Theoretical Time-Complexity of Memory Cache

Rasmus Tollund

March 20, 2025

DEIS Junior Club - Aalborg University

This talk is inspired by Myth of Ram - by Emil Ernerfeldt

Last Week on Juniour Club

Data-Oriented Programming

How to speeeeeeeeeeeeeed

Nicolaj Østerby Jensen

(Heavy inspiration from Nic Barker's talk; links on last slide)

Data Oriented Programming

Key points:

- Memory segmented into increasing sized blocks
- Larger memory blocks are slower to access
- Caching has big impact on performance

Chopping board, pan, pot ~ Registers
Knife, spoon, grater ~ Instructions
Table ~ L1 cache
Shelves ~ L2 cache
Storage room ~ L3 cache
Supermarket ~ RAM
The farm ~ Harddrive

Type	Size	Latency (cycles)
Registers (frying pan)	8-16 bytes	0
L1 (table)	64 kb	3
L2 (shelves)	2 mb	20
L3 (storage fridge)	32 mb	~100
DRAM (supermarket)	4 gb	2-300
Disk (farm)	200+ gb	300,000

Cache miss

Runtime complexity vs "cache-miss complexity"

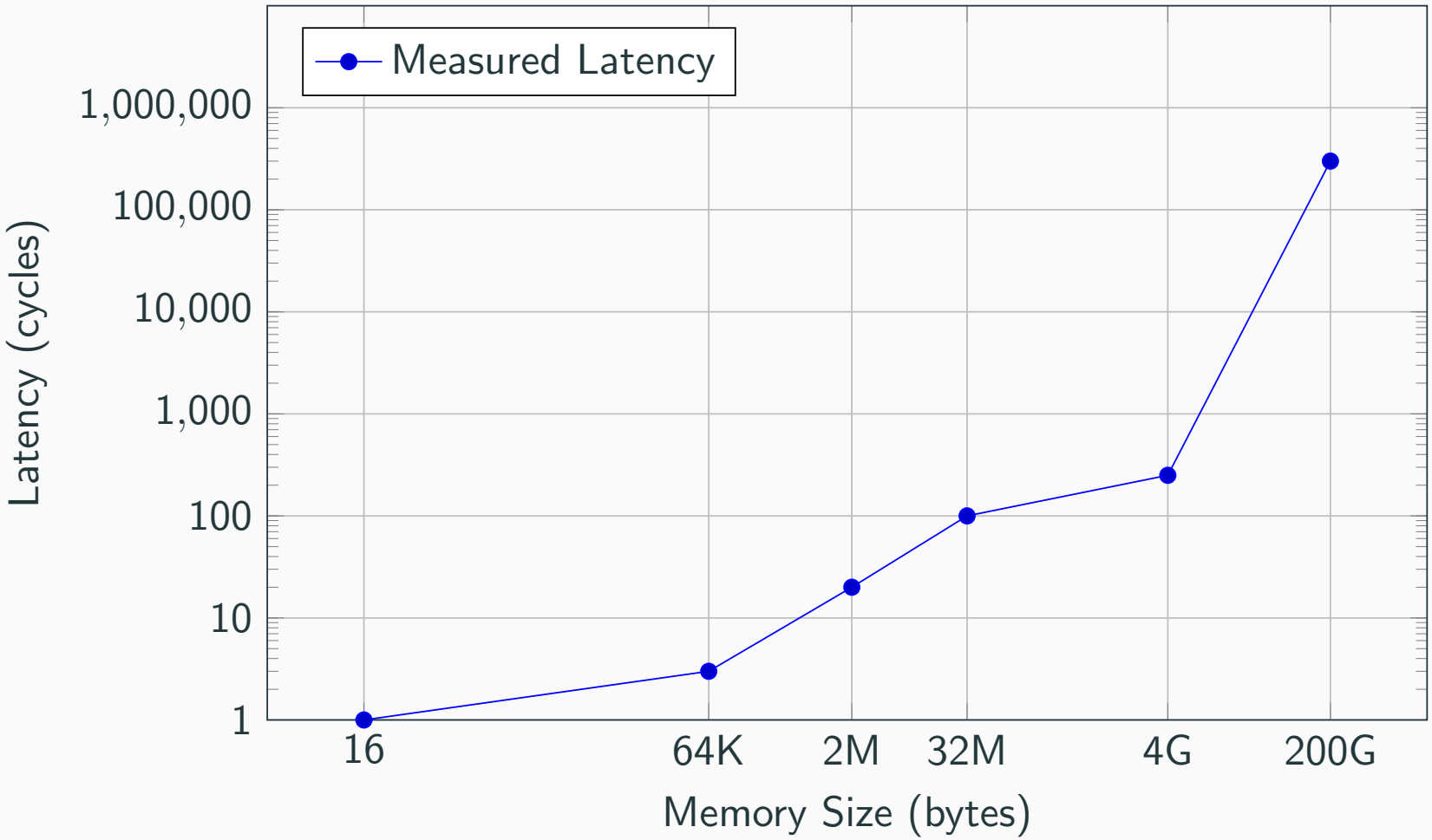
Always exponential

My CPU when the L1 cache misses

- But what if most instructions miss?
- 1000 misses ► 3 ms ► 36% of frame budget in game with 120 fps

This Week on Junior Club

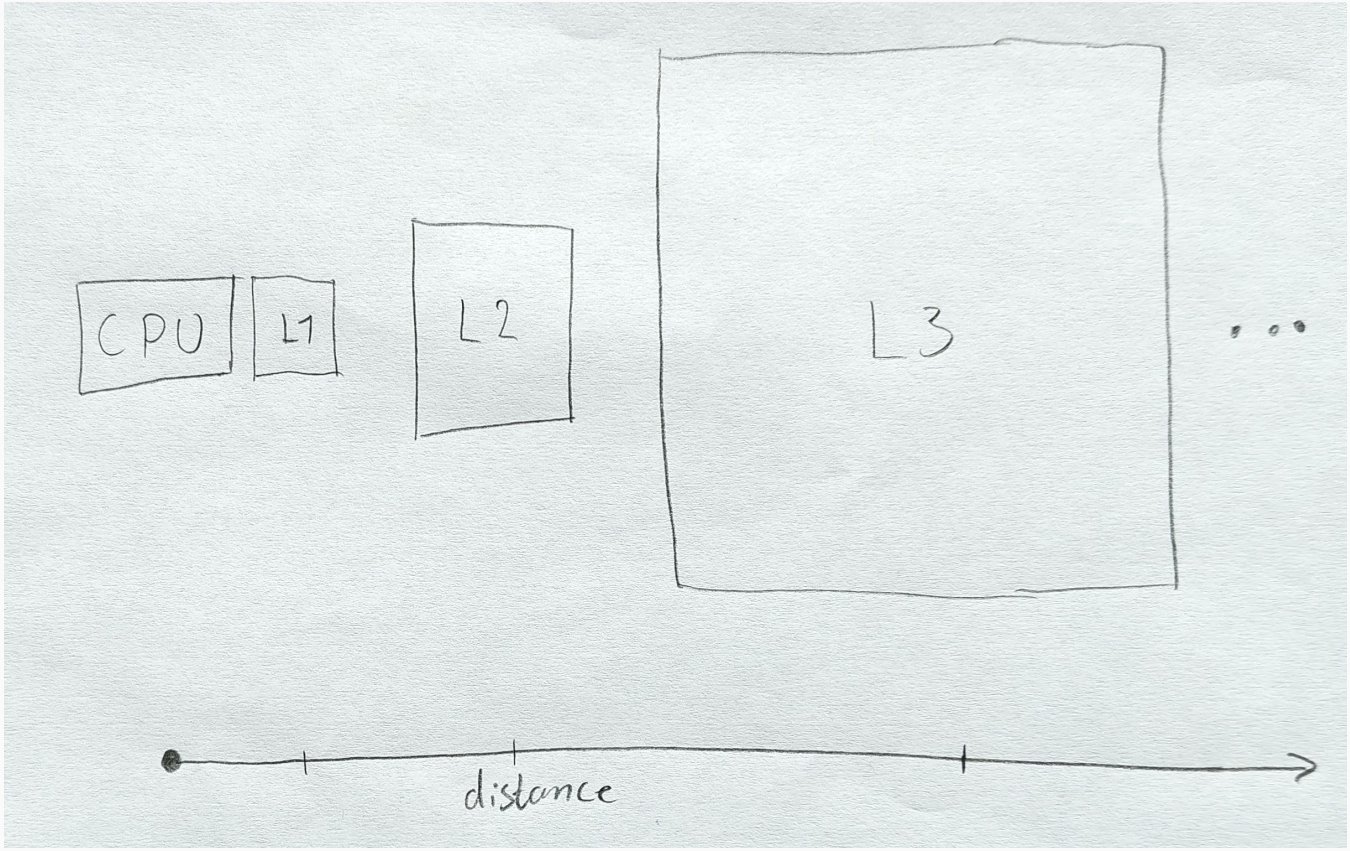
Latency as a Function of Memory Size



What is this function?

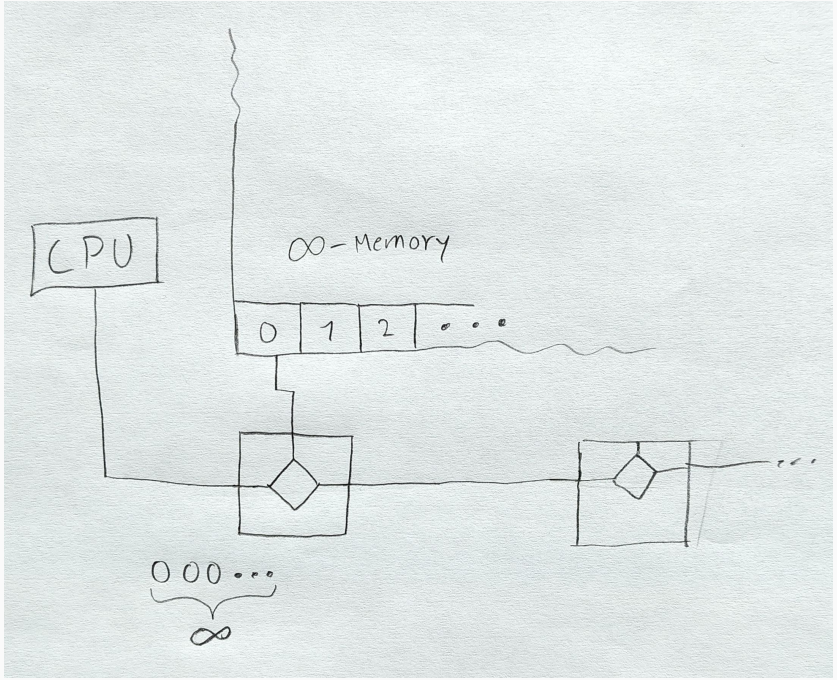
Caching of Memory

Memory is cached into levels of increasing size and distance from CPU



Why not just infinite block of memory?

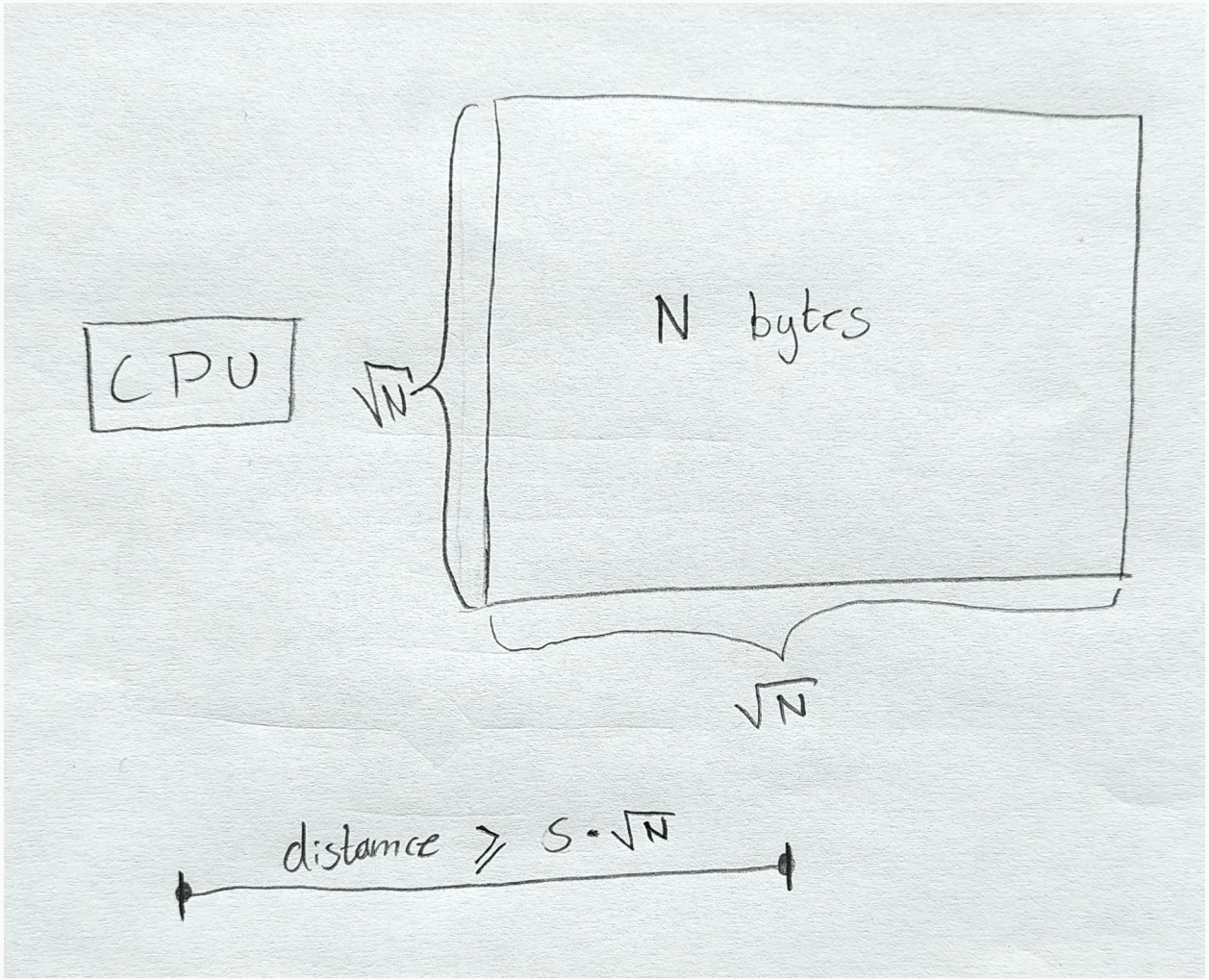
Memory must inherently be segmented into levels



How can the multiplexer decide if an infinite address refers to this memory?

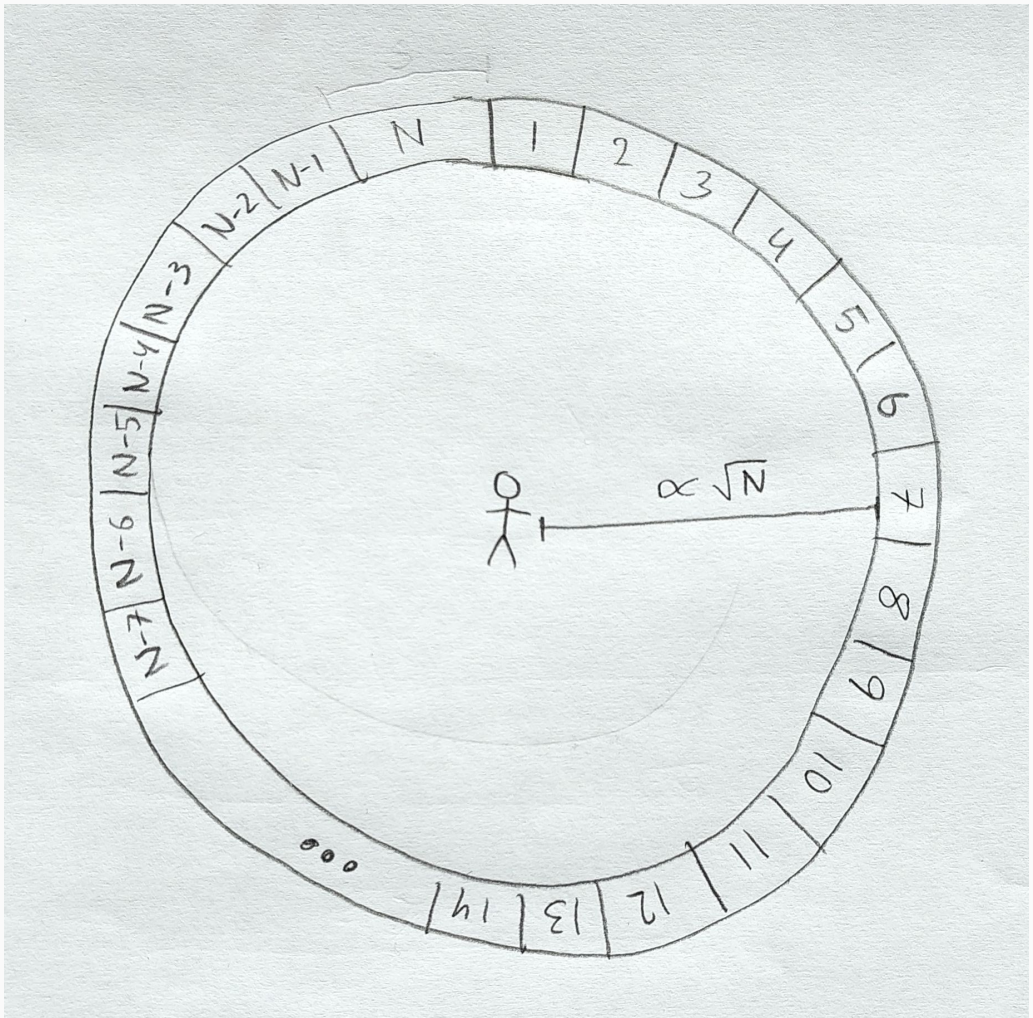
Distance to Memory Based on Size

Each byte (or bit) has a minimum non-zero size s.



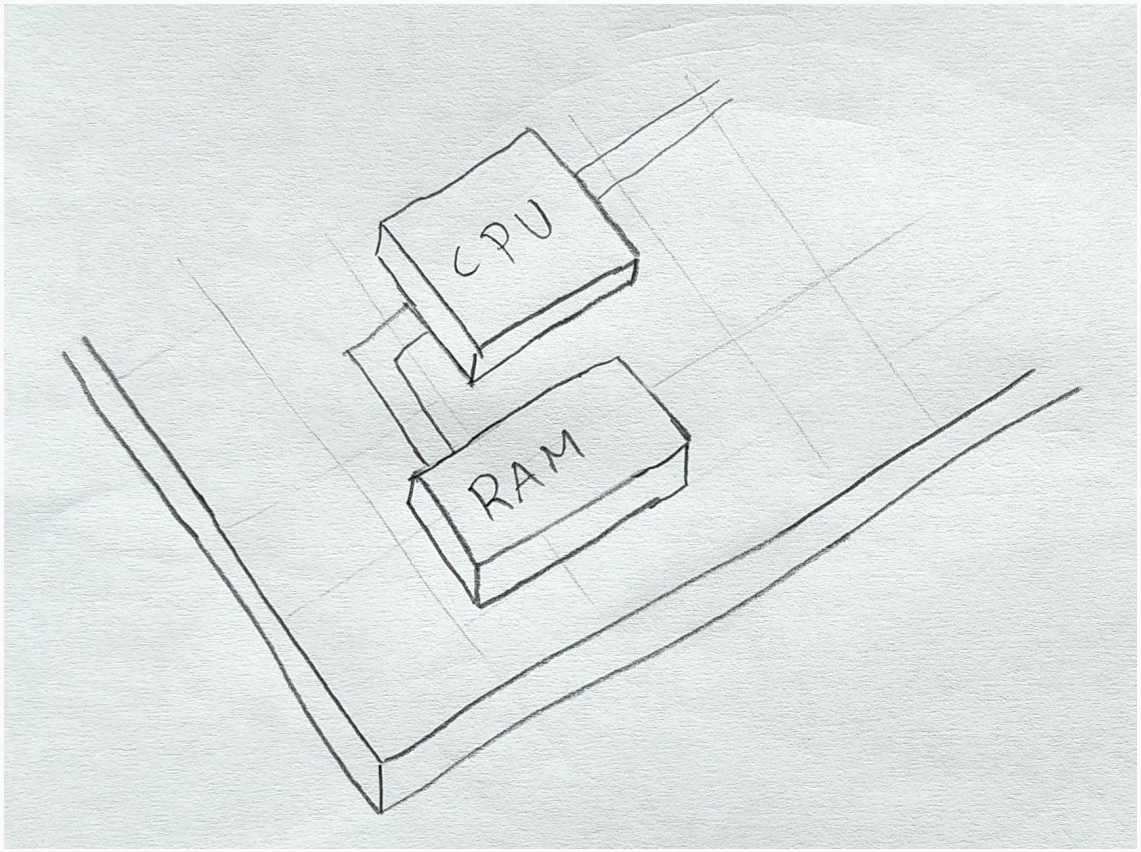
General Distance to Memory in 2 Dimensions

In general, in 2 dimensions, the distance access 1 memory unit within N units is proportional to $\sqrt{N}$.



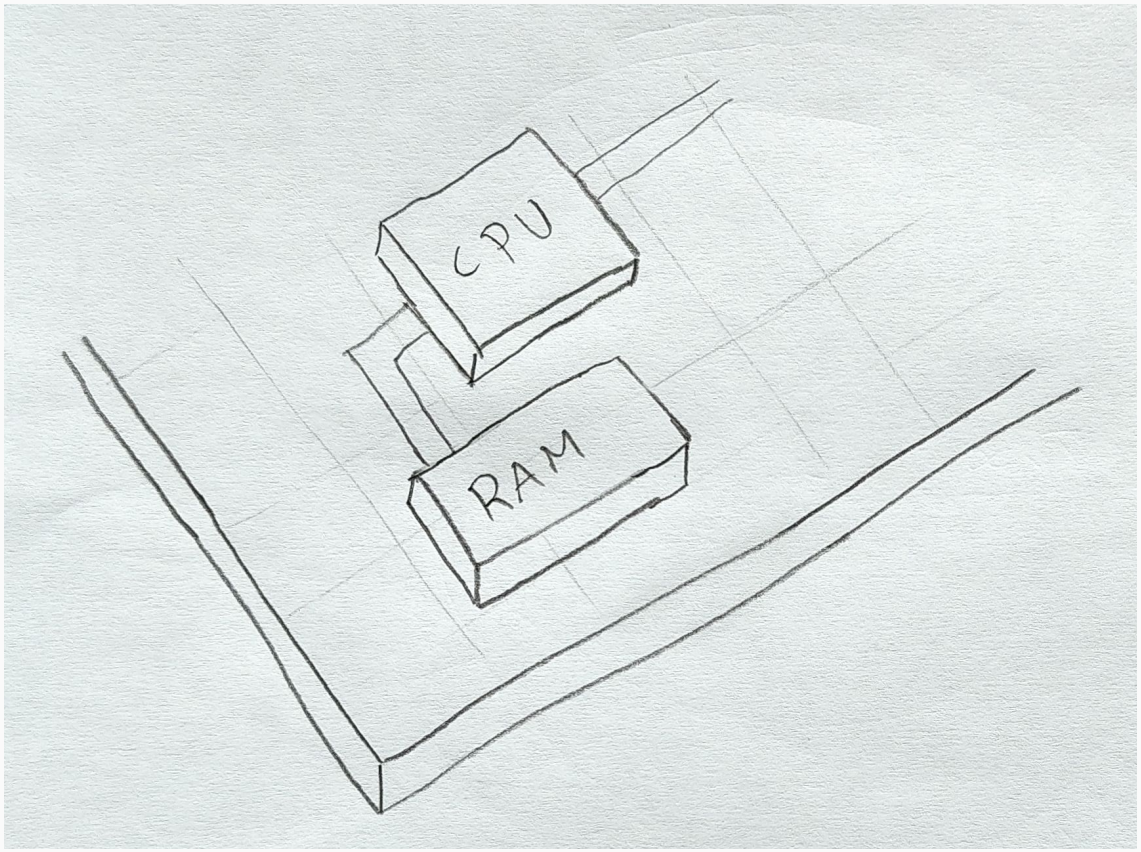
Computers Are Normally Quite 2-Dimensional

Computers are very 2-dimensional. At the level of the circut boards, the room, the buildings, distributed over the entire globe.



Computers Are Normally Quite 2-Dimensional

Computers are very 2-dimensional. At the level of the circut boards, the room, the buildings, distributed over the entire globe.



But space is 3-dimensional, right?

So How Much is Physically Possible?

The *Bekenstein Bound* is an upper bound on entropy (S) within a volume enclosed by a radius (R) containing mass-energy E.

$$S \leq \frac{2\pi kRE}{\hbar c}$$

Entropy is basically the amount of information, in bits, that exists in the system.

So How Much is Physically Possible?

The *Bekenstein Bound* is an upper bound on entropy (S) within a volume enclosed by a radius (R) containing mass-energy E.

$$S \leq \frac{2\pi kRE}{\hbar c}$$

Entropy is basically the amount of information, in bits, that exists in the system.

These are just some constants: $2, \pi, k, \hbar, c$, so

$$S \propto RE$$

So How Much is Physically Possible?

The *Bekenstein Bound* is an upper bound on entropy (S) within a volume enclosed by a radius (R) containing mass-energy E.

$$S \leq \frac{2\pi kRE}{\hbar c}$$

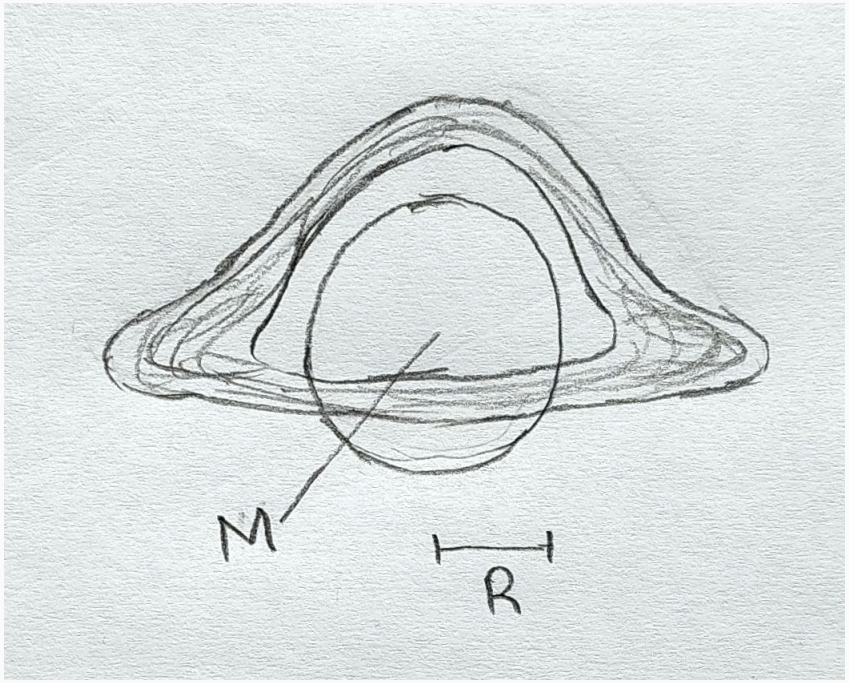
Entropy is basically the amount of information, in bits, that exists in the system.

These are just some constants: $2, \pi, k, \hbar, c$, so

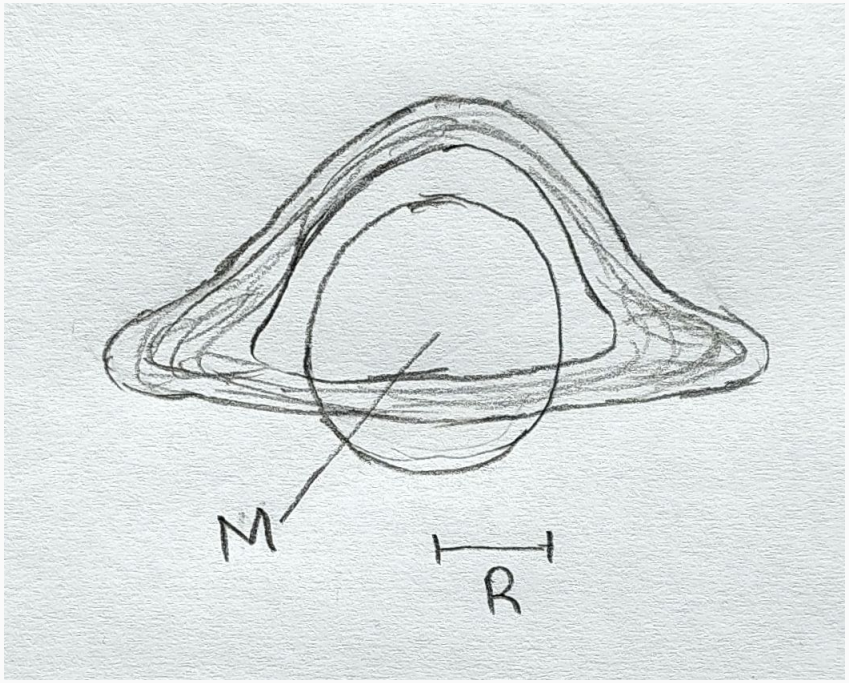
$$S \propto RE$$

What is the upper bound on mass-energy in a given radius?

A Black Hole



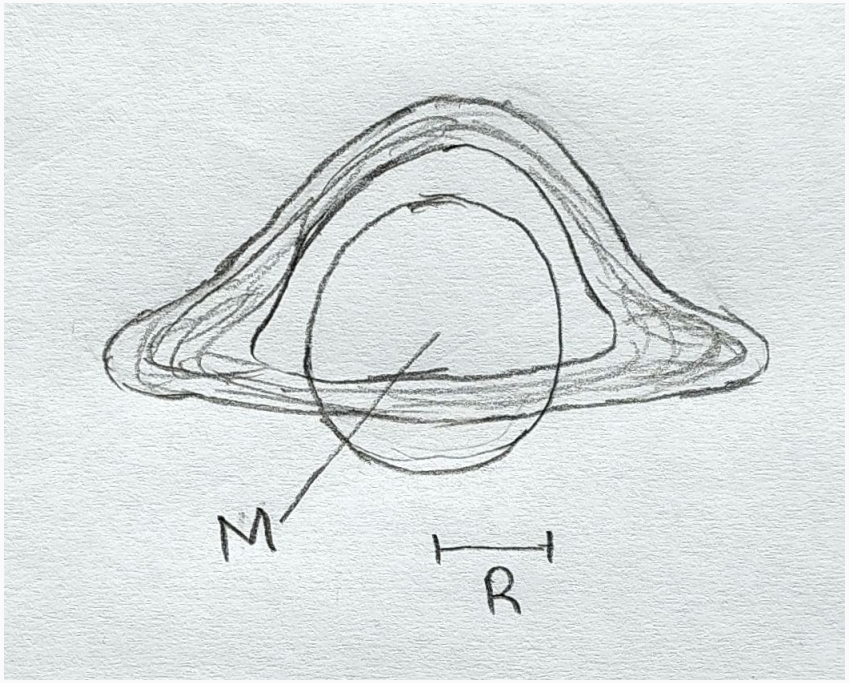
A Black Hole



The *Schwarzchild radius* is the radius defining the event horizon of a black hole.

$$R_s = \frac{2GM}{c^2}$$

A Black Hole



The *Schwarzchild radius* is the radius defining the event horizon of a black hole.

$$R_s = \frac{2GM}{c^2}$$

Again, some constants: $2, G, c$, so

$$R_s \propto M \quad \text{and thus} \quad M \propto R_s$$

Upper Bound on Entropy

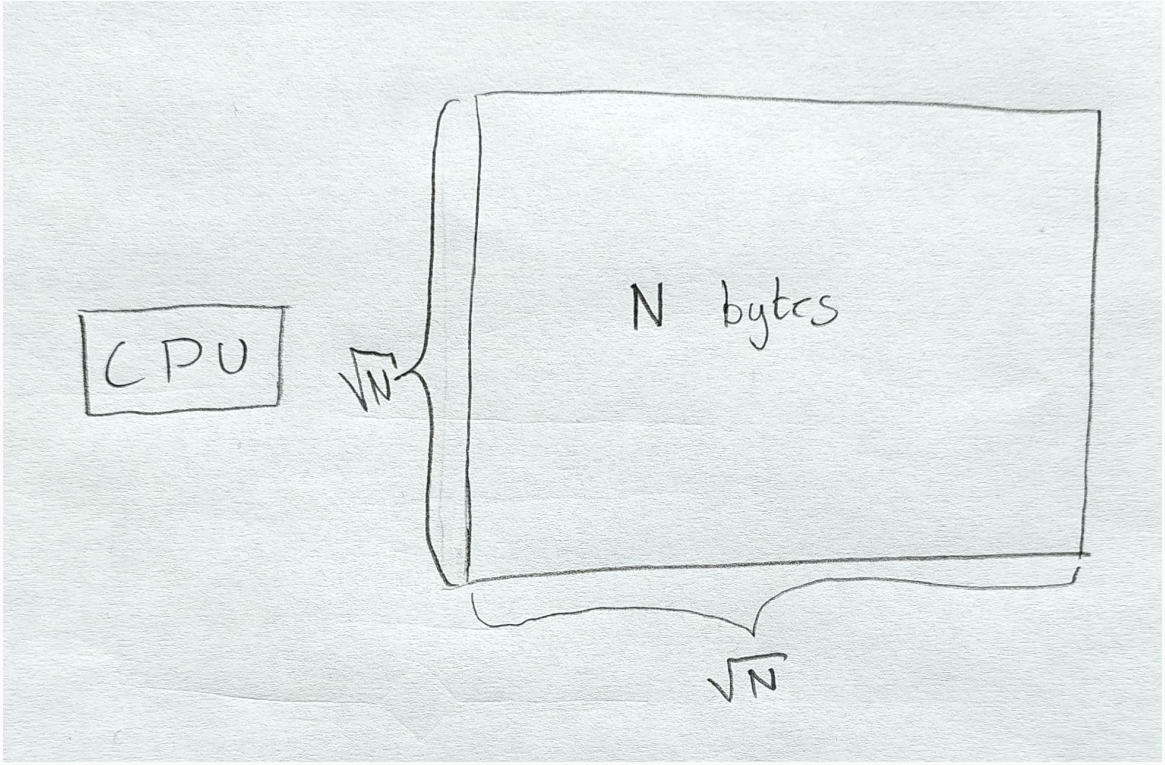
By applying this in the Bekenstein bound, we get

$$S \propto R^2.$$

Thus, the maximum amount of information in a sphere of radius R is proportional to R^2.

Memory Access is $\mathcal{O}(\sqrt{N})$

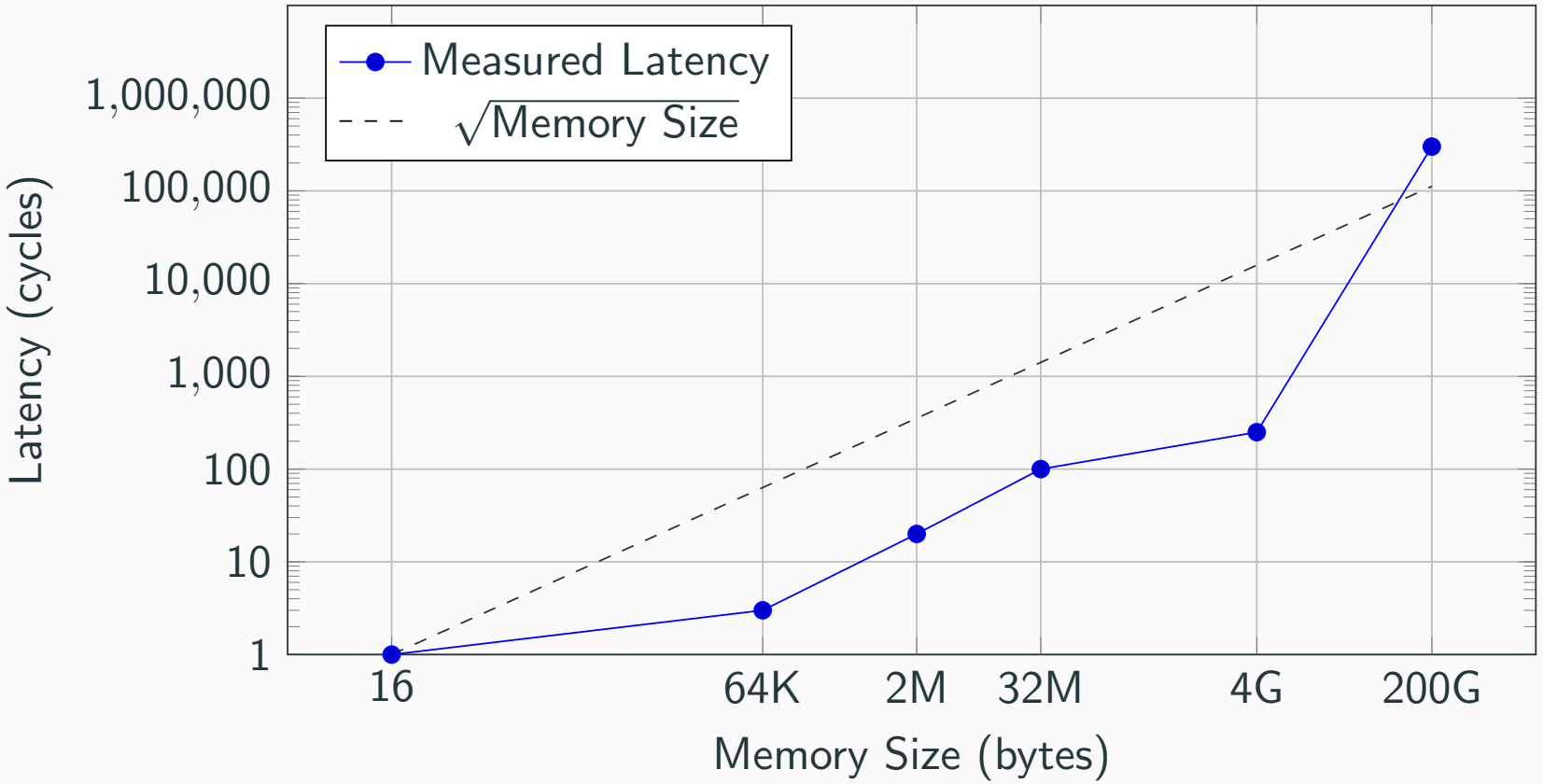
Since we are limited by the speed of light, the time to to extract data from N bits is proportional to the radius, and thus we will need $\mathcal{O}(\sqrt{N})$ time.



$$\text{access time} \geq cs\sqrt{N}$$

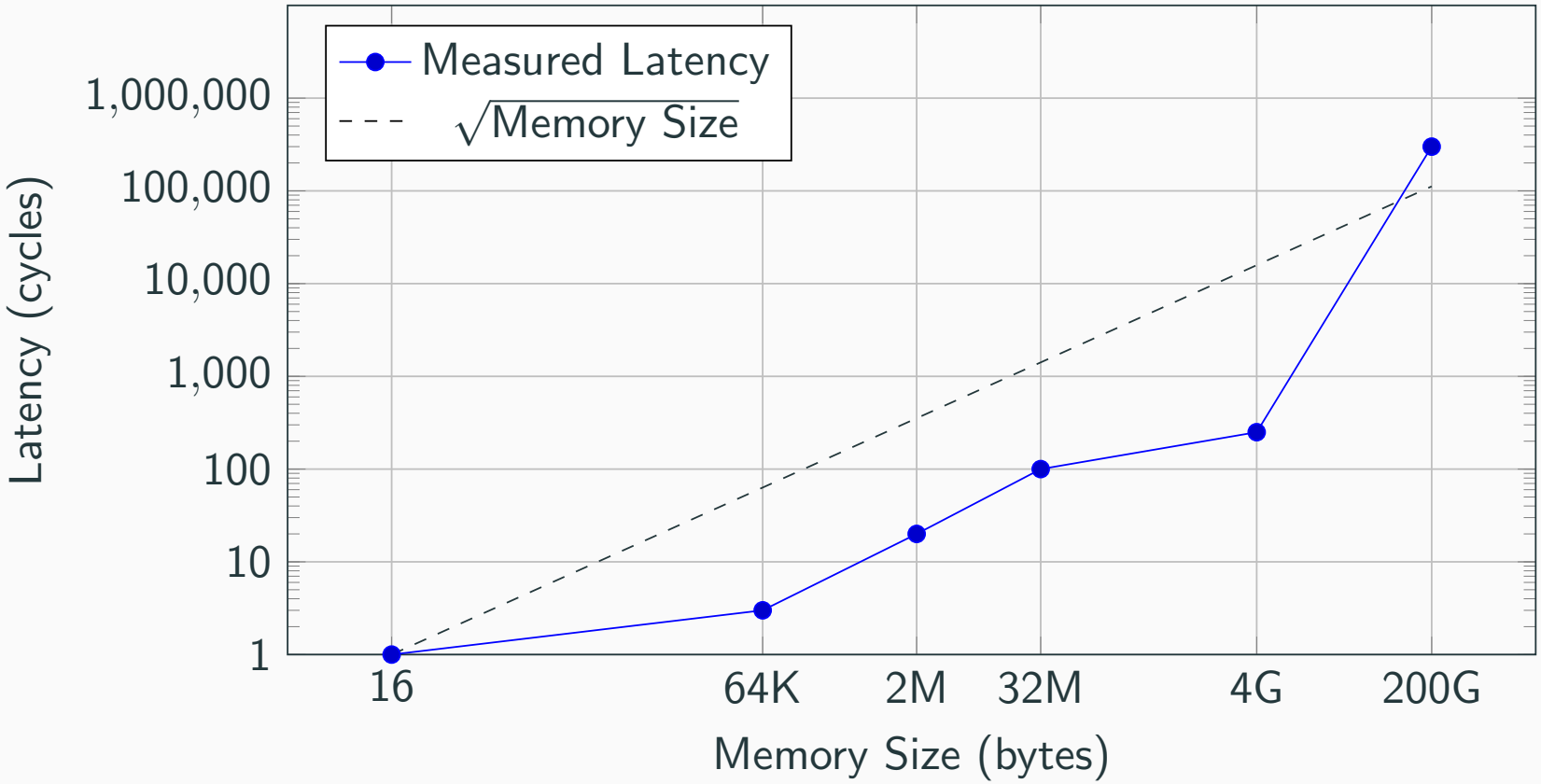
Does this match empirical data?

How does this fit into the graph we saw earlier?



Does this match empirical data?

How does this fit into the graph we saw earlier?



What does this mean for asymptotic analysis of algorithms?

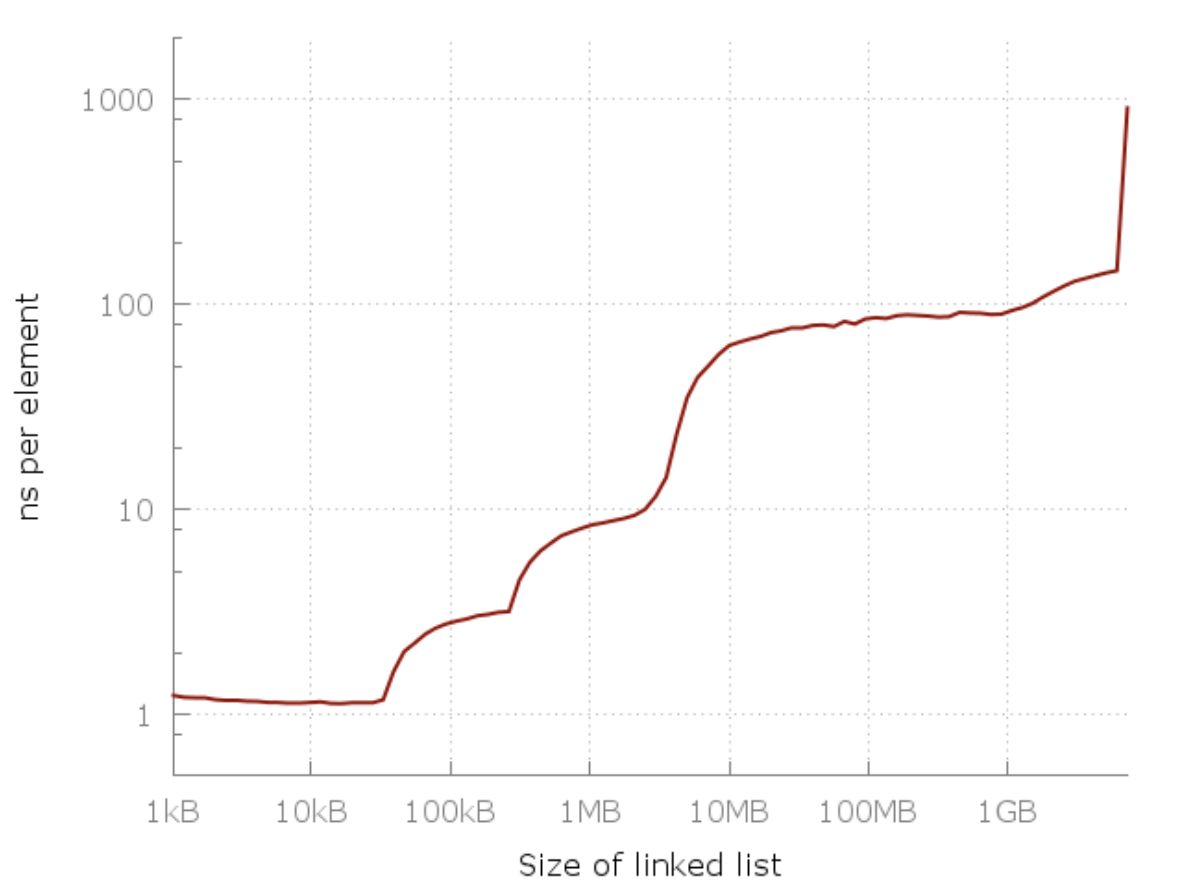
Applying The New Model

Time-Complexity of Search in Linked List

Traditionally, searching a linked list is $\Theta(N)$ time-complexity. However, all lookups are random access. Thus, the complexity in the new model, is actually $\Theta(N\sqrt{N})$.

Time-Complexity of Search in Linked List

Traditionally, searching a linked list is $\Theta(N)$ time-complexity. However, all lookups are random access. Thus, the complexity in the new model, is actually $\Theta(N\sqrt{N})$.



Time-Complexity of Bubble Sort

Assume all memory access is random. Notice, not a tight bound.

Algorithm 1: Bubble Sort

Input : Array A of length N

Output: Sorted array A in ascending order

```
1 for i ← 0 to N − 1 do                                  // O(√(log N))
2     for j ← 0 to N − i − 2 do                          // O(√(log N))
3         if A[j] > A[j + 1] then                        // O(√N)
4             Swap A[j] and A[j + 1]                     // O(√N)
```

Time-Complexity of Bubble Sort

Assume all memory access is random. Notice, not a tight bound.

Algorithm 2: Bubble Sort

Input : Array A of length N

Output: Sorted array A in ascending order

```
1 for i ← 0 to N − 1 do                          // O(√(log N))
2     for j ← 0 to N − i − 2 do                  // O(√(log N))
3         if A[j] > A[j + 1] then                // O(√N)
4             Swap A[j] and A[j + 1]             // O(√N)
```

Time-complexity

$$\mathcal{O}(N) \cdot \left(\mathcal{O}(\sqrt{\log N}) + \mathcal{O}(N) \cdot \left(\mathcal{O}(\sqrt{\log N}) + \mathcal{O}(\sqrt{N}) + \mathcal{O}(\sqrt{N})\right)\right)$$
$$= \mathcal{O}(N^2\sqrt{(N)})$$

Time-Complexity of Bubble Sort

Assume all memory access is random. Notice, not a tight bound.

Algorithm 3: Bubble Sort

Input : Array A of length N

Output: Sorted array A in ascending order

```
1 for i ← 0 to N − 1 do                          // O(√(log N))
2     for j ← 0 to N − i − 2 do                  // O(√(log N))
3         if A[j] > A[j + 1] then                // O(√N)
4             Swap A[j] and A[j + 1]             // O(√N)
```

Time-complexity

$$\mathcal{O}(N) \cdot \left(\mathcal{O}(\sqrt{\log N}) + \mathcal{O}(N) \cdot \left(\mathcal{O}(\sqrt{\log N}) + \mathcal{O}(\sqrt{N}) + \mathcal{O}(\sqrt{N})\right)\right)$$
$$= \mathcal{O}(N^2\sqrt{(N)})$$

Normally: $\Theta(N^2)$

When does this matter?

If we assume all memory access is random, this basically means:

Let algorithm have normal time-complexity $\Theta(t(n))$ and space-complexity (including input) $\Theta(s(n))$. The new time-complexity is $\mathcal{O}(t(n) \cdot \sqrt{s(n)})$

When does this matter?

If we assume all memory access is random, this basically means:

Let algorithm have normal time-complexity $\Theta(t(n))$ and space-complexity (including input) $\Theta(s(n))$. The new time-complexity is $\mathcal{O}(t(n) \cdot \sqrt{s(n)})$

	Normal Time Complexity	**Space Complexity**	**New Time Complexity**
Alg. 1	$\Theta(n^2)$	$\Theta(n^2)$	$\mathcal{O}(n^3)$
Alg. 2	$\Theta(n^2 \log n)$	$\Theta(n)$	$\mathcal{O}(n^2 \sqrt{n} \log n)$

So Does This Help With Considering Cache?

Not really...

So Does This Help With Considering Cache?

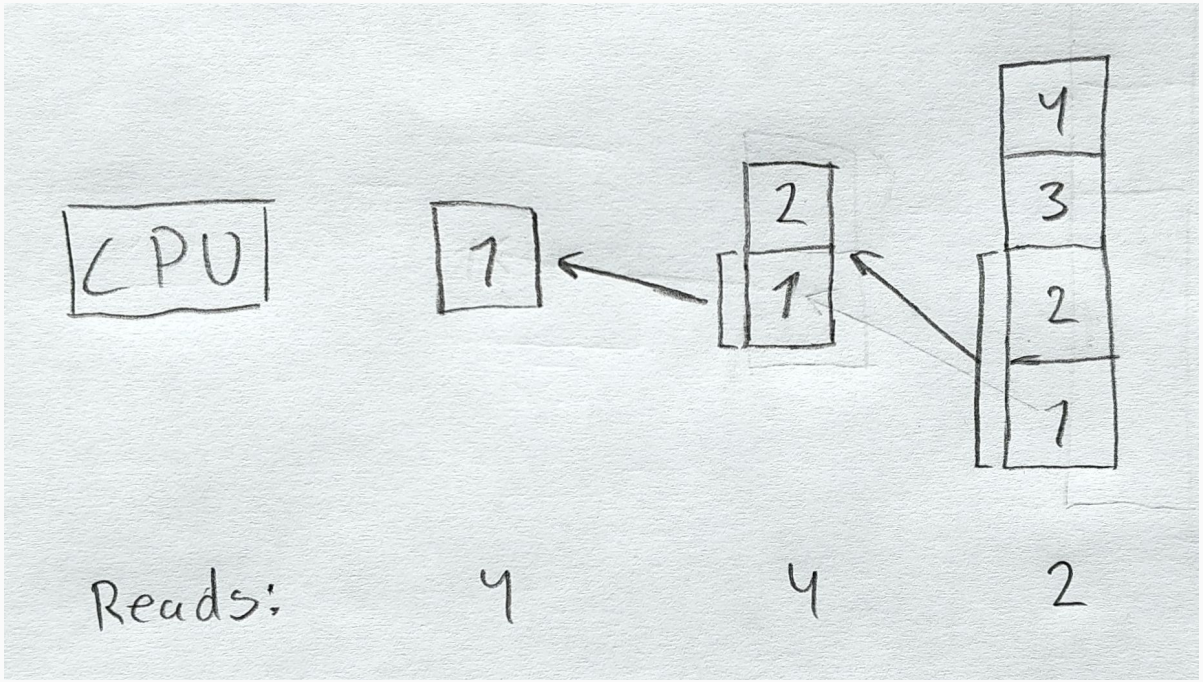
Not really...

Assuming all memory access is random gives us an upper bound, but it does not help us distinguish proper cache usage.

Not All Memory Access is Random

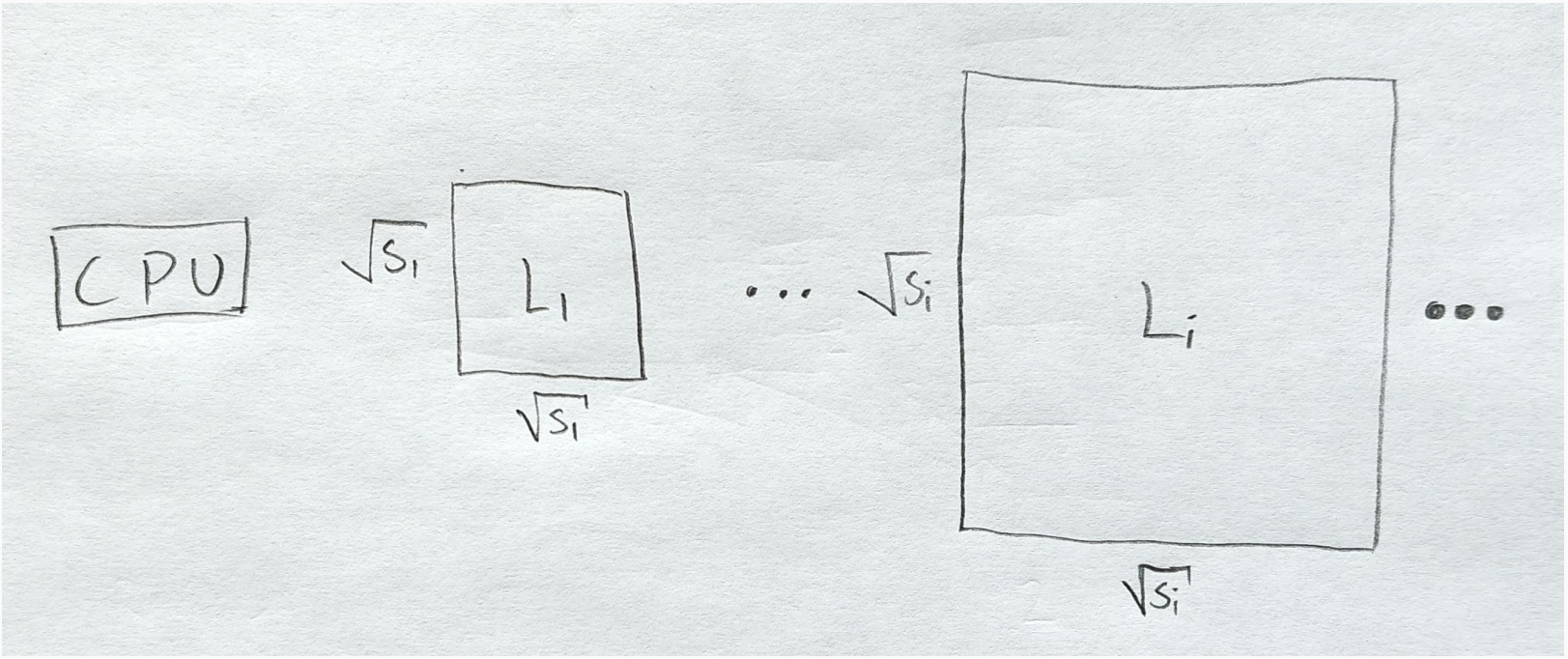
We know that not all memory access is random. Let us consider linear search through an array of N elements.

How can we analyze the complexity of this algorithm?



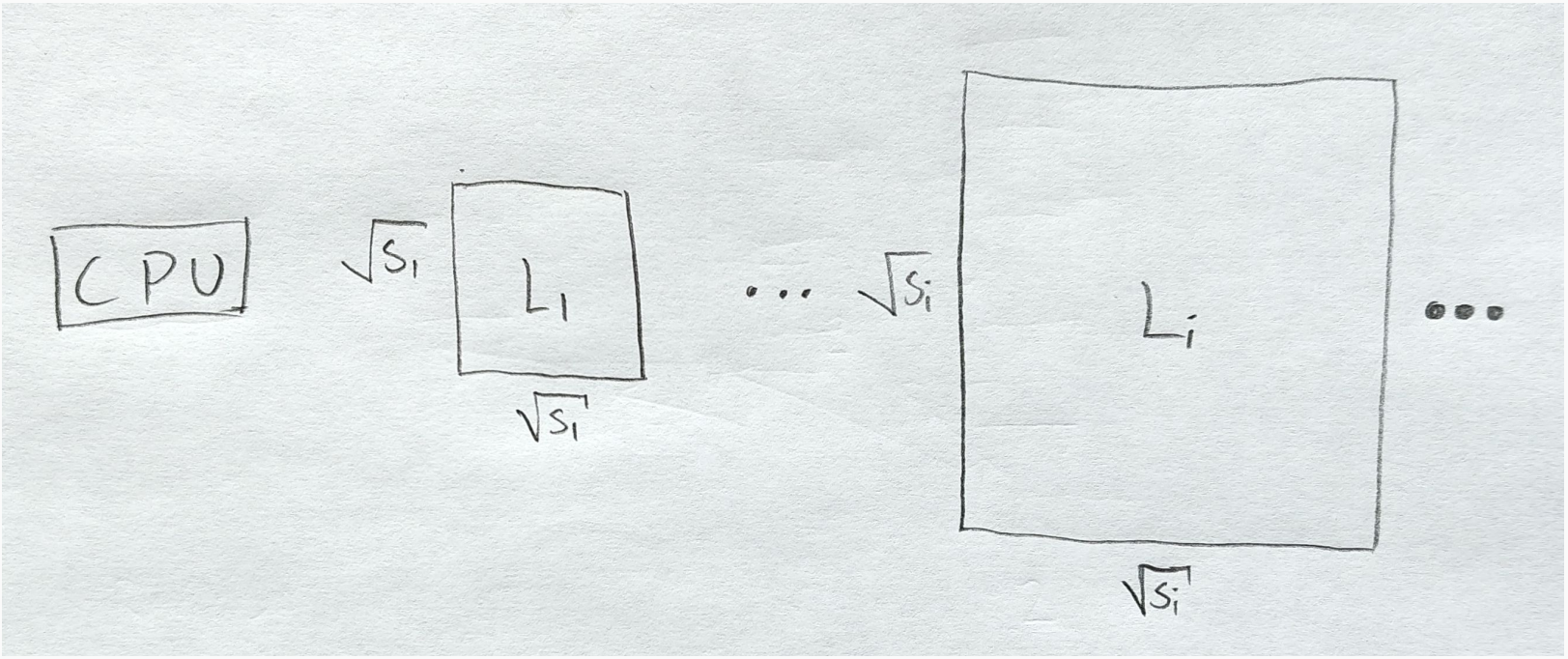
Generalizing Cache Layout

Let S_i be the size of the i'th cache L_i. Let us assume that filling L_i with S_i values of L_{i+1} can be done in time $\Theta(\sqrt{S_{i+1}})$.



Generalizing Cache Layout

Let S_i be the size of the i'th cache L_i. Let us assume that filling L_i with S_i values of L_{i+1} can be done in time $\Theta(\sqrt{S_{i+1}})$.



How big should L_i be?

Memory Accesses to Caches

In linear search, for each unique chunk in L_i, L_{i+1} will read it $\lceil \frac{S_i}{S_{i-1}} \rceil$ times. There are $\lceil \frac{N}{S_i} \rceil$ unique chunks. Thus, L_i receives $\lceil \frac{S_i}{S_{i-1}} \rceil \lceil \frac{N}{S_i} \rceil \approx \frac{N}{S_{i-1}}$ reads. Each has the complexity $\Theta(\sqrt{S_i})$.

Memory Accesses to Caches

In linear search, for each unique chunk in L_i, L_{i+1} will read it $\lceil \frac{S_i}{S_{i-1}} \rceil$ times. There are $\lceil \frac{N}{S_i} \rceil$ unique chunks. Thus, L_i receives $\lceil \frac{S_i}{S_{i-1}} \rceil \lceil \frac{N}{S_i} \rceil \approx \frac{N}{S_{i-1}}$ reads. Each has the complexity $\Theta(\sqrt{S_i})$.

The total number of reads in an N array will be

$$\sum_{i=1}^{k} \frac{N}{S_{i-1}} \Theta(\sqrt{S_i}),$$

where $k = \min\{i \in \mathbb{N} \mid N \leq S_i\}$.

Which Caching Size Strategy to Use

Let us try with $S_i = i$. This yields

$$\sum_{i=2}^{N} \frac{N}{i-1}\Theta(\sqrt{i}) = N\sum_{i=2}^{N}\Theta\left(\frac{1}{\sqrt{i}}\right) = \Theta(N\sqrt{N}).$$

Which Caching Size Strategy to Use

Let us try with $S_i = i$. This yields

$$\sum_{i=2}^{N} \frac{N}{i-1}\Theta(\sqrt{i}) = N\sum_{i=2}^{N}\Theta\left(\frac{1}{\sqrt{i}}\right) = \Theta(N\sqrt{N}).$$

What about $S_i = 2^i$?

$$\sum_{i=1}^{\log N} \frac{N}{2^{i-1}}\Theta(\sqrt{2^i}) = N\underbrace{\sum_{i=1}^{\log N}\frac{\Theta(\sqrt{2^i})}{2^{i-1}}}_{\text{bounded}} = \Theta(N)$$

Which Caching Size Strategy to Use

Let us try with $S_i = i$. This yields

$$\sum_{i=2}^{N} \frac{N}{i-1}\Theta(\sqrt{i}) = N\sum_{i=2}^{N}\Theta\left(\frac{1}{\sqrt{i}}\right) = \Theta(N\sqrt{N}).$$

What about $S_i = 2^i$?

$$\sum_{i=1}^{\log N} \frac{N}{2^{i-1}}\Theta(\sqrt{2^i}) = N\underbrace{\sum_{i=1}^{\log N}\frac{\Theta(\sqrt{2^i})}{2^{i-1}}}_{\text{bounded}} = \Theta(N)$$

Cool! Assuming exponentially increasing cache sizes, optimal locality usage is actually (amortized) constant access.

Merge Sort Vs. Heap Sort

Merge sort uses optimal memory locality, thus $\Theta(n \log n)$. Heap sort is mostly random access, thus $\Theta(n \log n \sqrt{n})$.

Conclusion

- Due to physics, random access is $\Theta(\sqrt{N})$
- With careful analysis, we can get cache-aware time-complexity
- Optimal locality usage is amortized constant time