DTU

02/11/23

Real-Time Visual and Machine Learning Systems

Questions or comments from last time?

Agenda – Module 1 part A

- Memory Hierarchies in Hardware
- Memory Hierarchies in Software
- Memory Allocations and Data Structures
- Smart Pointers
- Graph Structures
- Garbage Collectors
- Computational Graphs
- Exercises

Memory Hierarchies in Hardware

- The speed of RAM and disks vs. the speed of processors
- Not the whole picture – core count, cache sizes, efficiency, sophistication of branch prediction, bus bandwidth

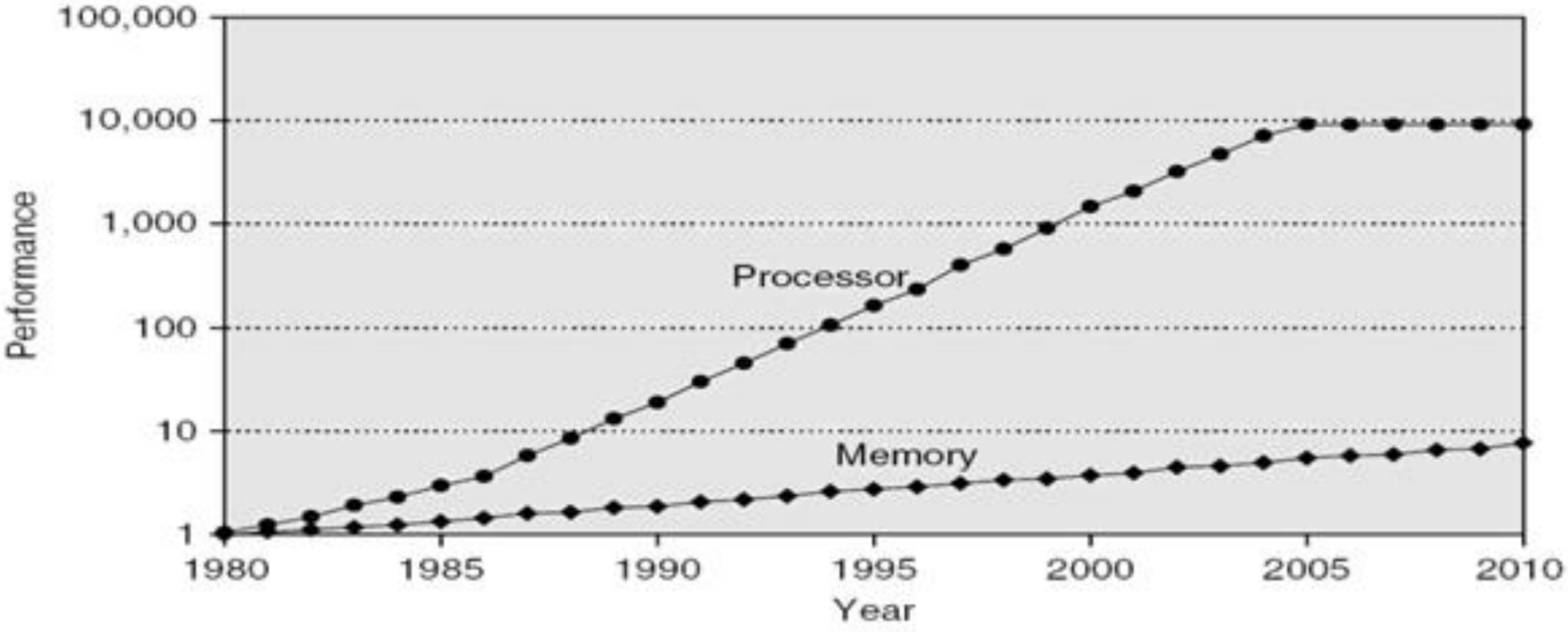
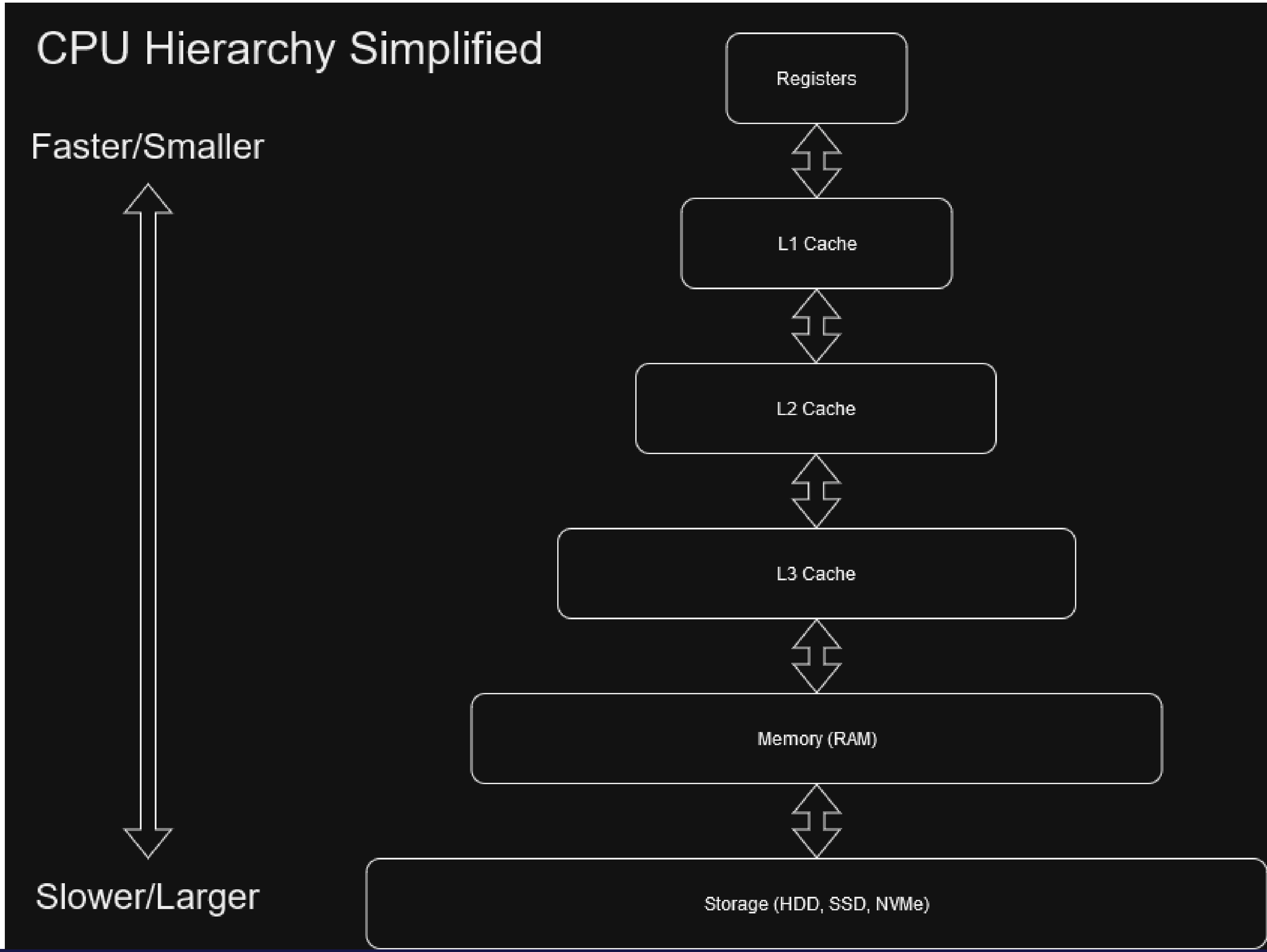


Figure 25.2

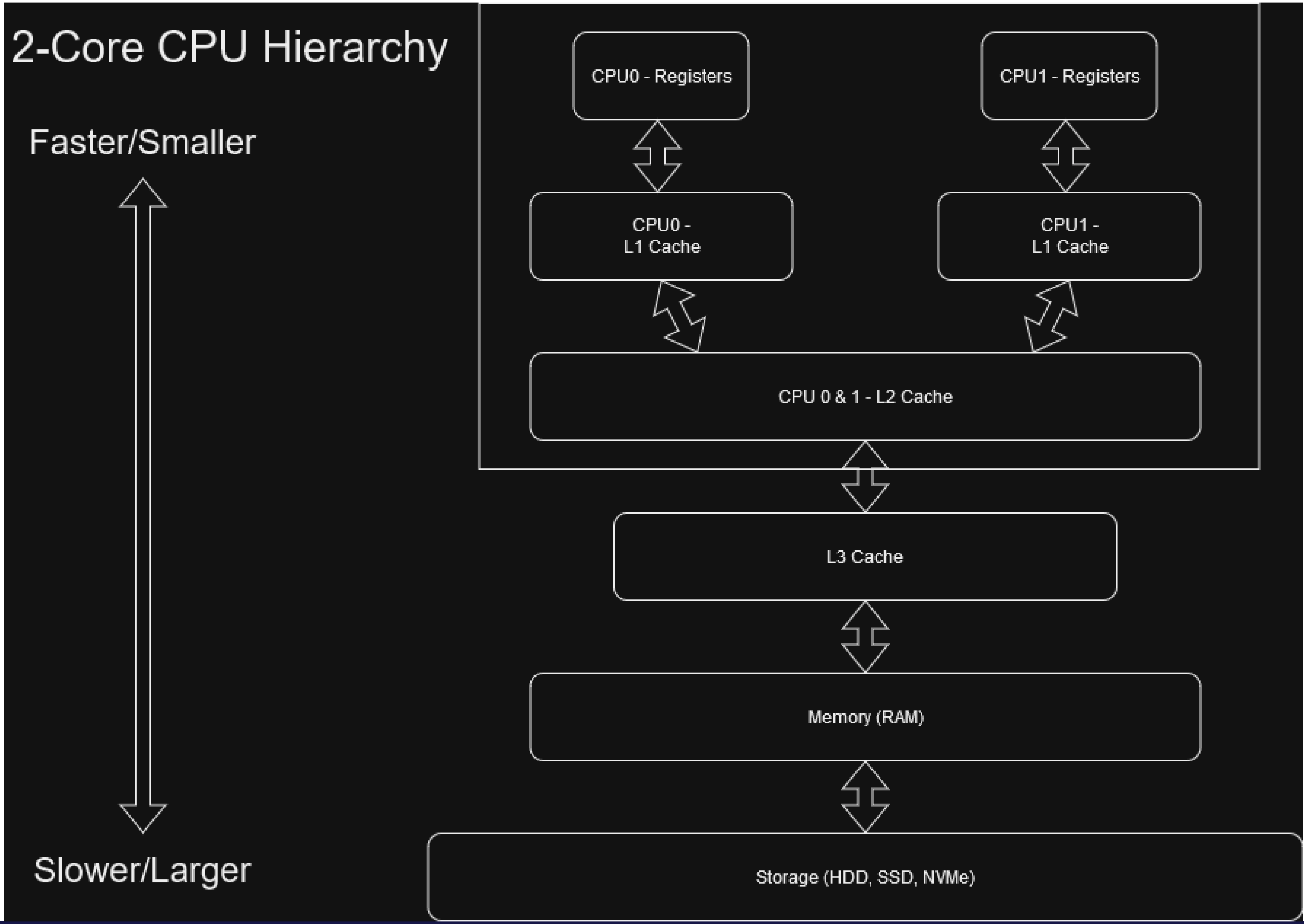
Memory Hierarchies in Hardware

From slowest to fastest



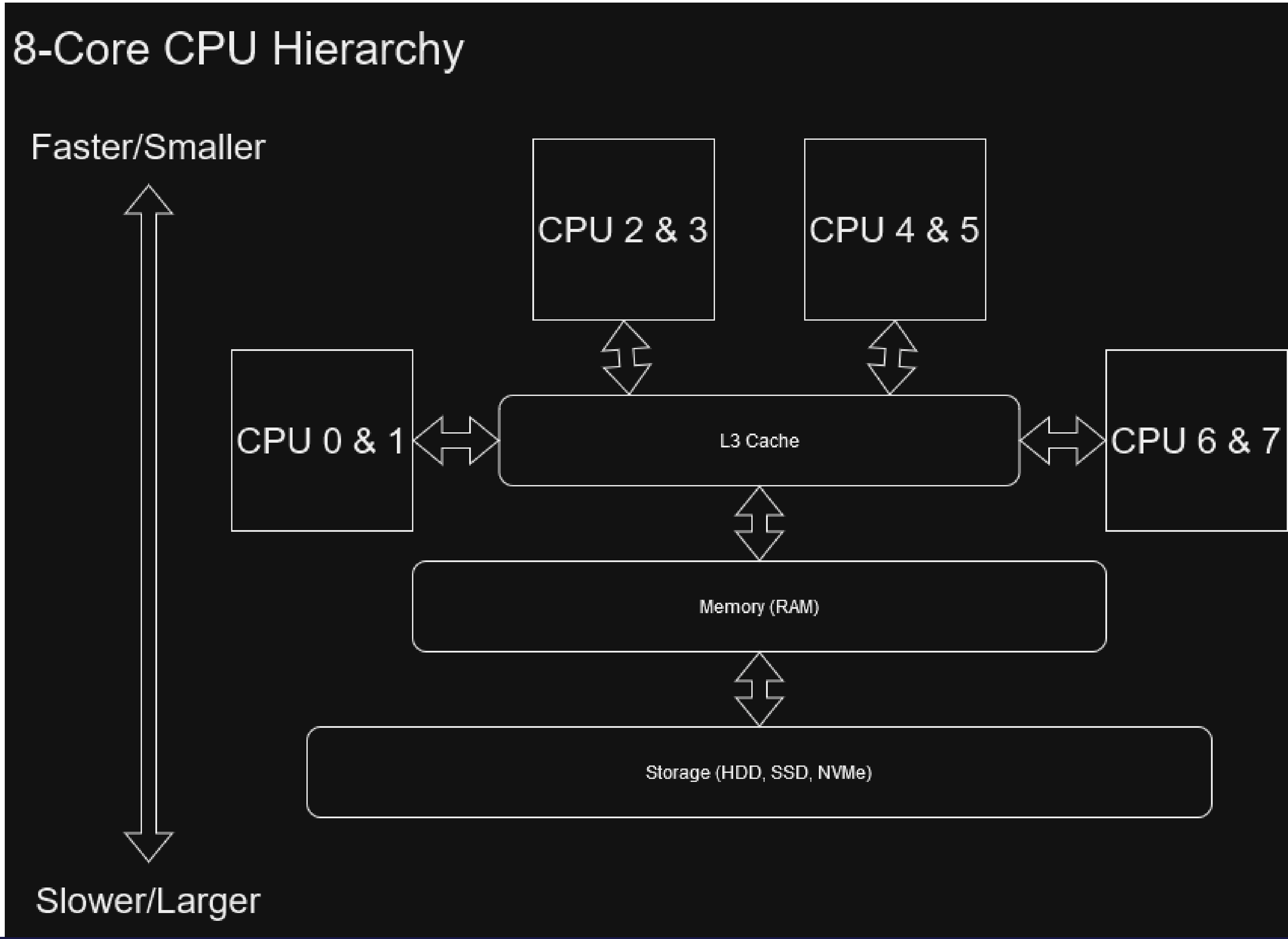
Memory Hierarchies in Hardware

Different levels can be shared differently



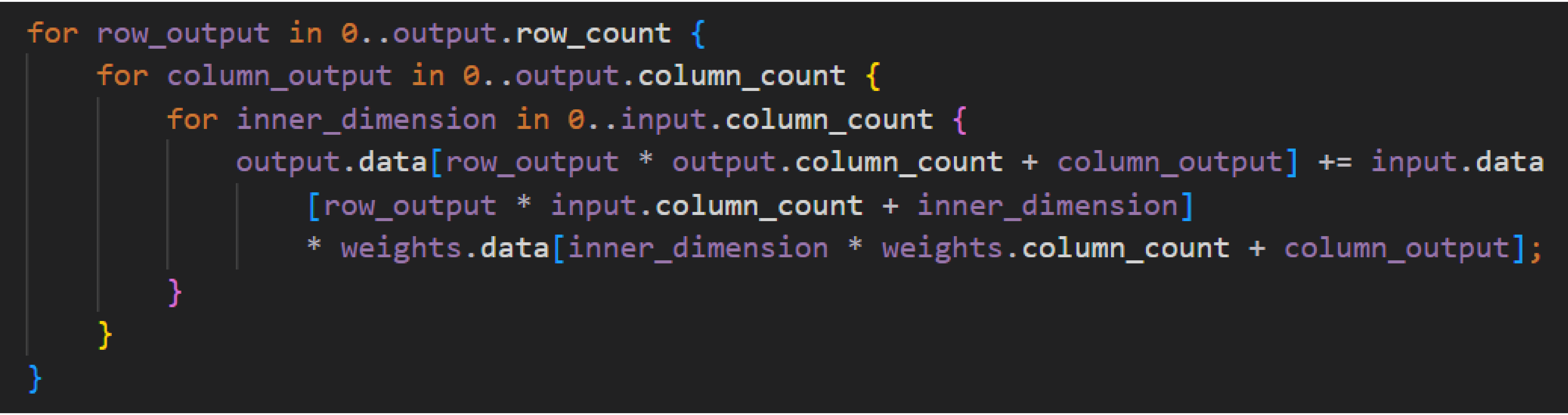
Memory Hierarchies in Hardware

Different levels can be shared differently



Memory Hierarchies in Hardware

We can’t manipulate the caches directly

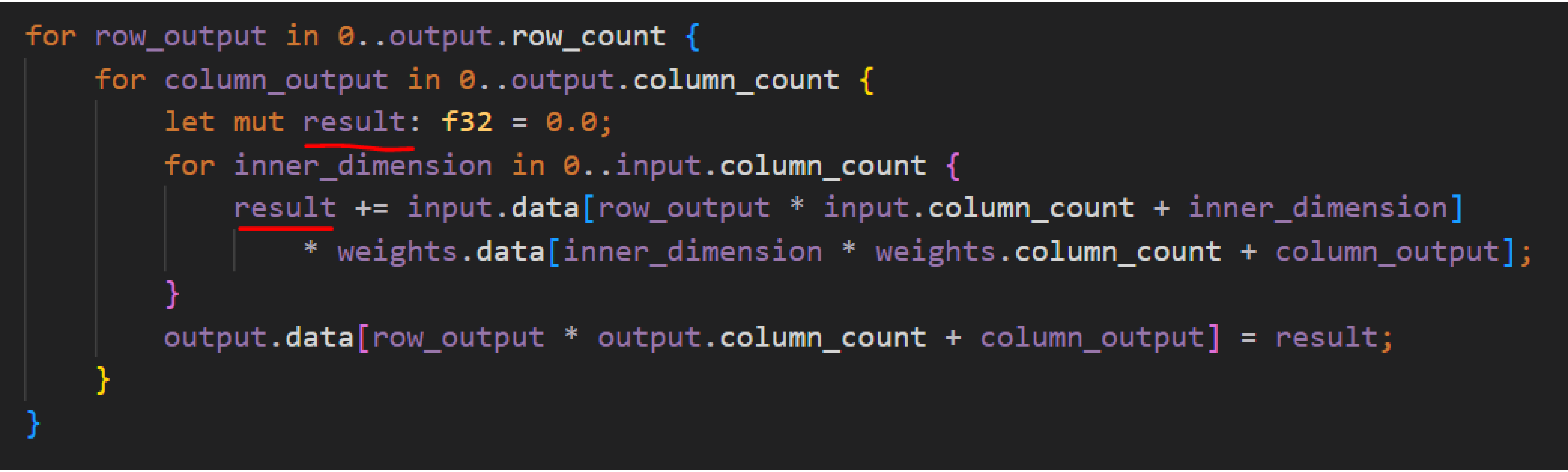


```
for row_output in 0..output.row_count {
    for column_output in 0..output.column_count {
        for inner_dimension in 0..input.column_count {
            output.data[row_output * output.column_count + column_output] += input.data
                [row_output * input.column_count + inner_dimension]
                * weights.data[inner_dimension * weights.column_count + column_output];
        }
    }
}
```

Should have cleaned up the linearized accesses

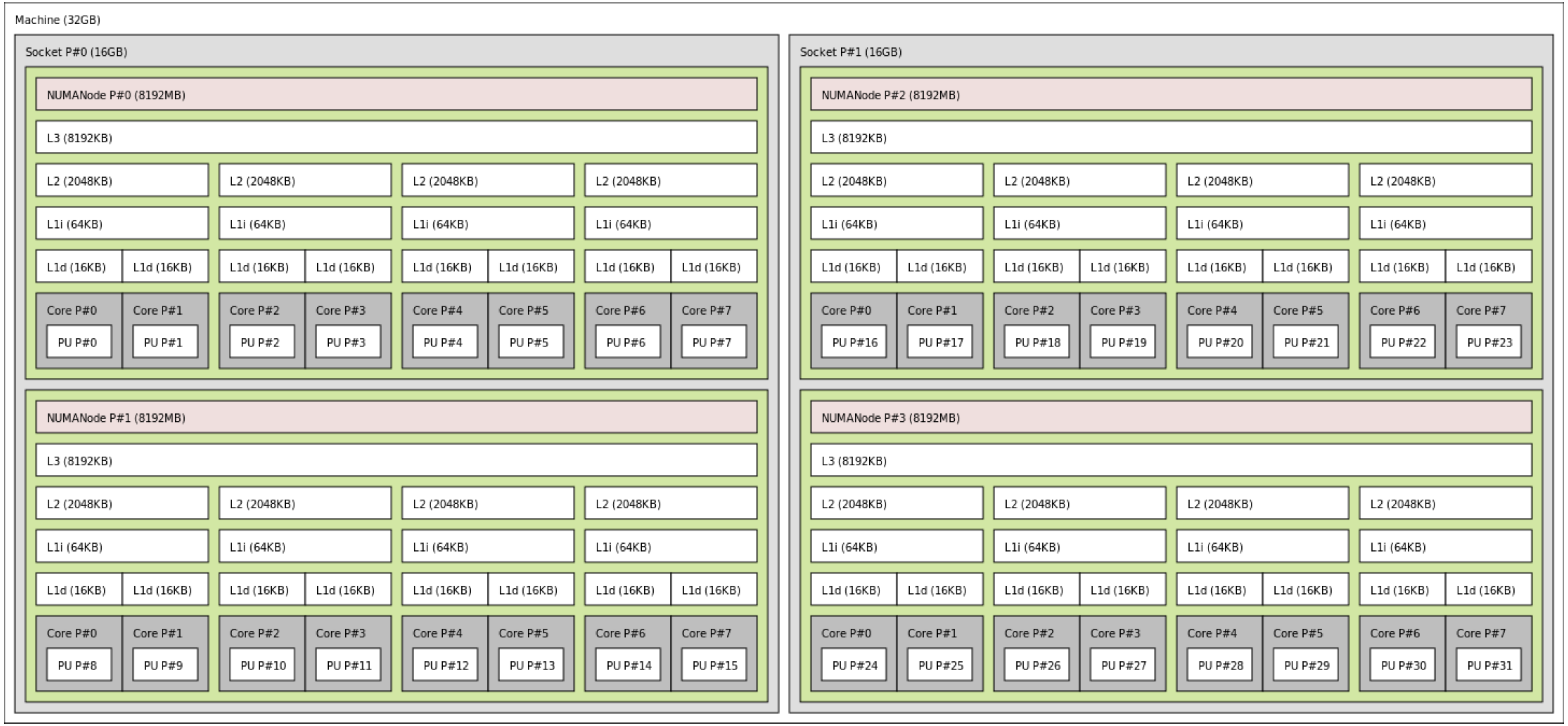
Memory Hierarchies in Hardware

But we can create stack variables of known sizes, which are likely to reside in registers and L1 cache



```
for row_output in 0..output.row_count {
    for column_output in 0..output.column_count {
        let mut result: f32 = 0.0;
        for inner_dimension in 0..input.column_count {
            result += input.data[row_output * input.column_count + inner_dimension]
                * weights.data[inner_dimension * weights.column_count + column_output];
        }
        output.data[row_output * output.column_count + column_output] = result;
    }
}
```

Memory Hierarchies in Hardware – AMD Bulldozer server



Machine (32GB)

Socket P#0 (16GB)

NUMANode P#0 (8192MB)

L3 (8192KB)

L2 (2048KB) | L2 (2048KB) | L2 (2048KB) | L2 (2048KB)

L1i (64KB) | L1i (64KB) | L1i (64KB) | L1i (64KB)

L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB)

Core P#0 PU P#0 | Core P#1 PU P#1 | Core P#2 PU P#2 | Core P#3 PU P#3 | Core P#4 PU P#4 | Core P#5 PU P#5 | Core P#6 PU P#6 | Core P#7 PU P#7

NUMANode P#1 (8192MB)

L3 (8192KB)

L2 (2048KB) | L2 (2048KB) | L2 (2048KB) | L2 (2048KB)

L1i (64KB) | L1i (64KB) | L1i (64KB) | L1i (64KB)

L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB)

Core P#0 PU P#8 | Core P#1 PU P#9 | Core P#2 PU P#10 | Core P#3 PU P#11 | Core P#4 PU P#12 | Core P#5 PU P#13 | Core P#6 PU P#14 | Core P#7 PU P#15

Socket P#1 (16GB)

NUMANode P#2 (8192MB)

L3 (8192KB)

L2 (2048KB) | L2 (2048KB) | L2 (2048KB) | L2 (2048KB)

L1i (64KB) | L1i (64KB) | L1i (64KB) | L1i (64KB)

L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB)

Core P#0 PU P#16 | Core P#1 PU P#17 | Core P#2 PU P#18 | Core P#3 PU P#19 | Core P#4 PU P#20 | Core P#5 PU P#21 | Core P#6 PU P#22 | Core P#7 PU P#23

NUMANode P#3 (8192MB)

L3 (8192KB)

L2 (2048KB) | L2 (2048KB) | L2 (2048KB) | L2 (2048KB)

L1i (64KB) | L1i (64KB) | L1i (64KB) | L1i (64KB)

L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB) | L1d (16KB)

Core P#0 PU P#24 | Core P#1 PU P#25 | Core P#2 PU P#26 | Core P#3 PU P#27 | Core P#4 PU P#28 | Core P#5 PU P#29 | Core P#6 PU P#30 | Core P#7 PU P#31

Image Link

Memory Hierarchies in Hardware – GPU (H100)

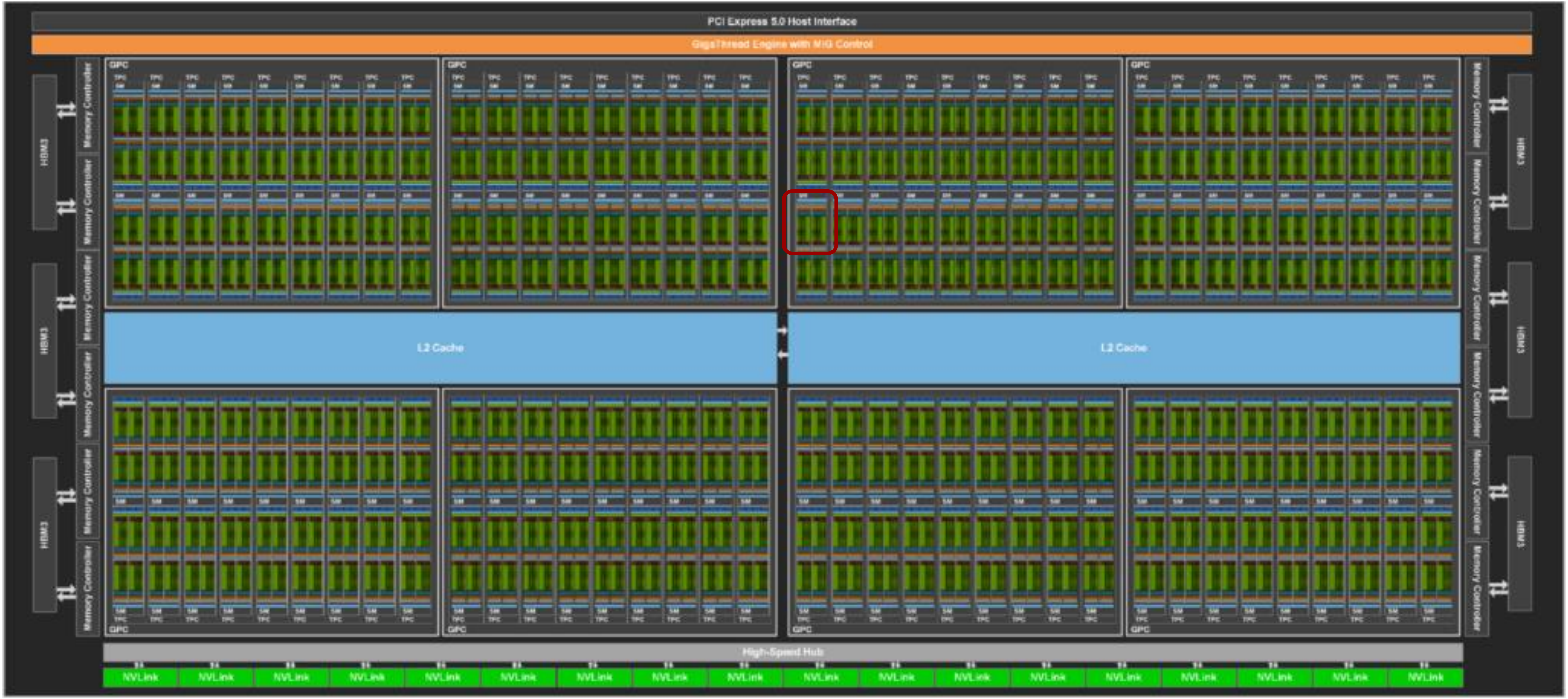


Image Link

Memory Hierarchies in Hardware GPU (H100)

More about GPU architecture next week

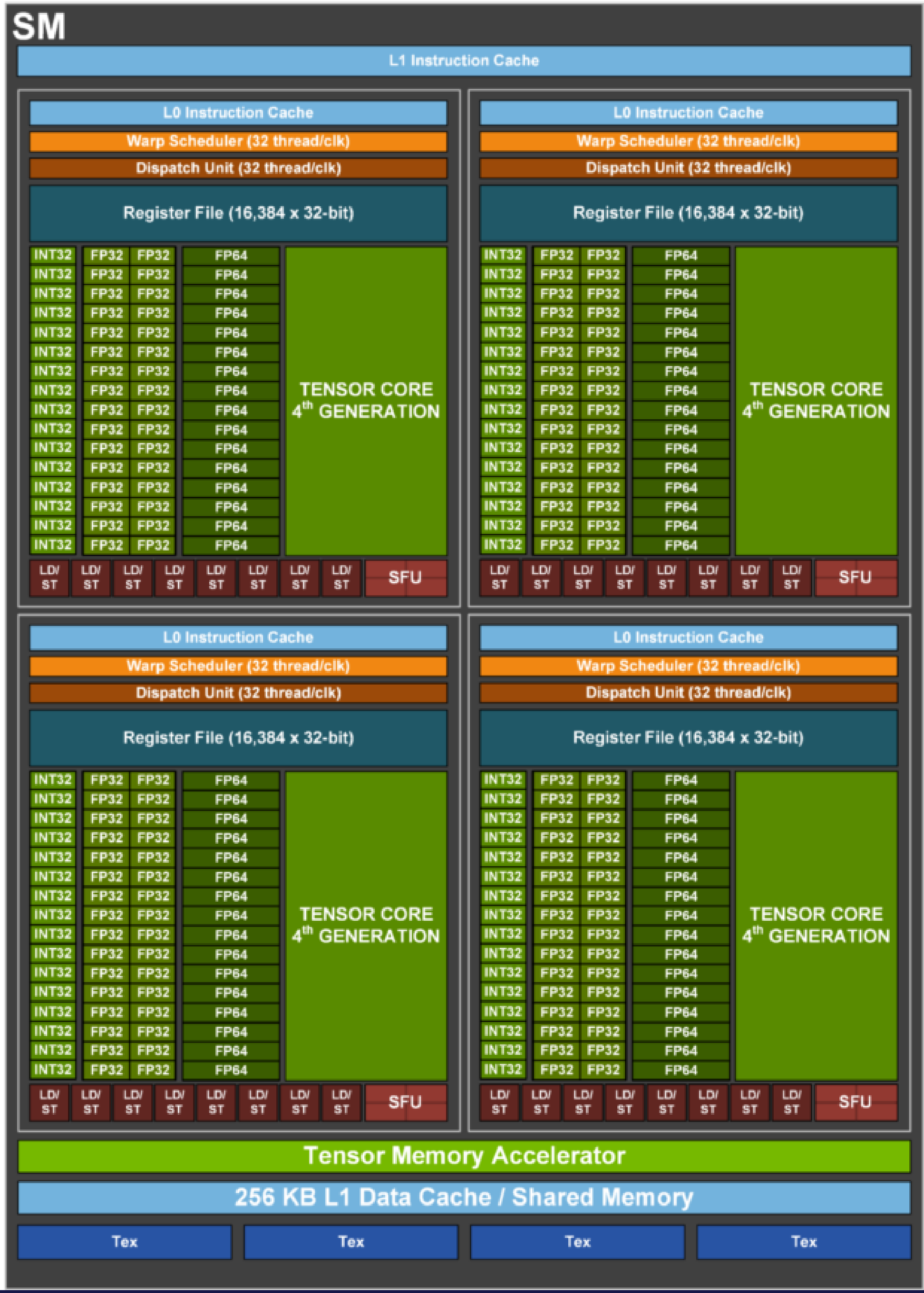
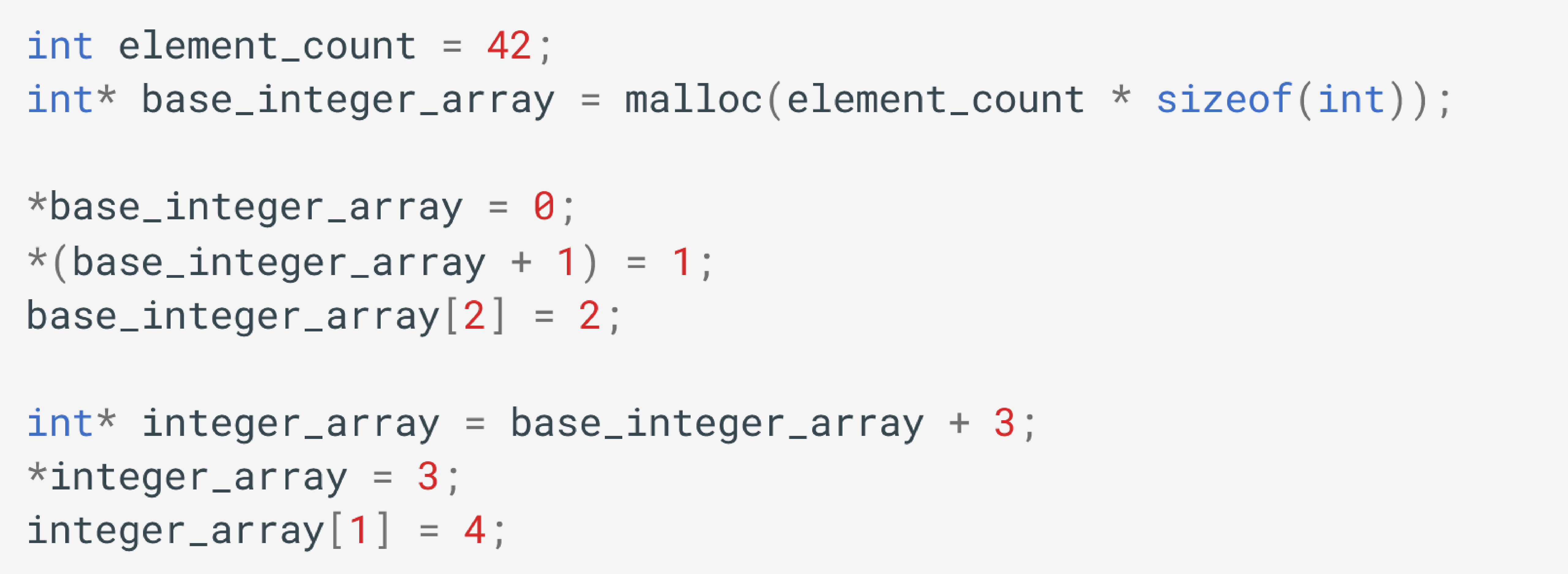

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE 4th GENERATION
LD/ ST
SFU
Tensor Memory Accelerator
256 KB L1 Data Cache / Shared Memory
Tex

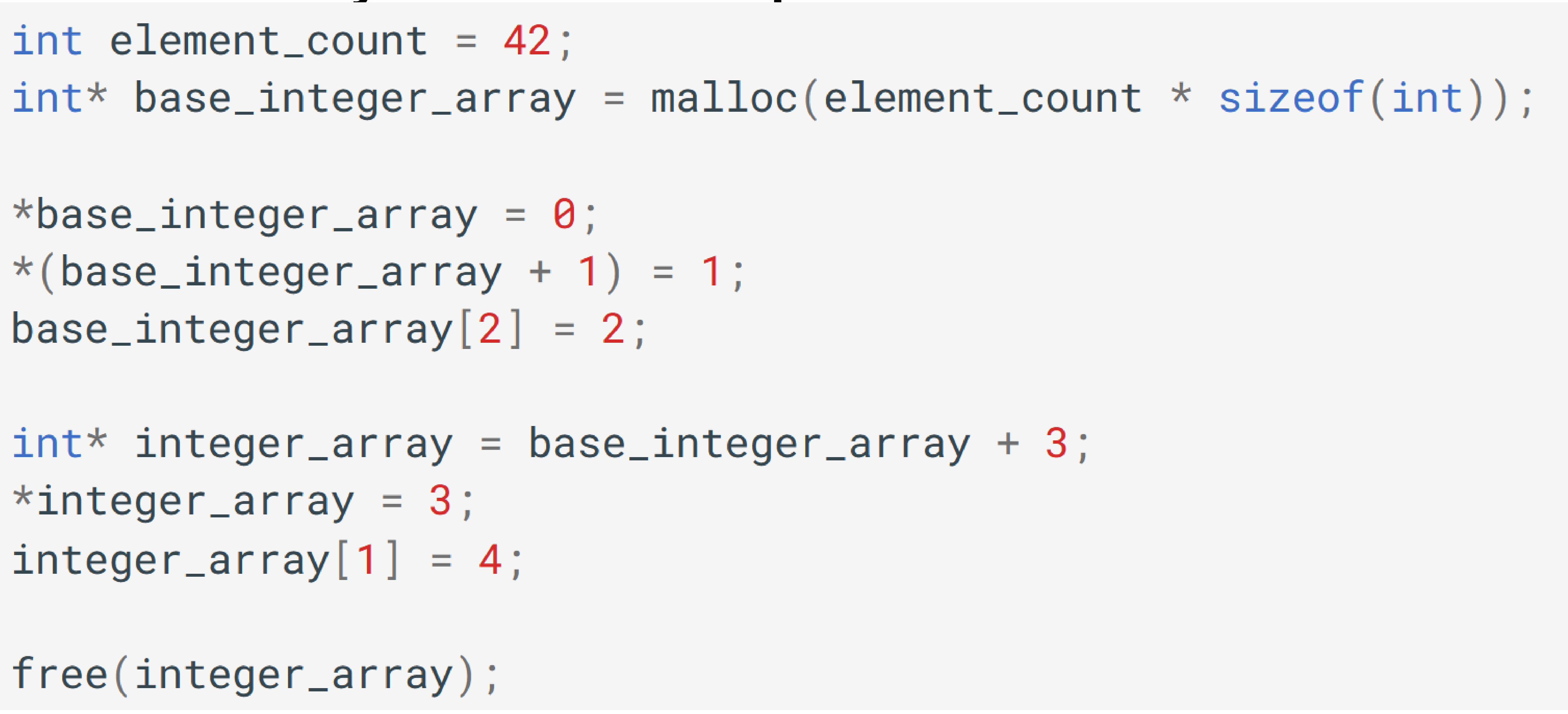
Image Link

Memory Hierarchies in Software
Memory Allocations – in C!



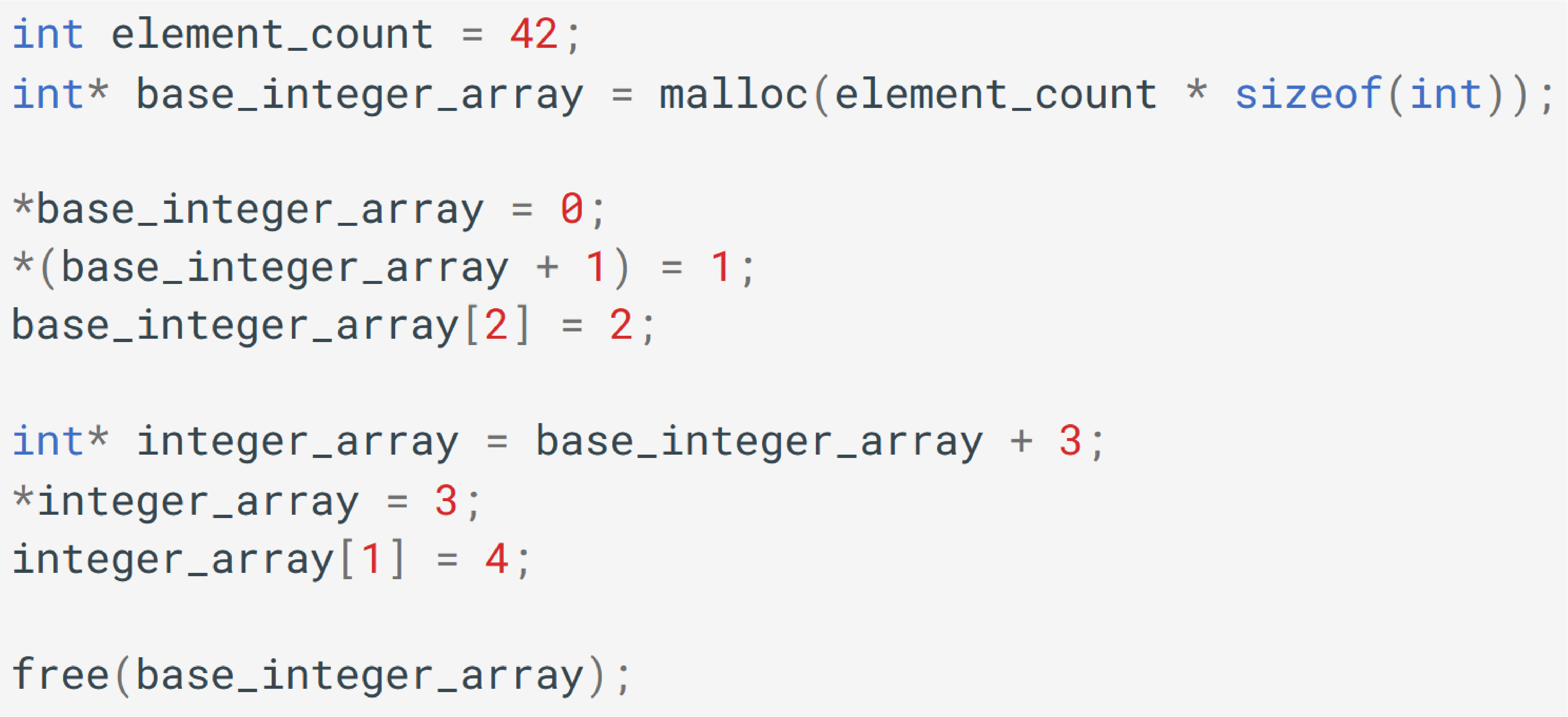
```c
int element_count = 42;
int* base_integer_array = malloc(element_count * sizeof(int));

*base_integer_array = 0;
*(base_integer_array + 1) = 1;
base_integer_array[2] = 2;

int* integer_array = base_integer_array + 3;
*integer_array = 3;
integer_array[1] = 4;
```

Memory Hierarchies in Software
Memory Allocations – Spot the Error!



```
int element_count = 42;
int* base_integer_array = malloc(element_count * sizeof(int));

*base_integer_array = 0;
*(base_integer_array + 1) = 1;
base_integer_array[2] = 2;

int* integer_array = base_integer_array + 3;
*integer_array = 3;
integer_array[1] = 4;

free(integer_array);
```

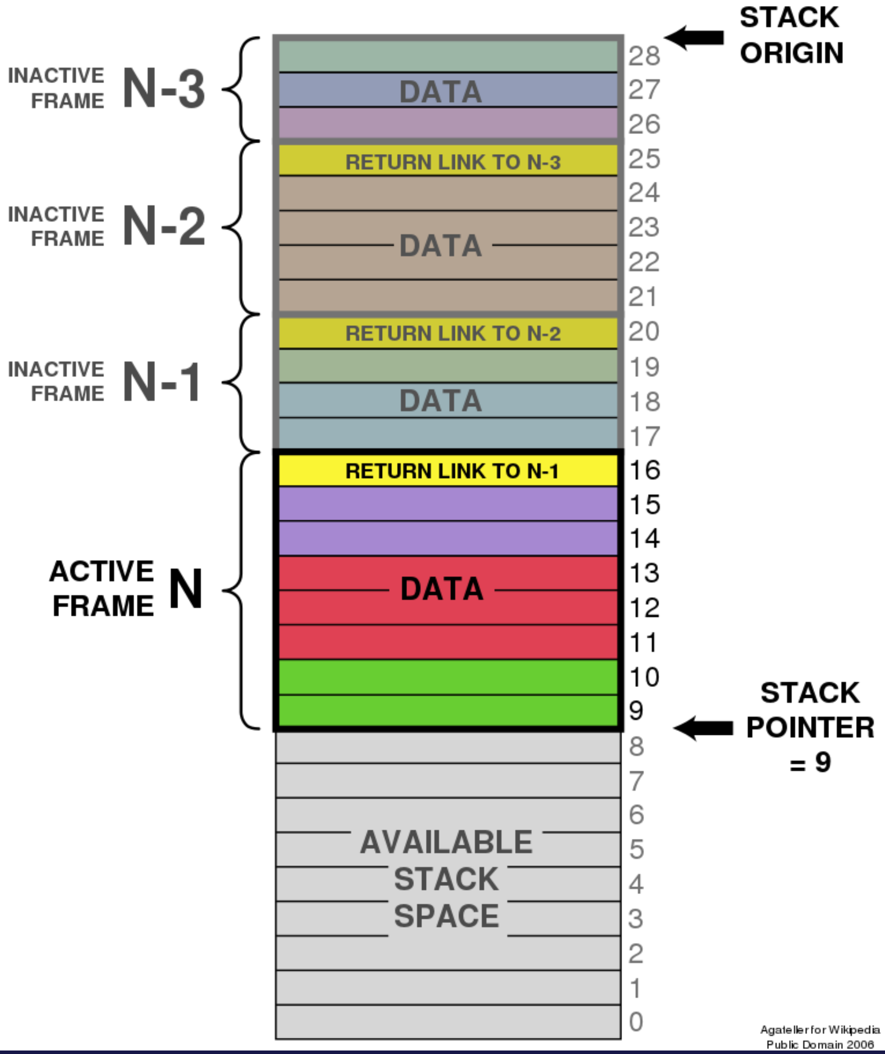
Memory Hierarchies in Software
Memory Allocations – Spot the Error!



```c
int element_count = 42;
int* base_integer_array = malloc(element_count * sizeof(int));

*base_integer_array = 0;
*(base_integer_array + 1) = 1;
base_integer_array[2] = 2;

int* integer_array = base_integer_array + 3;
*integer_array = 3;
integer_array[1] = 4;

free(base_integer_array);
```

Memory Hierarchies in Software
Where does the memory come from?

- Function calls
- Variables with a size known at compile time



Agateller for Wikipedia
Public Domain 2006

Image Link

Memory Hierarchies in Software
Where does the memory come from?

- When we malloc'ed, the data was allocated on the heap

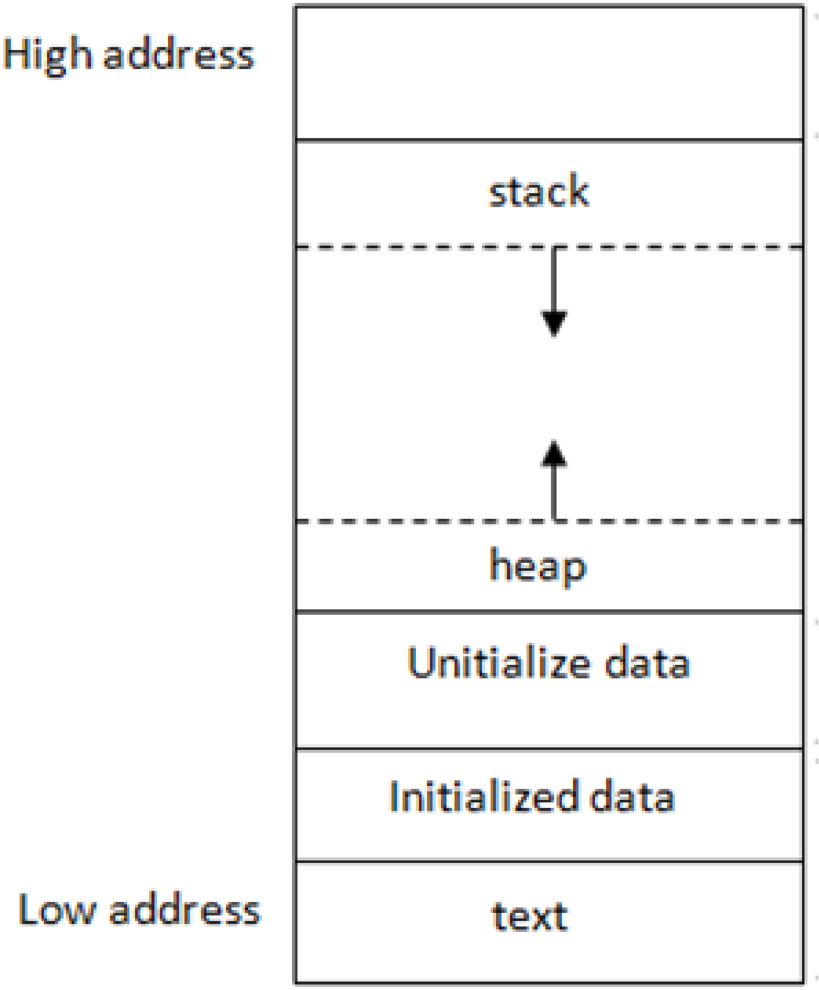
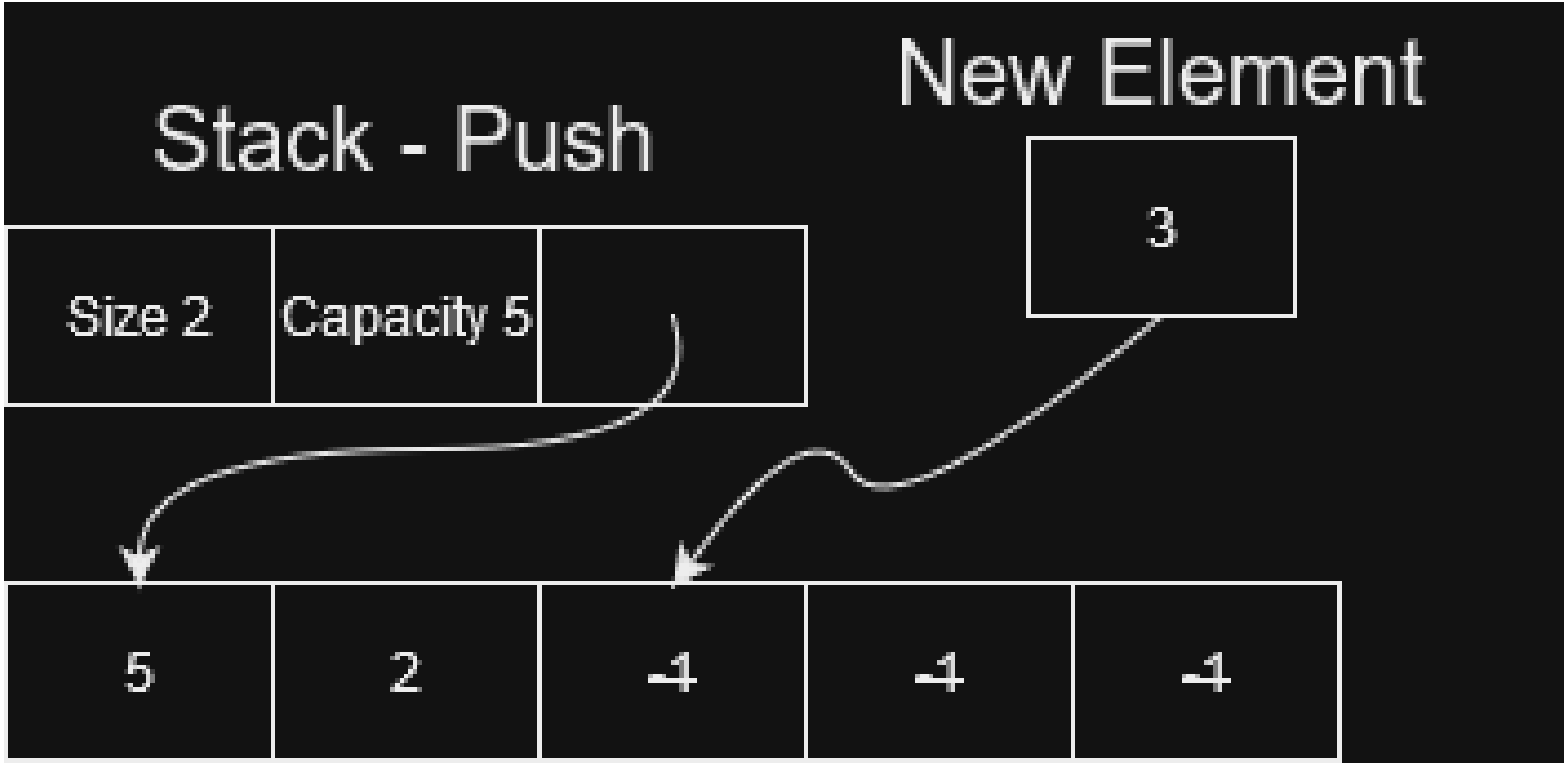


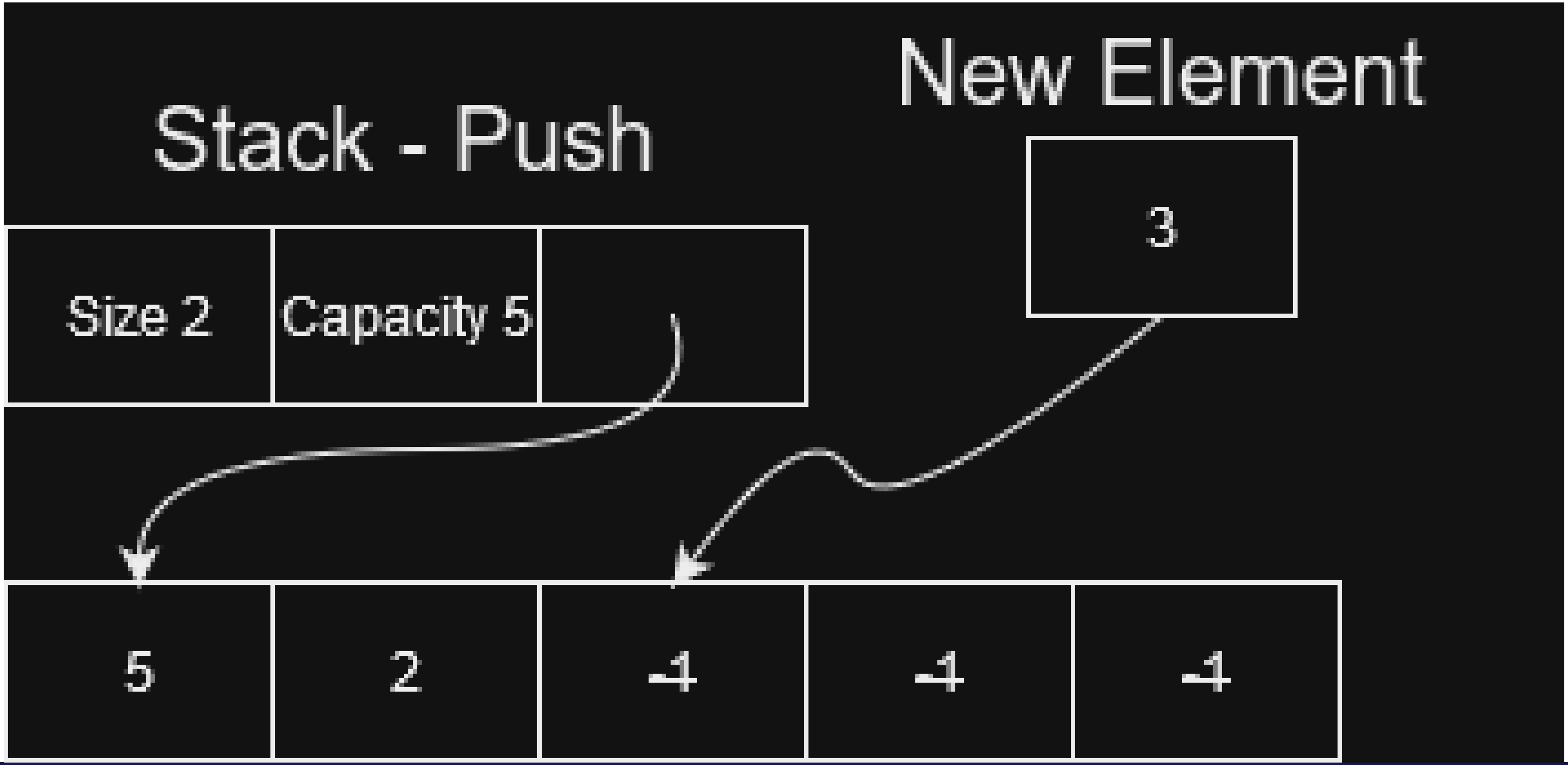
Image Link

Memory Allocations and Data Structures
Dynamic Arrays



Memory Allocations and Data Structures
Dynamic Arrays

Why resizing can be problematic for memory safety (pointers and references).



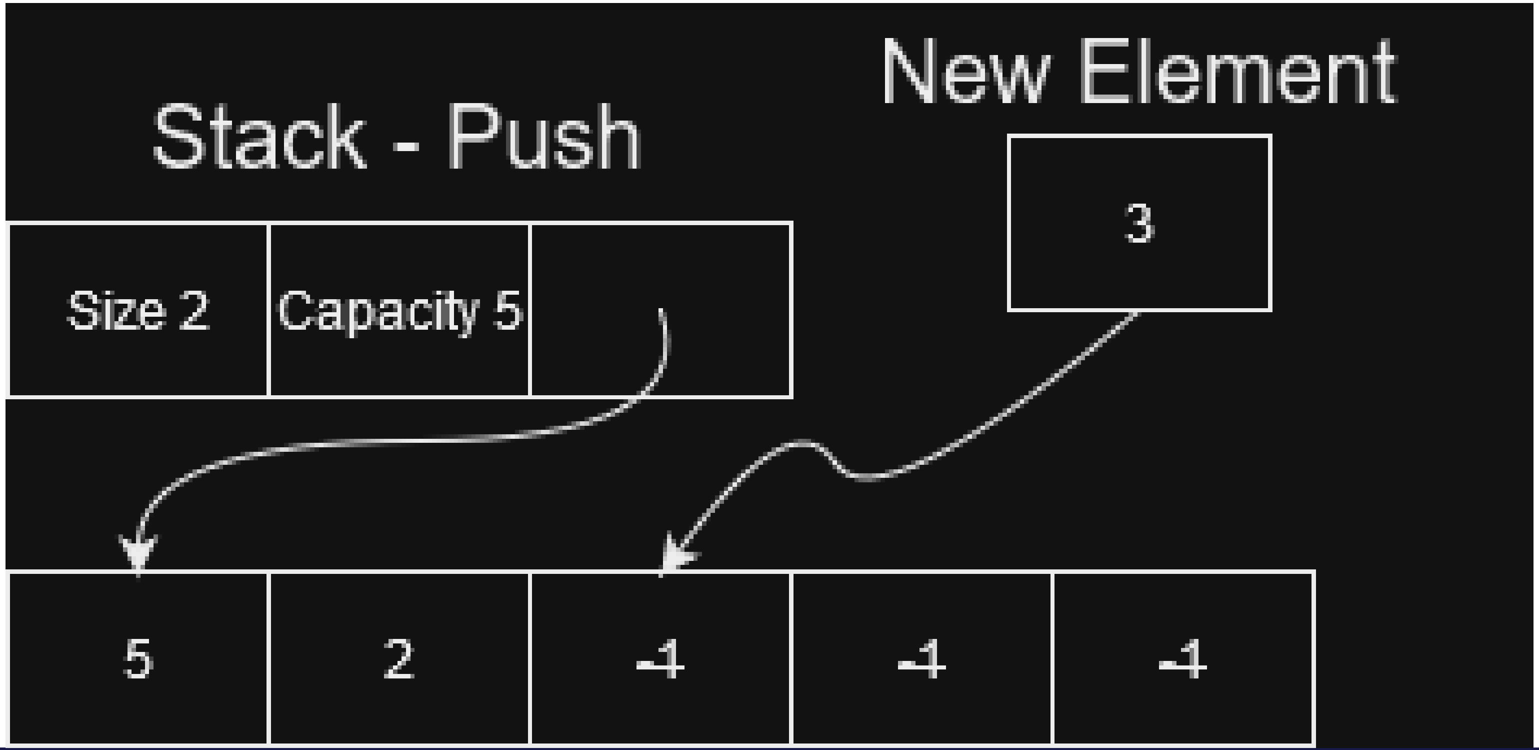
Memory Hierarchies in Software
Reference, Copy, Clone, Move

Adding references is the default in Python

Clone is the default in C++ for most types like vector<float>

Move is the default in Rust for most types like Vec<f32>

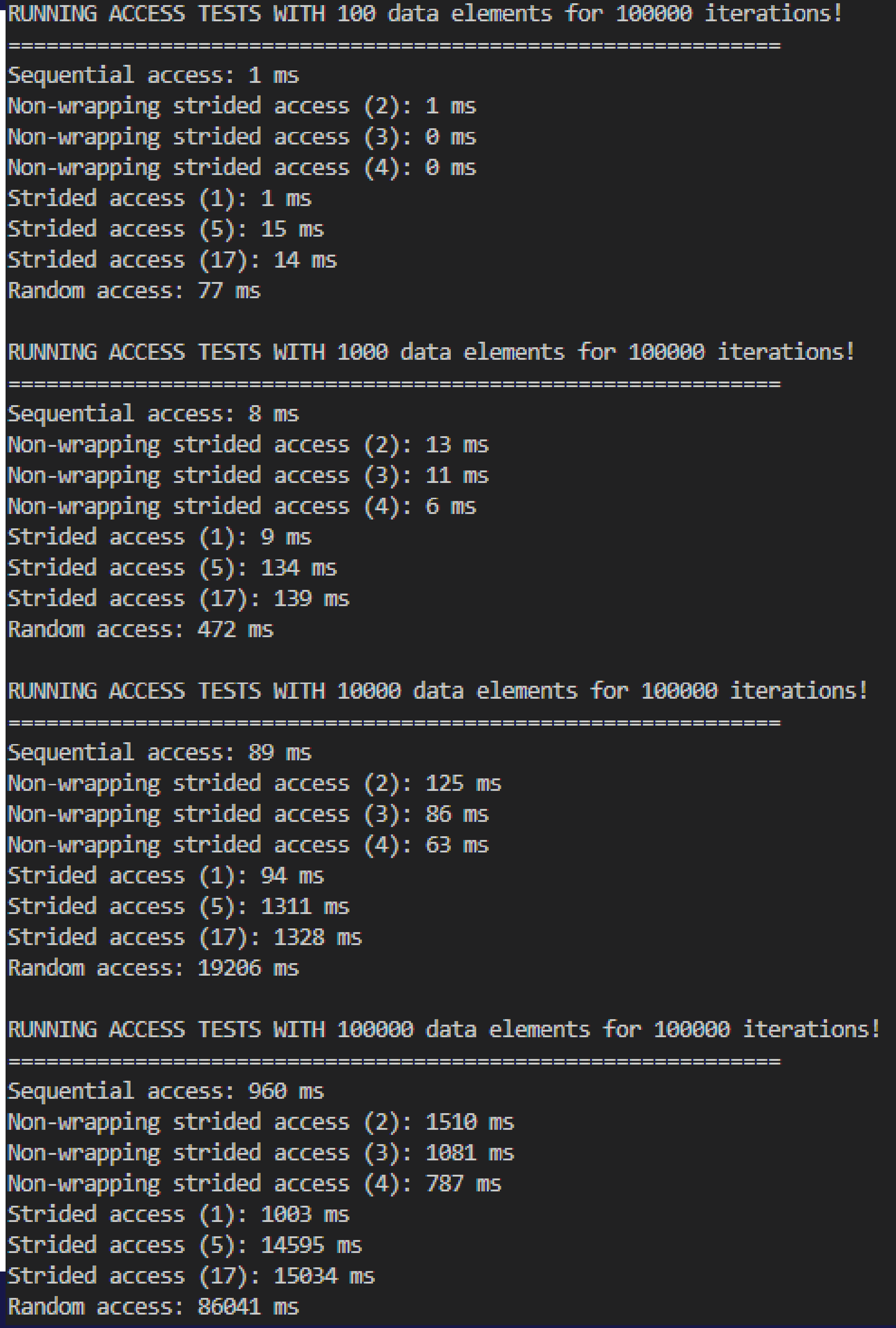
Copy is standard for very small types like f32 and u32 for both languages



Memory Allocations and Data Structures Accesses and Strides

Now that we know how a list/vector/dynamic array works, how do we access it?

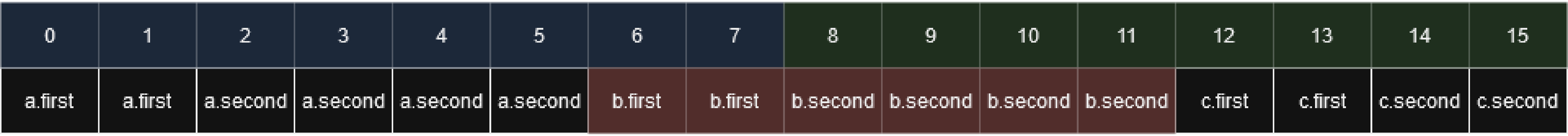
Strides



```
RUNNING ACCESS TESTS WITH 100 data elements for 100000 iterations!
=============================================================
Sequential access: 1 ms
Non-wrapping strided access (2): 1 ms
Non-wrapping strided access (3): 0 ms
Non-wrapping strided access (4): 0 ms
Strided access (1): 1 ms
Strided access (5): 15 ms
Strided access (17): 14 ms
Random access: 77 ms

RUNNING ACCESS TESTS WITH 1000 data elements for 100000 iterations!
=============================================================
Sequential access: 8 ms
Non-wrapping strided access (2): 13 ms
Non-wrapping strided access (3): 11 ms
Non-wrapping strided access (4): 6 ms
Strided access (1): 9 ms
Strided access (5): 134 ms
Strided access (17): 139 ms
Random access: 472 ms

RUNNING ACCESS TESTS WITH 10000 data elements for 100000 iterations!
=============================================================
Sequential access: 89 ms
Non-wrapping strided access (2): 125 ms
Non-wrapping strided access (3): 86 ms
Non-wrapping strided access (4): 63 ms
Strided access (1): 94 ms
Strided access (5): 1311 ms
Strided access (17): 1328 ms
Random access: 19206 ms

RUNNING ACCESS TESTS WITH 100000 data elements for 100000 iterations!
=============================================================
Sequential access: 960 ms
Non-wrapping strided access (2): 1510 ms
Non-wrapping strided access (3): 1081 ms
Non-wrapping strided access (4): 787 ms
Strided access (1): 1003 ms
Strided access (5): 14595 ms
Strided access (17): 15034 ms
Random access: 86041 ms
```

Memory Allocations and Data Structures Accesses and Strides

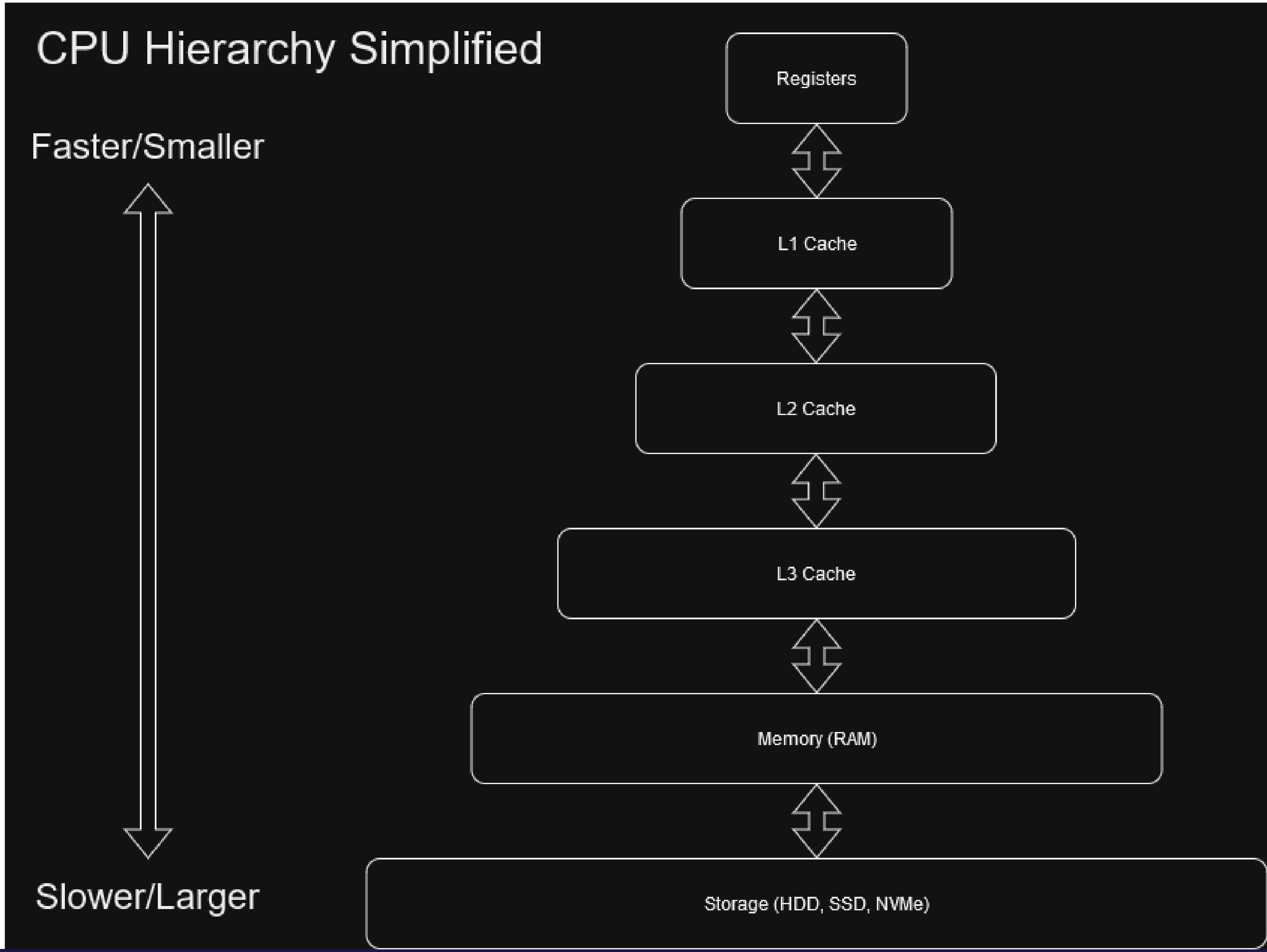
Cache Lines



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a.first	a.first	a.second	a.second	a.second	a.second	b.first	b.first	b.second	b.second	b.second	b.second	c.first	c.first	c.second	c.second

Memory Hierarchies in Hardware

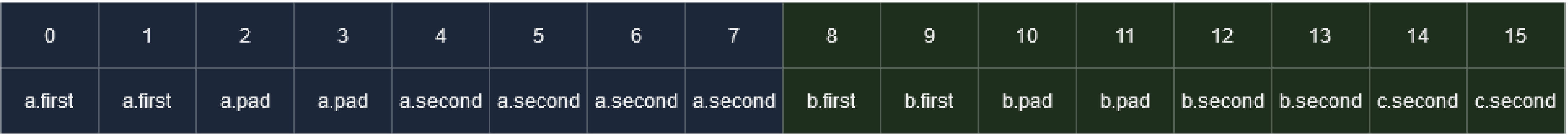
From slowest to fastest



Memory Allocations and Data Structures
Accesses and Strides

Pad for alignment

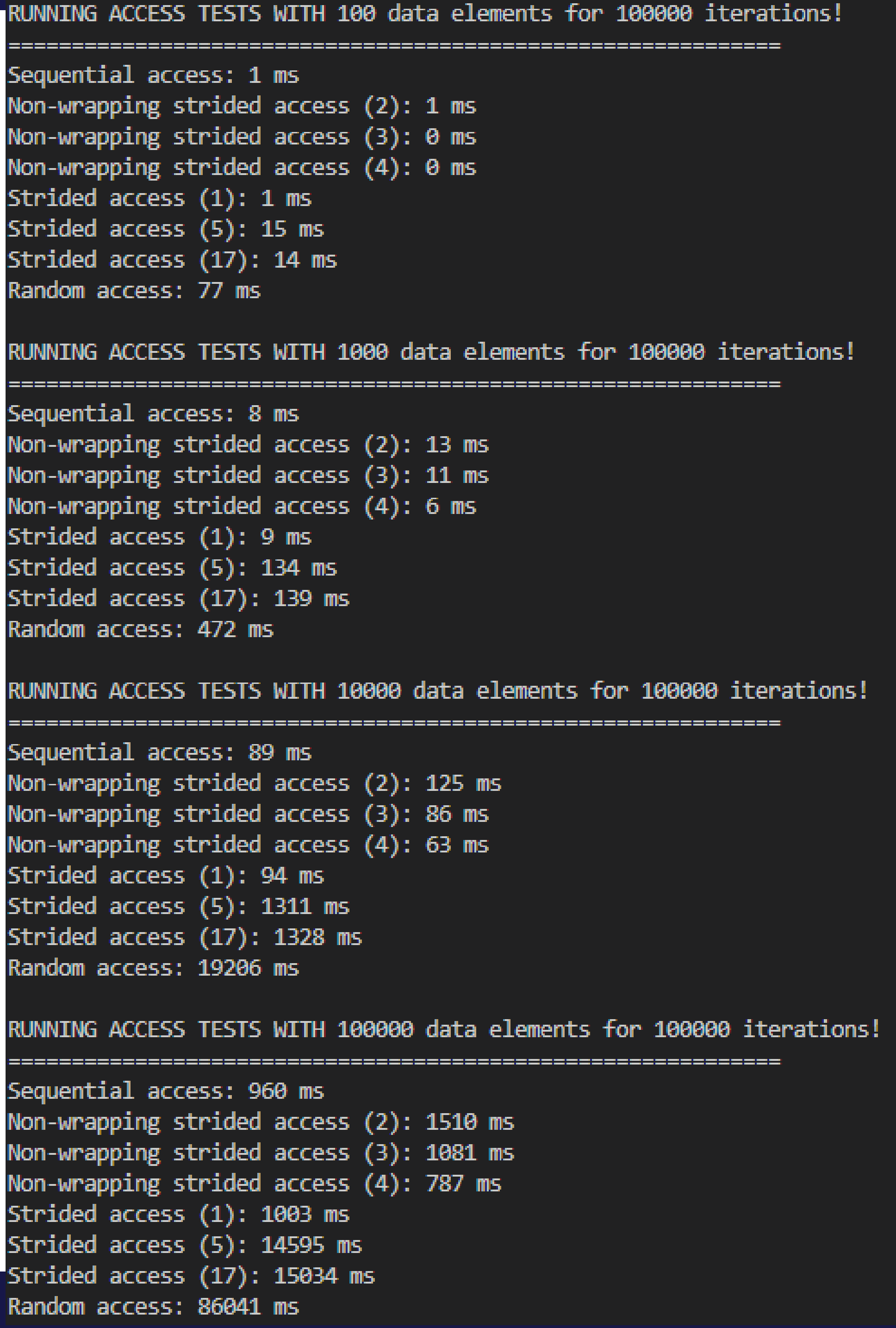
Remove pad for storage



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a.first	a.first	a.pad	a.pad	a.second	a.second	a.second	a.second	b.first	b.first	b.pad	b.pad	b.second	b.second	c.second	c.second

Memory Allocations and Data Structures Accesses and Strides

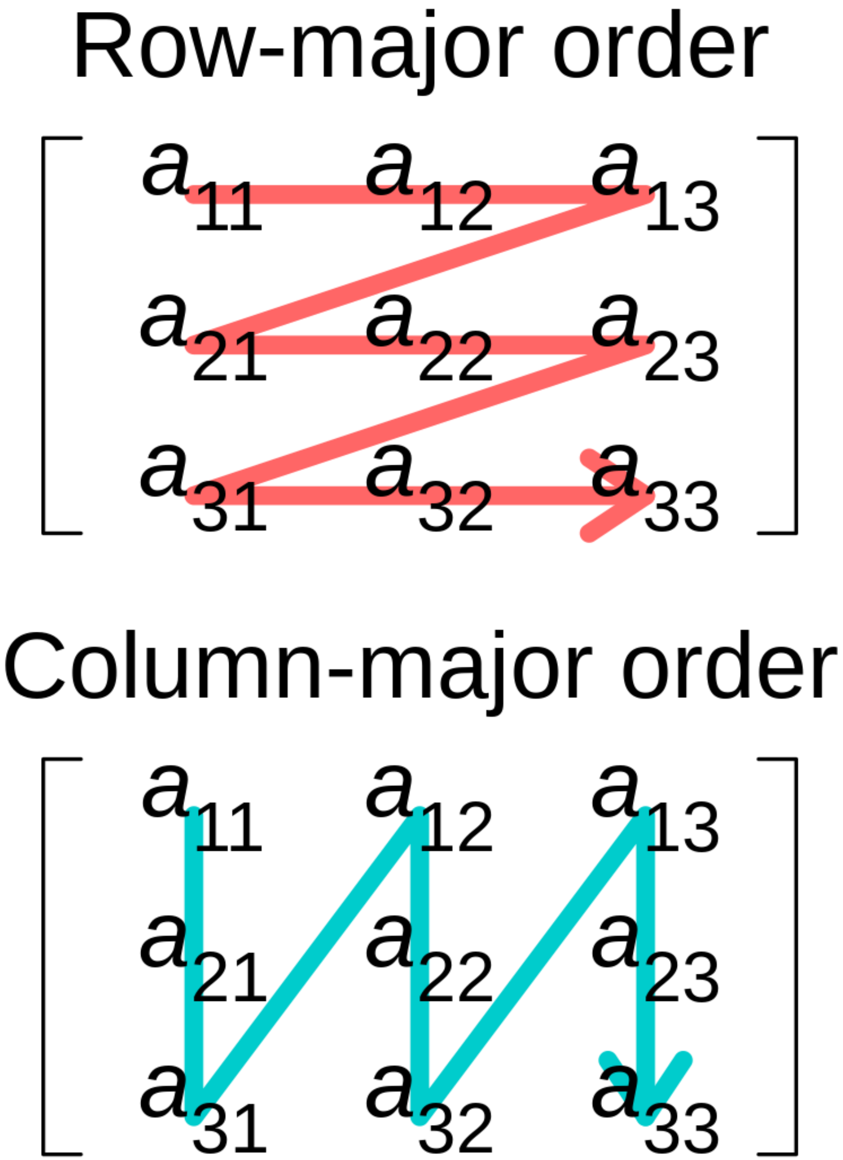
Strides – are you getting the most out of your cache line?



```
RUNNING ACCESS TESTS WITH 100 data elements for 100000 iterations!
=============================================================
Sequential access: 1 ms
Non-wrapping strided access (2): 1 ms
Non-wrapping strided access (3): 0 ms
Non-wrapping strided access (4): 0 ms
Strided access (1): 1 ms
Strided access (5): 15 ms
Strided access (17): 14 ms
Random access: 77 ms

RUNNING ACCESS TESTS WITH 1000 data elements for 100000 iterations!
=============================================================
Sequential access: 8 ms
Non-wrapping strided access (2): 13 ms
Non-wrapping strided access (3): 11 ms
Non-wrapping strided access (4): 6 ms
Strided access (1): 9 ms
Strided access (5): 134 ms
Strided access (17): 139 ms
Random access: 472 ms

RUNNING ACCESS TESTS WITH 10000 data elements for 100000 iterations!
=============================================================
Sequential access: 89 ms
Non-wrapping strided access (2): 125 ms
Non-wrapping strided access (3): 86 ms
Non-wrapping strided access (4): 63 ms
Strided access (1): 94 ms
Strided access (5): 1311 ms
Strided access (17): 1328 ms
Random access: 19206 ms

RUNNING ACCESS TESTS WITH 100000 data elements for 100000 iterations!
=============================================================
Sequential access: 960 ms
Non-wrapping strided access (2): 1510 ms
Non-wrapping strided access (3): 1081 ms
Non-wrapping strided access (4): 787 ms
Strided access (1): 1003 ms
Strided access (5): 14595 ms
Strided access (17): 15034 ms
Random access: 86041 ms
```

Memory Allocations and Data Structures
ND Arrays

Row-major and Column-major ordering



Row-major order

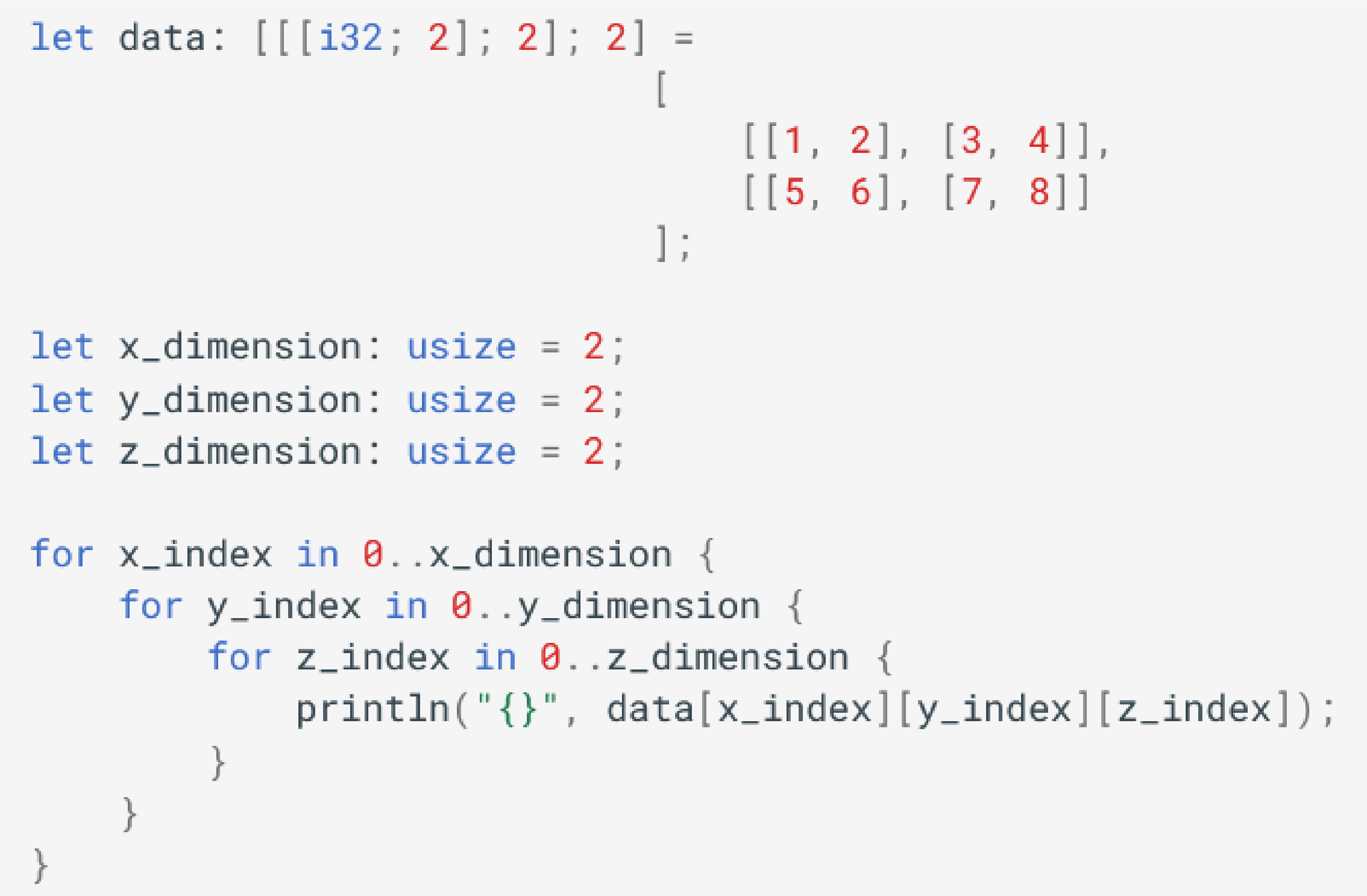
$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Column-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Memory Allocations and Data Structures
ND Arrays

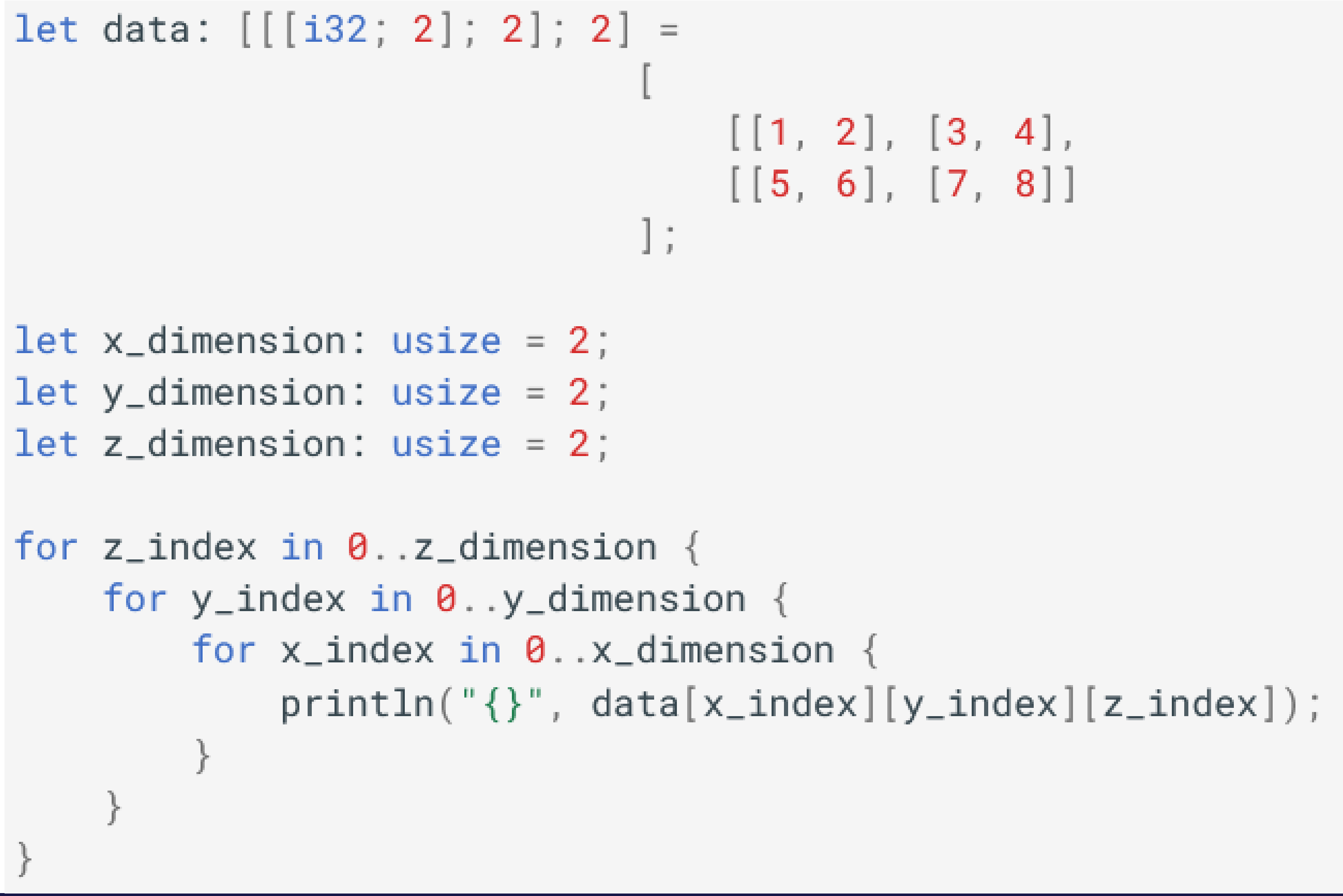
We can do this in 2D with row major ordering



```
let data: [[[i32; 2]; 2]; 2] =
                            [
                                [[1, 2], [3, 4]],
                                [[5, 6], [7, 8]]
                            ];

let x_dimension: usize = 2;
let y_dimension: usize = 2;
let z_dimension: usize = 2;

for x_index in 0..x_dimension {
    for y_index in 0..y_dimension {
        for z_index in 0..z_dimension {
            println("{}", data[x_index][y_index][z_index]);
        }
    }
}
```

Memory Allocations and Data Structures
ND Arrays

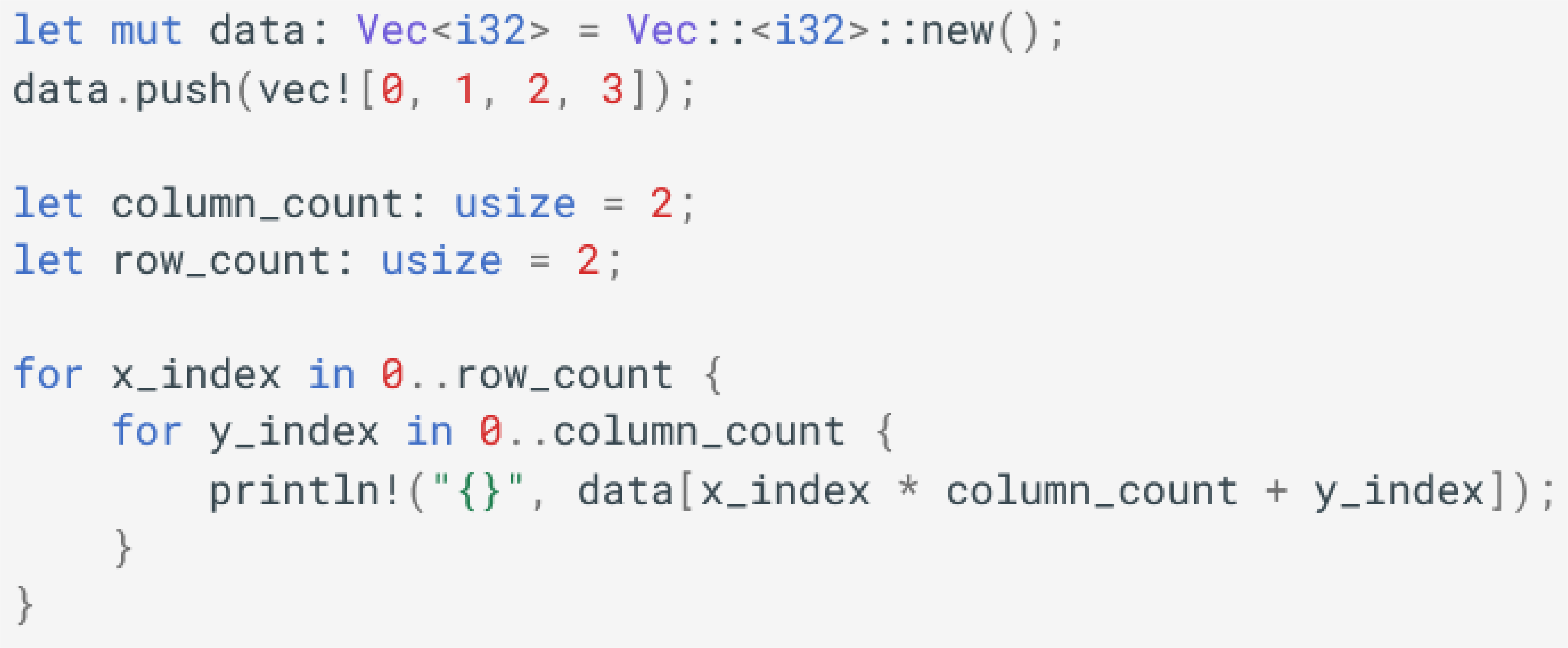
We can do this in
2D with column major
ordering



```
let data: [[[i32; 2]; 2]; 2] =
                            [
                                [[1, 2], [3, 4]],
                                [[5, 6], [7, 8]]
                            ];

let x_dimension: usize = 2;
let y_dimension: usize = 2;
let z_dimension: usize = 2;

for z_index in 0..z_dimension {
    for y_index in 0..y_dimension {
        for x_index in 0..x_dimension {
            println("{}", data[x_index][y_index][z_index]);
        }
    }
}
```

Memory Allocations and Data Structures
ND Arrays – Linearized Indexing

Or in 1D and gain nice properties



```rust
let mut data: Vec<i32> = Vec::<i32>::new();
data.push(vec![0, 1, 2, 3]);

let column_count: usize = 2;
let row_count: usize = 2;

for x_index in 0..row_count {
    for y_index in 0..column_count {
        println!("{}", data[x_index * column_count + y_index]);
    }
}
```

Memory Allocations and Data Structures
ND Arrays – Linearized Indexing

```
x_index * y_size * z_size + y_index * z_size + z_index
```

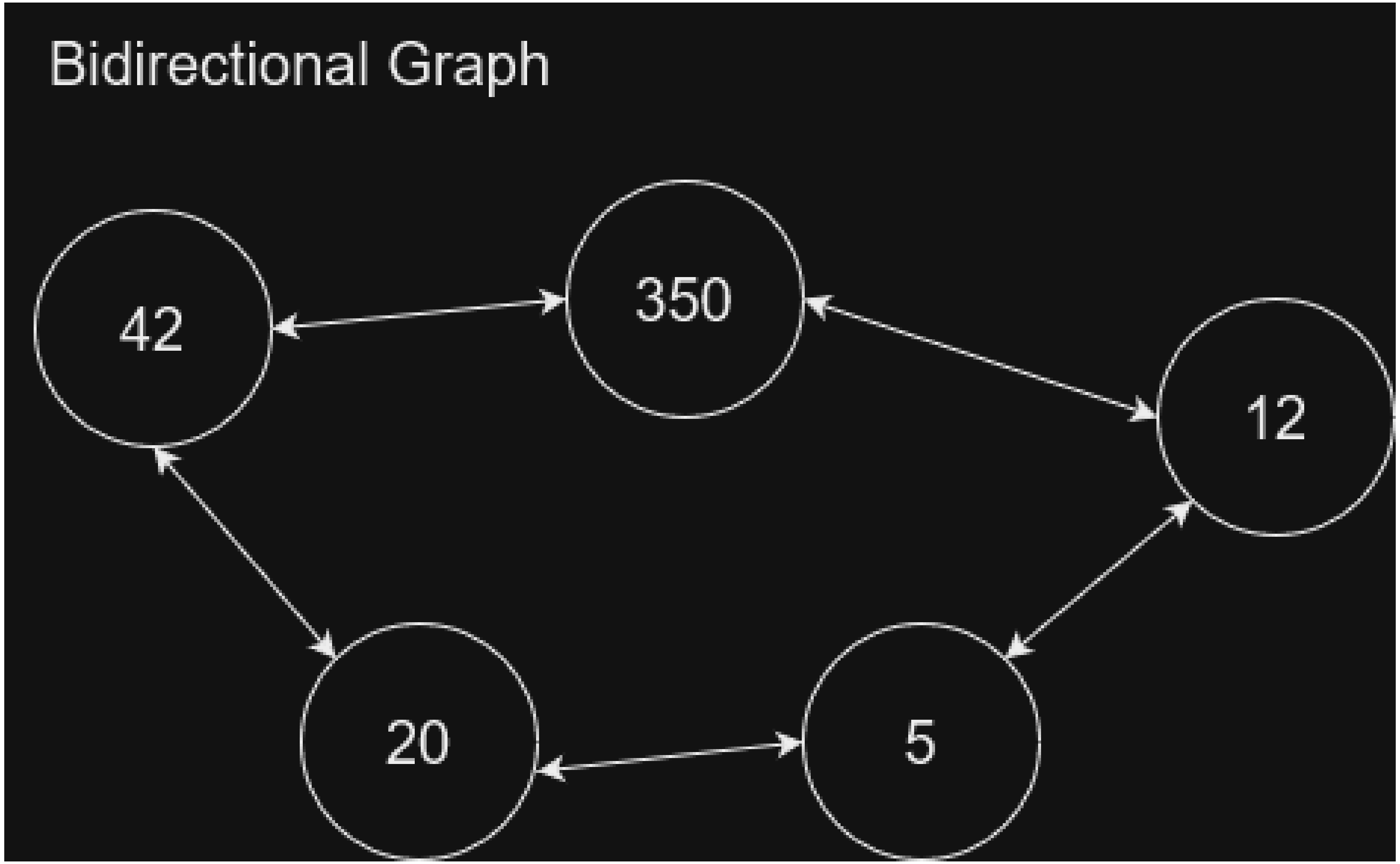
With linearized indexing we gain nice properties like different views on the same data, permutations, easier data transfers and better performance.

It also becomes a lot easier to move your code to the GPU.

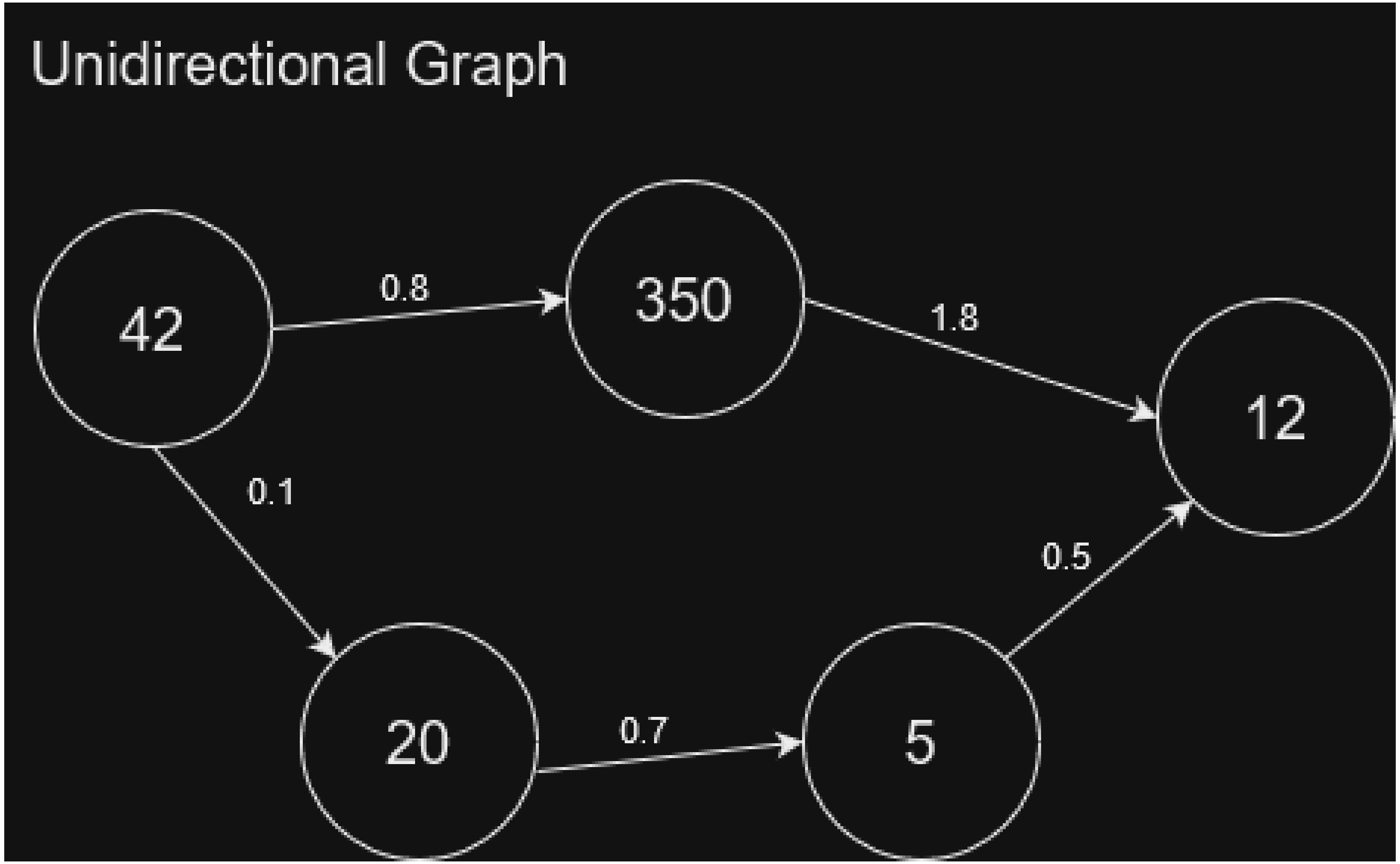
GRAPHS!

Pointers, pointers, pointers

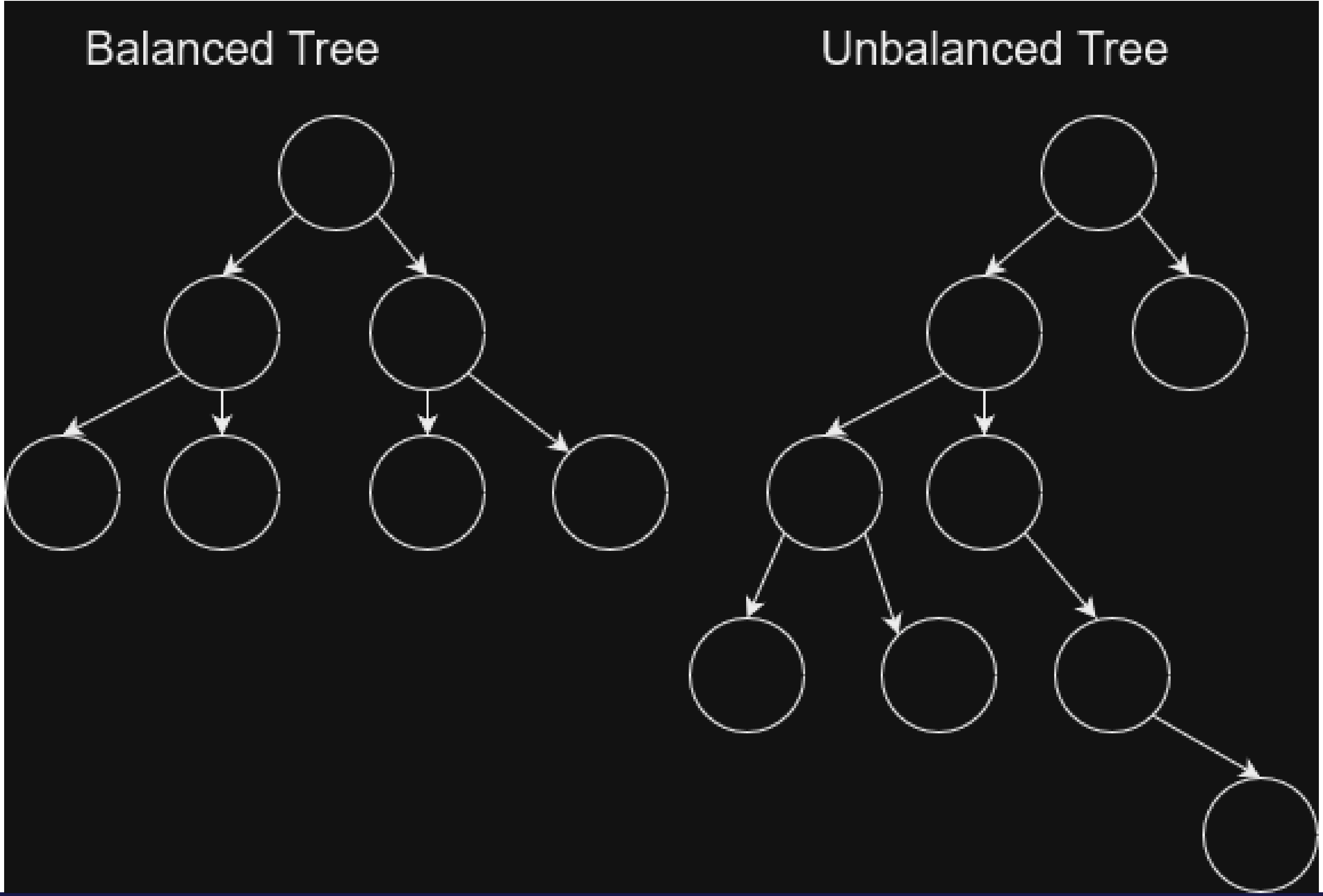
Graph Structures - Bidirectional



Graph Structures – Unidirectional with weighted edges



Graph Structures - Trees

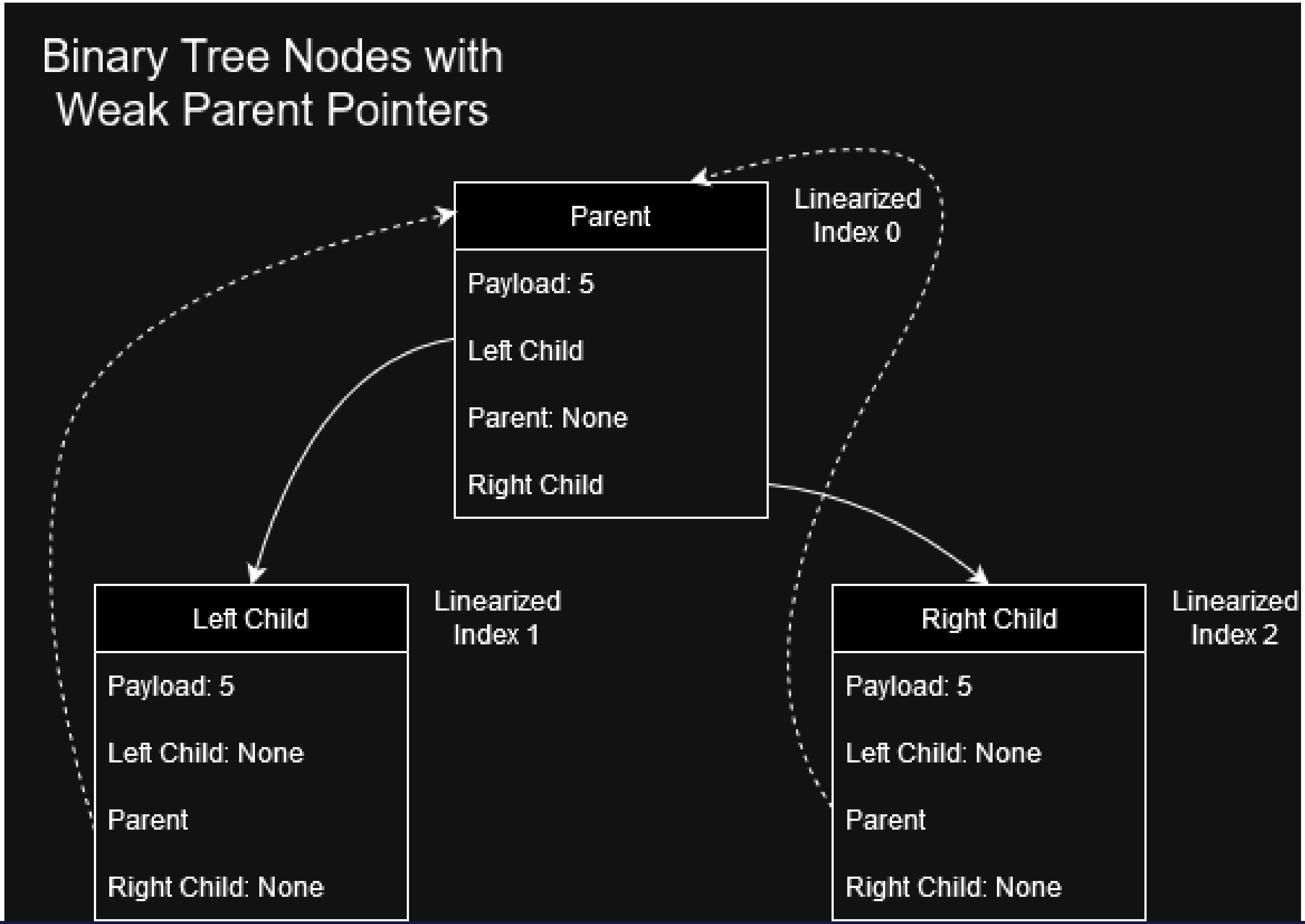


Smart Pointers

Rust and C++11+

- Box<T> or unique_ptr<T>
- Rc<T>, Arc<T> and shared_ptr<T>
- Weak<T> and weak_ptr<T>

Graph Structures



Binary Tree Nodes with
Weak Parent Pointers

Parent

Payload: 5

Left Child

Parent: None

Right Child

Linearized
Index 0

Left Child

Payload: 5

Left Child: None

Parent

Right Child: None

Linearized
Index 1

Right Child

Payload: 5

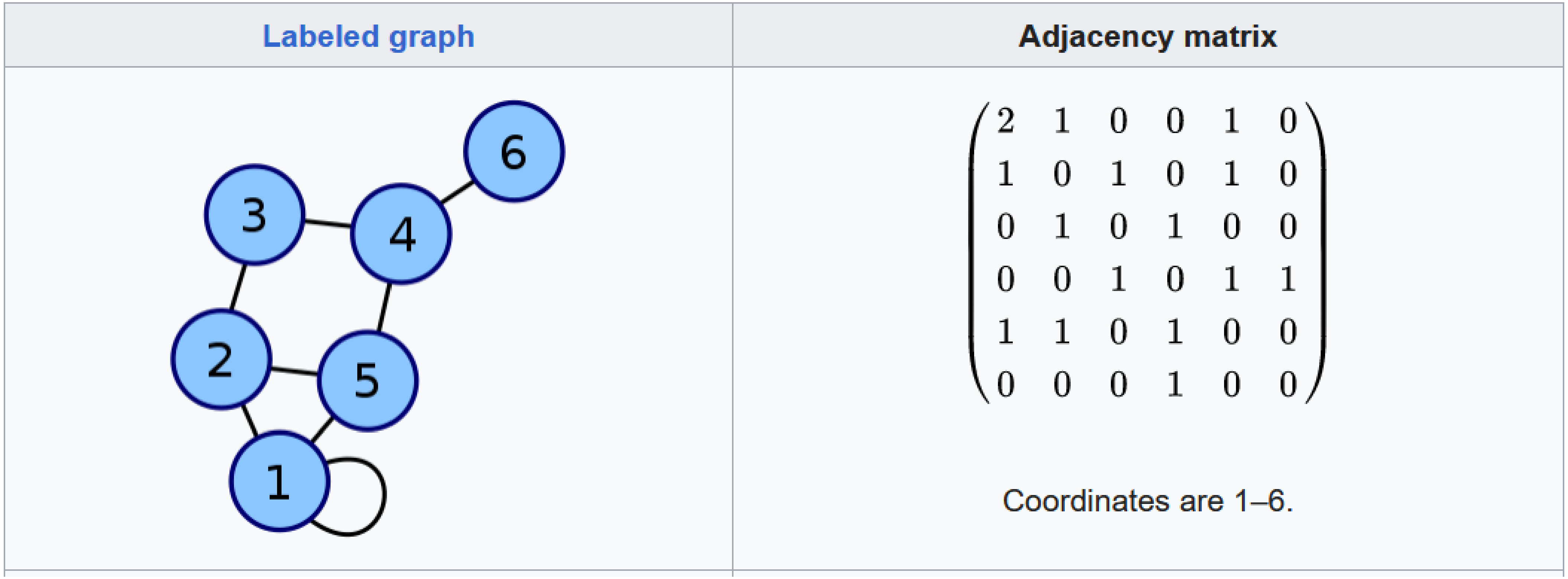
Left Child: None

Parent

Right Child: None

Linearized
Index 2

Graph Structures with Indices



Labeled graph	Adjacency matrix
	$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$ Coordinates are 1–6.

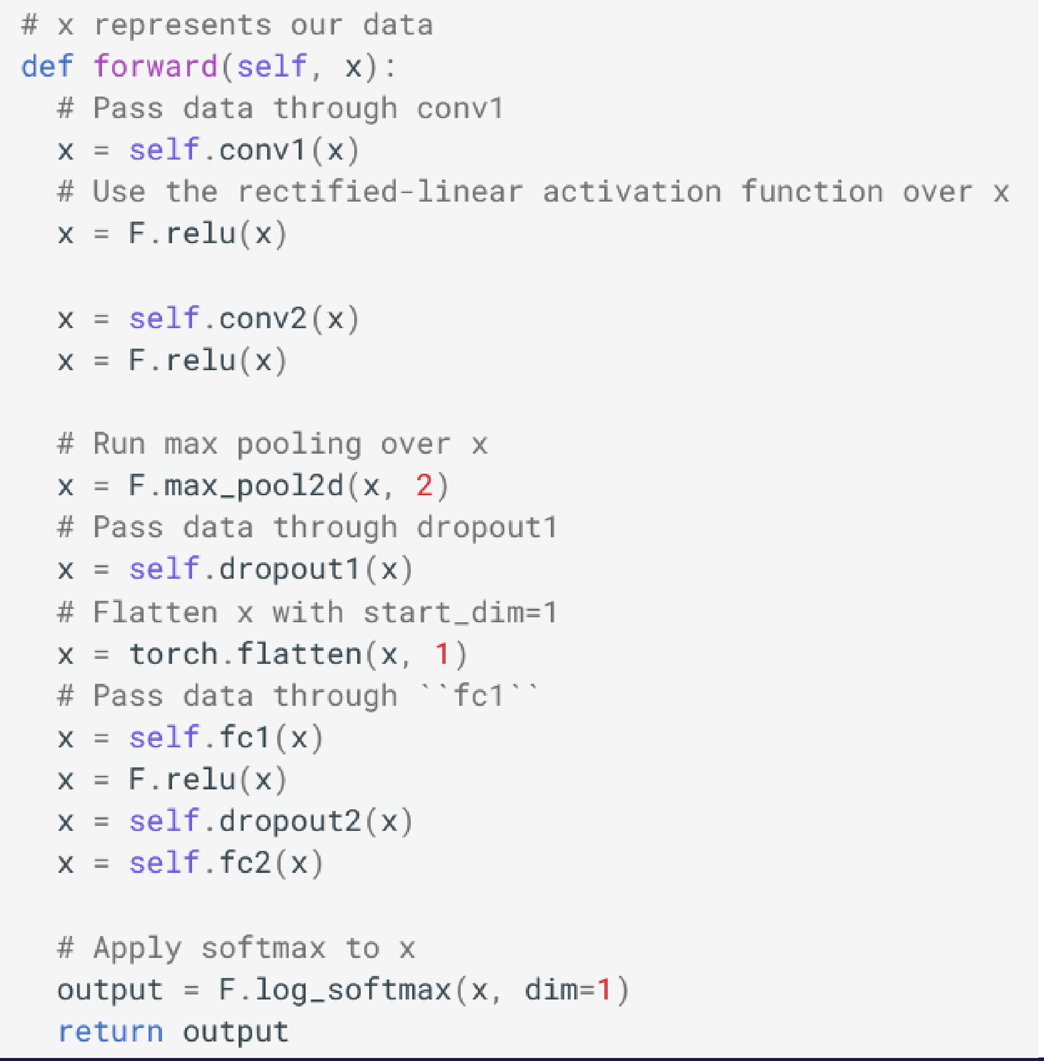
Image Link

Safety and Pointer Based Data Structures

Such a hard topic that there is a small book written about how to even do this safely in Rust

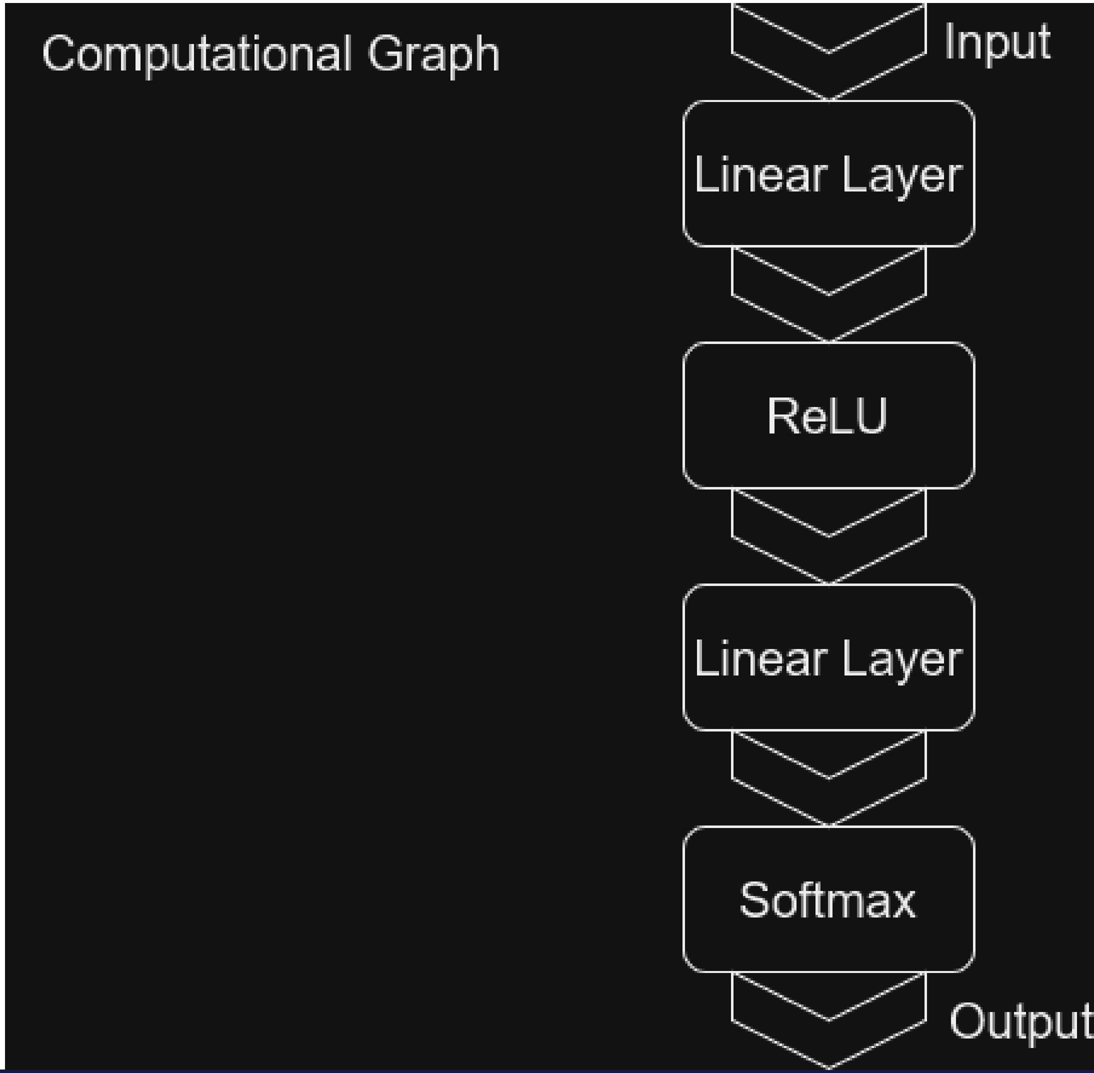
Learning Rust With Entirely Too Many Linked Lists

Computational Graphs

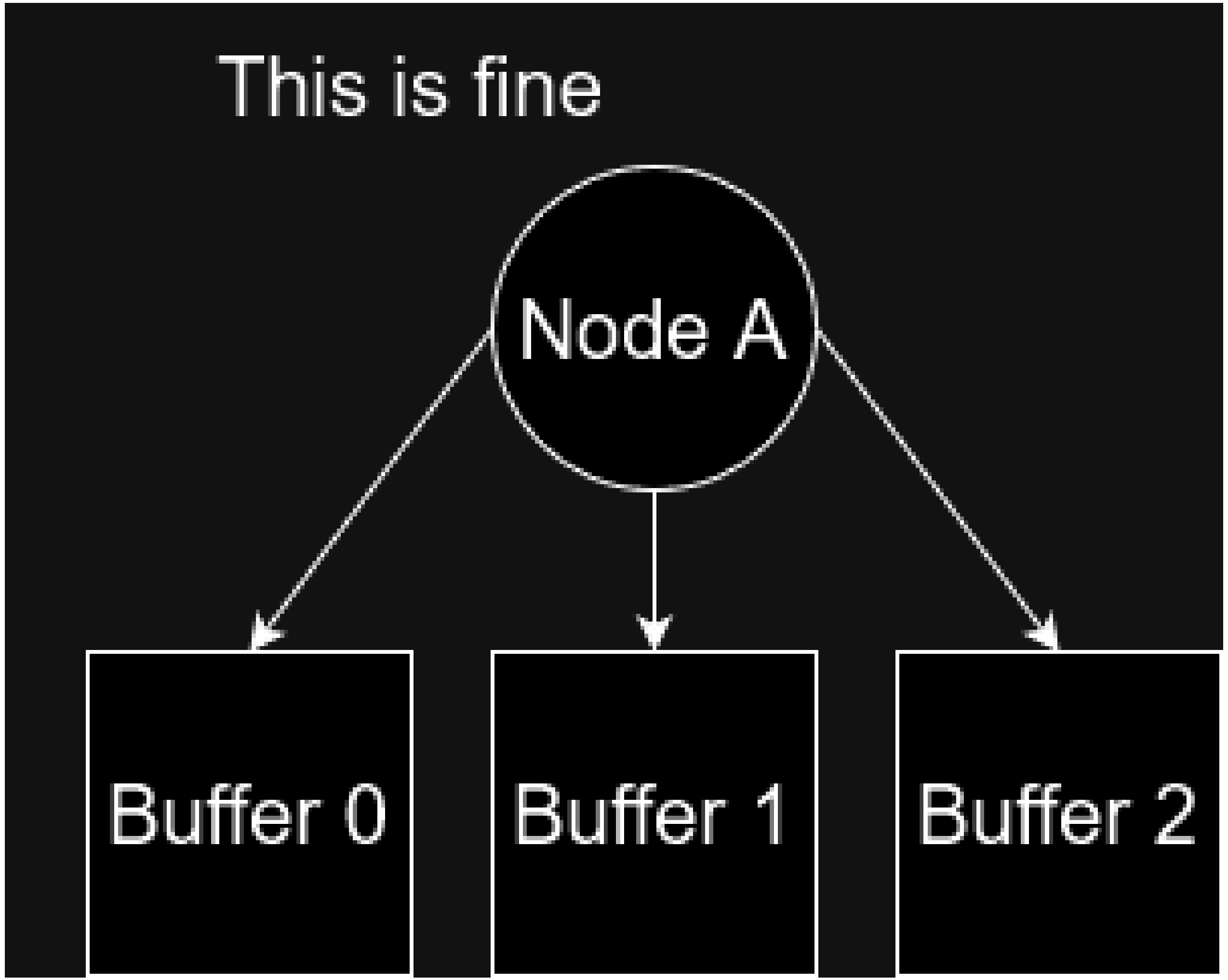


```
# x represents our data
def forward(self, x):
  # Pass data through conv1
  x = self.conv1(x)
  # Use the rectified-linear activation function over x
  x = F.relu(x)

  x = self.conv2(x)
  x = F.relu(x)

  # Run max pooling over x
  x = F.max_pool2d(x, 2)
  # Pass data through dropout1
  x = self.dropout1(x)
  # Flatten x with start_dim=1
  x = torch.flatten(x, 1)
  # Pass data through ``fc1``
  x = self.fc1(x)
  x = F.relu(x)
  x = self.dropout2(x)
  x = self.fc2(x)

  # Apply softmax to x
  output = F.log_softmax(x, dim=1)
  return output
```

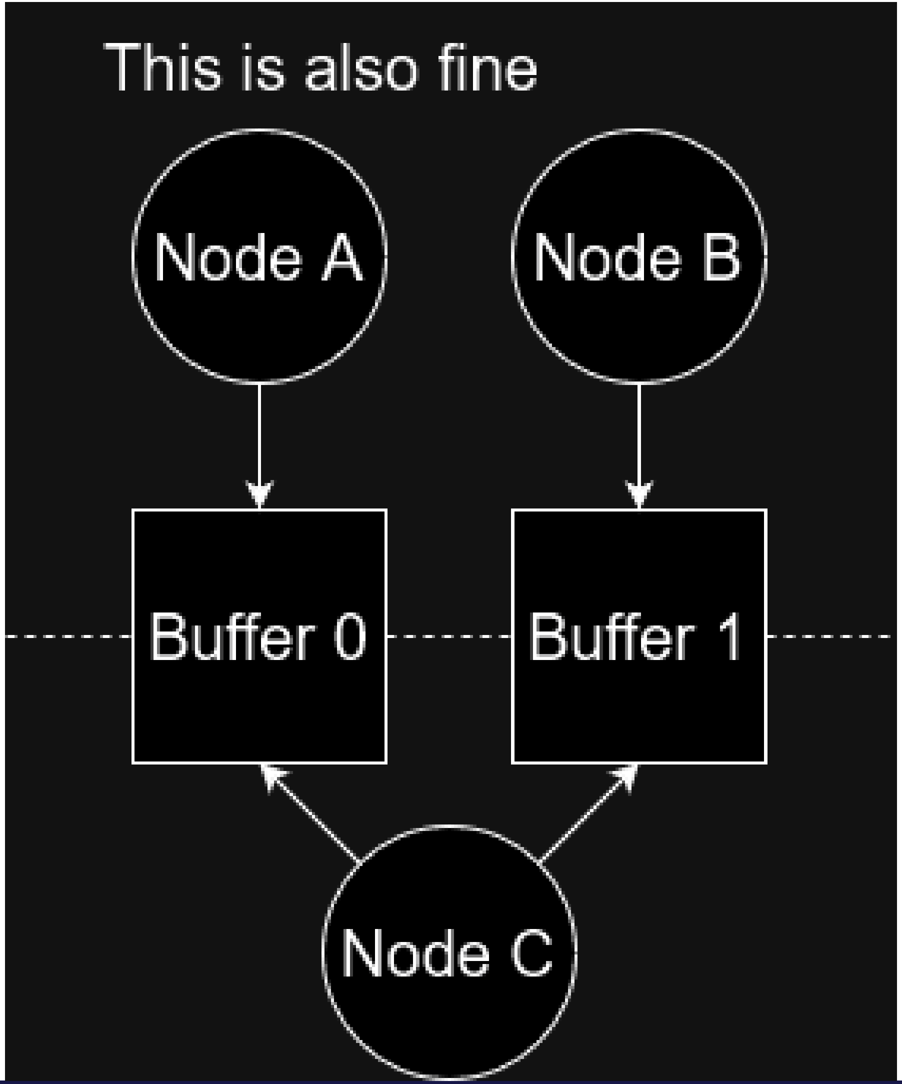
Image Link



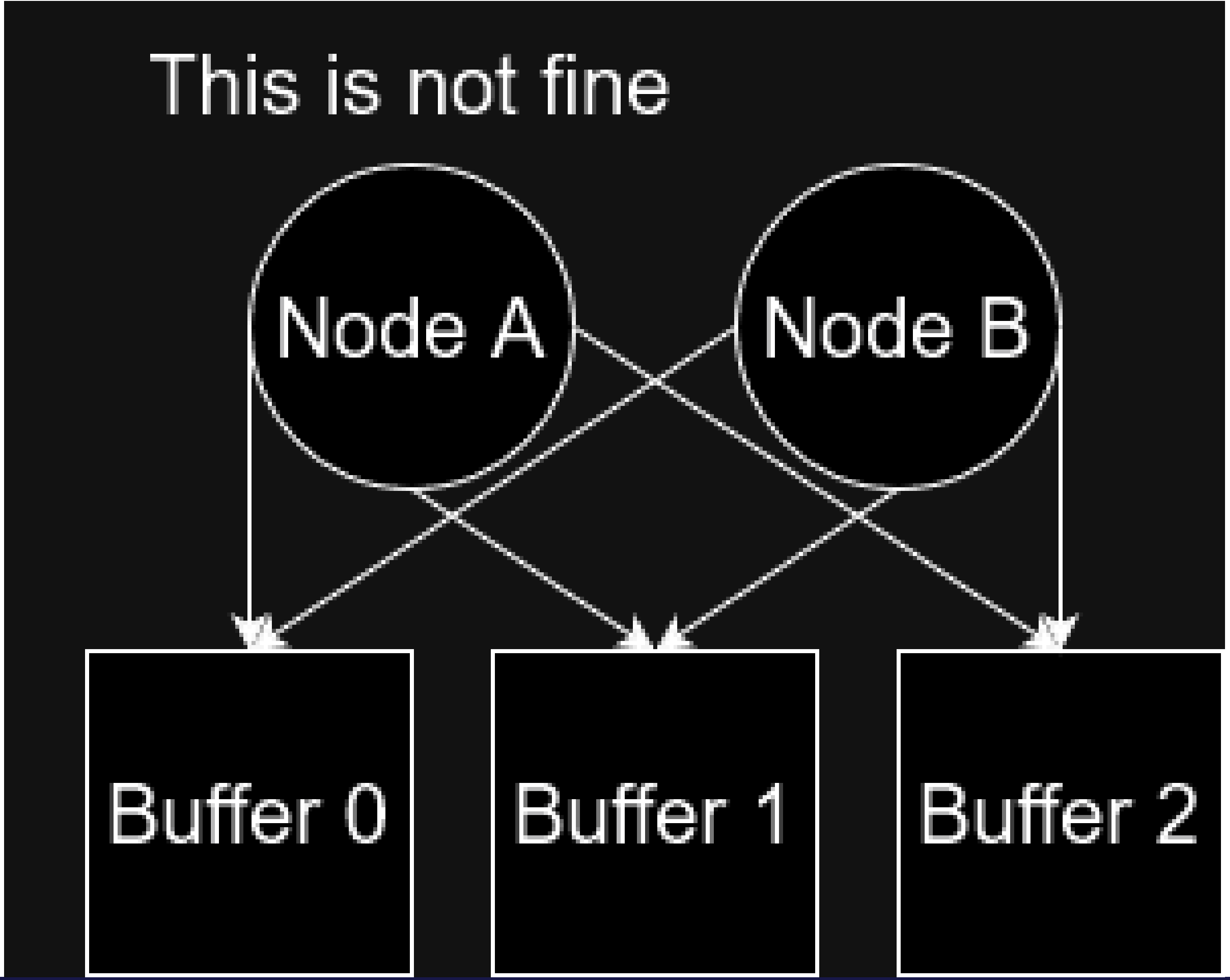
Computational Graphs



Computational Graphs



Computational Graphs & The Borrow Checker



Computational Graphs

These are the same restrictions enforced by Rust’s borrow checker

If we build our own computational graph system, we can enforce our own restrictions and checks

This can happen either at compile time, just-in-time or at runtime

Garbage Collectors

Reference counting in languages like Python, C# and Java

Everything is hidden behind a shared pointer!

Add a garbage collector!

Exercises

Draw, write and discuss (the top exercises)

Once you are done

Do more Advent of Code or other Rust tutorials if you feel like you need to do more.

or

Do Ray Tracing In One Weekend – it will confront you with the borrow checker, smart pointers, traits and dyn. I have a code snippet for showing the ray traced image on your screen.

If you complete your 1-to-1 Rust implementation of RTIOW, try to do the following:

- Remove the use of dynamic dispatching by not using dyn anywhere. (Hint: use enums)
- Remove the use of smart pointers. (Hint: my implementation was based on indices and dependency injection of a geometry service)
- Parallelize the application by using the rayon library