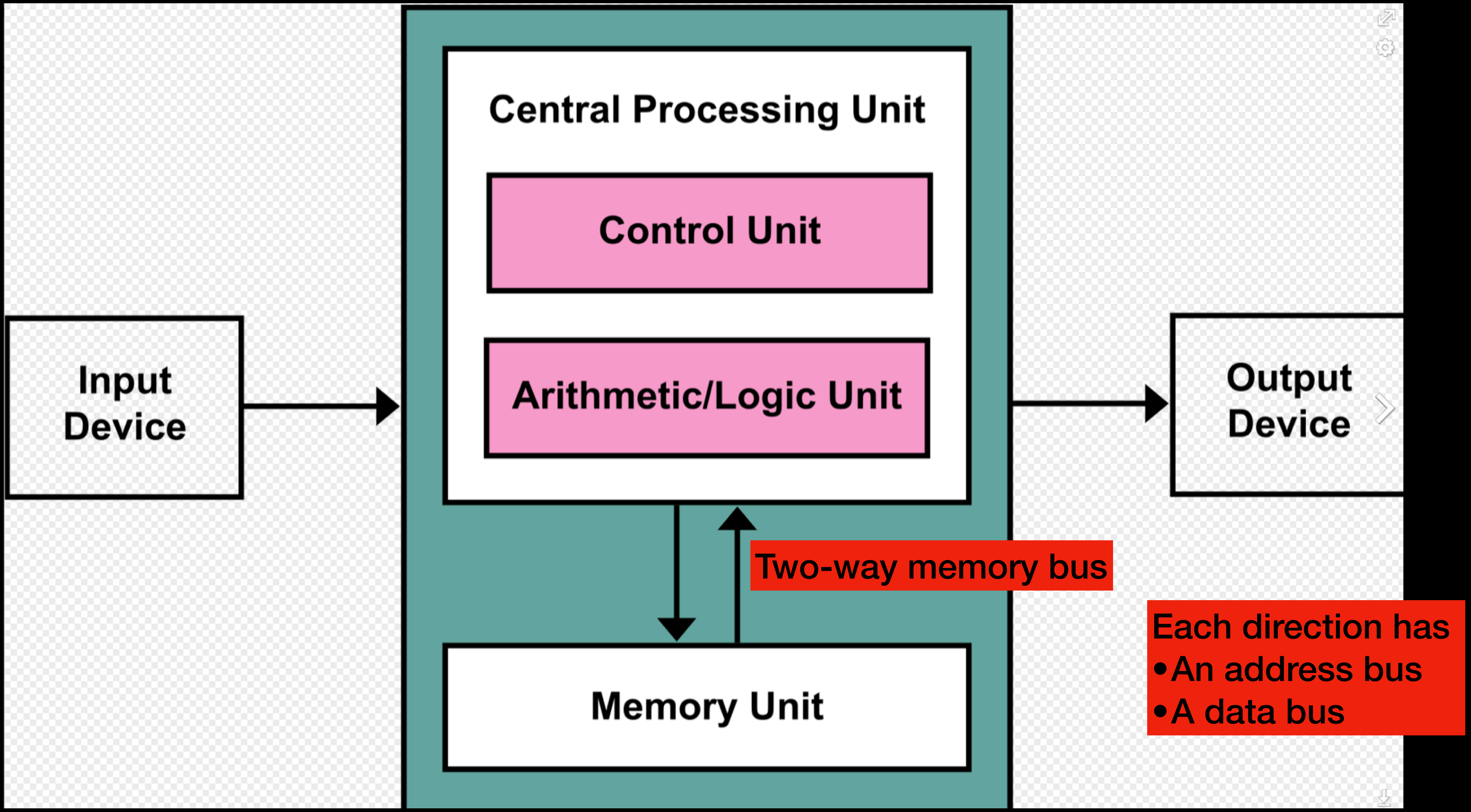
## Overview

### Why Are We Talking about Computer Architecture?

- Even in *shared memory parallelism*, we have to move data back and forth between processors and memory
- This is a hidden *communication cost*.
- This communication cost may be significant.
- We canmitigate it to improve the performance of shared memory parallelism.

# The von Neumann Architecture

## The Foundation of Digital Computers



# The von Neumann Architecture

## Published in 1945 by John von Neumann (1903 – 1957)

## The von Neumann Bottleneck

### Background

- A *bus* is a collection of wires or connections.
- The *memory bus* has as many connections as bits it carries.
  - Address bus—e.g., 64 connections for 64-bit addresses
  - Data bus—e.g., 64 connections for a 64-bit number
- *Fetch instruction*
  - CPU sets the bits for the address on the bus, along with a fetch signal
  - The memory unit sets the bits on the data bus corresponding to the value stored in memory
- *Store instruction*
  - The CPU sets the bits for the address on the bus, along with a store signal
  - It also sets the bits on the data bus corresponding to the value in the register to be stored

# The von Neumann Bottleneck

## The Problem

- Transmission of a signal across the bus is *much slower* than transmission between a register and the ALU in a CPU.
- Bus speed: 66 – 800 MHz
- Cache speed: > 1.8 GHz
- An instruction that requires a value from memory must cause the CPU to *block* until the value arrives.

## The von Neumann Bottleneck
### Example

- Processing Speed
  $8 \text{ cores} \times 3 \text{ MHz clock speed} \times 16 \text{ Flops per cucle} = 384 \text{ GFlops/s.}$
- Data Transfer Speed
  $800 \text{ MHz} \times 8 \text{ cores} \times 8 \text{ bytes per cycle} = 51.2 \text{ GB/s.}$
- Problem
  Dot product of 2 vectors, length $2^{30} \rightarrow 2^{31}$ floating point operations (Flops) (2 GFlops).
- Data
  $2^{31}$ 8-bit (double-precision) floating point numbers $\rightarrow 2^{34}$ bytes (16 GB)

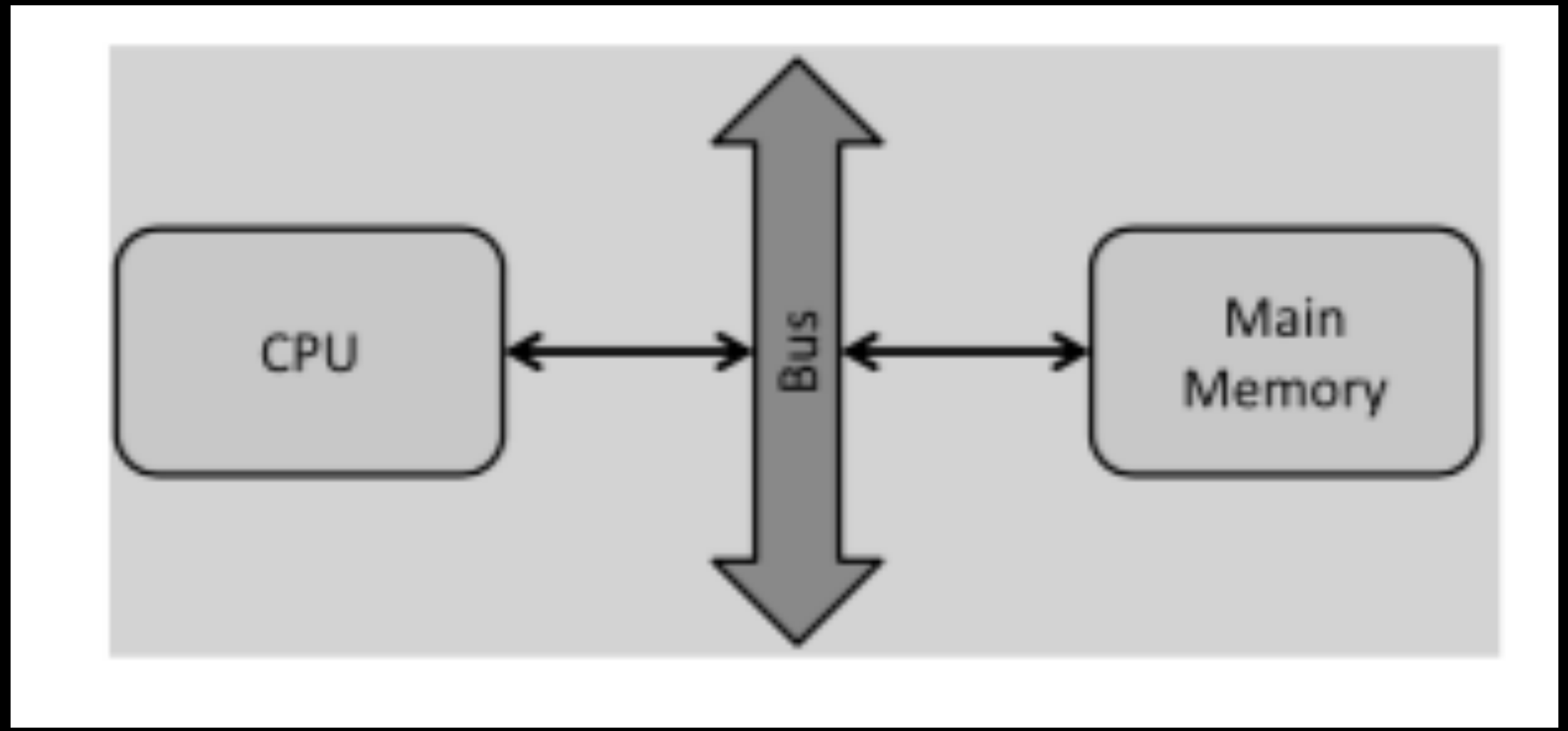
## The von Neumann Bottleneck

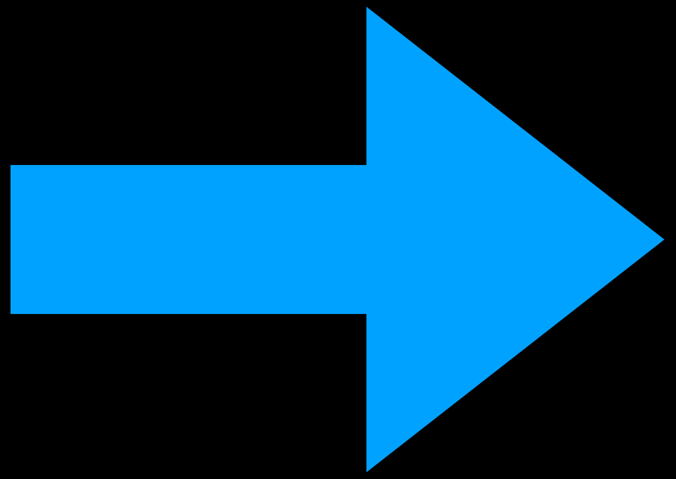
### Example

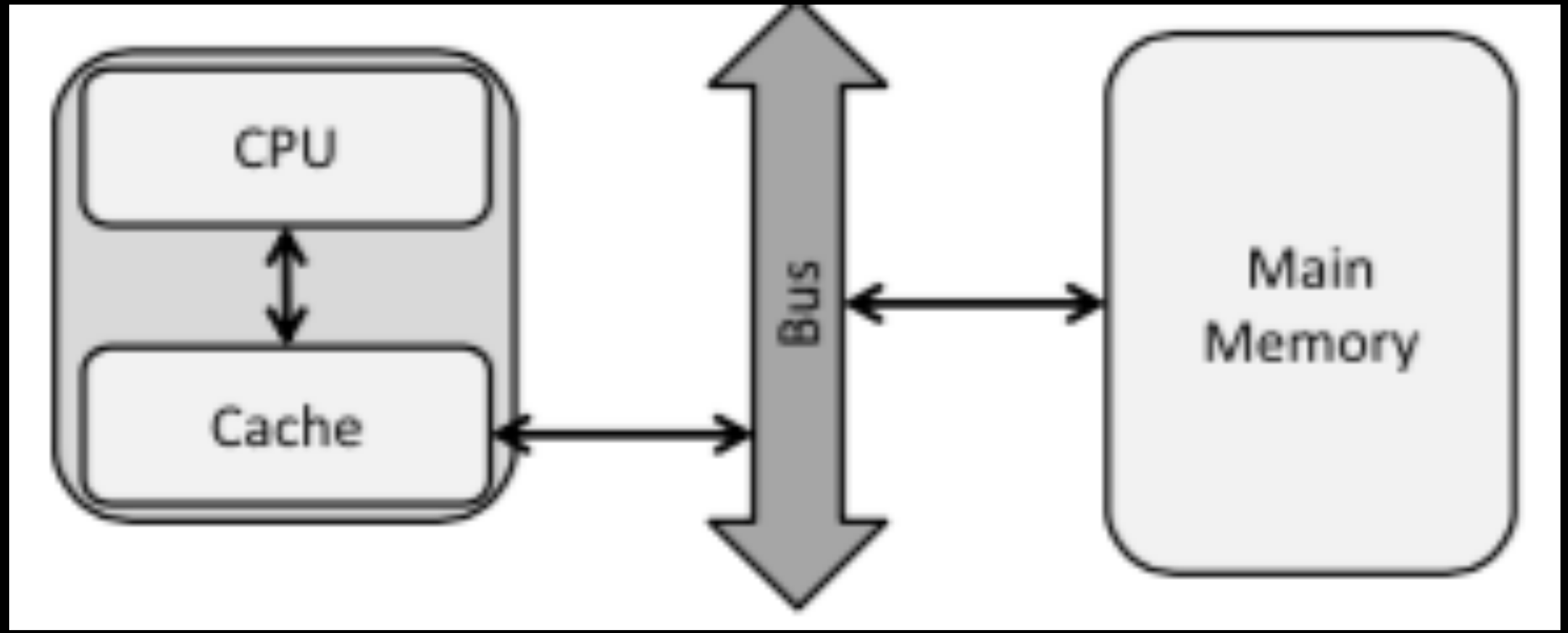
- Computation time: $t_{\text{comp}} = \dfrac{2 \text{ GFlops}}{384 \text{ GFlops/s}} = 5.2 \text{ ms}$
- Data transfer time: $t_{mem} = \dfrac{16 \text{ GB}}{51.2 \text{ GB/s}} = 312.5 \text{ ms}$ Memory bound
- Assuming that memory access and computation are concurrent, $t_{\text{exec}} \geq \max(t_{\text{comp}}, t_{\text{data}}) = 312.5 \text{ ms}.$
- Performance: $\dfrac{2 \text{ GFlops}}{312 \text{ ms}} = 6.4 \text{ GFlops/s}$, and $\dfrac{6.4 \text{ GFlops/s}}{384 \text{ GFlops/s}} = 1.67\,\%$

# Solution

## The Memory Cache (or Cache Memory)







The cache is a smaller intermediate container that holds copies of portions of the main memory that are likely to be used immediately.
CPU-cache communication is *much faster*l than CPU-memory communication.

# Memory Cache

## Example

- Data Storage Capacity
  512 KB
- Data Transfer Speed
  The same as for registers, so counted in with computation.
- Problem
  Matrix multiplication with $n \times n$ matrices, $n = 128$.
  For each element, $n$ multiplication + $n$ addition = $2n$ Flops.
  $2n \times n^2 \text{ Flops} = 2n^3 \text{ Flops} = 2 \cdot 128^3 \text{ Flops} = 2^{22} \text{ Flops} = 4 \text{ GFlops}$
- Data
  $3 \times 128 \times 128 \times 8 \text{ B} = 3 \times 128 \text{ KB} = 384 \text{ KB} < 512 \text{ KB}.$

```
for (size_t i = 0; i < n; ++i) {
    for (size_t j = 0; j < n; ++j) {
        for (size_t k = 0; k < n; ++k) {
            W[i][j] += U[i][k] * V[k][j];
        }
    }
}
```

# Memory Cache

## Example

- $t_{\text{comp}} = \dfrac{4\ \text{GFlops}}{384\text{GFlops/s}} = 10.4\ \mu\text{s}$

Compute bound

- $t_{\text{mem}} = \dfrac{384\ \text{KB}}{51.2\ \text{GB/s}} = 7.5\ \mu\text{s}$

We fetch data into the cache once at
the beginning and store it back at the end,
so computing and memory times do not overlap.

- $t_{\text{exec}} \geq t_{\text{comp}} + t_{\text{mem}} = 10.4\ \mu\text{s} + 7.5\ \mu\text{s} = 17.9\ \mu\text{s}$

- Performance
  $\dfrac{4\ \text{GFlops}}{17.9\ \mu\text{s}} = 223\ \text{GFlops/s}$, and $\dfrac{223\ \text{GFlops/s}}{384\ \text{GFlops/s}} = 58\ \%$

## Cache Algorithms

### Common Solutions

- Cache access is much faster than main memory access, but a cache is much more expensive, so caches need to be small.
- Spatial locality — fetch data in *lines* — line size is, say, 64 bytes.
- Temporal locality — to make space for  anew line, we evict the least-recently used one (LRU).
- Mapping of memory to cache addresses
  - Direct mapping — undermines LRU policy — higher miss rate.
  - Two-way (or *n*-way) mapping — a compromise.
  - Fully associative mapping — too expensive.

# Cache-Based Optimization

## Matrix Multiplication

- In the naïve algorithm, $B[k][j]$ skips from row to row (with the same column), so it causes a high miss rate.
- If you transpose *B*, you can get a significant improvement (by a factor of 2.2 in the Schmidt's example).

```
for (uint64_t i = 0; i < n; ++i) {
    for (uint64_t j = 0; j < n; ++j) {
        for (uint64_t k = 0; k < n; ++k) {
            C[i][j] += A[i][k] * Bt[j][k]
        }
}
```

## Cache Coherency

### The Serial Case

- If the core needs to write to location *x* in main memory, a copy of which is stored in the cache, it would write to the cache so as to have the value immediately available.
- But this makes the cache line no longer a copy of main memory!
- Solutions:
  - *Write through*: always write *both* to the cache *and* to main memory. This is as slow as not having a cache.
  - *Write back*: mark the cache line as *dirty* and copy it to main memory when it is evicted. This introduces a period of cache incoherence.

# Cache Coherency

## The Parallel Case

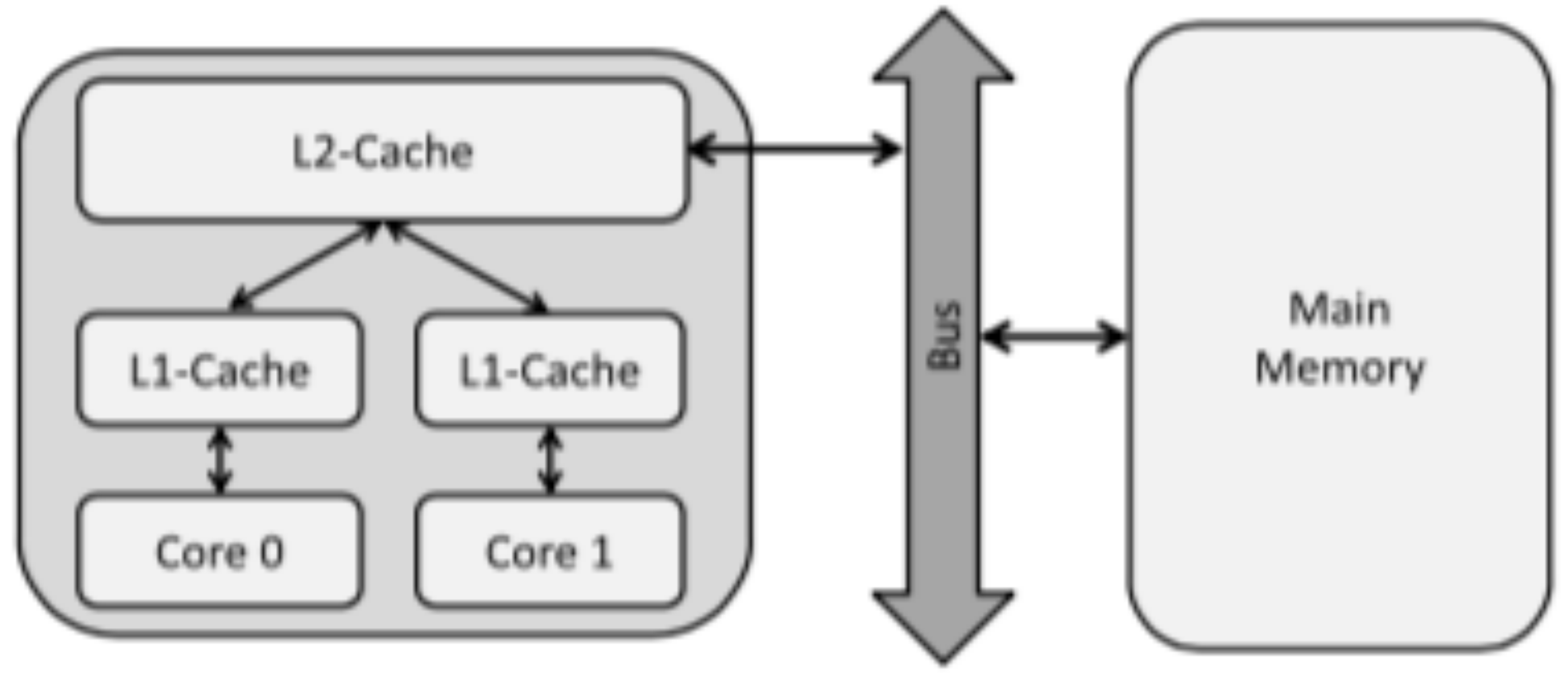


**FIGURE 3.5**

Illustration of a CPU-chip with two cores and two levels of cache: a private L1-cache and a shared a L2-cache.

See MESI Protocol:
https://en.wikipedia.org/wiki/MESI_protocol

In some cases, parallel cache interactions may cause a parallel program to run more slowly than its sequential equivalent.

## Levels of Parallelism

# Flynn’s Taxonomy

## Kinds of Parallel Architecture

- SISD — Single Instruction, Single Datum (not parallel)
- SIMD — Single Instruction, Multiple Data (vector instructions)
- Multiple Instructions, Multiple Data (independent processing elements)
- Multiple Instructions, Single Datum (pipelines)

Michael J. Flynn, Stanford, 1966

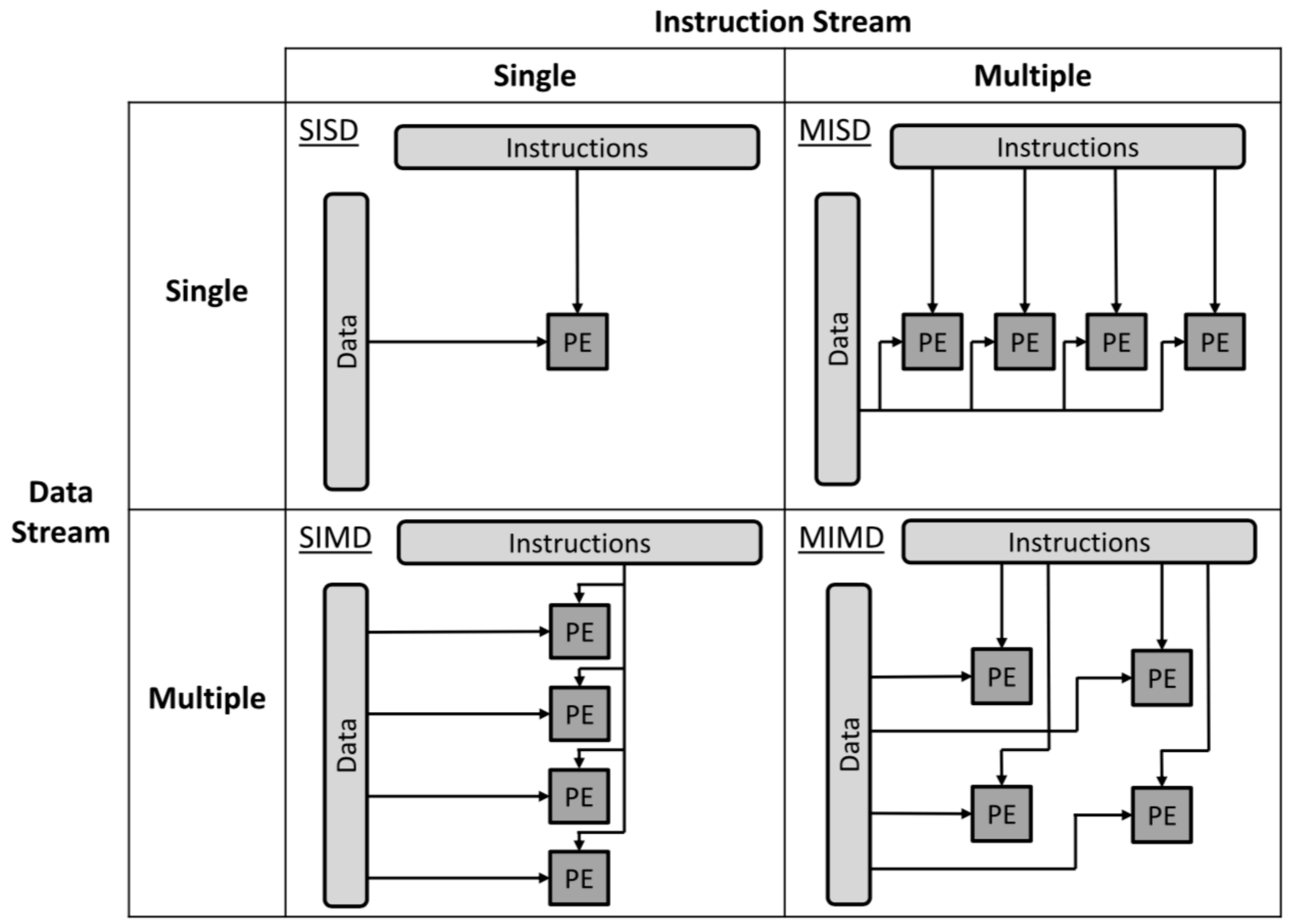


**FIGURE 3.7**

Flynn's taxonomy classifies computer architectures into four classes according to the number of instruction streams and data streams: SISD, MISD, SIMD, and MIMD.

## Microarchitecture

### Support for Flynn Classes of Parallelism

- MIMD — multiple cores.
- SIMD — vector registers, vector units, and vector instructions (“intrinsics”).
- MISD
  - Pipelining — simultaneous instruction fetching, instruction decoding, and data fetching.
  - Superscalar execution — pipelining of instructions in the ALU — requires out-of-order execution to avoid dependencies.

## SIMD Instructions

### Vectorized Registers and Operations

- Intel supports SIMD operations through *intrinsics*.
- For reference, see the Intel Intrinsics Guide.

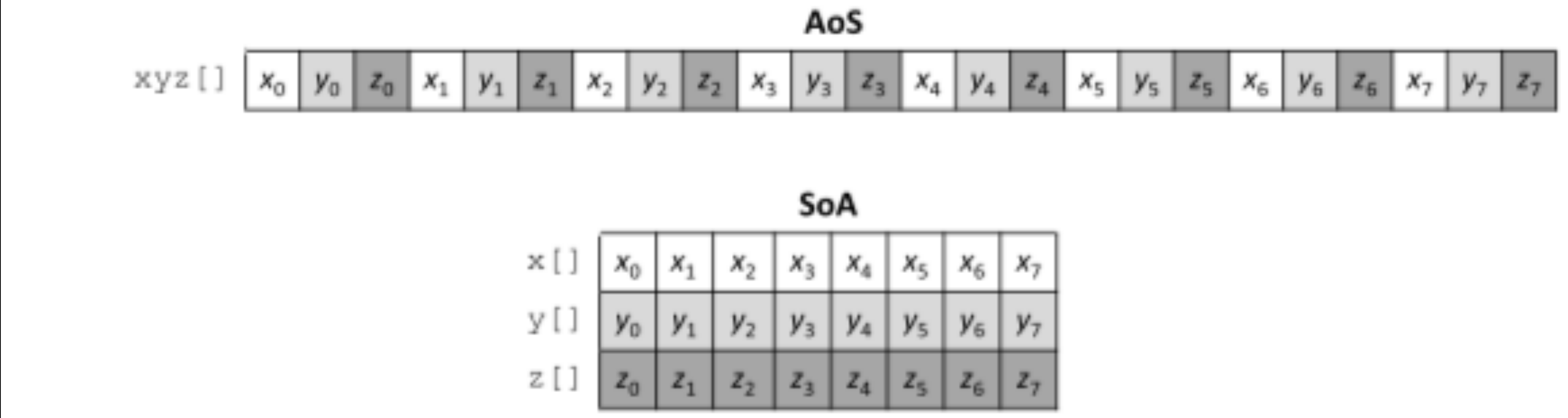
# SIMD Optimizations

## AoS versus SoA

- A SIMD operation over an array of structures (AoS) may be inefficient for vector hardware.
- Example
    - Array of length-3 arrays (vectors) of floating point numbers, representing an array of points in 3-dimensional space.
    - Each vector $v_i$ must be *normalized*:
      $$\hat{v}_i = \frac{v_i}{\|v_i\|} = \frac{x_i + y_i + z_i}{1/\sqrt{x_i^2 + y_i^2 + z_i^2}}$$
    - The triple $(x_i, y_i, z_i)$ wastes space on vector hardware sized for a power of 2.
    - The sum of squares would have to work across vector hardware lanes, which is impossible, so it must be serialized.
- Translating the data to three arrays—a structure of arrays (SoA)—enables us to use vector hardware efficiently.

# SIMD Optimization

## AoS versus SoA



AoS
xyz[]
$x_0$ $y_0$ $z_0$ $x_1$ $y_1$ $z_1$ $x_2$ $y_2$ $z_2$ $x_3$ $y_3$ $z_3$ $x_4$ $y_4$ $z_4$ $x_5$ $y_5$ $z_5$ $x_6$ $y_6$ $z_6$ $x_7$ $y_7$ $z_7$
SoA
x[]
$x_0$ $x_1$ $x_2$ $x_3$ $x_4$ $x_5$ $x_6$ $x_7$
y[]
$y_0$ $y_1$ $y_2$ $y_3$ $y_4$ $y_5$ $y_6$ $y_7$
z[]
$z_0$ $z_1$ $z_2$ $z_3$ $z_4$ $z_5$ $z_6$ $z_7$

**FIGURE 3.12**

Comparison of the AoS and the SoA memory layout of a collection of eight 3D vectors:

## Summary

### Parallelism in the Machine Architecture Context

- Memory accesses can degrade the performance of parallel programs in a way comparable to the effect of excess communication between compute nodes.
- The memory cache mitigates this problem—if programs are refactored to take advantage of cache locality.
- Flynn’s taxonomy of types of parallelism corresponds to various computer architecture features.
- Vector hardware and intrinsics support SIMD processing at the individual machine level.
- Programs can be refactored to benefit from vector hardware.