# CPSC 313: Computer Hardware and Operating Systems

Unit 3: The Memory Hierarchy
From Arrays to Strided Access

# Administration

- Quiz 3 is next week (sign up for Quiz 3, viewing, and retake times!)
- Lab 6: due shortly
- Lab 7: out Friday (not as big as Lab 5, but still substantial)
- Tutorial 6: this week
- Drop with W deadline on Friday: We want you in the class! (But... if you're certain for some reason that you want to drop, don't miss the deadline!)
- Poll: What's your opinion on the retakes? https://piazza.com/class/m0cup2q4ff54h3/post/671

- **As always: Check the syllabus for details and deadlines!**

# Today

- Learning Outcomes
  - Define row-major/column-major layout
  - Differentiate access patterns -- what C constructs will produce sequential access; what C constructs will produce strided access.
  - Evaluate cache performance in the presence of strided access.

- All code for today is in course repo … go there now …
  - https://github.students.cs.ubc.ca/cpsc313/CPSC313-2023w2/tree/master/3.6-strided-performance/Inclass

# Spatial Locality and Clustering

- What aspect of cache design takes advantage of spatial locality?
- Under what conditions is this effective?
- What does this say about how data should be ***clustered*** in memory
  - i.e., what data should be next to what?
- Examples - good and bad?

# Spatial Locality and Clustering

- What aspect of cache design takes advantage of spatial locality?
  - Cache lines: Move data into caches in units larger than a single access
- Under what conditions is this effective?
  - When we access more than one data item per line.
- What does this say about how data should be ***clustered*** in memory
  - i.e., what data should be next to what?
  - Make sure data accessed together is in the same cacheline (if possible)
- Examples - good and bad?
  - Good: Iterating in sequence over an array.
  - Bad: Random array accesses and some pointer data structures (lists/trees/...).

# Consider this Code … Spatial Locality?

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        struct person {
                char name[20];
                int age;
                char address[20];
                char city[20];
                char postal_code[6];
                char province[2];
        };

        const int NUM_PEOPLE = 100;
        struct person people[NUM_PEOPLE];

        /* Find the youngest person. */
        int youngest_ndx = -1;
        int youngest_age = 1000;
        for (int i = 0; i < NUM_PEOPLE; i++) {
                if (people[i].age < youngest_age) {
                        youngest_ndx = i;
                        youngest_age = people[i].age;
                }
        }
}
```

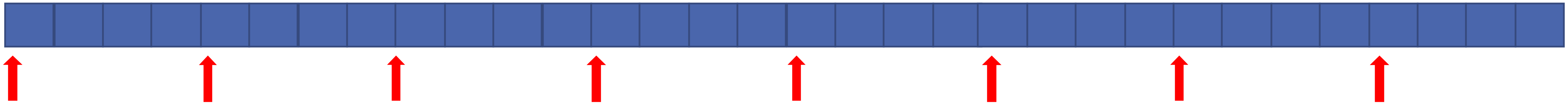
See ./stride_struct in the code handout

# Stride: Distance Between Elements Accessed Consecutively

Measured in elements or bytes. For example, in an array of 4-byte items:

- Stride of 1 element is a stride of 4 bytes, and accesses all elements

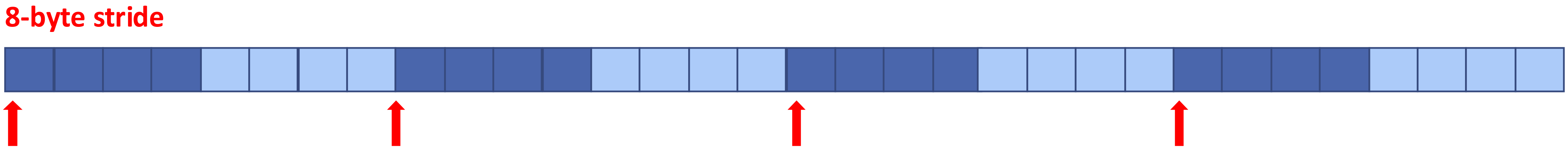
**4-byte stride**



# Stride: Distance Between Elements Accessed Consecutively

Measured in elements or bytes. For example, in an array of 4-byte items:

- Stride of 1 element is a stride of 4 bytes, and accesses all elements
- Stride of 2 elements is a stride of 8 bytes, and accesses every-other element

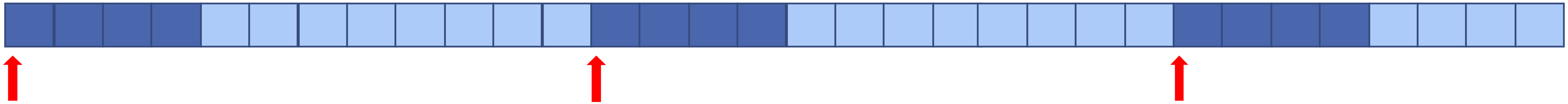


# Stride: Distance Between Elements Accessed Consecutively

Measured in elements or bytes. For example, in an array of 4-byte items:

- Stride of 1 element is a stride of 4 bytes, and accesses all elements
- Stride of 2 elements is a stride of 8 bytes, and accesses every-other element
- Stride of 3 elements is a stride of 12 bytes, and accesses every-third element

**12-byte stride**



# Strided Access, Real Example

The data – a stream of time-stamped data

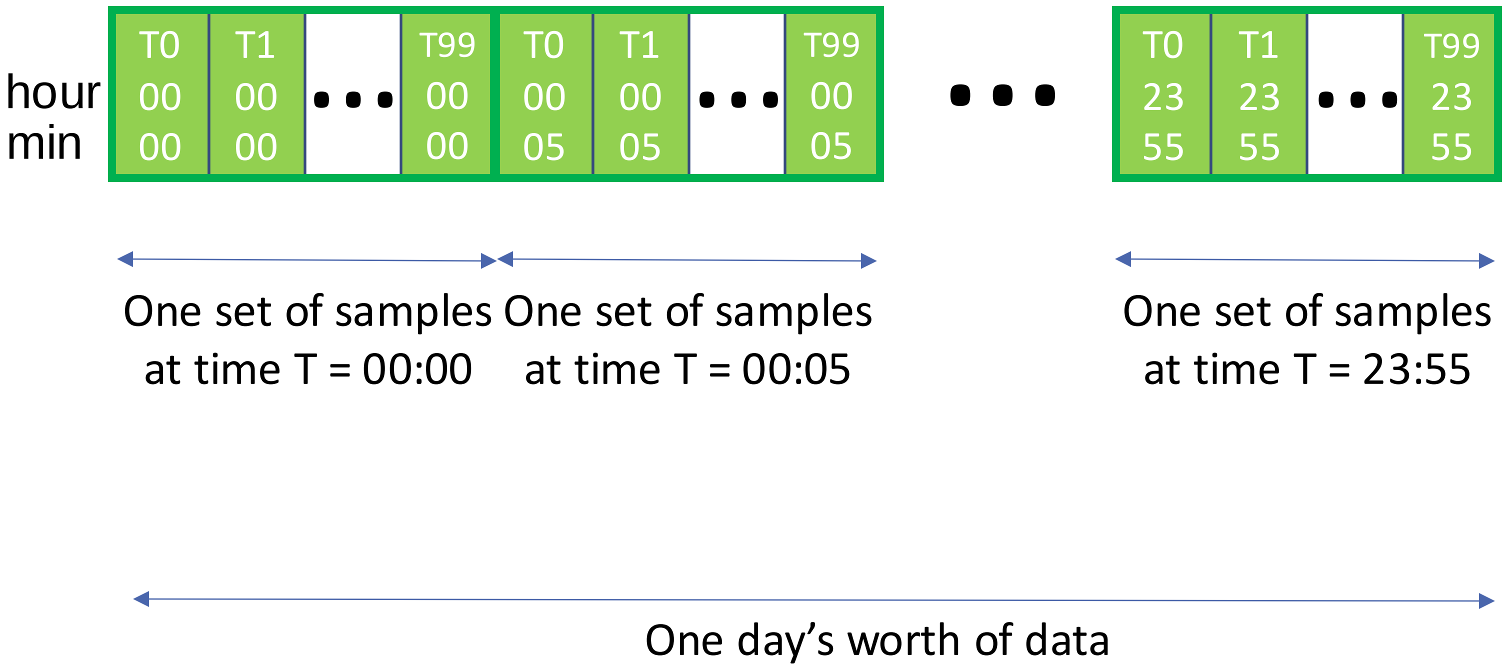
- 100 trees monitored with hydrological sensors.
- Each sensor produces one double value (8 bytes) every 5 minutes.
- We have one year's worth of data (~ 80MB)
- Stored in order it is received (i.e., clustered by time)

Two ecological studies:

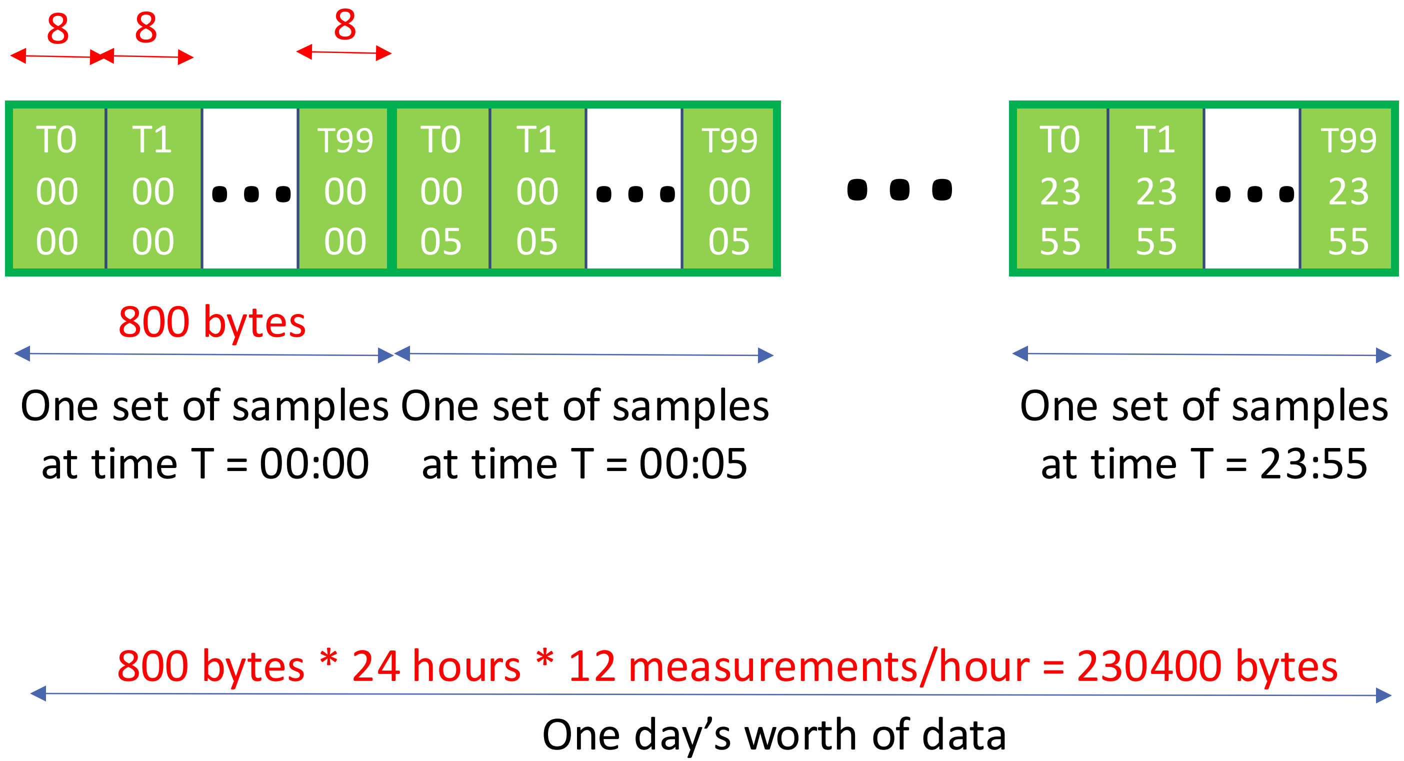
- Linh analyzes all the data for a particular tree.
- Aaron analyzes all the data for 9:00-10:00 AM every morning.

What is the stride for each experiment?

# The Ecology Data Layout (1)



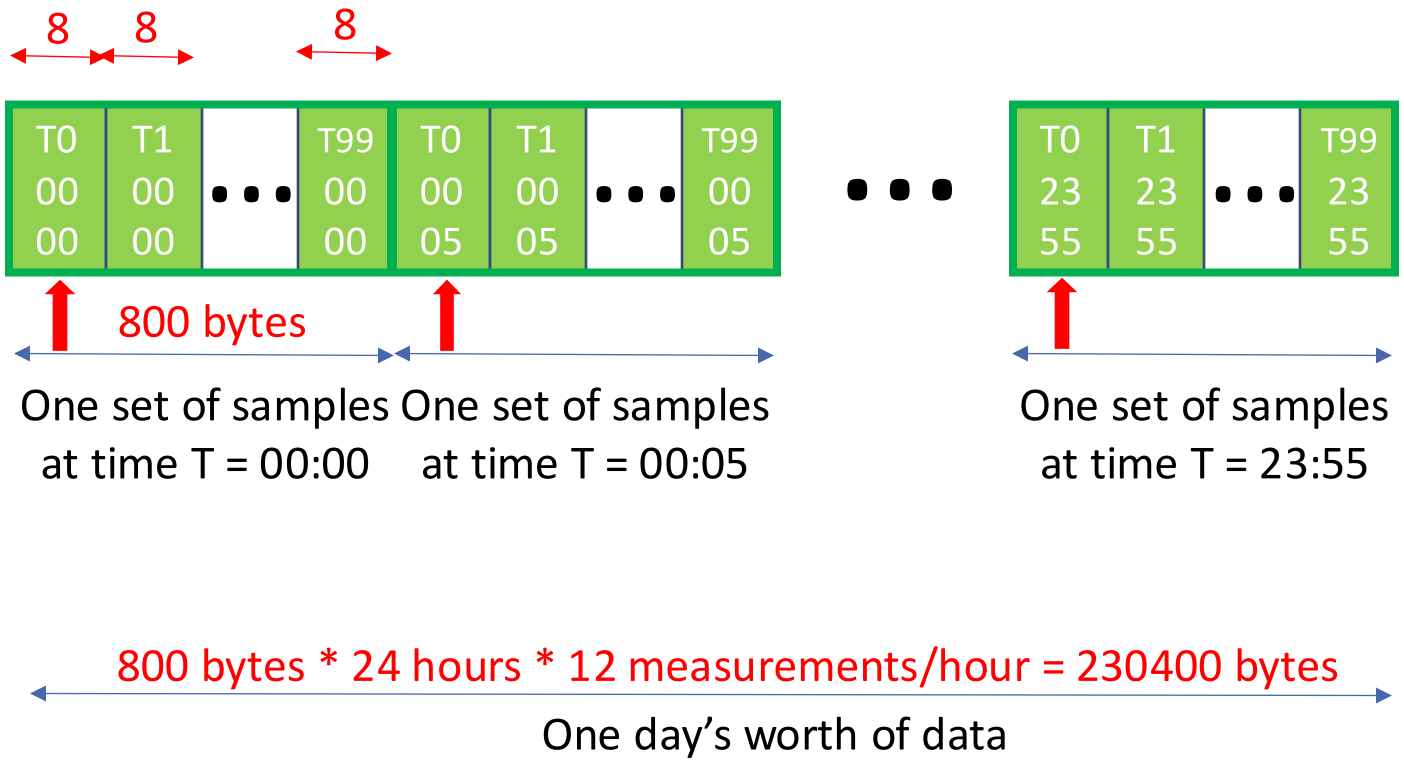
# The Ecology Data Layout (2)



# Linh's Access Pattern

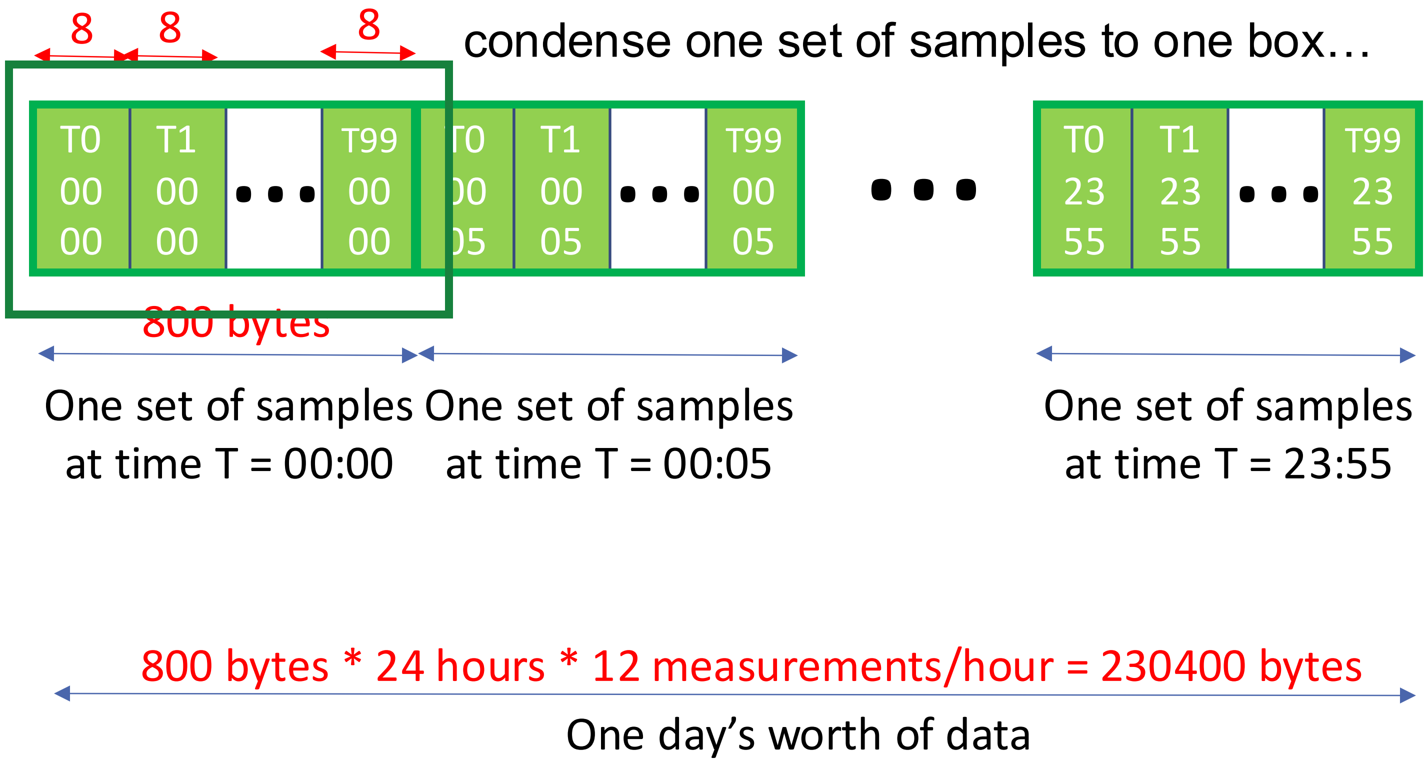
800-byte stride

All the data for a particular tree

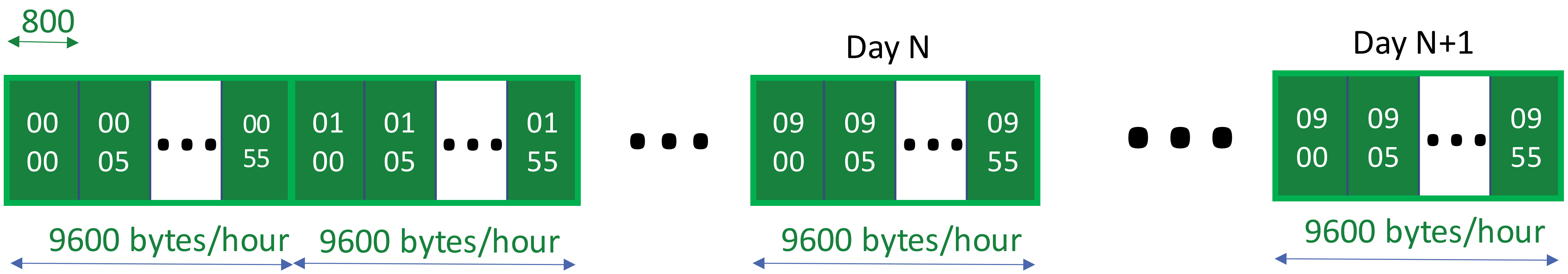


# Aaron's Access Pattern

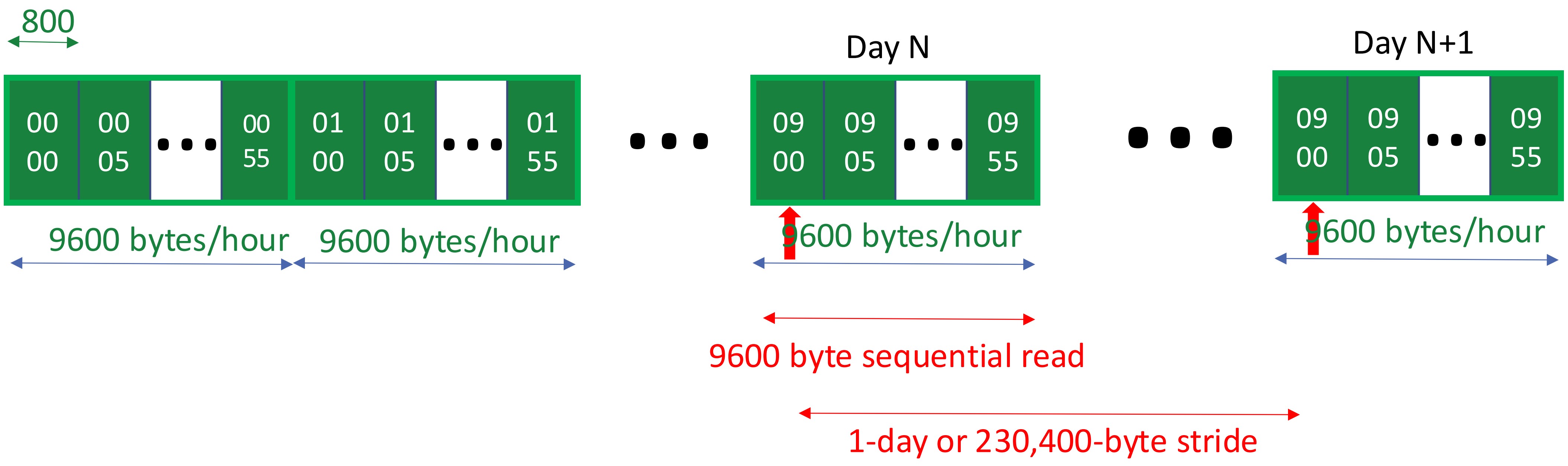
All the data from 9:00-10:00 AM every morning



# Aaron’s Access Pattern



# Aaron’s Access Pattern



# Who will read more data?

- Linh:
  - 8 bytes every 5 minutes
  - 8 * 12 = 96 bytes per hour
  - 96 * 24 = 2304 bytes per day
  - 2304 * 365 = 840,960 bytes in total
- Aaron
  - 9600 bytes per day (1 hour’s worth)
  - 9600 * 365 = 3,504,000 bytes in total
- Aaron reads about 4 times as much data *(is it really?)*

**How much data is fetched from the memory?**

Assuming 64-byte cachelines

Aaron accesses 365 x 9600 *consecutive* bytes.
-> 365 x 150 cachelines

Linh accesses 8 bytes with 800 byte stride.
-> Needs to fetch one cacheline per 8 byte access.
-> 365 x 12 * 24 = 365 x 288 cachelines

# Who reads data at a faster rate?

- Should Aaron's run take 4x as long as Linh's? Why or why not?
  - Aaron reads large chunks contiguously
  - Linh read 8-bytes chunks scattered in memory.
  - Aaron's pattern produces better cache behaviour!
    (Hardware will recognize this, and prefetch the cachelines!)

- But.. on my computer: doesn't matter. They both take ~1ms or less. Let's discuss one other point and then try something that does **not** run instantly on my computer.

# Strided Access in Real Life: 2D Arrays

- **Linh supposes:** The problem is using a 1D array. If we switch to a 2D array that stores all samples for a particular time in each row, then all samples for a particular tree are in the same column and performance will get better.

```
double data[time][tree#];
int COL = 42;
for (int row = 0; row < …; row++)
    // operate on data[row][COL];
```

- Do you agree?

# Strided Access in Real Life: 2D Arrays

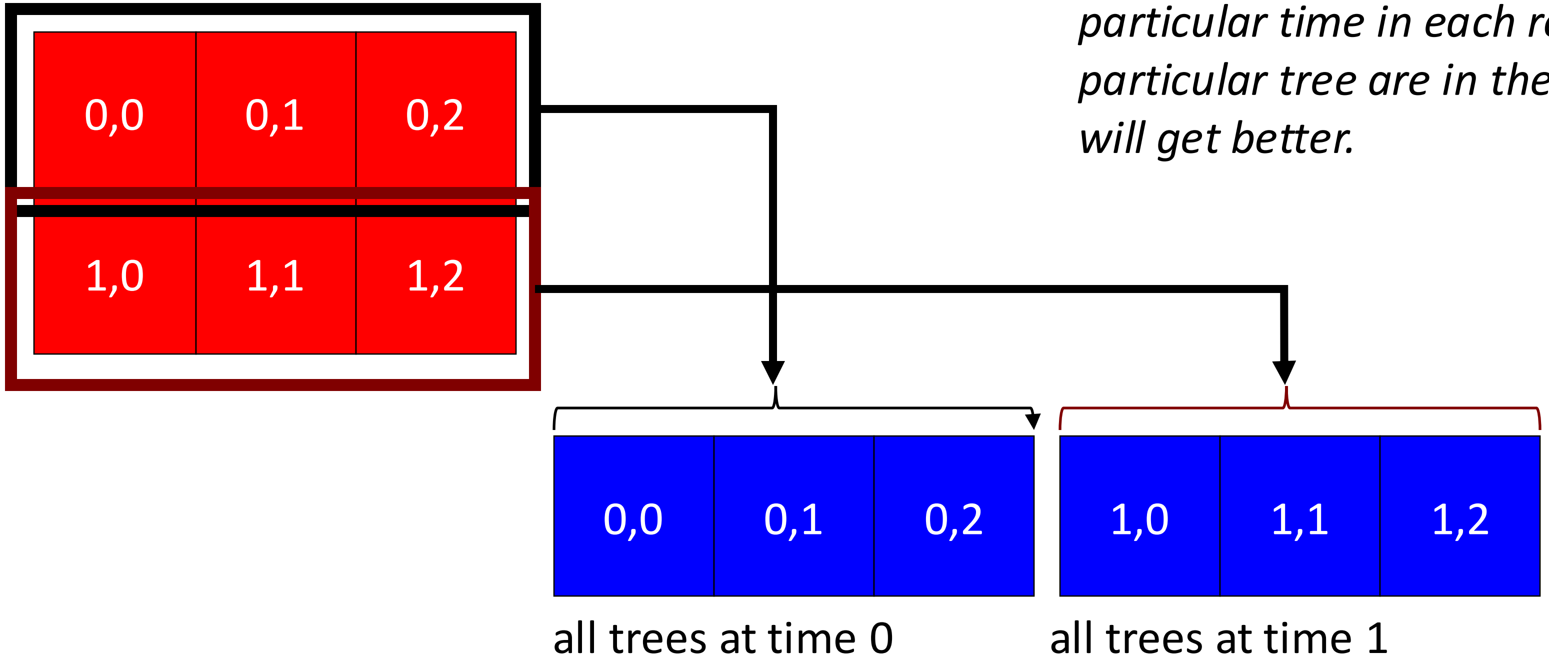
- **Linh supposes:** The problem is using a 1D array. If we switch to a 2D array that stores all samples for a particular time in each row, then all samples for a particular tree are in the same column and performance will get better.

```
double data[time][tree#];
int COL = 42;
for (int row = 0; row < …; row++)
    // operate on data[row][COL];
```

- Do you agree?
  No! Let's look at how 2-D arrays are allocated…

# Array Layout in C: **Row Major Form**

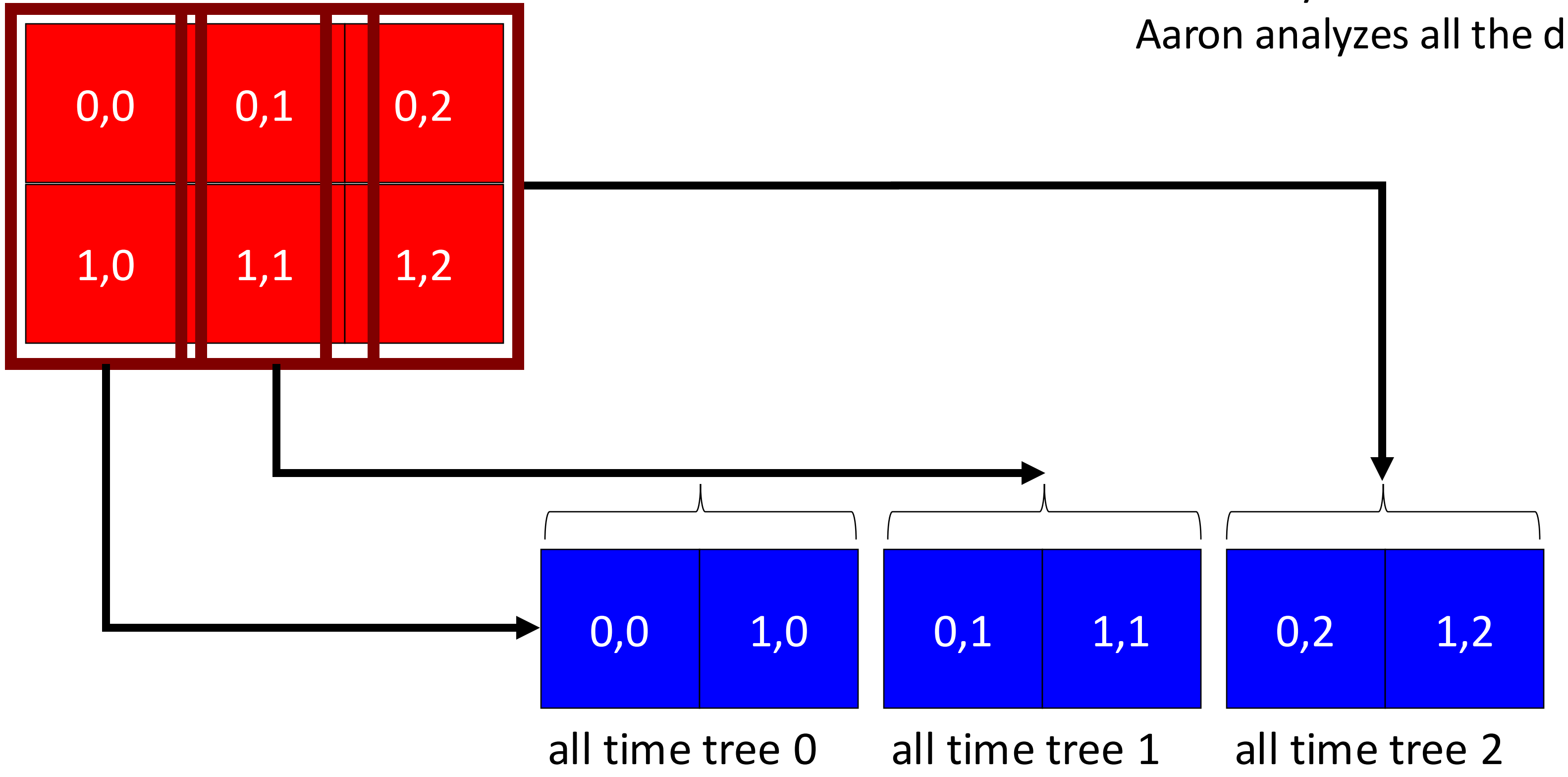
*Linh supposes: The problem is using a 1D array. If we switch to a 2D array that stores all samples for a particular time in each row, then all samples for a particular tree are in the same column and performance will get better.*




```
double data[ROW=time][COL=tree#];
```

The 1D array and the 2D array have identical layouts!

# Column Major Form (*not* what C does):




Better?

Linh analyzes all the data for a particular tree.
Aaron analyzes all the data for 9:00-10:00 AM every morning.

Bottom line

* clustering matters
* no perfect clustering for all workloads

# Determining Cache Parameters Empirically

# Consider the following workload

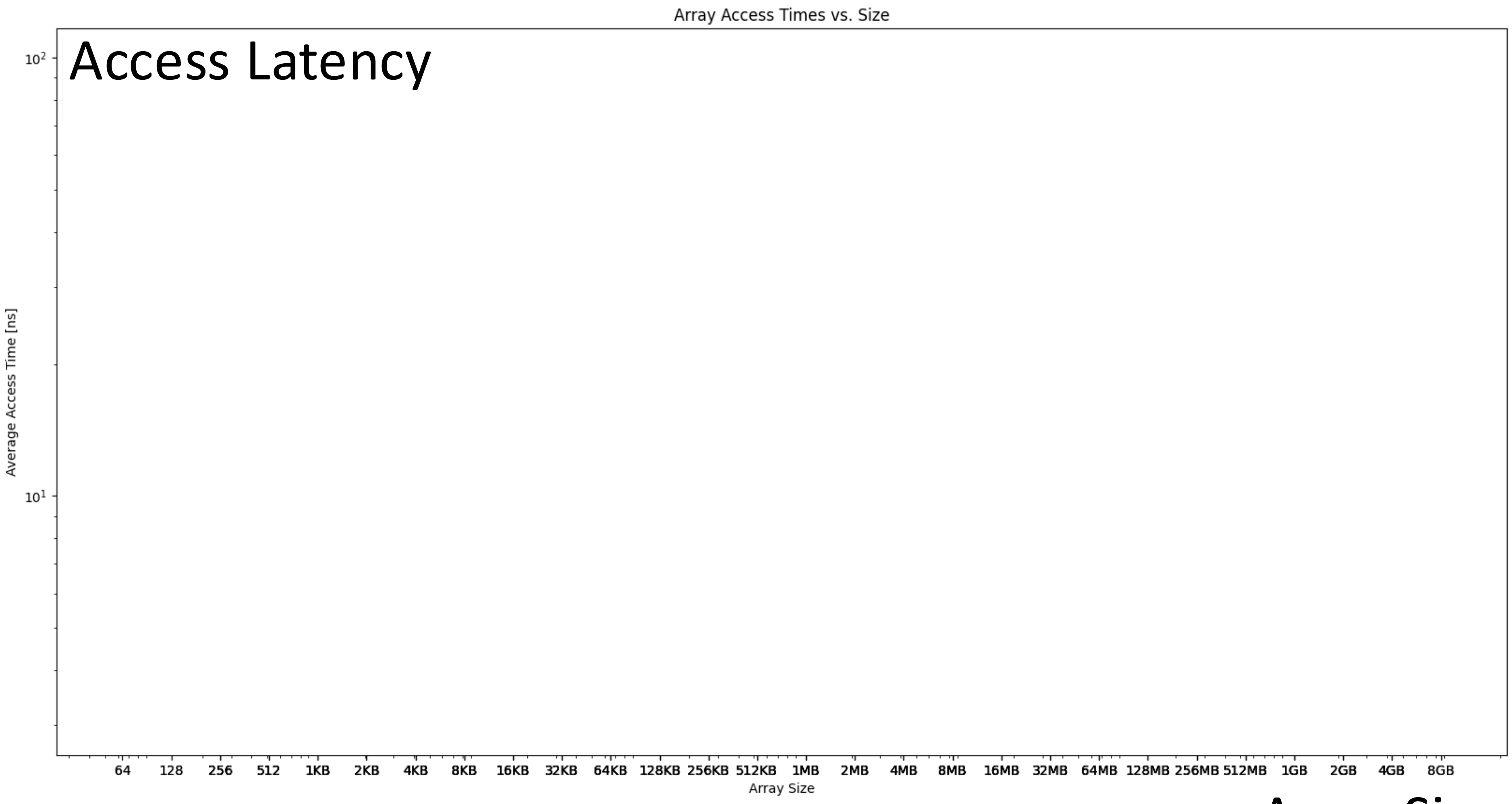
- We have an array of size A.
- We traverse the array sequentially or randomly.
- We can vary two parameters:
  - The **size** of the array (A)
  - The **stride**:
    - Here, we effectively change the padding of the array elements accordingly

```
struct array_type {
  uint64_t next;
  uint64_t pad[PADDING];
};
```

See empirical.c in the handout!

# Test your Understanding

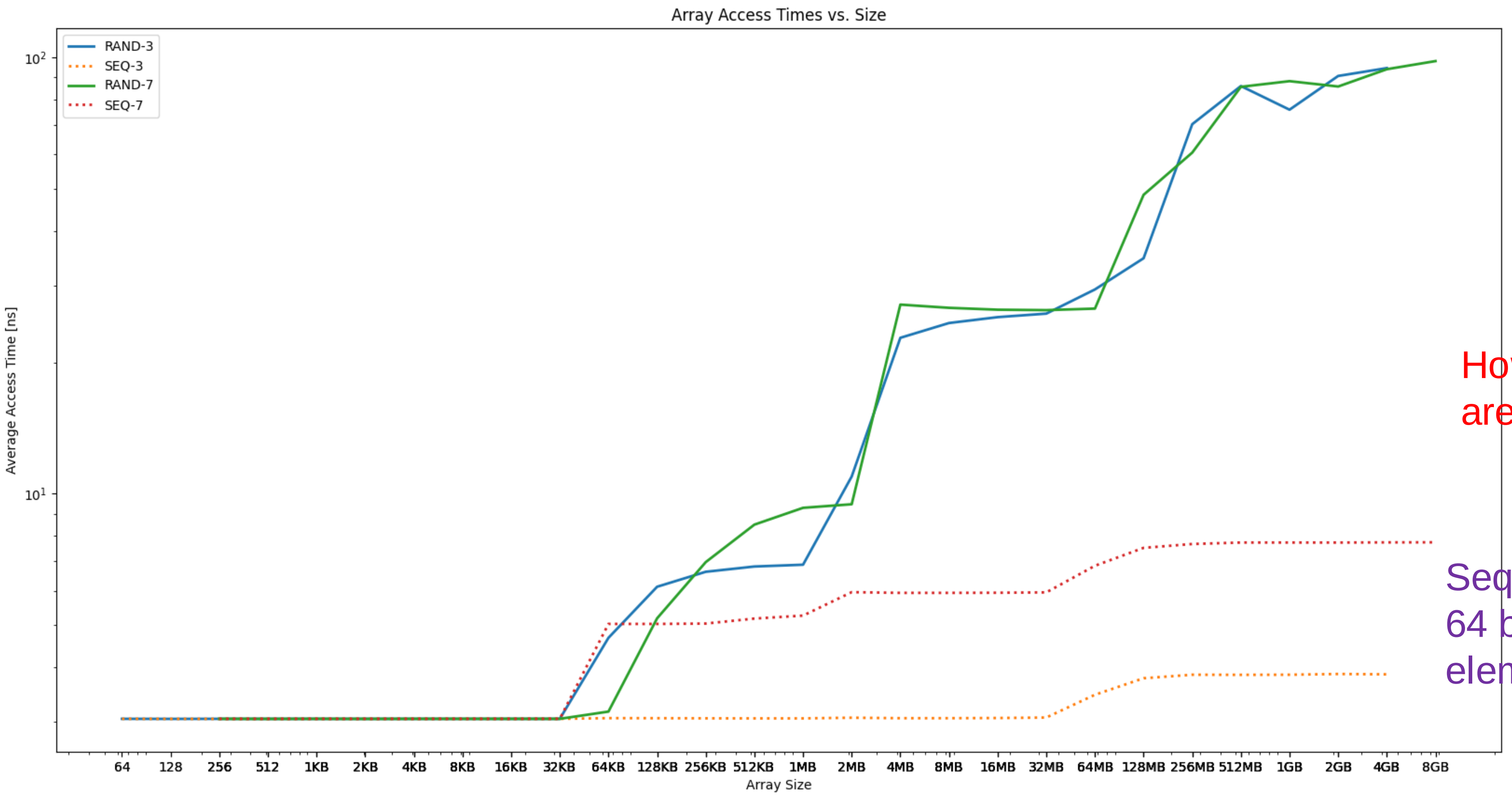
What do you think this graph will look like?




Array Size

Intel Xeon Platinum
8380 CPU @ 2.30GHz

# Test your Understanding



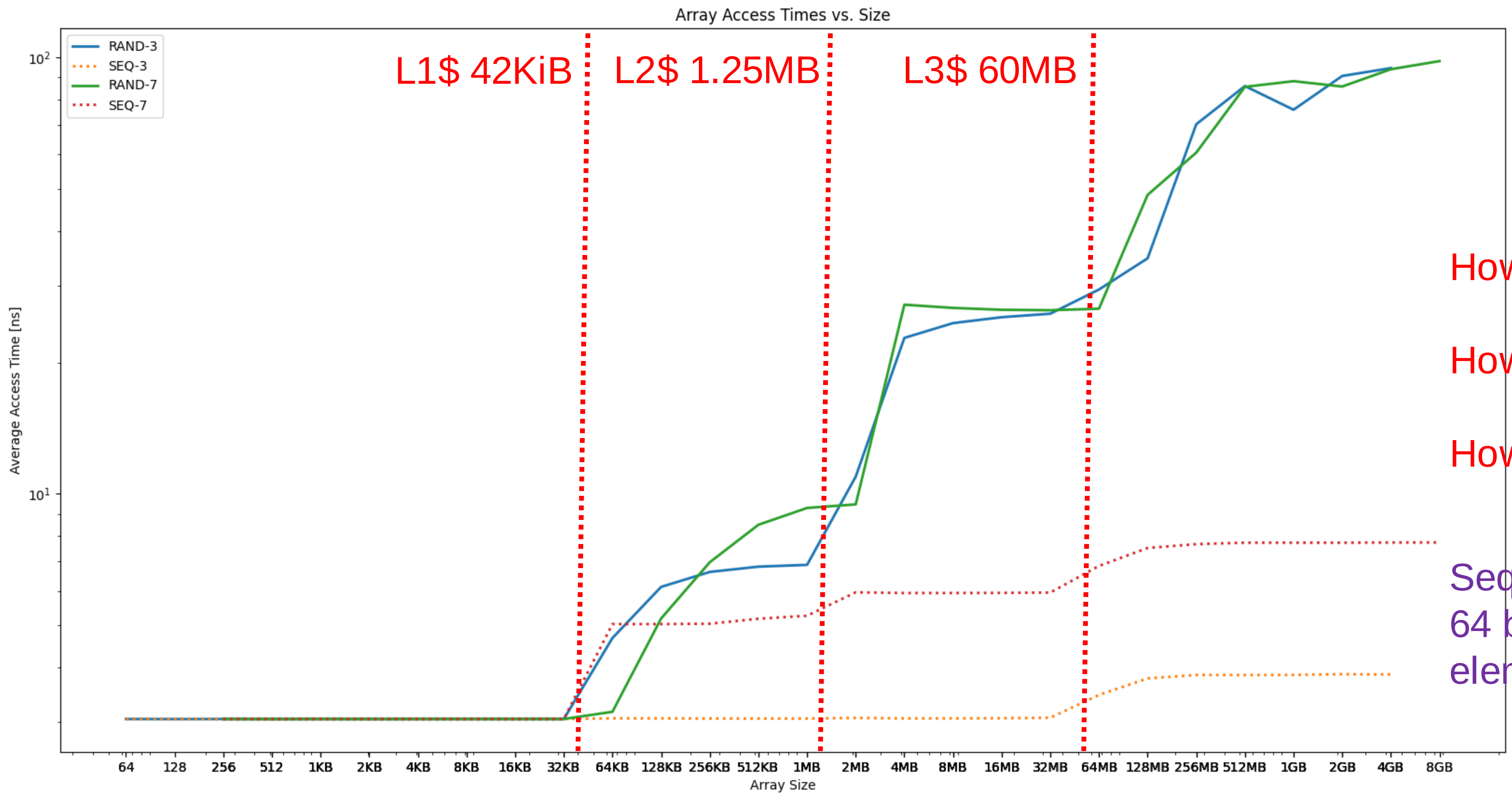

Random Accesses with 64 byte or 32 byte array elements

How many cache levels are there?

Sequential Accesses with 64 byte or 32 byte array elements

Intel Xeon Platinum
8380 CPU @ 2.30GHz

# Test your Understanding



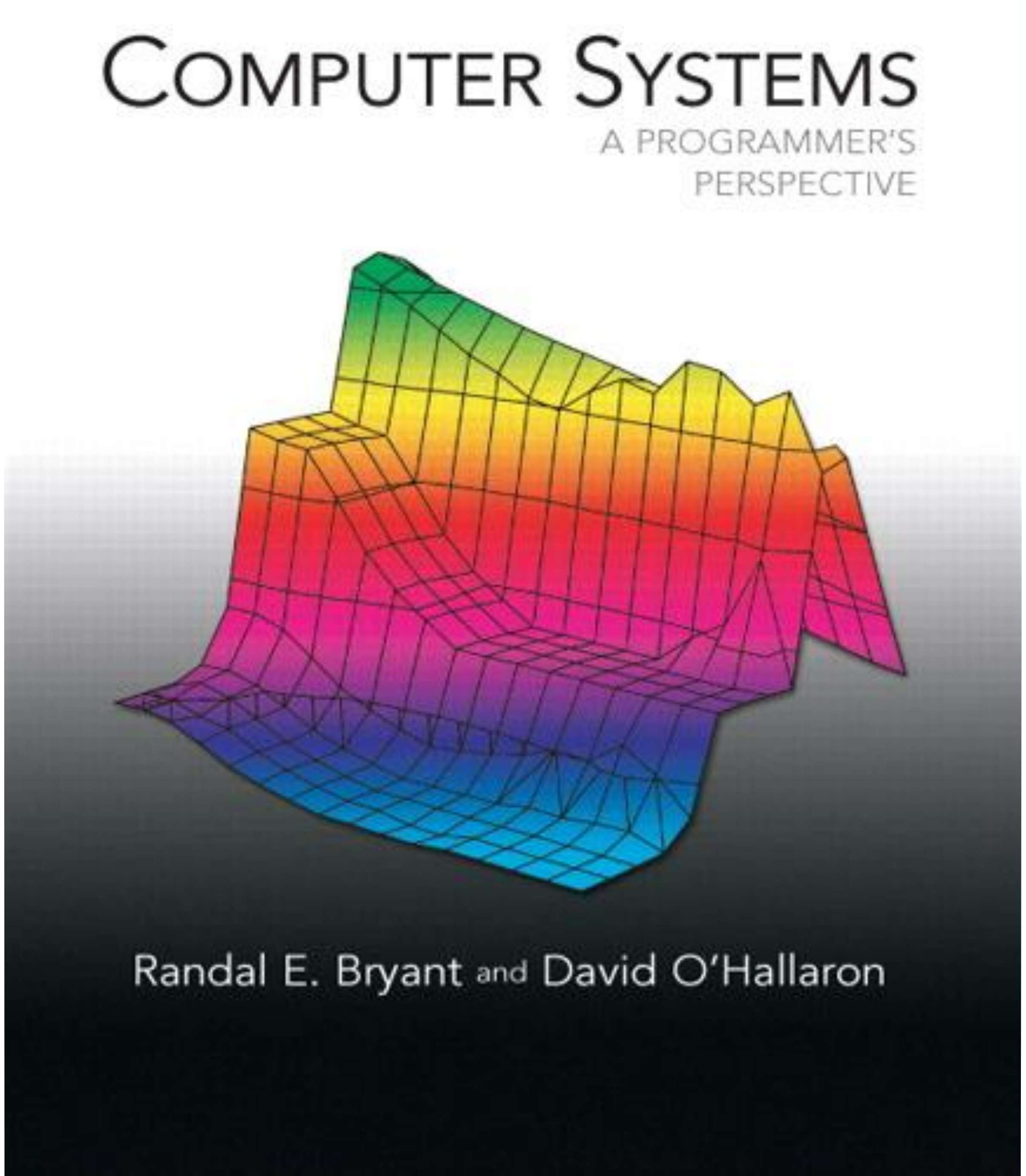

Random Accesses with 64 byte or 32 byte array elements

How big is the L1 cache?

How big is the L2 cache?

How big is the L3 cache?

Sequential Accesses with 64 byte or 32 byte array elements


COMPUTER SYSTEMS
A PROGRAMMER'S PERSPECTIVE
Randal E. Bryant and David O'Hallaron

# Now it’s your turn …

- Matrix multiply as an example of a real program whose performance matters!

# Wrapping Up

- Caches work best when access patterns exhibit spatial locality.
- Different access patterns can produce enormous performance differences, even if accessing the same amount of data.
- **Strided access** is common.
- Strided access patterns can be used to determine cache characteristics.