Cpp North 2025

Gotta Cache 'Em All

Optimize Your C++ Code By Utilizing Your Cache!

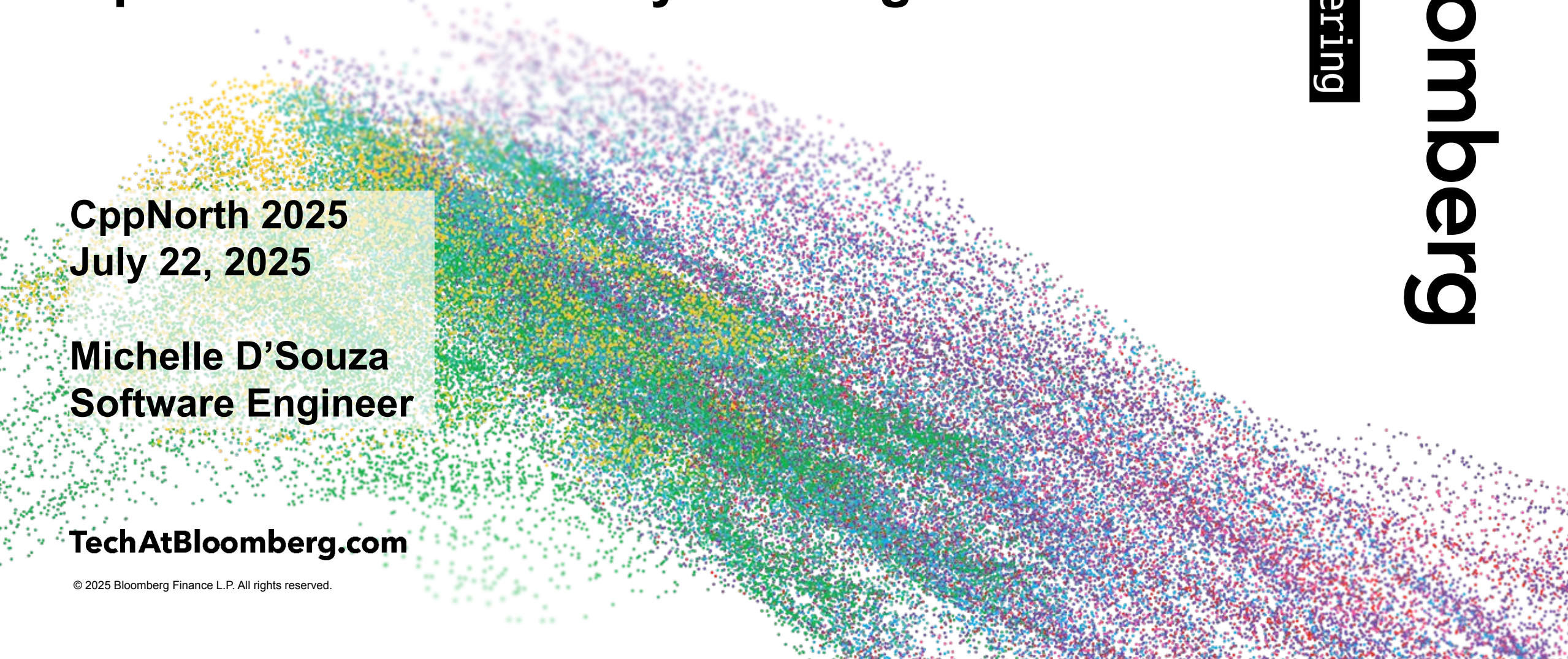
Michelle D'Souza

Gotta Cache ‘Em All:

Optimize Your Code By Utilizing Your Cache!

Bloomberg

Engineering



CppNorth 2025
July 22, 2025

Michelle D’Souza
Software Engineer

TechAtBloomberg.com

Inspired by a real 10x speedup

Chris Cotter commented on May 24, 2024 Member • • •

Do we even need to maintain the unordered_map at all? #8270 runs 10x faster by only using the vector.

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Basic ~~CompSci Facts~~
TRUST ISSUES

For any specific operation

Time complexity of a vector ≥ Time complexity of an unordered map

TechAtBloomberg.com

Bloomberg

Engineering

✨DetectiveD’Souza

🕵️ Undercover Detective

☕ Powered by curiosity, caffeine & Stack Overflow

👥 No case is ever cracked alone

Have something to share?
I’d love to hear at the end (just for the sake of time)!

Slide numbers here fyi!

TechAtBloomberg.com

Bloomberg

Engineering

Back on the Case

Chris Cotter commented on May 24, 2024 Member • • •

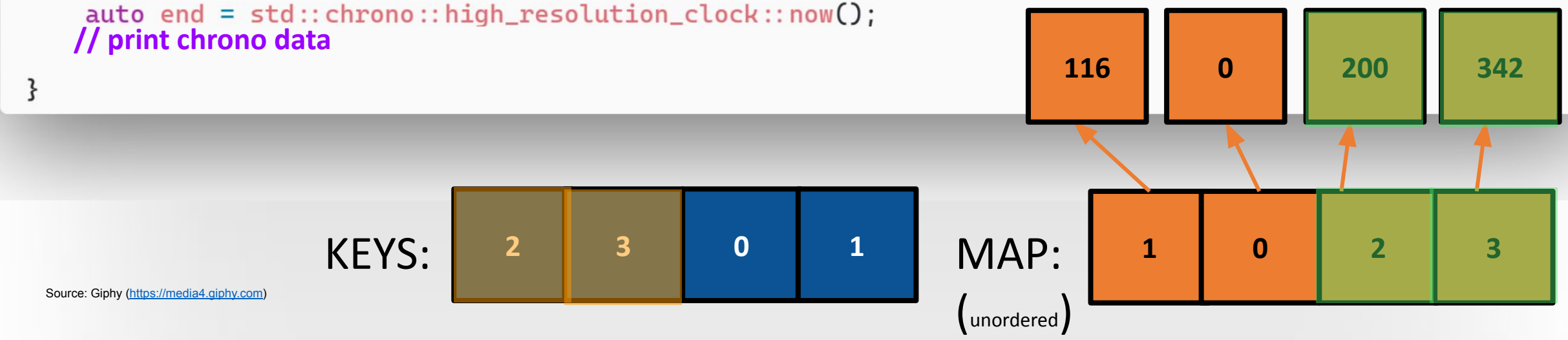
Do we even need to maintain the unordered_map at all? #8270 runs 10x faster by only using the vector.

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

The Case of Catching the Missing Complexity (Simplified)

```
constexpr size_t N = 1'000'000; // number of elements in map


/* Access each value and sum it up */
void benchmark_unordered_map(const std::unordered_map<i
    volatile int sum = 0;
    auto start = std::chrono::high_resolution_clock::no
    for (size_t i = 0; i < LOOKUPS; ++i) {
        sum += map.at(keys[i]);
    }
    auto end = std::chrono::high_resolution_clock::now();
    // print chrono data
}
```

O(K)
K is the number of keys



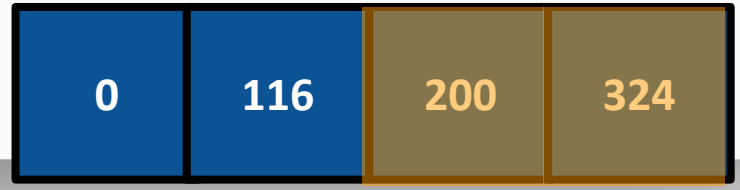
Source: Giphy (https://media4.giphy.com)

The Case of Catching the Missing Complexity (Simplified)

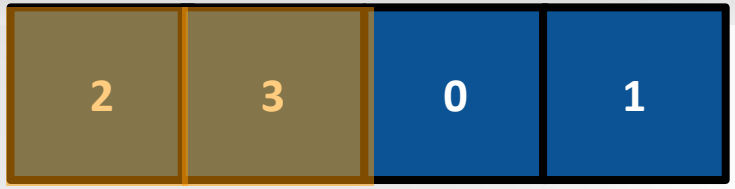
```
constexpr size_t N = 1'000'000; // number of elements in map


/* Access each value and sum it up */
void benchmark_vector(const std::vector<int>& vec, cons
    volatile int sum = 0;
    auto start = std::chrono::high_resolution_clock::no
    for (size_t i = 0; i < LOOKUPS; ++i) {
        sum += vec[keys[i]];
    }
    auto end = std::chrono::high_resolution_clock::now();
    // print chrono data
}
```

O(K)
K is the number of keys

VEC:



KEYS:



Source: Giphy (https://media4.giphy.com)

The Vector Wins!

```
[vector] Elapsed: 23.9168 ms
[unordered_map] Elapsed: 361.732 ms

[vector] Elapsed: 21.505 ms
[unordered_map] Elapsed: 353.587 ms

[vector] Elapsed: 22.4668 ms
[unordered_map] Elapsed: 338.601 ms
```

93.58% Speedup

15.53x faster!

Bloomberg

Engineering

Gathering Intel (and not the x86 kind)

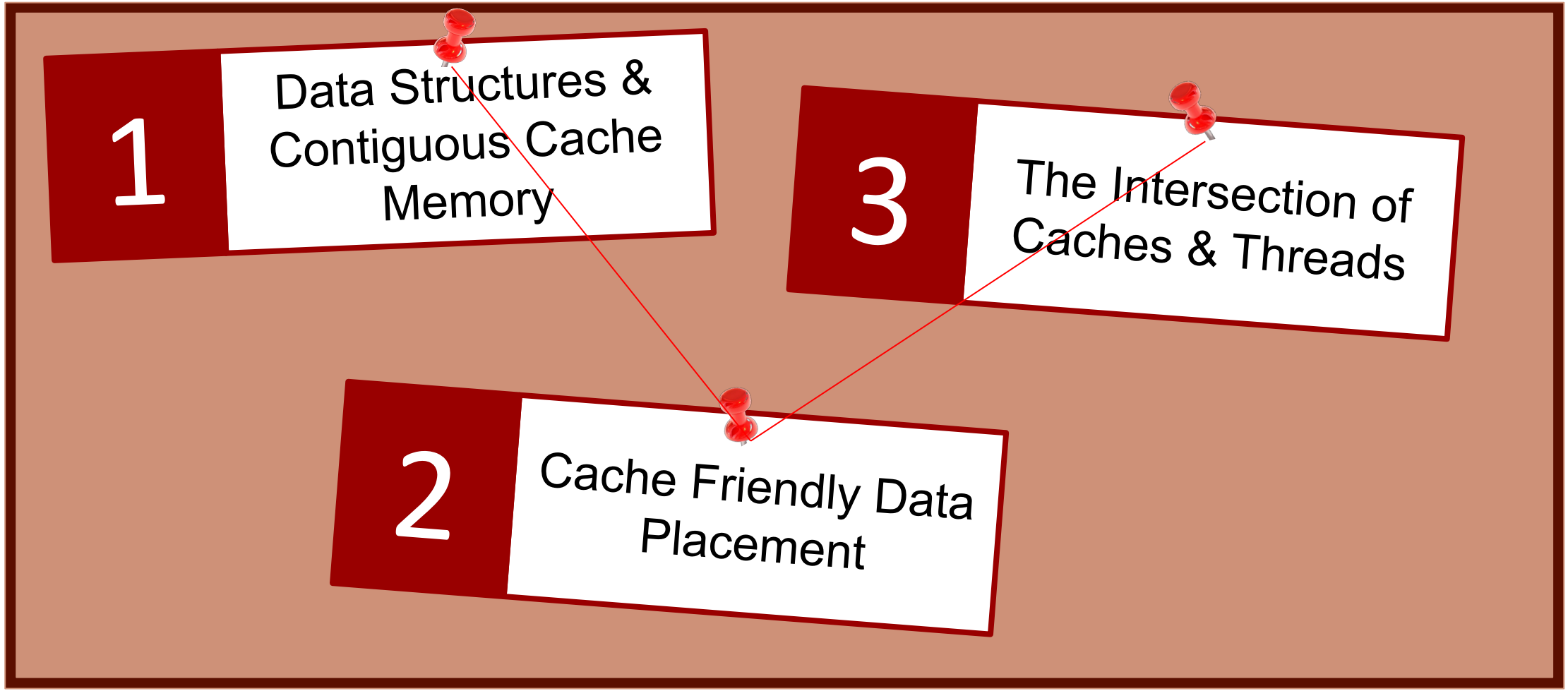
Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Code Sleuths Assemble! Can YOU Crack the Case?

- Hashing takes time
- Potential collisions in the unordered_map?
- No pointer chasing in the vector!
- Vector has contiguous memory!

How many detectives in the room knew this already?

~~Agenda~~ The Case File: What We'll Be Investigating Today



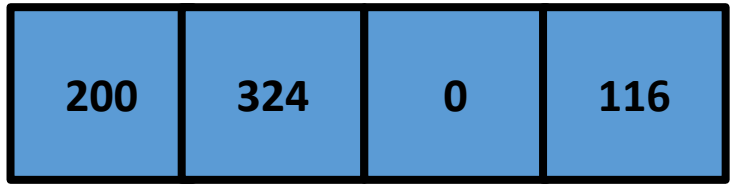
Bloomberg

Engineering

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

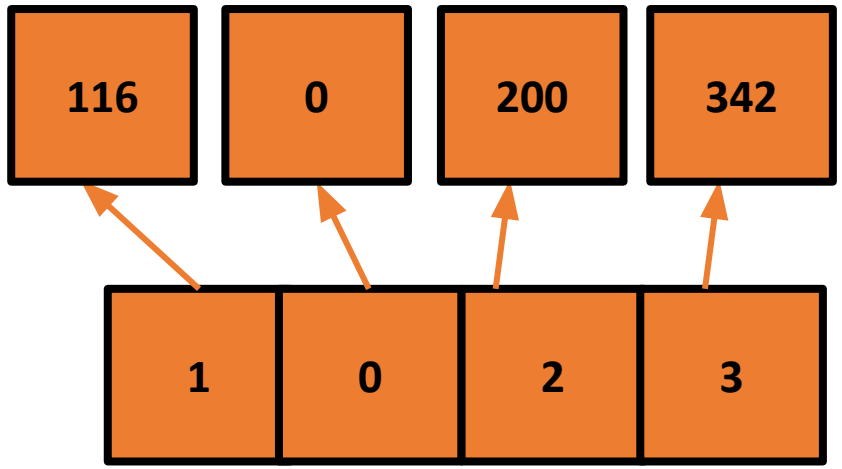
Why the vector was faster than the unordered_map

Contiguous memory!



Why the vector was faster than the unordered_map

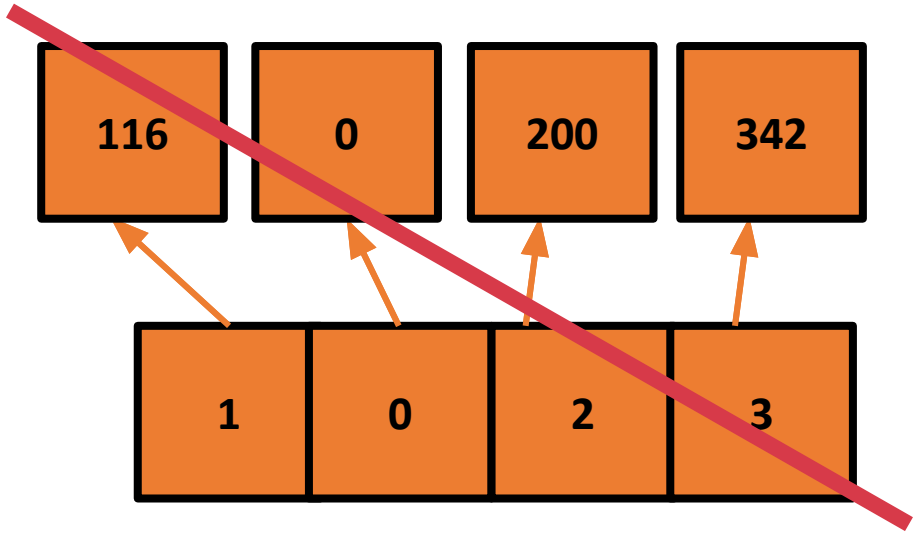
Contiguous memory!



The first rule of the game? Trust no one.
Including me.

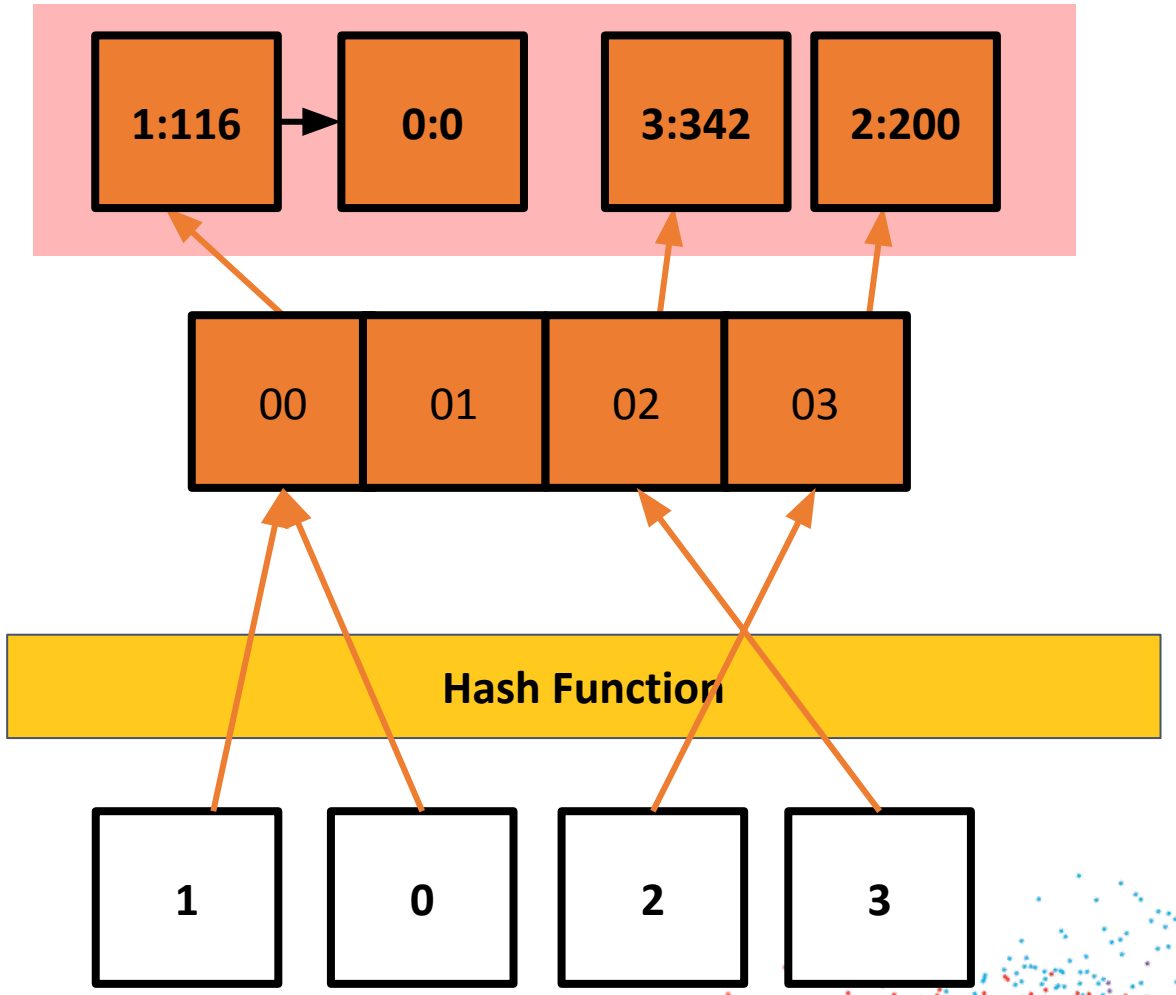
Why the vector was faster than the unordered_map

Contiguous memory!



Why the vector was faster than the unordered_map

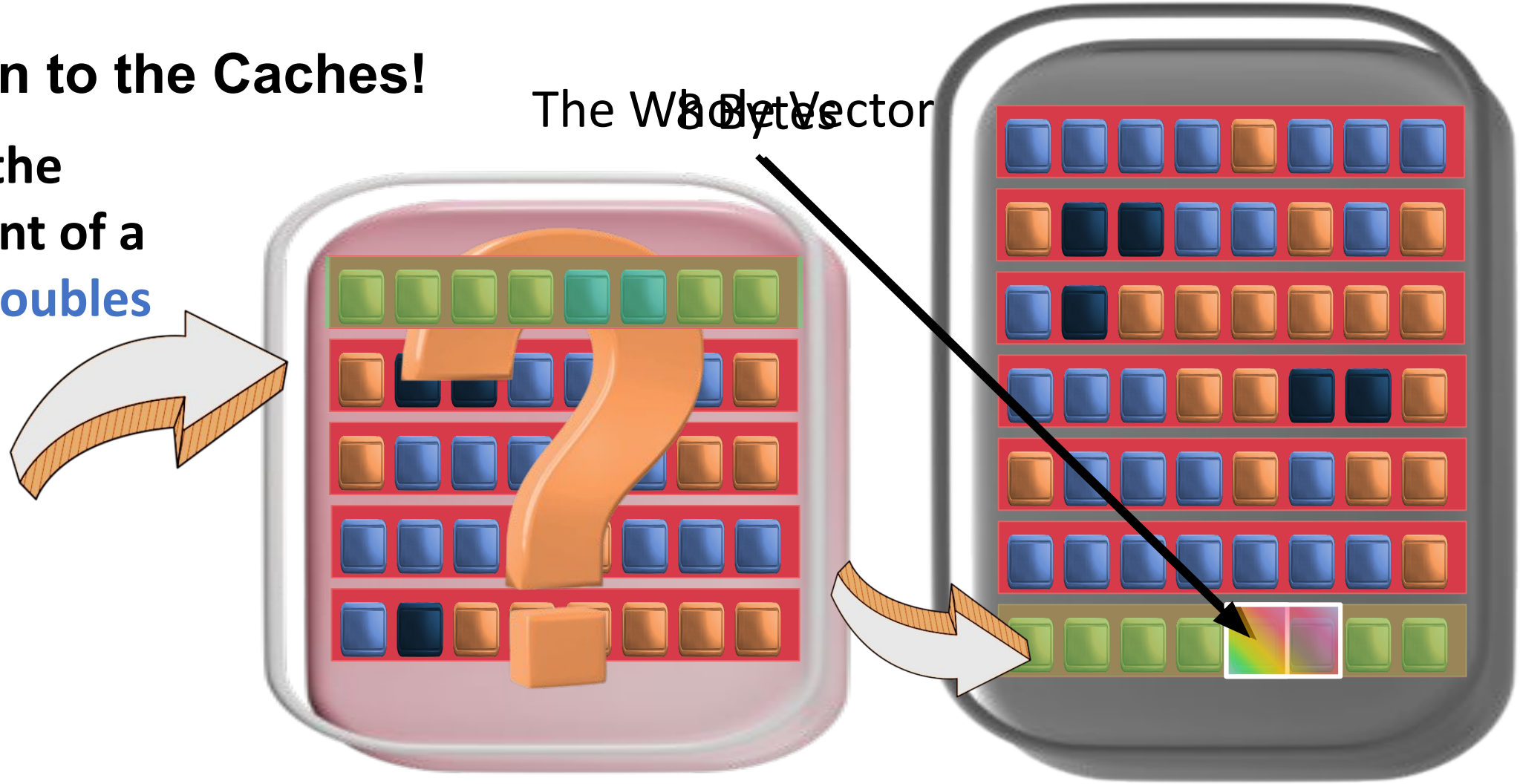
NOT Contiguous memory! **But why does that matter?**



Drill Down to the Caches!

Accessing the first element of a vector of doubles

Cache Miss



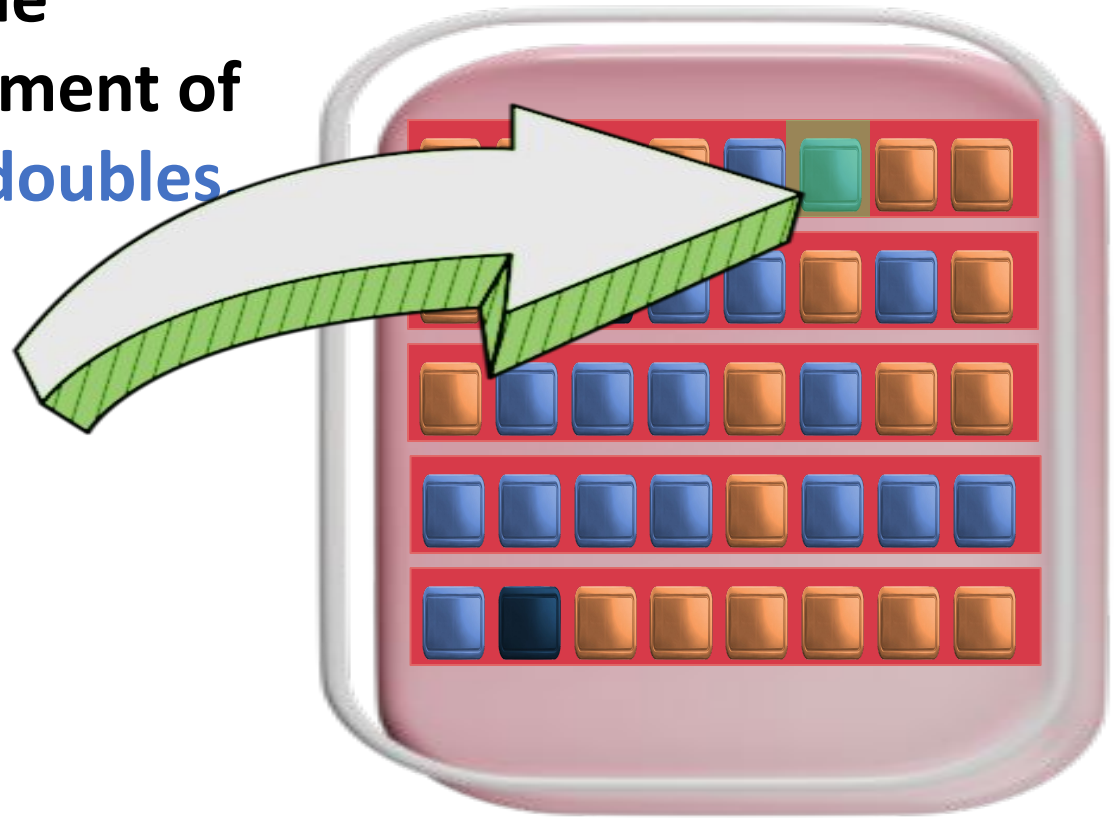
Cache line is typically 64 Bytes in x86, 32 Bytes in ARM

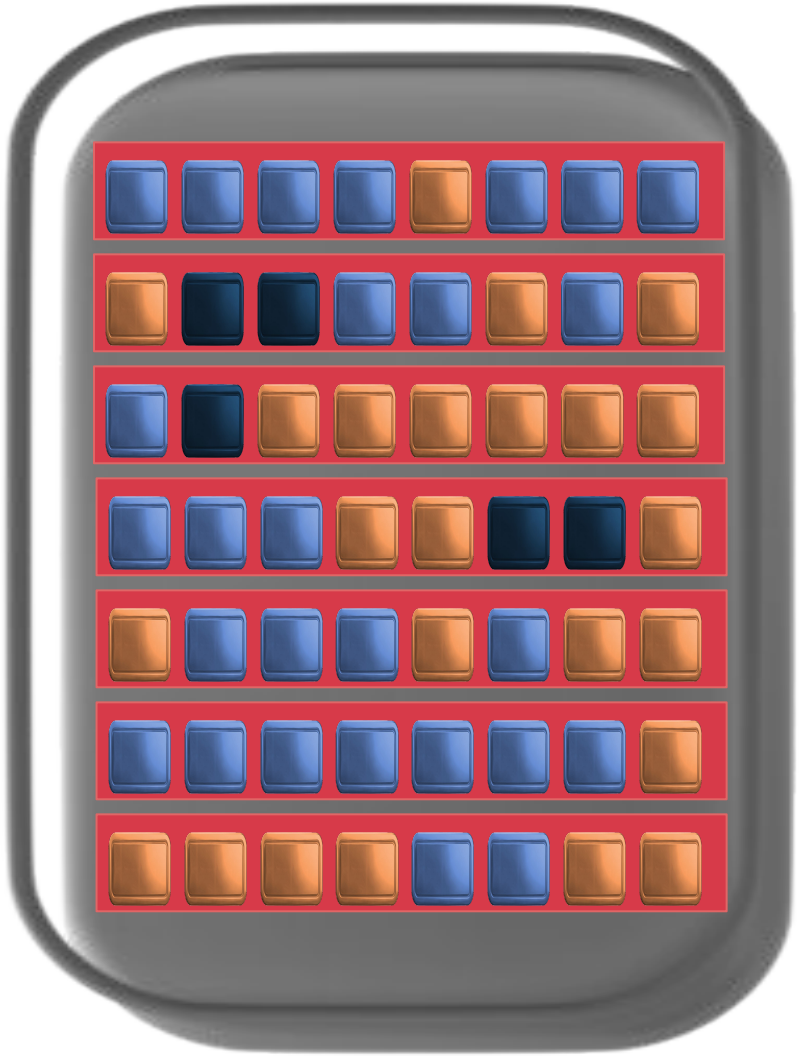
Source: Vecteezy (www.vecteezy.com)

Drill Down to the Caches!

Accessing the
SECOND element of
a vector of doubles

Cache hit





Cache

Main memory

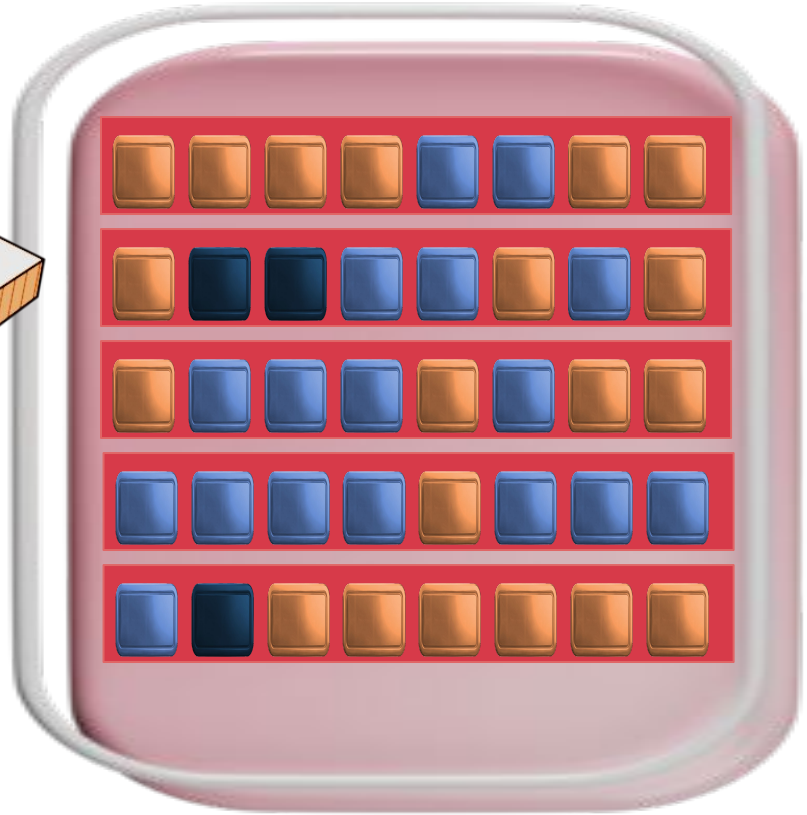
TechAtBloomberg.com

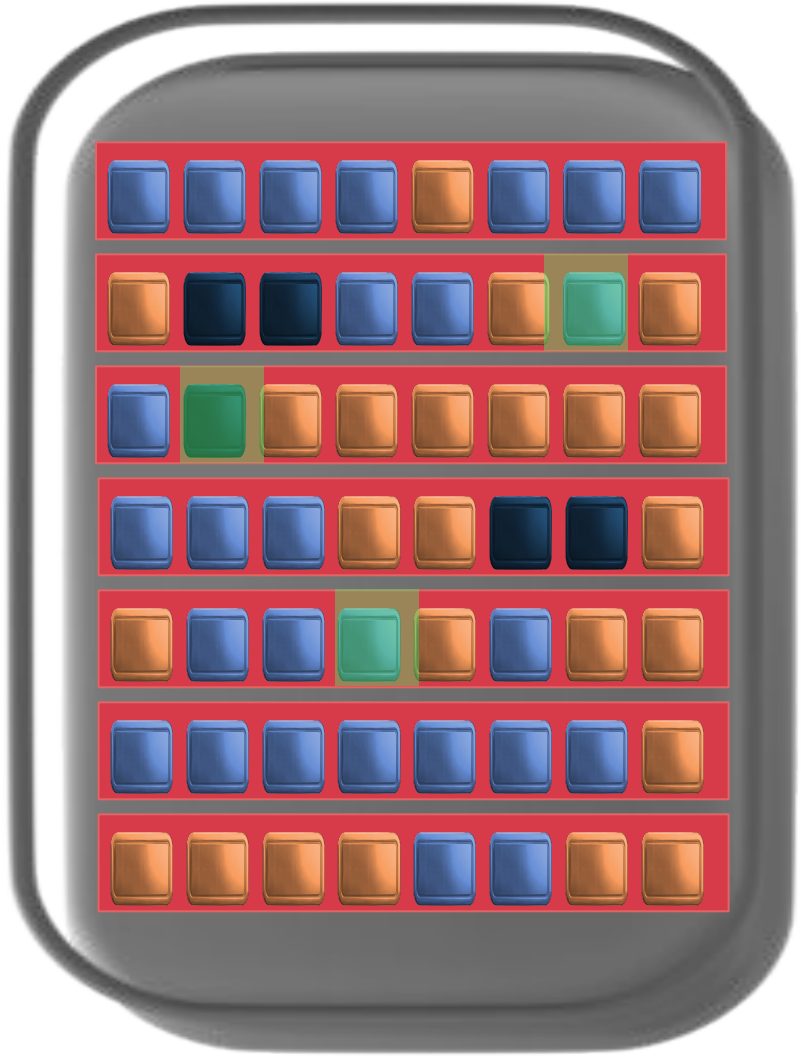
Source: Vecteezy (www.vecteezy.com)

Drill Down to the Caches!

Accessing the SECOND element of an unordered map.

Cache Miss





Cache

Main memory

TechAtBloomberg.com

Source: Vecteezy (www.vecteezy.com)

So what if there is a cache miss?

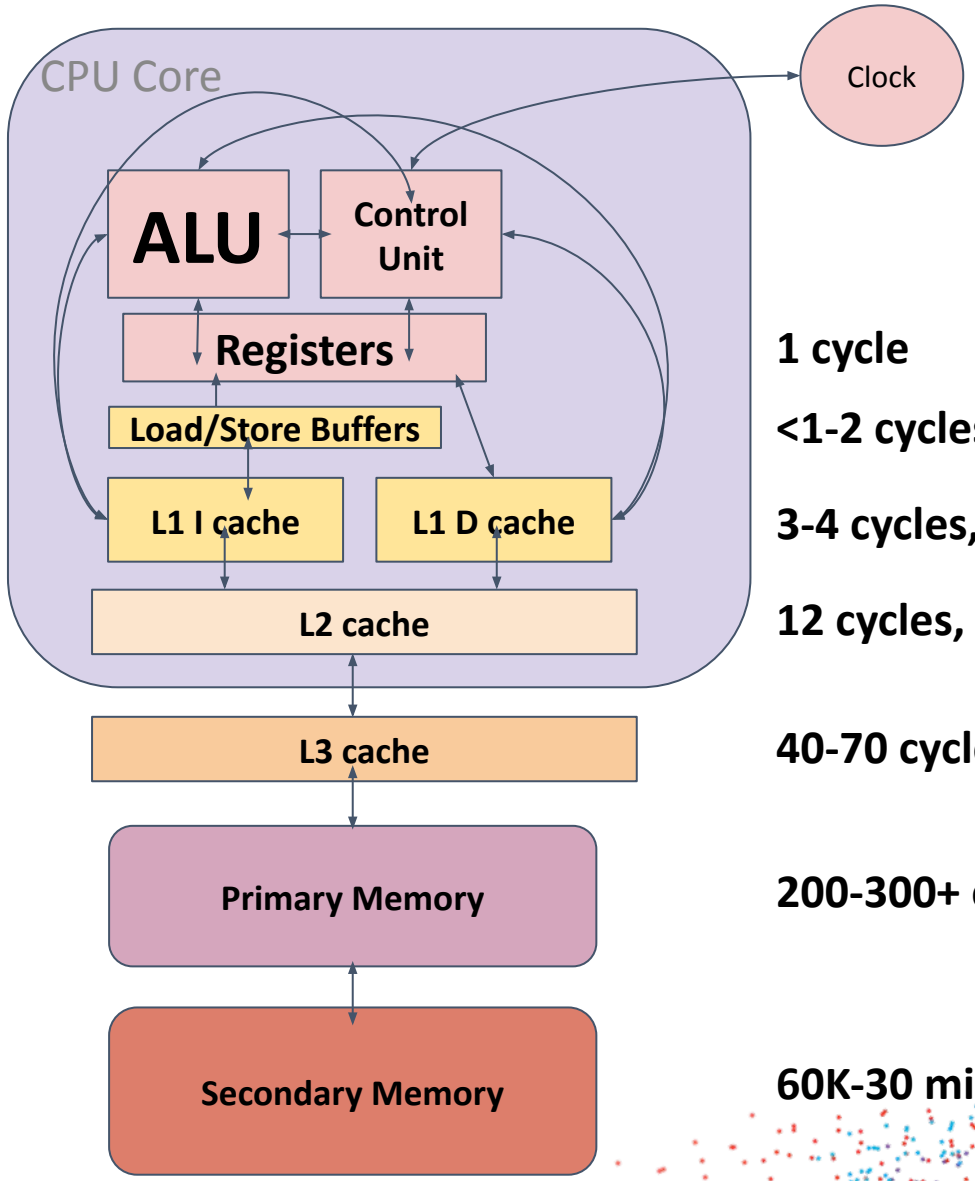
TechAtBloomberg.com

Bloomberg

Engineering

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Caches 101 - Contemporary CPUs



1 cycle

<1-2 cycles, 0.3 ns

3-4 cycles, 1 ns

12 cycles, 3-4 ns

40-70 cycles, 10-20 ns

200-300+ cycles, 60-120ns

60K-30 mil cycles,0.2-10ms

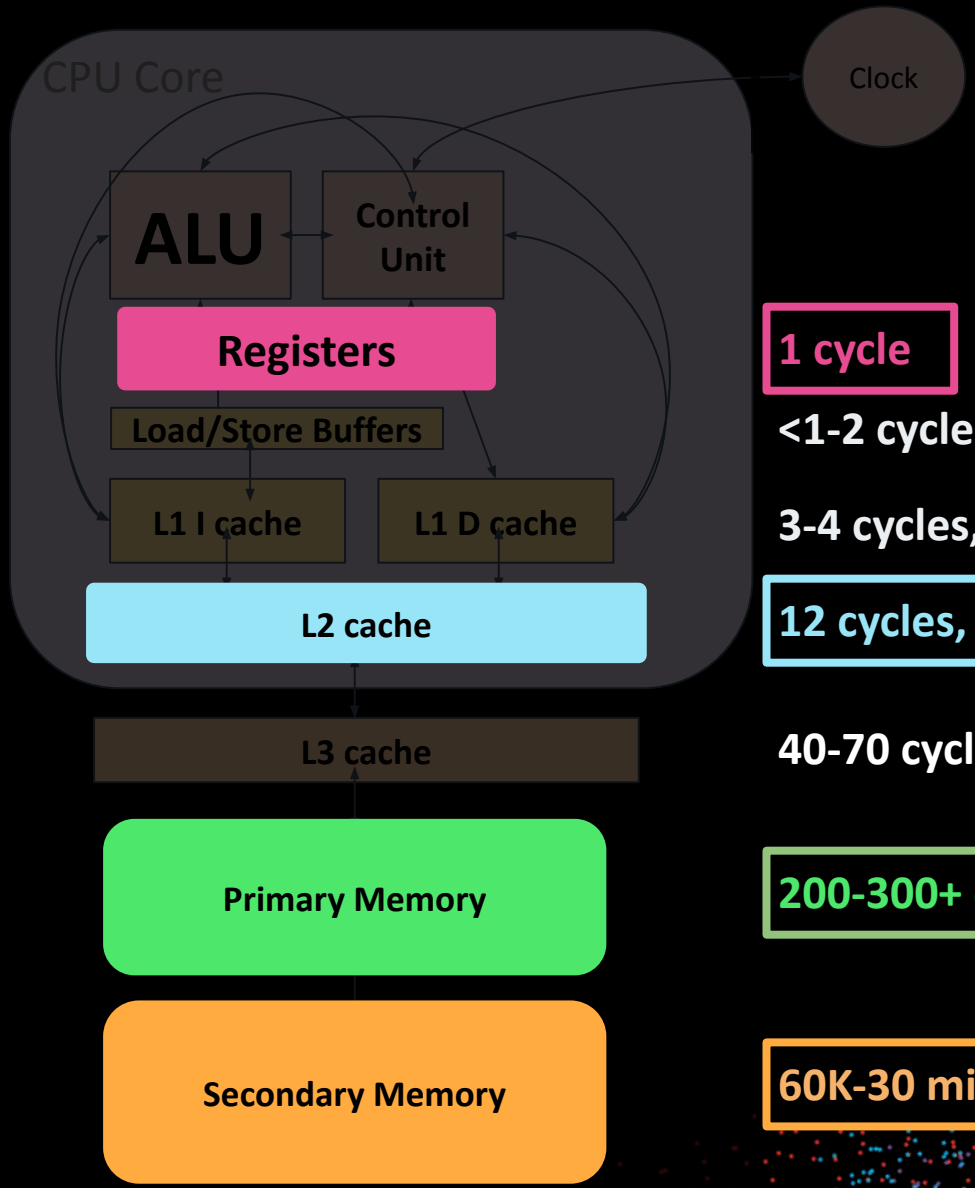
Slower access time

More storage space

(because less $ per KB)

* Diagram leaves out TLB, MMR, branch predictor, etc.
* There are also different kinds of architectures!

Caches 101 - Contemporary CPUs



1 cycle

<1-2 cycles, 0.3 ns

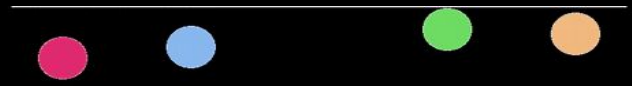
3-4 cycles, 1 ns

12 cycles, 3-4 ns

40-70 cycles, 10-20 ns

200-300+ cycles, 60-120ns

60K-30 mil cycles,0.2-10ms



In this investigation, the cache is sacred.

Don't miss it.

NOT CONTIGUOUS CONTIGUOUS

★ WANTED ~~DEAD~~ OR ~~ALIVE~~ ★

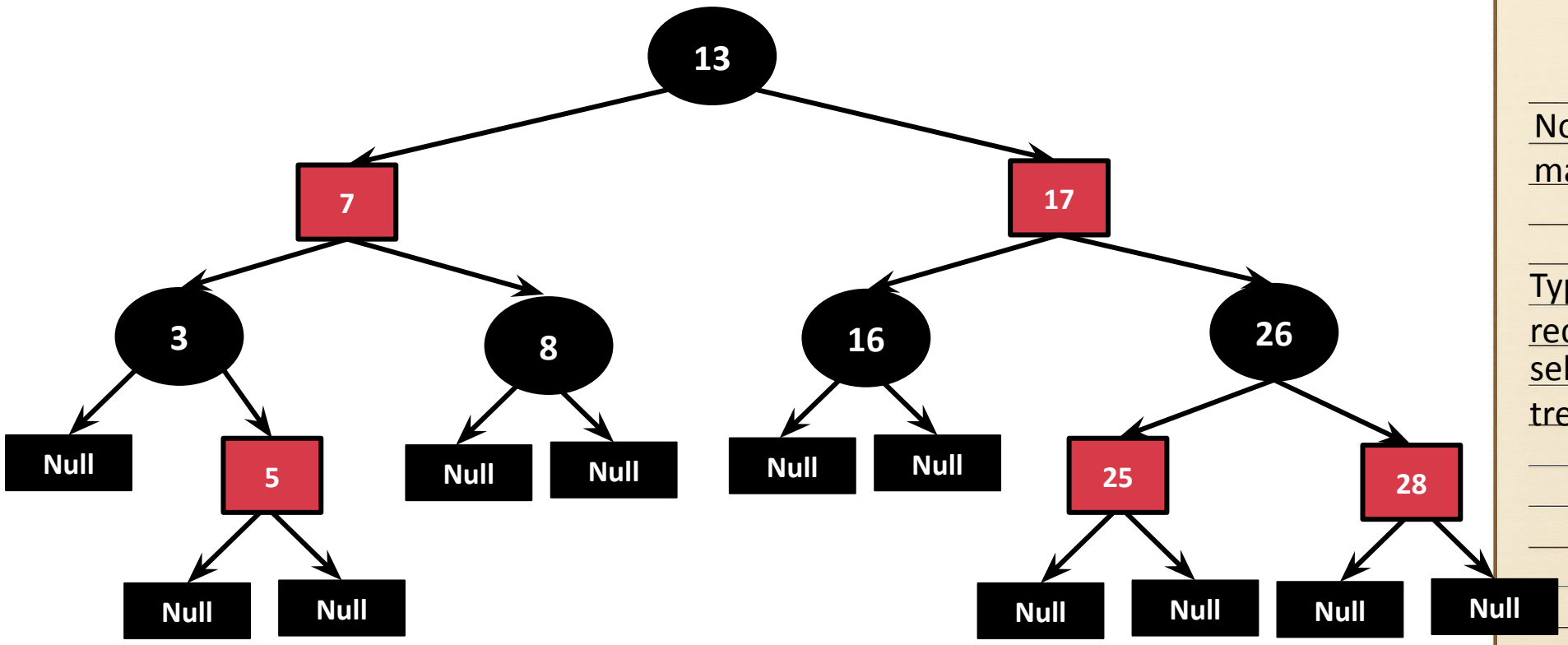
std::map
std::unordered_map
std::list
std::forward_list
std::set
std:unordered_set
std::multiset
std::unordered_multiset
std::multimap
std::unordered_multimap
boost::unordered_set

std::vector
std::array
C-style arrays (e.g int[])
std::valarray
std::string
std::flat_map (C++26)?
std::flat_set (C++26)?
absl::flat_hash_set
boost::flat_set
folly::F14FastSet

(C++ 26?, https://wg21.link/p0429)

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Understanding the enemy…



SUSPECT:

STD::SET

No specific implementation mandated by the standard.

Typically implemented as a red-black tree, or a self-balancing binary search tree.

TechAtBloomberg.com

Bloomberg

Engineering

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Understanding our allies …

3	5	7	8	10	11	13	16	17	25	26	28

SUSPECT:

BOOST::FLAT_SET

No tree. Just contiguous memory (array / vector), with sorted elements.

Sorted elements == binary search! (O log(N))

TechAtBloomberg.com

Bloomberg

Engineering

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Non-Contiguous vs. Contiguous

STD::SET wins

INSERTION / DELETION
O(log N) to find position (and rebalance) for std::set, O(N) for flat_set.

BOOST::FLAT_SET wins

LOOKUP
Both have O(logN) search time! (Flat_set might be faster because of SIMD though).

ITERATION
Faster (contiguous memory + cache friendly).

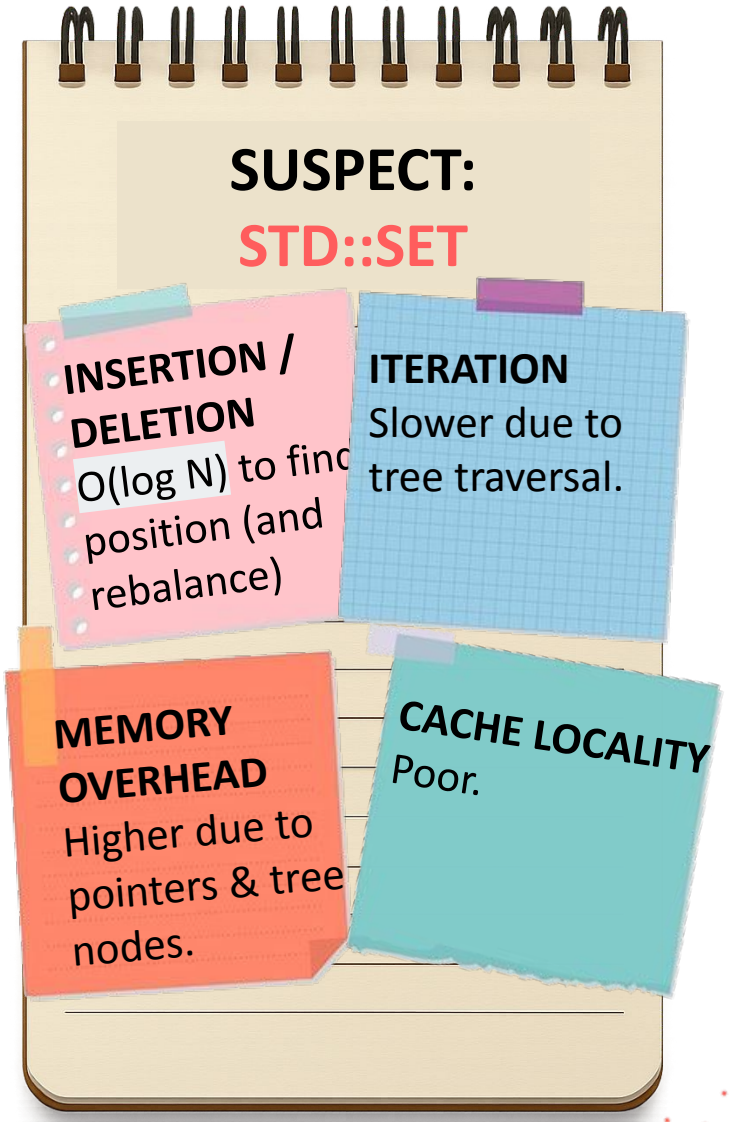
MEMORY OVERHEAD
Lower: compact array.

CACHE LOCALITY
Excellent.

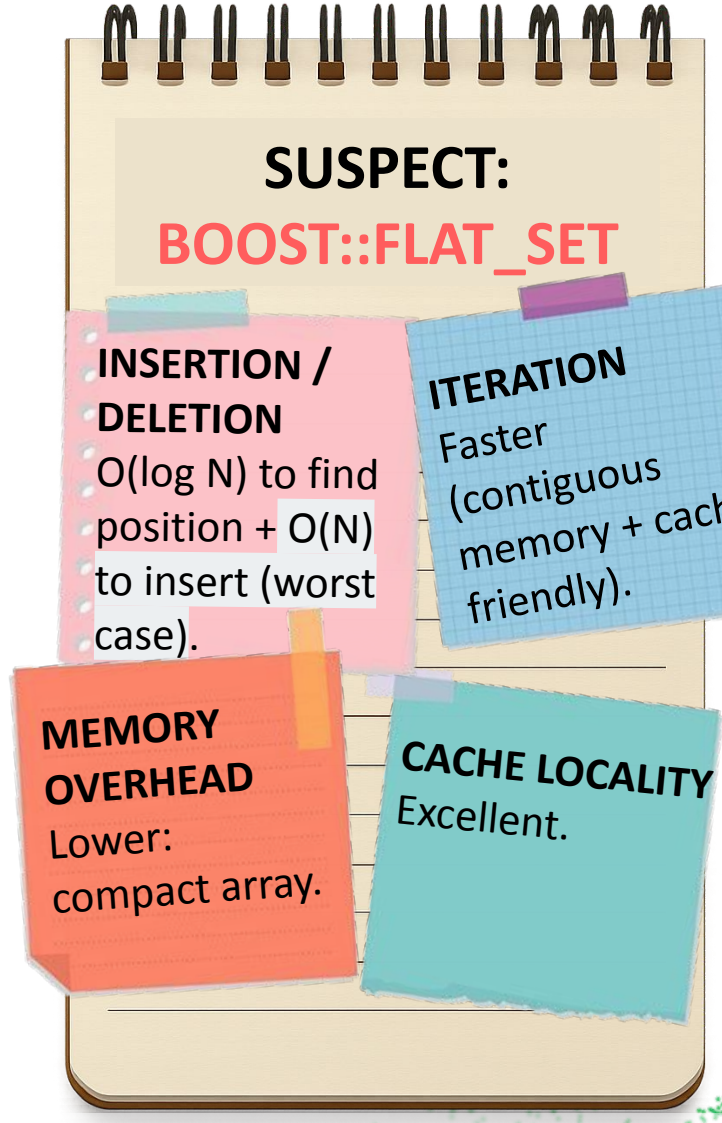
Source: pngall (www.pngall.com)

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Non-Contiguous vs. Contiguous



LOOKUP
Both have O(logN) search time! (Flat_set might be faster because of SIMD though).



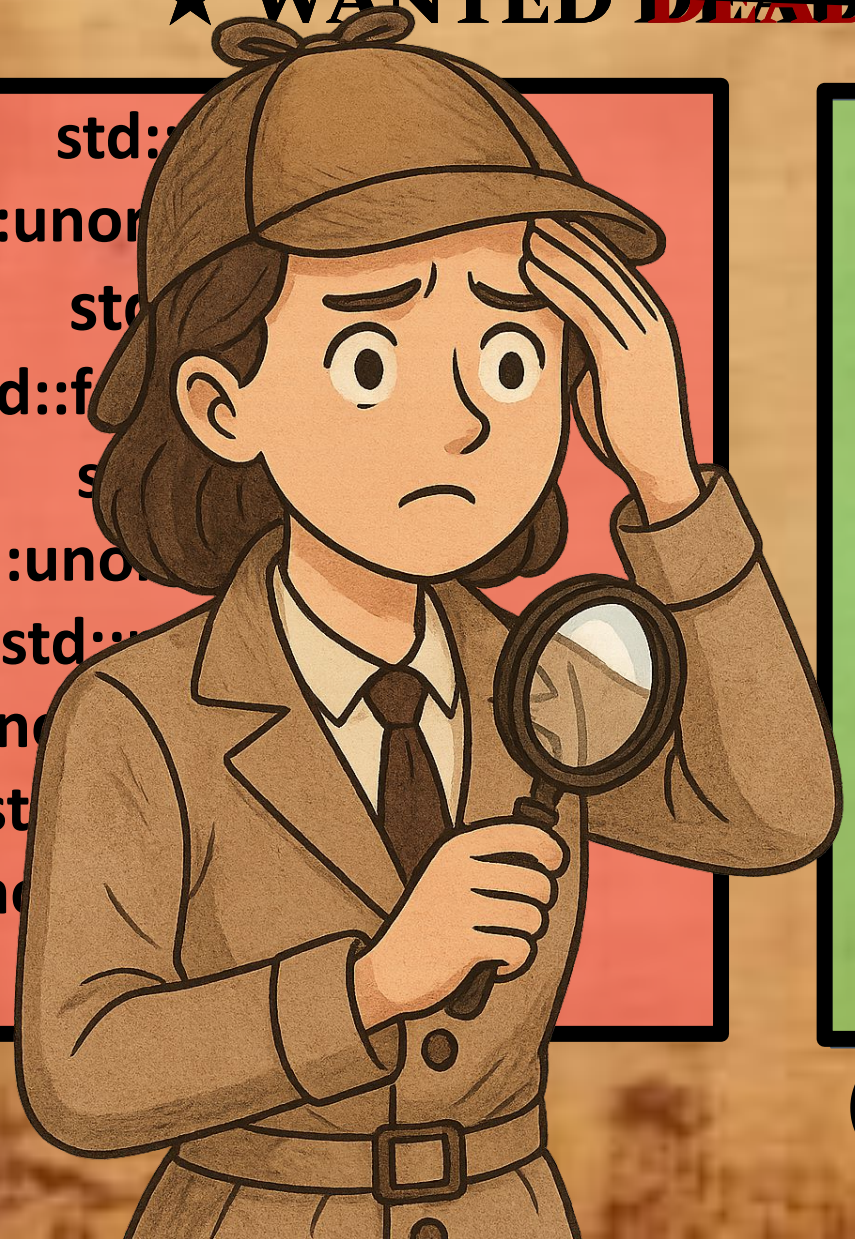
Source: pngall (www.pngall.com)

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

NOT CONTIGUOUS

CONTIGUOUS

★ WANTED ~~DEAD~~ OR ~~ALIVE~~ ★



std::

std::unor

st

std::f

s

std::uno

std::

std::unc

st

std::unc

std::vector

std::array

C-style arrays (e.g int[])

std::valarray

std::string

std::flat_map (C++26)?

std::flat_set (C++26)?

absl::flat_hash_set

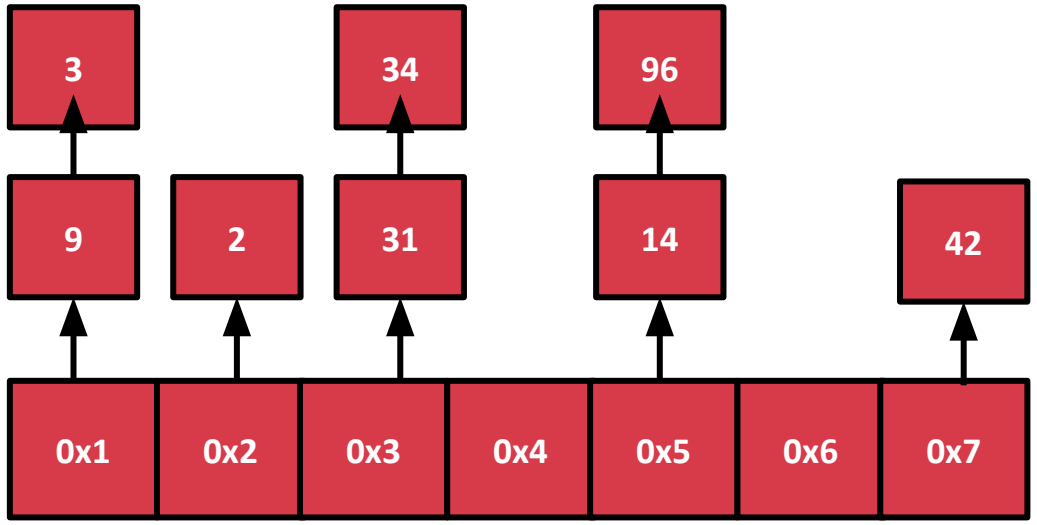
boost::flat_set

folly::F14FastSet

(C++ 26?, https://wg21.link/p0429)

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Investigating std::unordered_set!



SUSPECT:

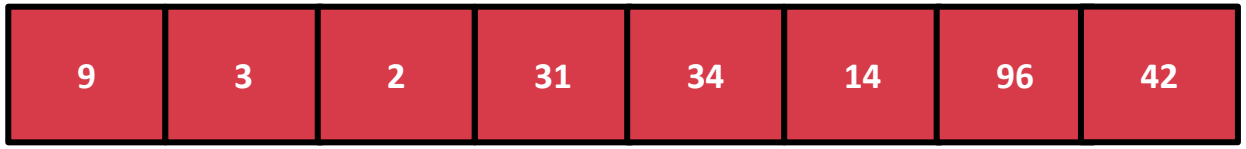
STD::UNORDERED_SET

Each bucket corresponds to a group of elements that hash to the same value modulo the bucket count.

Buckets are contiguous (usually a std::vector<std::forward_list<key >> or similar).

Nodes are scattered (forward or linked list), with separate chaining.

Investigating boost::flat_unordered_set!



SUSPECT:
BOOST::UNORDERED_FLAT_SET

Put everything in its own bucket (open-addressing), so contiguous!

What if a collision happens?

Limited API to access buckets, so not standard compliant.

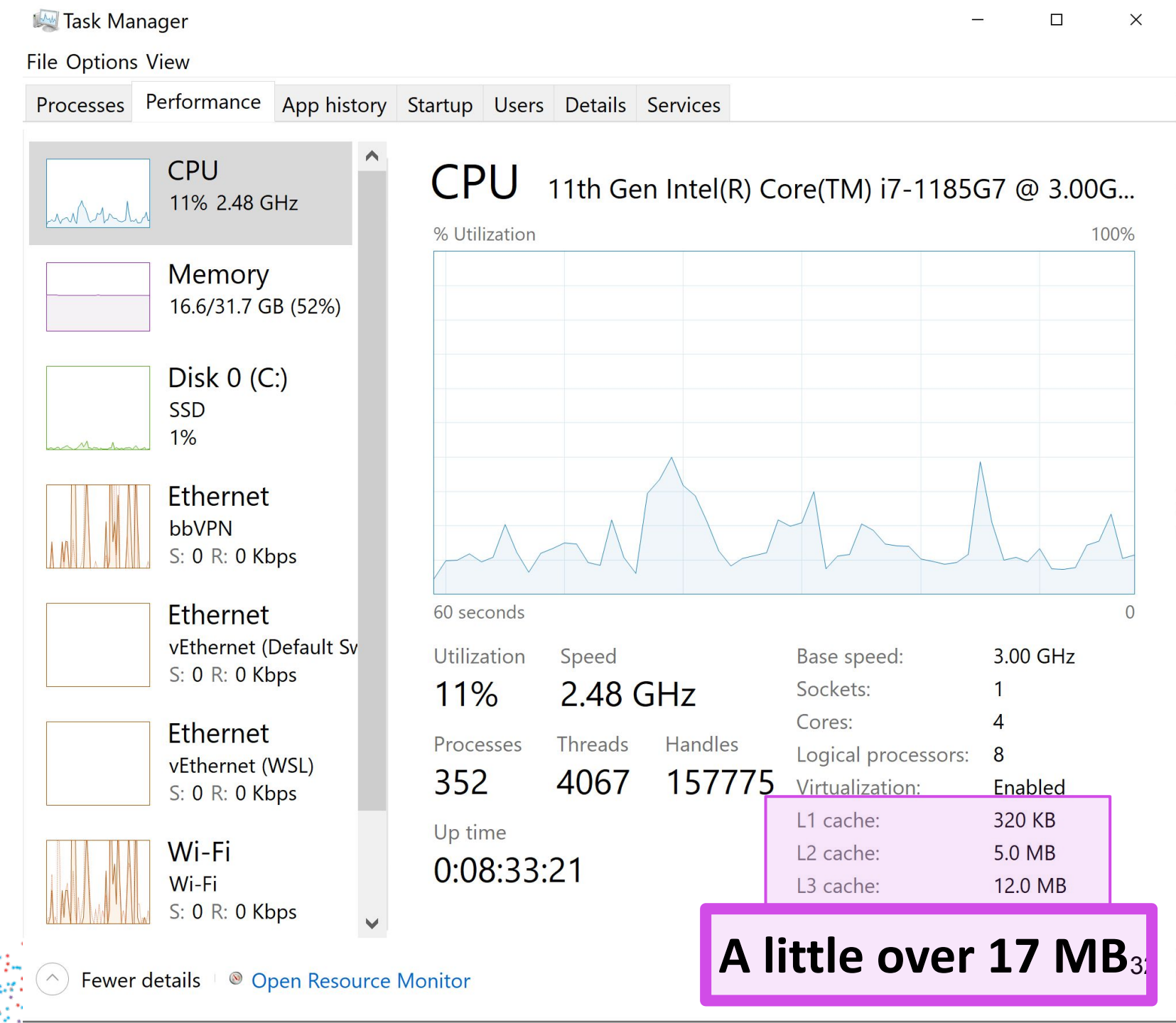
- No bucket(const& key), which is meant to return the bucket that an element is in.
- No max_bucket_count(), which is an upper bound on the number of buckets!

Source: Boost.org (boost.org)

The Curious Case of the Heavy Work Search

What if you've got a search heavy algorithm…
and stuff in between each search changes the L1 + L2 + L3 cache?

STEP 1: Understand Your Cache



A little over 17 MB

TechAtBloomberg.com

```cpp
void simulateHeavyWork() {
    const size_t arraySize = 512 * 1024 * 1024; // 512 MB
    std::vector<uint8_t> tempArray(arraySize);

    // some more heavy work :)
    for (int pass = 0; pass < 2; ++pass) {
        unsigned int x = 0xDEADBEEF + pass;
        for (size_t i = 0; i < arraySize; ++i) {
            x ^= x << 13;
            x ^= x >> 17;
            x ^= x << 5;
            size_t idx = x % arraySize;

            tempArray[idx] = (uint8_t)(i % 256);
            volatile auto dummy = tempArray[idx];
            (void)dummy;
        }
    }
}
```

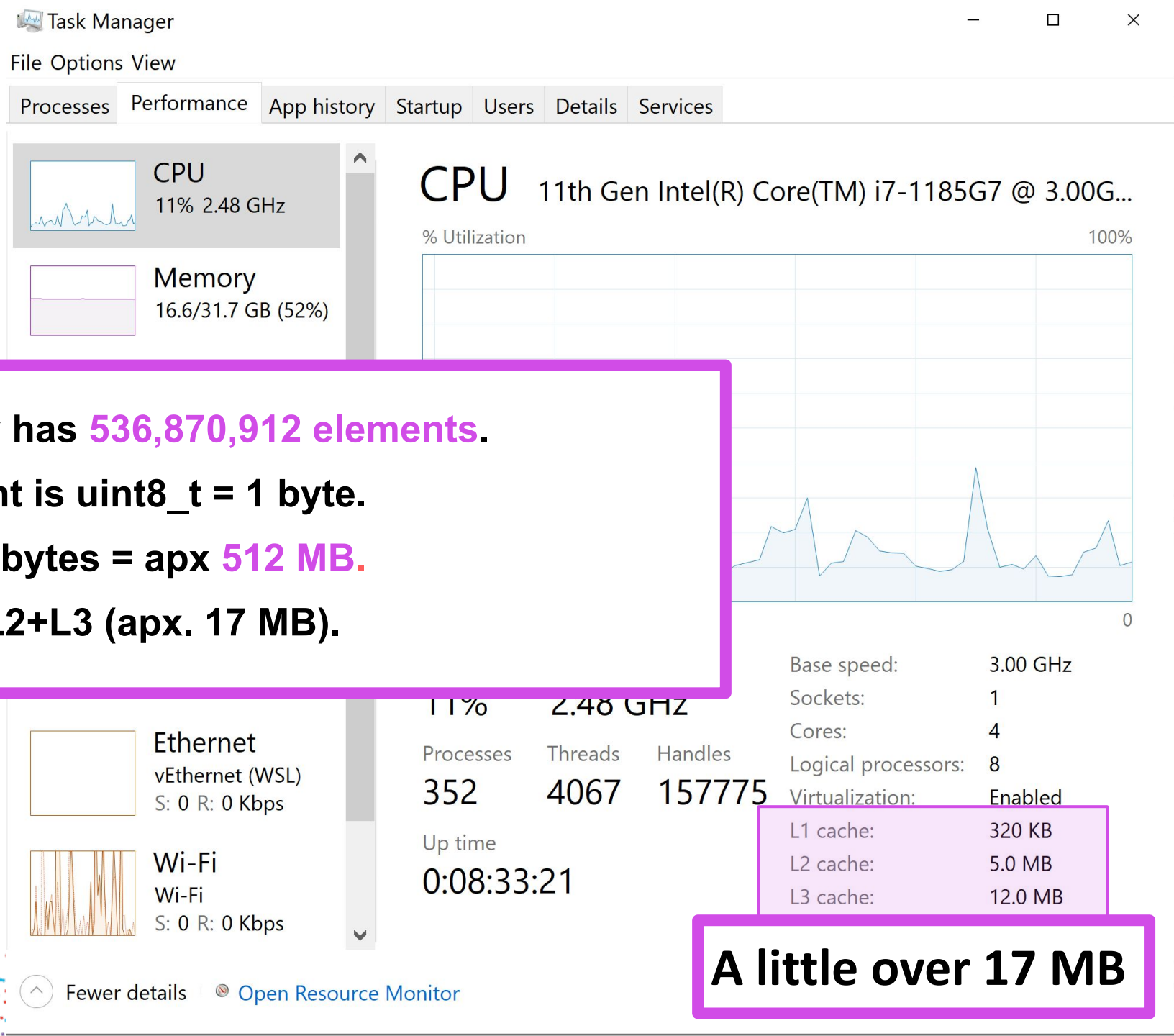
STEP 1: Understand Your Cache

Heavy work's array has 536,870,912 elements.

Each element is uint8_t = 1 byte.

536,870,912 bytes = apx 512 MB.

Flush L1+L2+L3 (apx. 17 MB).



A little over 17 MB

TechAtBloomberg.com

```cpp
template <typename T>
long long measureFlatSetSearch(const boost::container::flat_set<T>& flatSet, const T& target) {
    simulateHeavyWork();

    // Time search for element in data structure
    auto start = std::chrono::high_resolution_clock::now();
    auto it = flatSet.find(target);
    auto end = std::chrono::high_resolution_clock::now();
    // use it variable so it is not optimized away
    return std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();
}
```

Repeat for different data structures.

Repeat for 100 different search targets.

Repeat for different data structure sizes.

Repeat 25 times for each data structure size.

This code took 1 day to run.

Source: Giphy (https://media4.giphy.com)

The data structures!

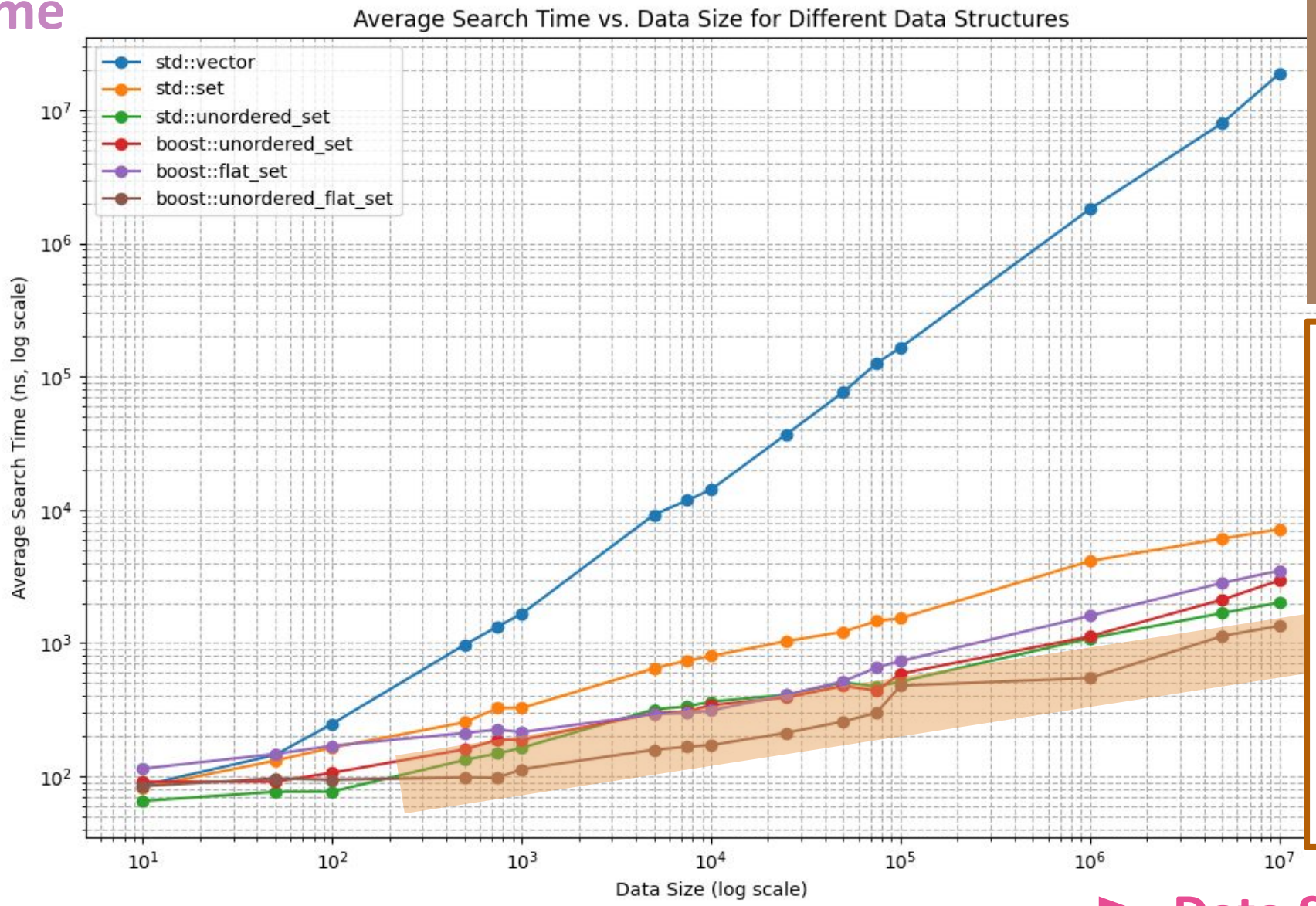
- std::vector?
- std::set?
- std::unordered_set?
- boost::flat_set?
- boost::unordered_set?
- boost::unordered_flat_set?

Which one performed best for large data structure sizes (> 10K)?

- std::vector?
- std::set?
- std::unordered_set?
- boost::flat_set?
- boost::unordered_set?
- boost::unordered_flat_set?

TechAtBloomberg.com

Time

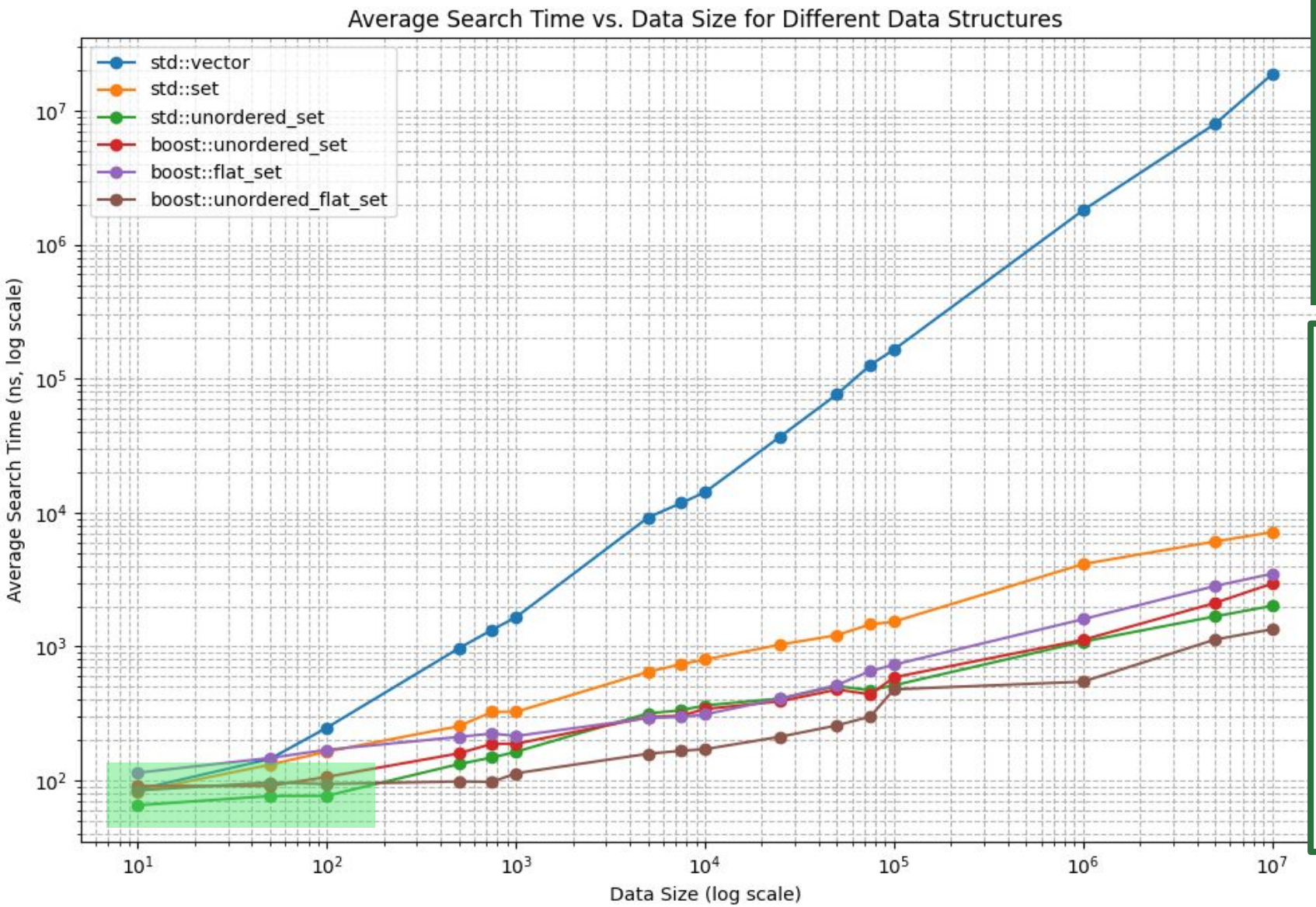


boost::unordered_flat_set wins!

As the data got larger, boost::unordered_set and std::unordered_set weren't too far behind.

Boost::flat_set followed after, then std::set, and then std::vector.

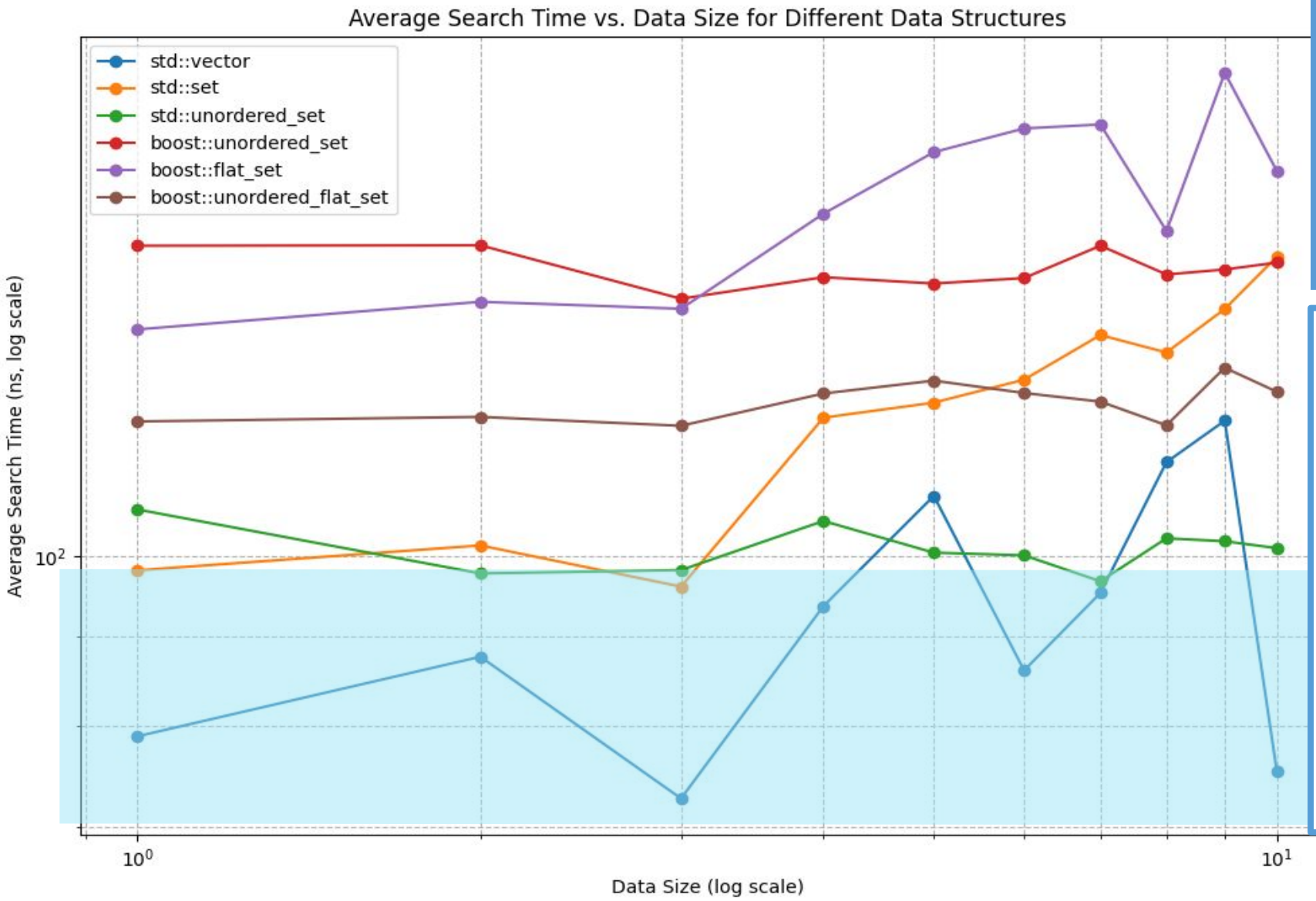
Data Size



std::unordered_set vs boost::unordered_flat_set.

For very small data sizes (e.g., N < 100), std::unordered_set won!

Std::unordered_set: Follow a pointer to a node. For a small, non-colliding set, this is incredibly direct and simple. The find operation is just a few machine instructions.



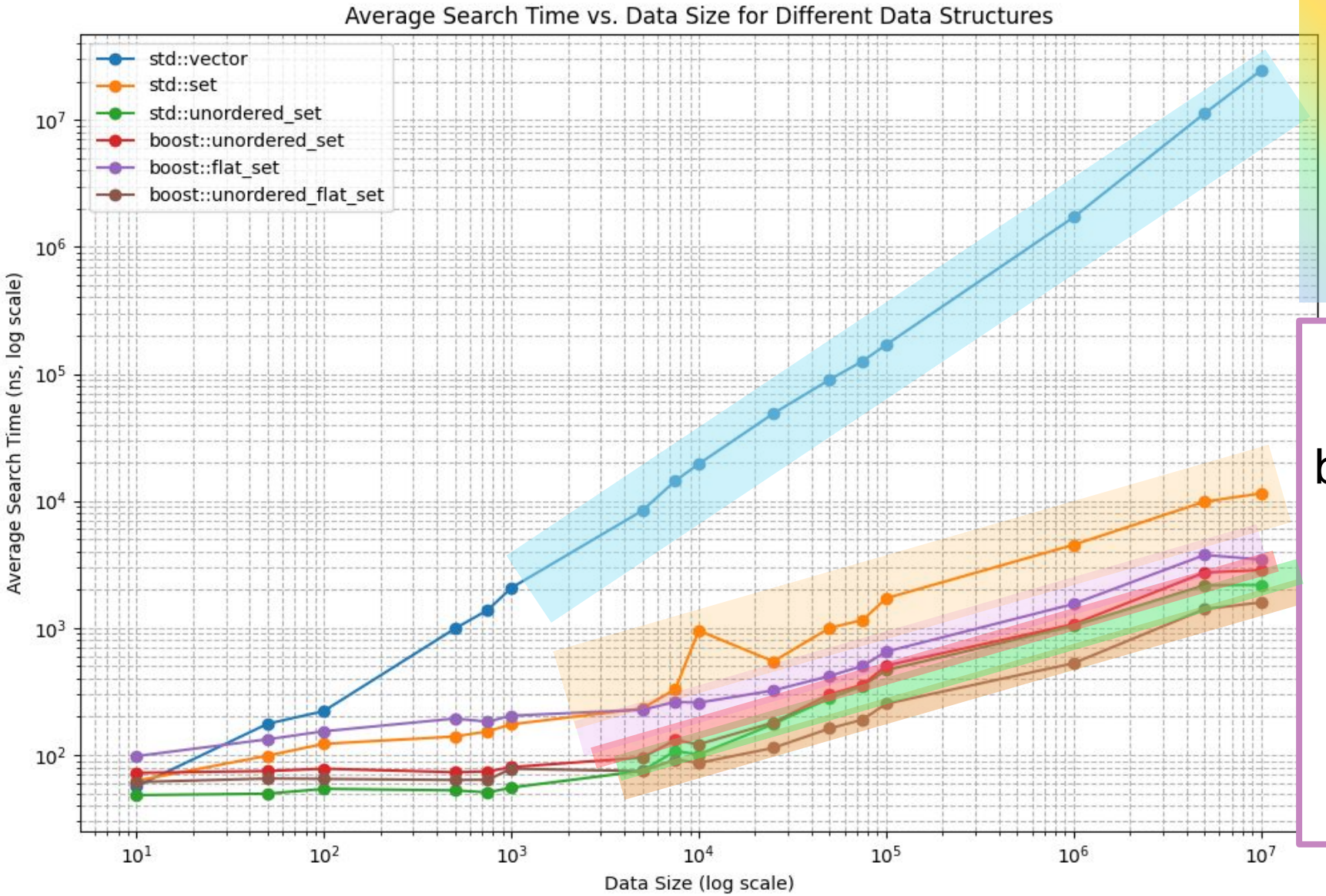
Vector

Reminder: Big O applies mainly on big data!

For small N (<=10) vector can be faster than the other data structures!

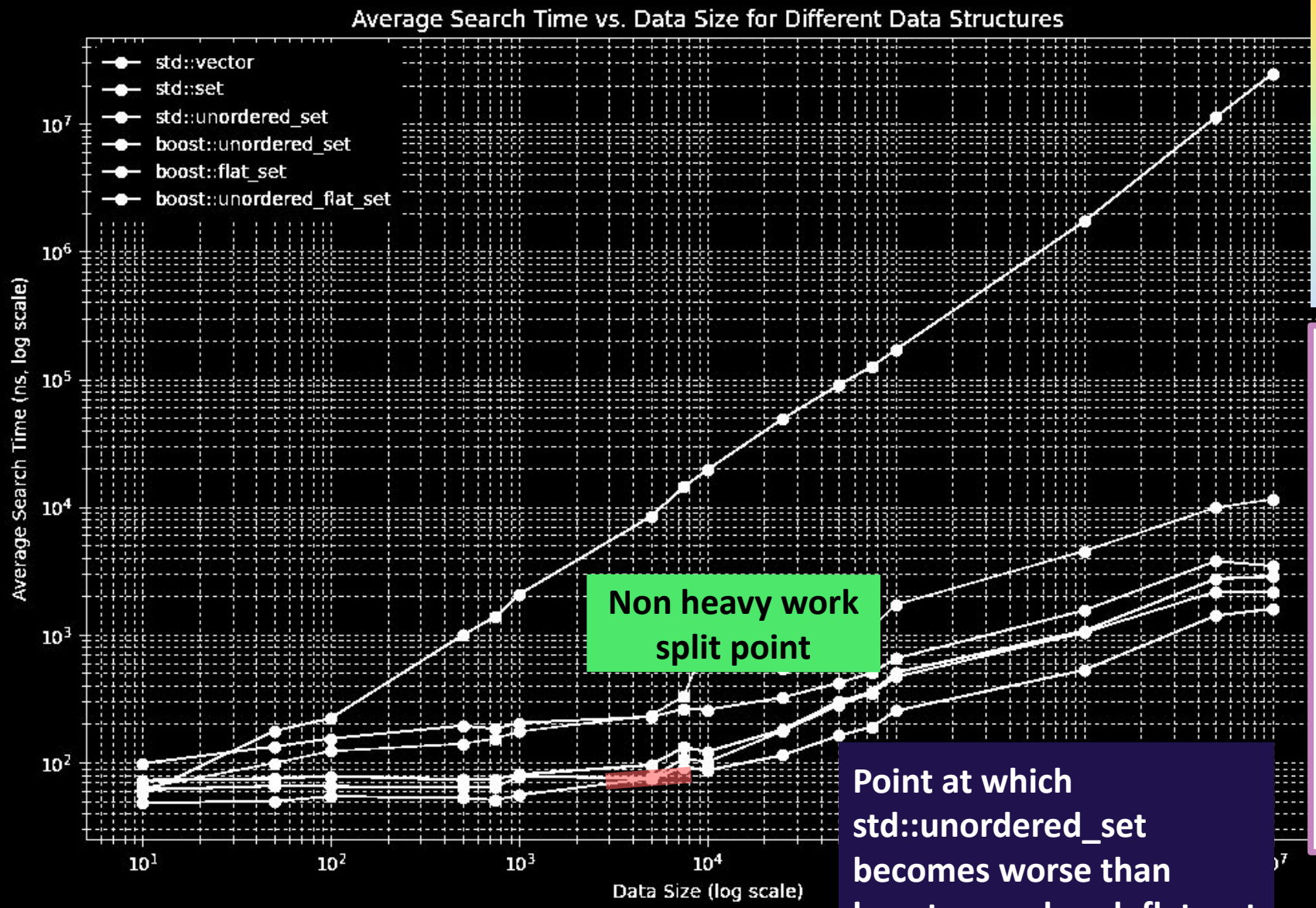
What about WITHOUT heavy work?

Without Heavy Work



The pattern persists!

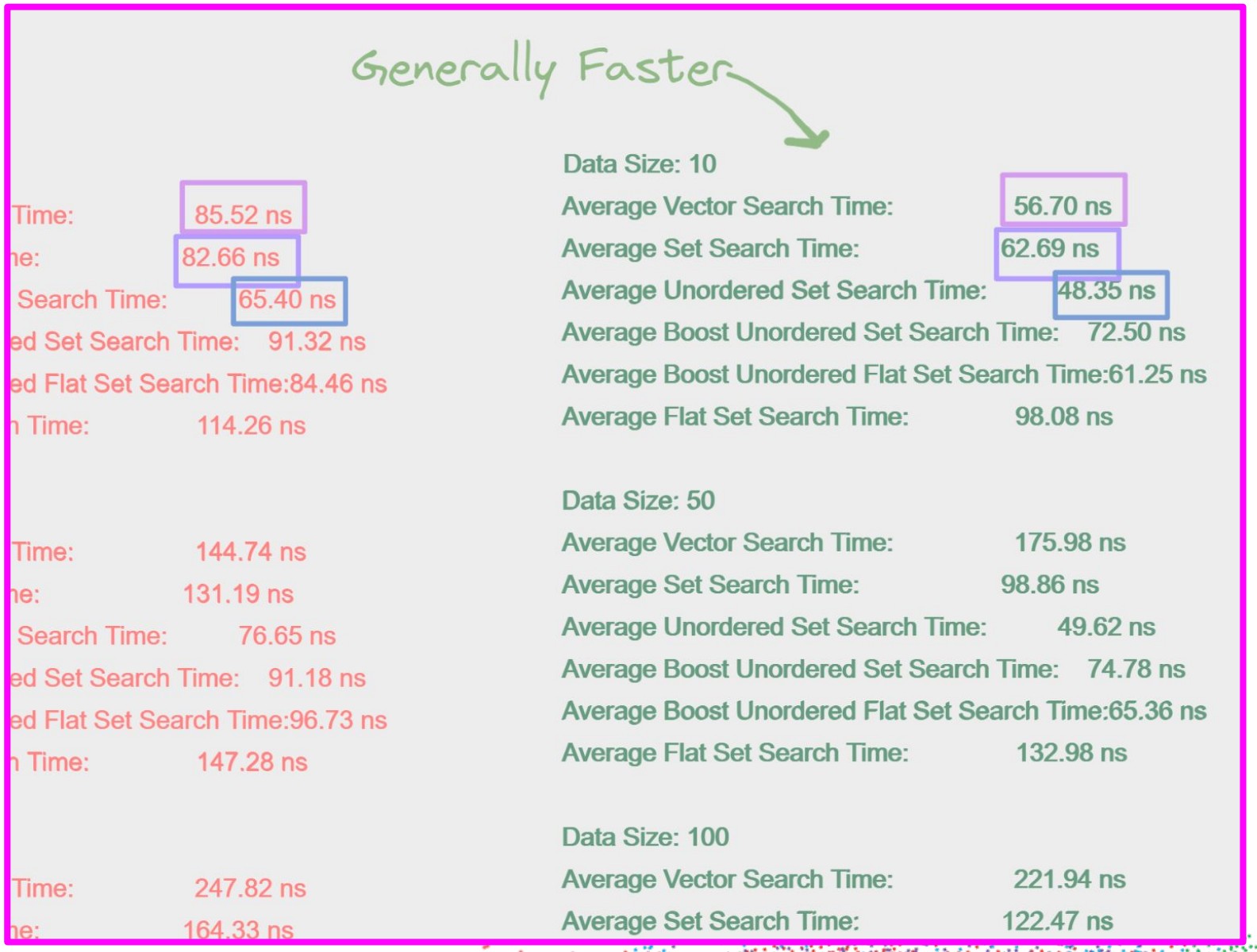
For large N,
boost::unordered_flat_set
<= std::unordered_set
<= boost::unordred_set
<= boost::flat_set
<= std::set
<= std::vector



However, the contiguous data structures start becoming better at larger N!

Although boost::unordered_flat_set becomes better at a larger N because the std::set benefits from the warm cache :) !

Without heavy work



Without heavy work == faster results!

All data structures generally performed better when no heavy work!

Ideally - we profile!

Warning: what we just saw are micro-benchmarks

But good to know about them to build intuition!

Also, the harsh reality is that you may not always get time to profile :’)

~~Agenda~~ The Case File: What We'll Be Investigating Today

1. Data Structures & Contiguous Cache Memory
2. Cache Friendly Data Placement
3. The Intersection of Caches & Threads

Agenda – The Case File: What We'll Be Investigating Today

1 Data & Caches

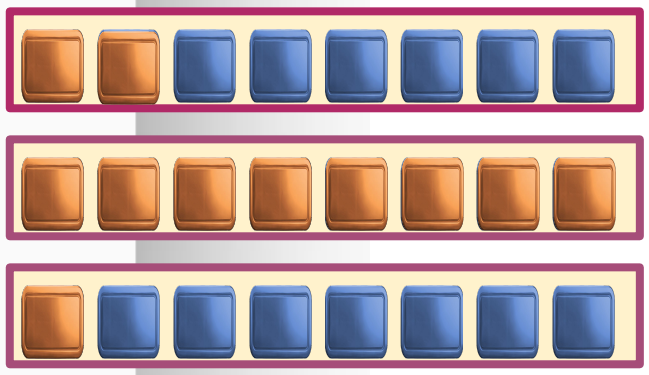
2 Cache Friendly Data Placement

3 Caches & Threads

Declaration Order

```
struct Good {
    char a;
    char c;
    double b;
};
```

Total size = **16 bytes**



1 byte for a, then 7 byte padding

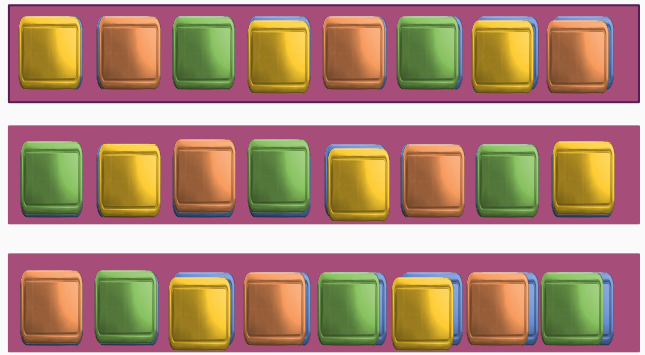
8 bytes for b

1 byte for c, then 7 byte padding

How do you improve this?

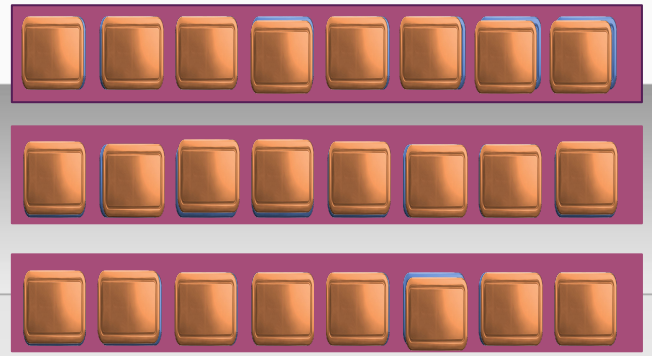
Array of Structs vs. Struct of Arrays (AoS vs. SoA)

```cpp
struct Detective {
    double age;
    double numberOfSolvedCases;
    double classificationLevel;
};

std::vector<Detective> detectives;
```



Array of Structs vs. Struct of Arrays (AoS vs. SoA)

```
struct Detectives {
    std::vector<double> age;
    std::vector<double> numberOfSolvedCases;
    std::vector<double> classificationLevel;
};
```

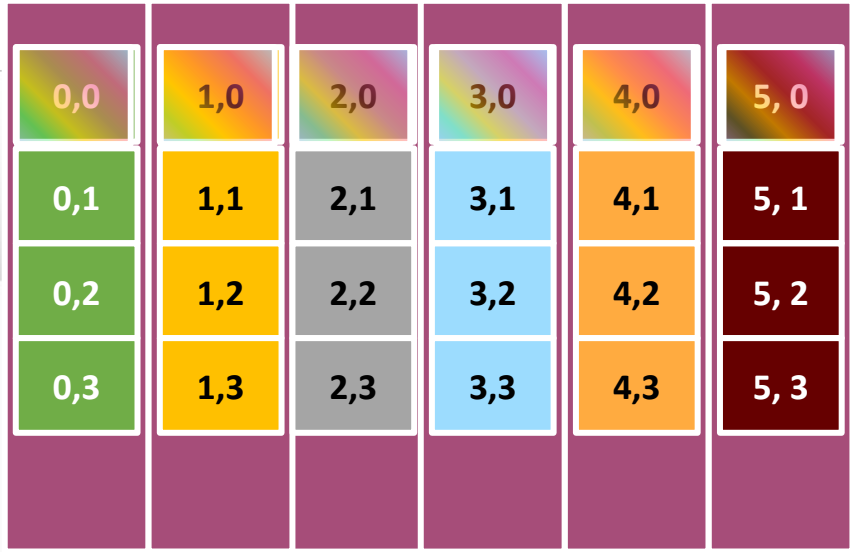


Fits more in!

Less Cache Misses*

Row Wise Over Column Wise

mat[i][j]




```
int mat[1000][1000];
for (int j = 0; j < 1000; ++j) {
    for (int i = 0; i < 1000; ++i) {
        mat[i][j] = 1;
    }
}
```

Next i

Source: Are.na (https://www.are.na.com)

Row Wise Over Column Wise


```
int mat[1000][1000];
for (int i = 0; i < 1000; ++i) {
    for (int j = 0; j < 1000; ++j) {
        mat[i][j] = 1;
    }
}
```

... more i

Next i

Declaration Order Matters … but trust no one…

```
volatile int x = rand();
heavyWork();
int y = x + 5;
```

```
volatile int x = rand();
int y = x + 5;
heavyWork();
```

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Even Declaration Order Matters!

```
void scenarioA() {
    volatile int x = 42;
    doHeavyWork(); // Pollute Cache

    auto start = std::chrono::high_resolution_clock::now();
    int y = x + 1;
    auto end = std::chrono::high_resolution_clock::now();

    // print time taken
    // prevent y to prevent optimization
}
```

A: Declare variable → heavy cache work → access variable

Source: Giphy (https://media4.giphy.com)

Even Declaration Order Matters!

```
void scenarioB() {
    volatile int x = 42;
    auto start = std::chrono::high_resolution_clock::now();
    int y = x + 1;
    auto end = std::chrono::high_resolution_clock::now();

    doHeavyWork(); // Pollute Cache

    // print time taken
    // prevent y to prevent optimization
}
```

B: Declare variable → access variable → heavy cache work

Source: Giphy (https://media4.giphy.com)

Even Declaration Order Matters!

```
int main() {
    scenarioA();
    scenarioB();
    return 0;
}
```

A: Declare variable → heavy cache work → access variable

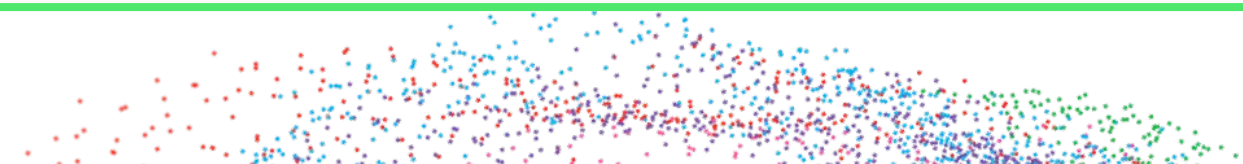
B: Declare variable → access variable → heavy cache work

Even Declaration Order Matters!

Run	Scenario A Time (ns)	Scenario B Time (ns)	Speedup (A → B)
1	36	28	1.29×
2	48	23	2.09×
3	43	34	1.26×

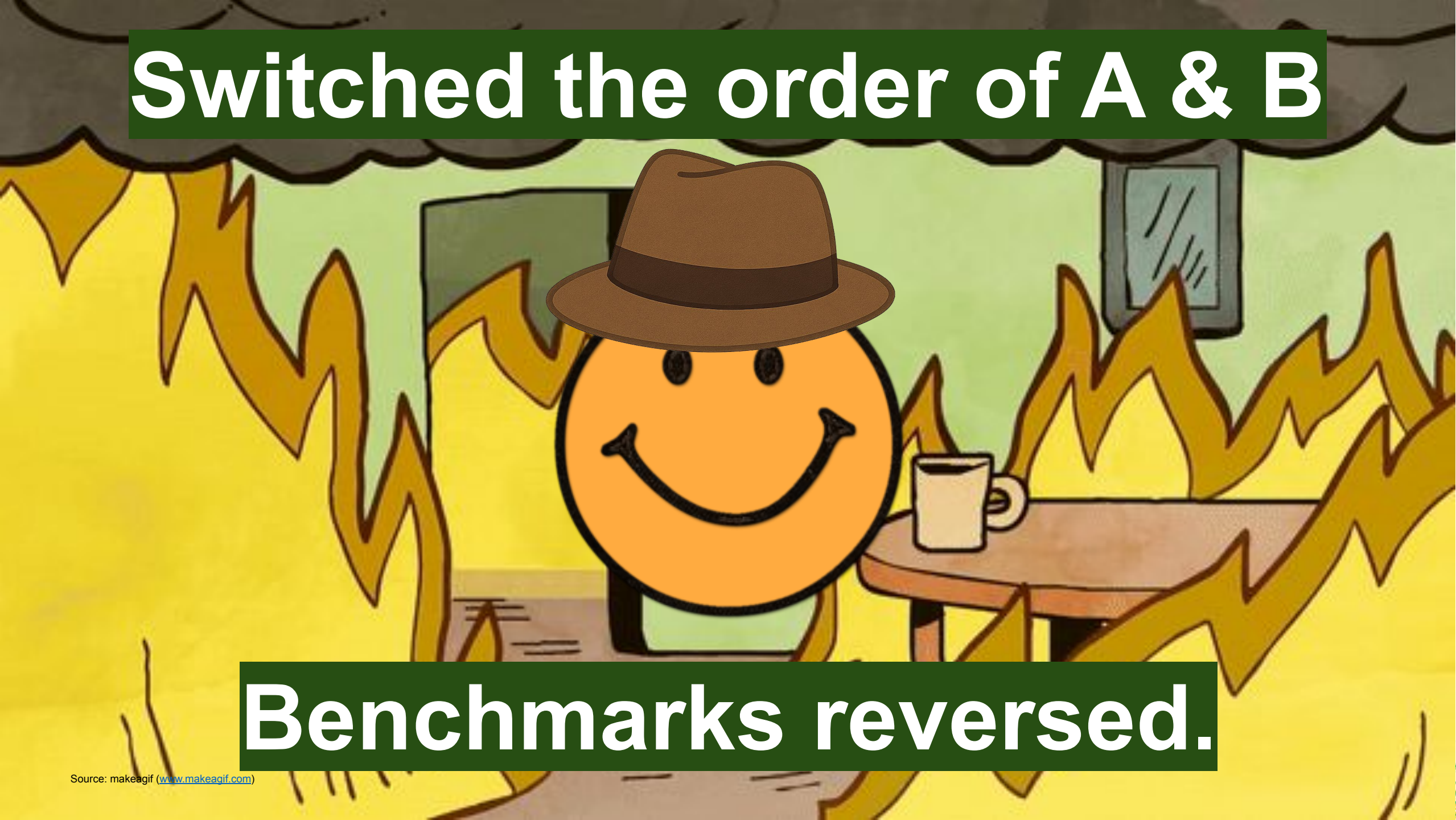
A: Declare variable → heavy cache work → access variable

B: Declare variable → access variable → heavy cache work



Switched the order of A & B

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.



Source: makeagif (www.makeagif.com)

What Just Happened? Why did A get faster than B when B was run first?

Any guesses?

```
int main() {
    scenarioA();
    scenarioB();
    return 0;
}
```

```
int main() {
    scenarioB();
    scenarioA();
    return 0;
}
```

A: Declare variable → heavy cache work → access variable

B: Declare variable → access variable → heavy cache work

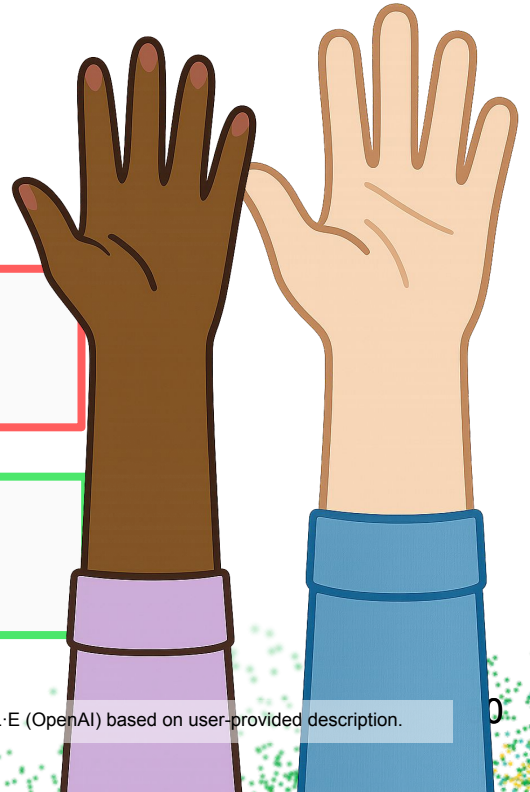
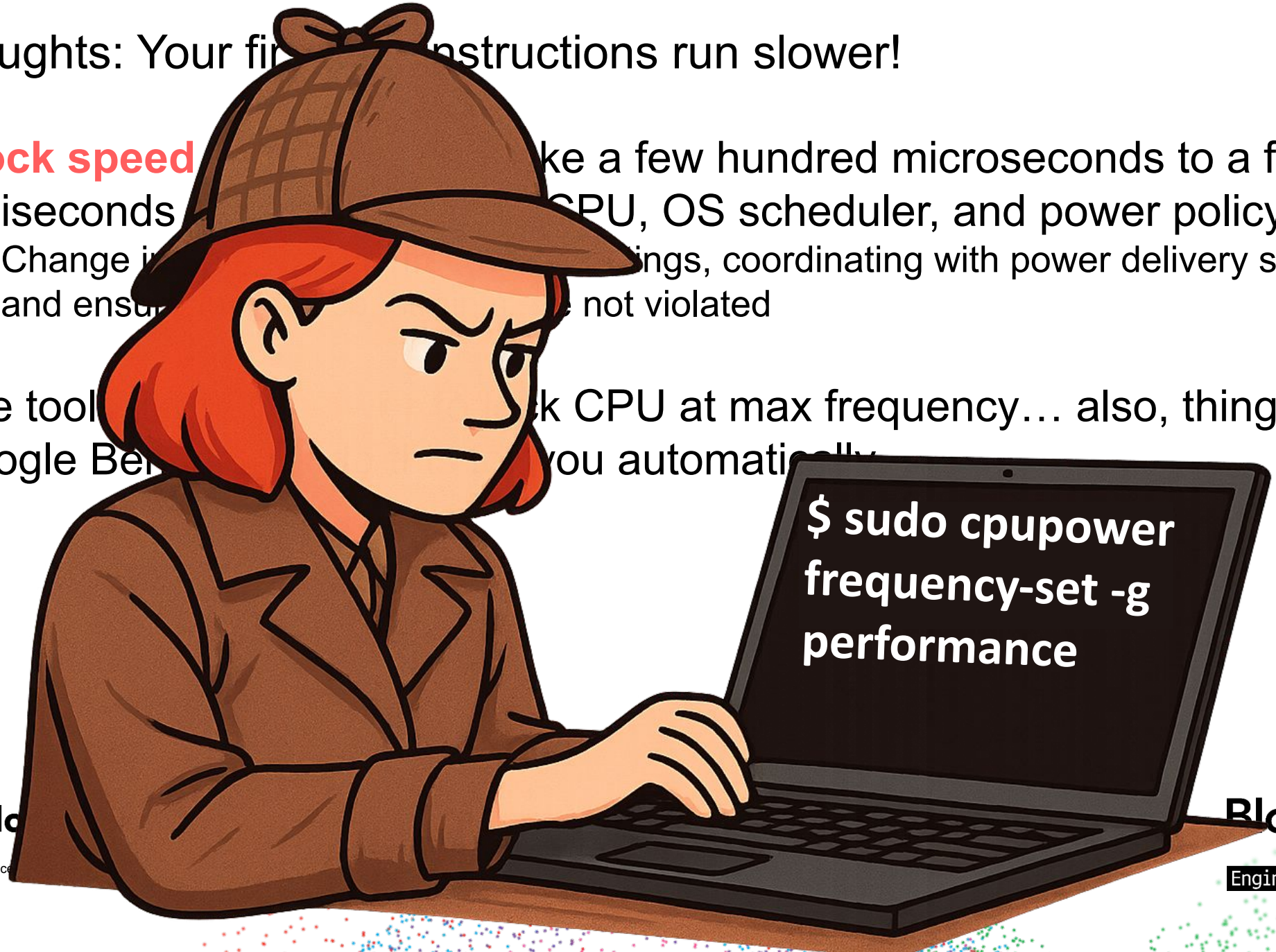


Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

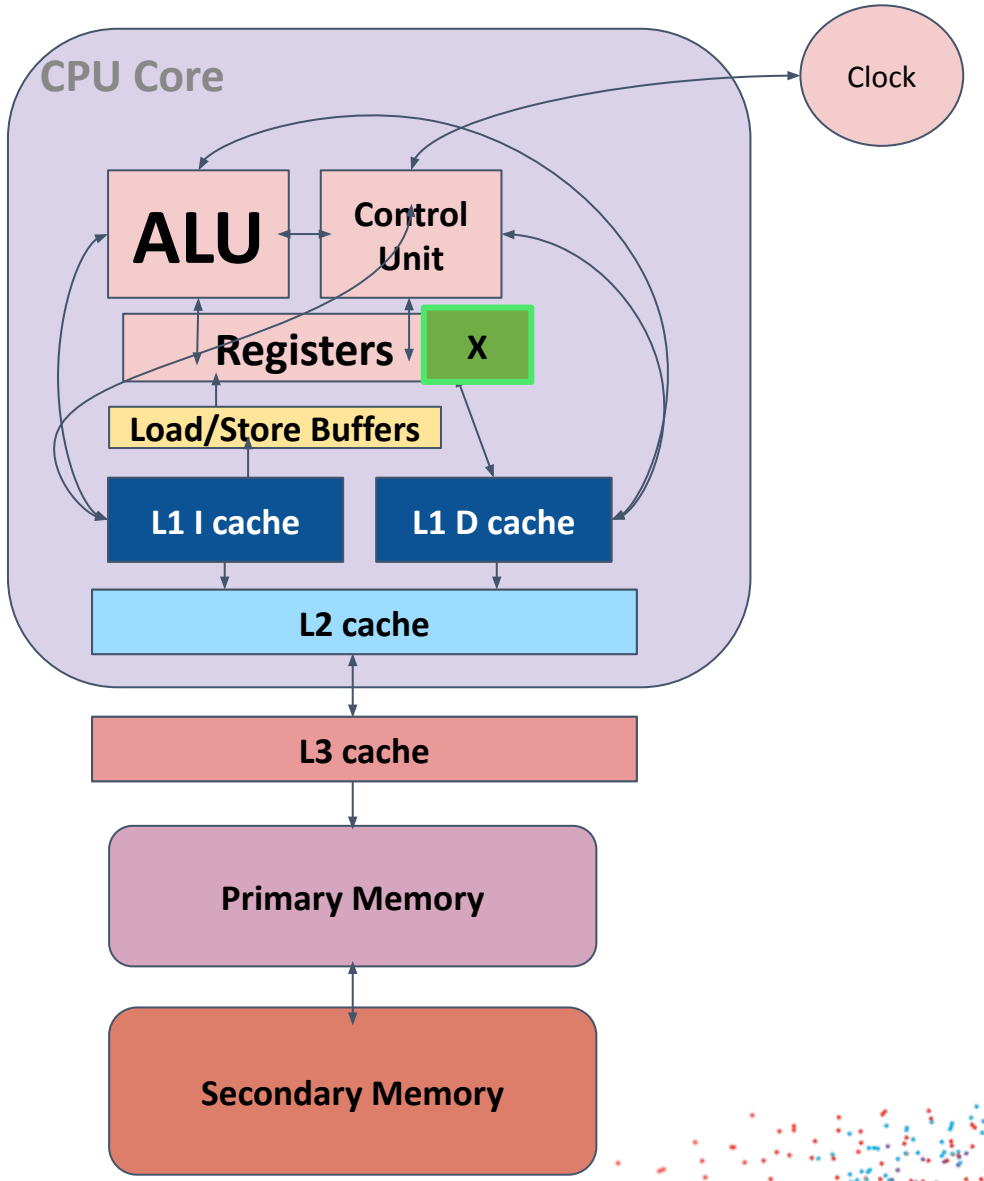
My thoughts: Your fi nstructions run slower!

- **Clock speed** ke a few hundred microseconds to a few milliseconds PU, OS scheduler, and power policy
  - Change i ings, coordinating with power delivery systems, and ensu not violated

- Use tool k CPU at max frequency… also, things like Google Be ou automati



Still no clear winner! The evidence was inconclusive



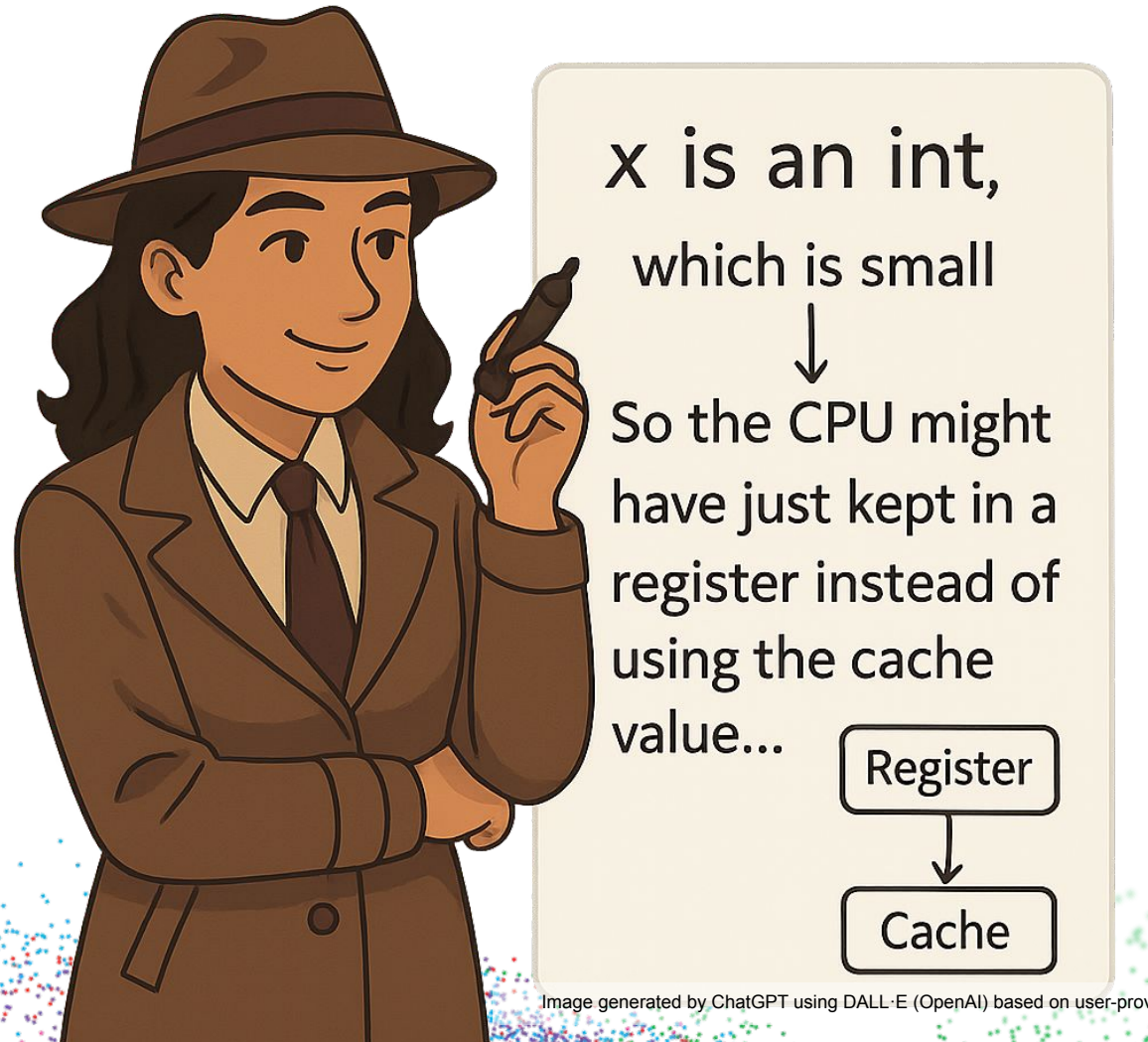
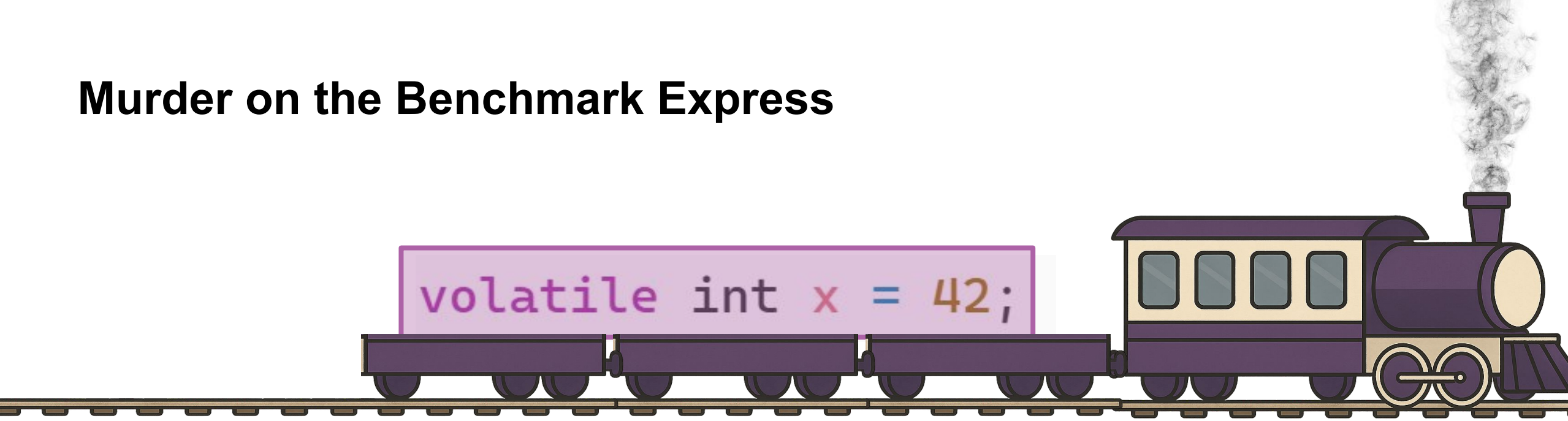


Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Murder on the Benchmark Express



“This variable might be changed or accessed outside the current code's control. My dear compiler, do not optimize access to it.” - **volatile**

It **can interfere with realistic behavior** (e.g., inhibits certain optimizations like register usage).

benchmark::DoNotOptimize(x) prevents something from being optimized away by the compiler in a more controlled and performance-safe way.

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Tried Making x BIG

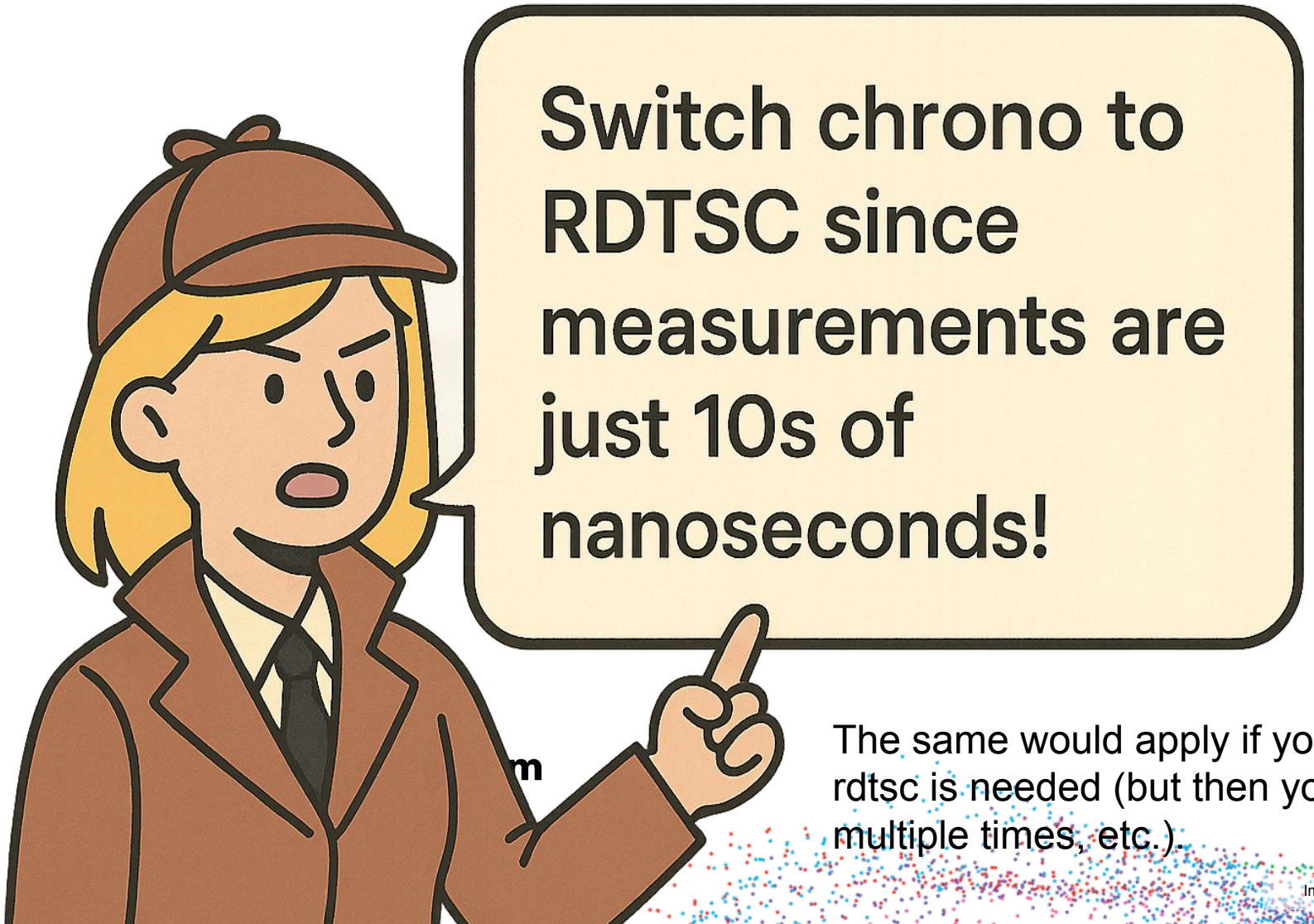
std::vector<size_t> x , with 4,194,304 elements (4 * 1024 * 1024).

But still no clear winner!

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

TechAtBloomberg.com

Switch to RDTSC



The same would apply if you used Google Benchmark … the switch to rdtsc is needed (but then you need to make sure you are running things multiple times, etc.).

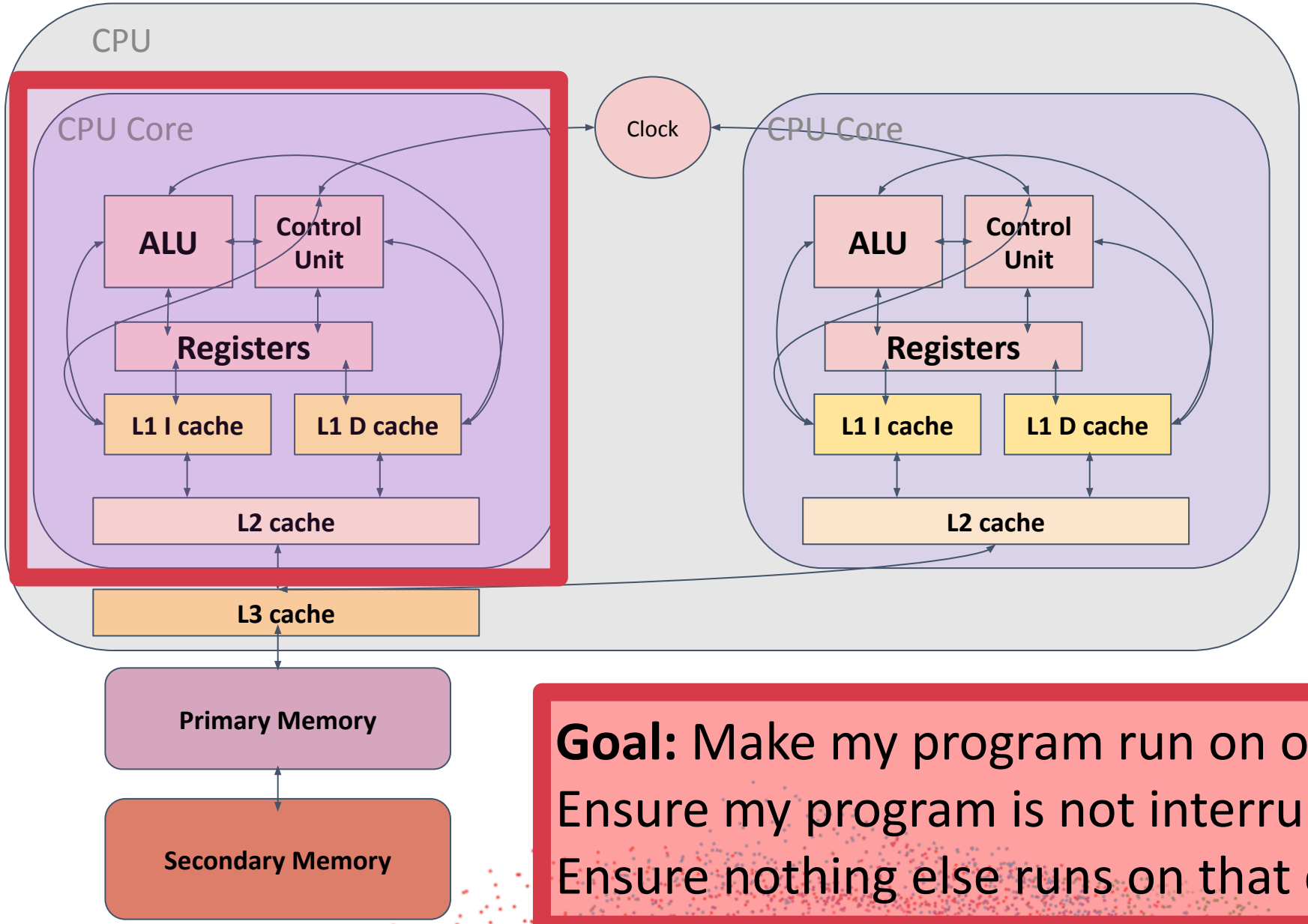
Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

STILL NO CLEAR WINNER!

AAAAAAAAAAAAAAHHHHHHHHHHHHHHHHHHHH!!!!!!!!!!!!

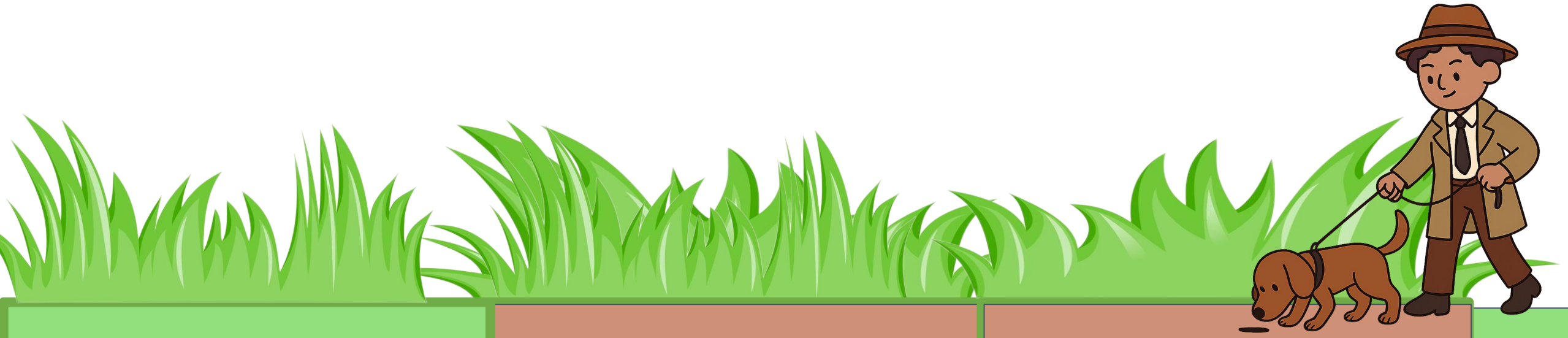
TechAtBloomberg.com

First Rule of being a Detective? Trust no one…



Goal: Make my program run on only one core.
Ensure my program is not interrupted mid run.
Ensure nothing else runs on that core.

Make sure the task is run on only one core and nothing else interrupts it!



BIOS > SMT off

Simultaneous Multi-Threading off prevents multiple threads from running on the same **physical core.**

With SMT off, each physical core exposes one logical core.

ONLY 1 PROGRAM WILL RUN ON THIS PHYSICAL CORE

THE OS DOESN'T RUN ITS STUFF ON THIS CORE + DOESN'T INTERRUPT ANY TASK ON IT

RUN MY PROGRAM ON THIS CORE

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Inconsistent!! Even though they aren’t too different!

In terms of cycles!

Run	Avg Cycles A	Avg Cycles B	Winner
1	51.19M	51.84M	**B**
2	58.94M	59.81M	**A**
3	59.24M	59.41M	**A**
4	58.79M	56.81M	**B**

Can the Compiler Move the Declaration of x?

Assign values to vector x
heavyWork();
Use values in vector x

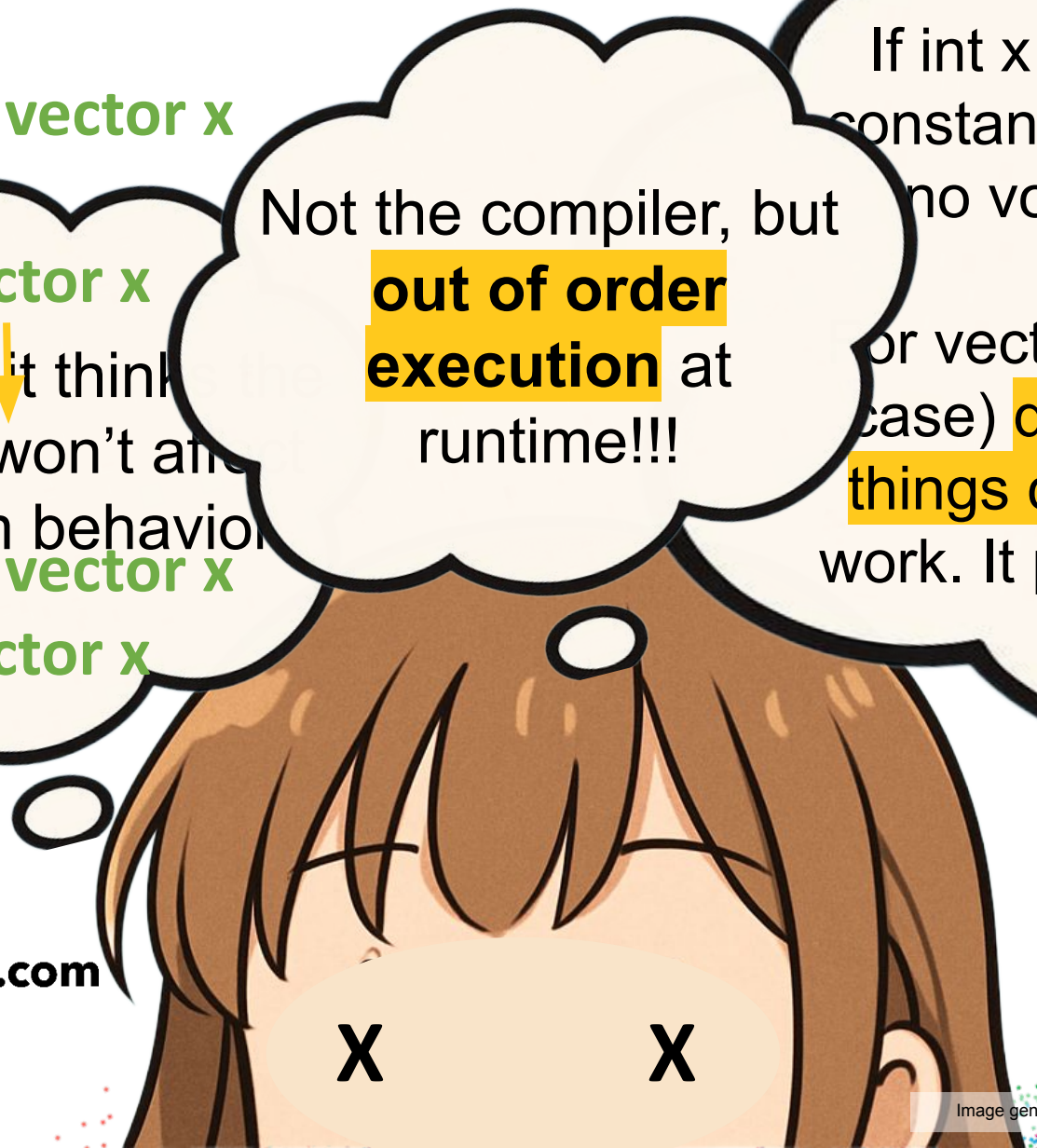
… only if it thinks the change won't affect program behavior

heavyWork();
Assign values to vector x
Use values in vector x

Not the compiler, but **out of order execution** at runtime!!!

If int x = 42; yes it will move the constant automatically if there was no volatile (constant folding).

For vector, the compiler (gcc in my case) did not automatically move things closer together w.r.t heavy work. It preserved the source order as per assembly.



TechAtBloomberg.com

Bloomberg
Engineering

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Out of Order Execution…

Can’t see in assembly directly, since it’s done at runtime!

But can identify candidates!

```
mov eax, [rdi]          ; load from memory (slow)
add ebx, 1              ; independent (can go OoO)
imul ecx, 5             ; independent (can go OoO)
add edx, eax            ; must wait for load
```

Out of Order Execution…

```
int a = load_from_memory();   // SLOW
int b = 5 + 2;                // FAST
int c = b * 3;                // FAST
int d = a + c;
```

P.s. you can’t see this directly in assembly since it is done by the CPU for independent instructions.

Driven by the CPU’s:

- **Instruction scheduler**
- **Reservation stations**
- **Reorder buffer (ROB)**
- **Dependency tracking logic**

OoO will kick in … depending on the platform

x86/x86-64 (Intel & AMD)
OoO happens!

But not for legacy/older CPUs like Intel 486 & Pentium (original).

PowerPC
OoO happens!

RISC-V
Depends on implementation

MIPS
Not for many embedded MIPS chips (E.g. MIPS32)

ARM
OoO for Cortex-A72, A75, A76, A78, X1, X2; Apple M1/M2 (ARM64); Cortex-A510 (ARMv9)!

Not for embedded microcontrollers, like the Cortex-A53, A55 cpu lines (energy efficient cores)

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

How to test the OoO hypothesis?

```
std::vector<int> x = {1, 2, 3, 4, 5, …}; // With benchmark::DoNotOptimize(x);

heavyWork();

// Use values in vector x
```

```
heavyWork();

std::vector<int> x = {1, 2, 3, 4, 5, …}; // With benchmark::DoNotOptimize(x);

// Use values in vector x
```

Bloomberg

Engineering

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

How to test the OoO hypothesis? Memory fences!

```
std::vector<int> x = {1, 2, 3, 4, 5, …};
_mm_mfence();
heavyWork();
_mm_mfence();
// Use values in vector x
```

```
heavyWork();
_mm_mfence();
std::vector<int> x = {1, 2, 3, 4, 5, …};
_mm_mfence();
// Use values in vector x
```

Bloomberg

Engineering

Image generated by ChatGPT using DALL·E (OpenAI) based on user-provided description.

Consistent results, finally!

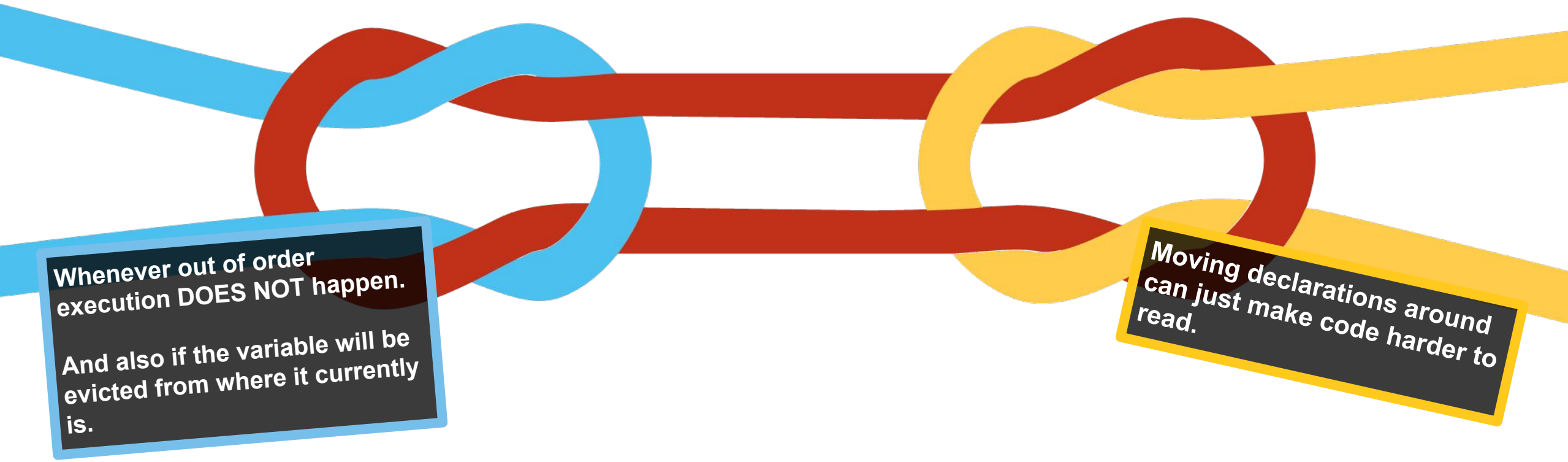
Run	Avg Cycles A	Avg Cycles B	Winner
1	63.74M	59.70M	B
2	65.64M	64.07M	B
3	66.67M	60.84M	B
4	64.09M	62.03M	B

B faster than A

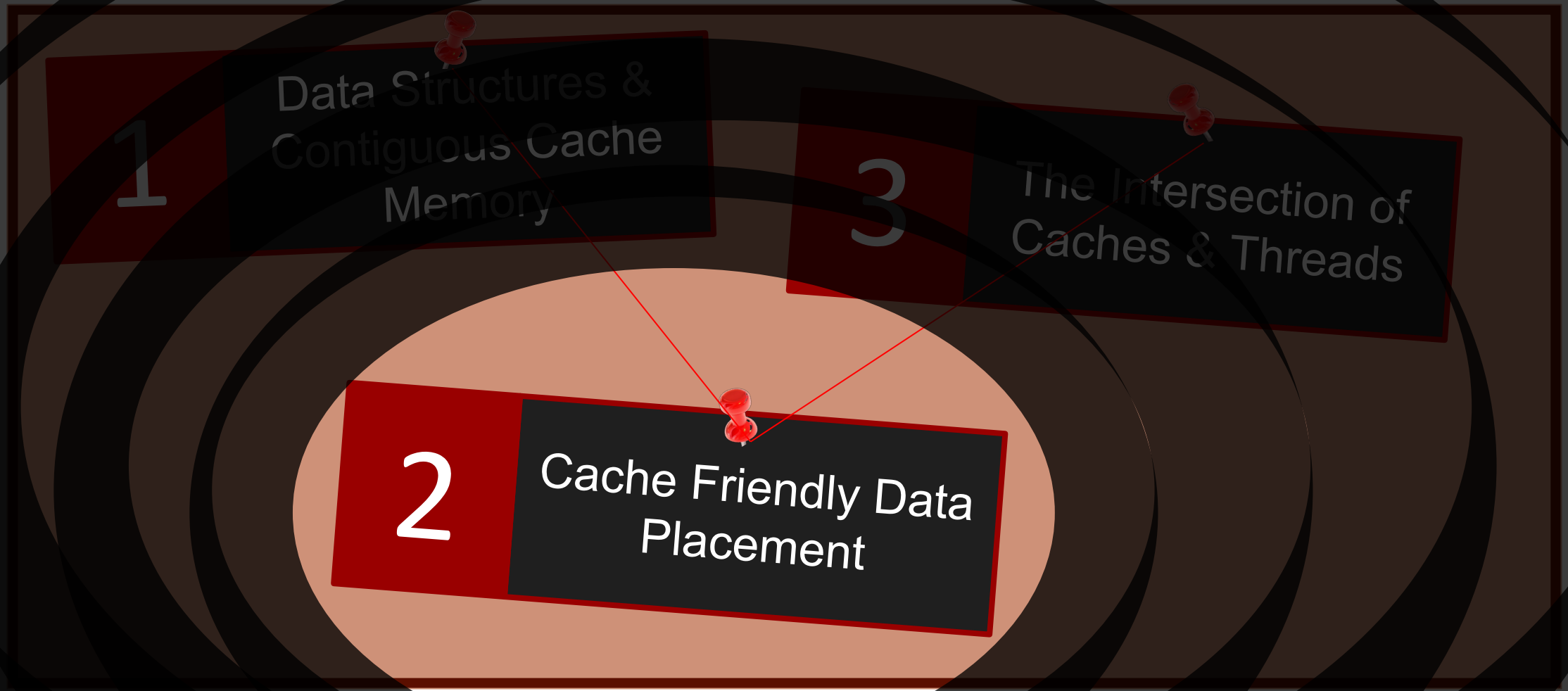
(Irrespective of which is run first)

But more clock cycles than before :’)

Variable Order Matters*



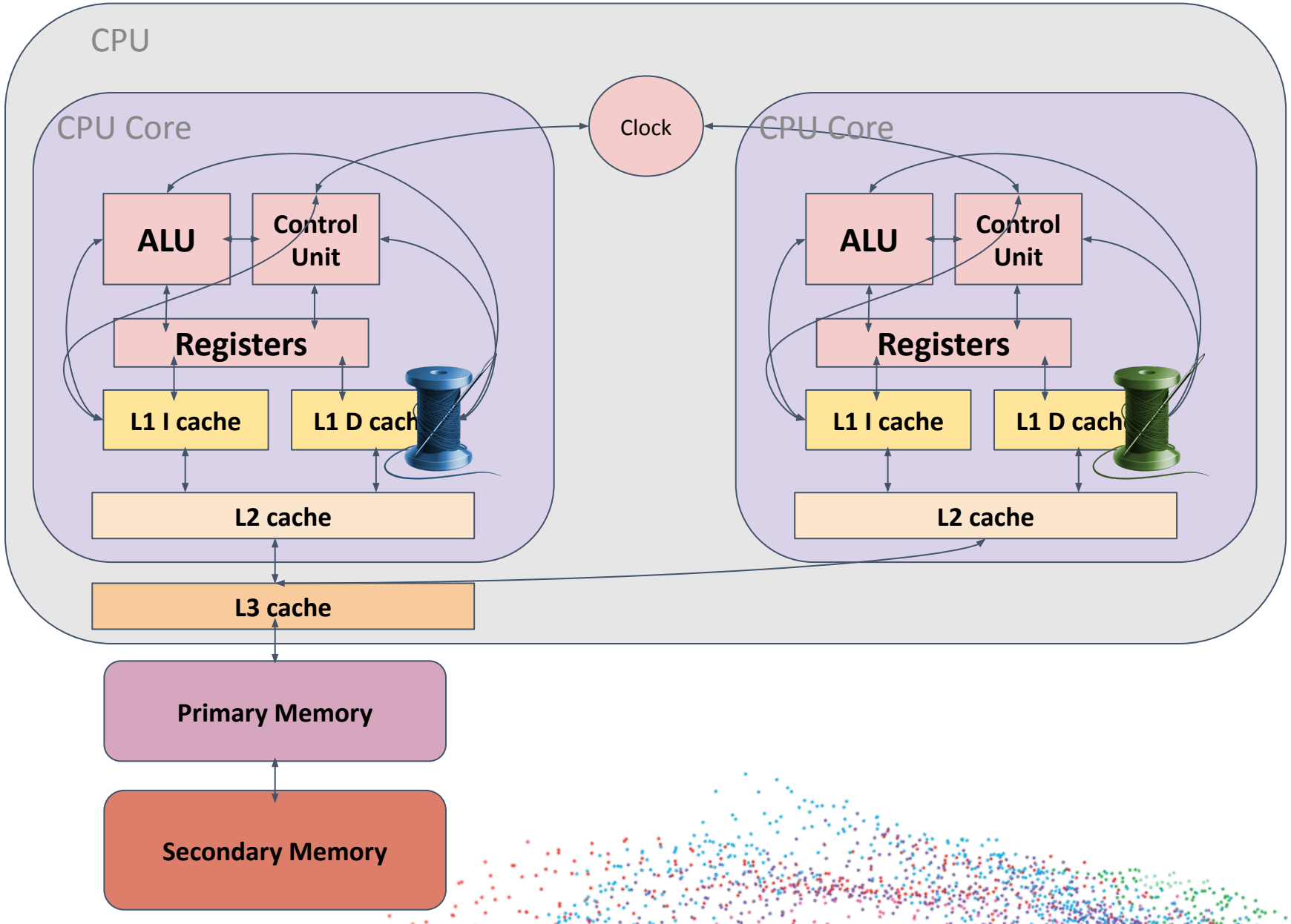
Agenda – The Case File: What We'll Be Investigating Today



Agenda – The Case File: What We'll Be Investigating Today

1. Data Structures & Contiguous Cache Memory
2. Cache Friendly Data Placement
3. The Intersection of Caches & Threads

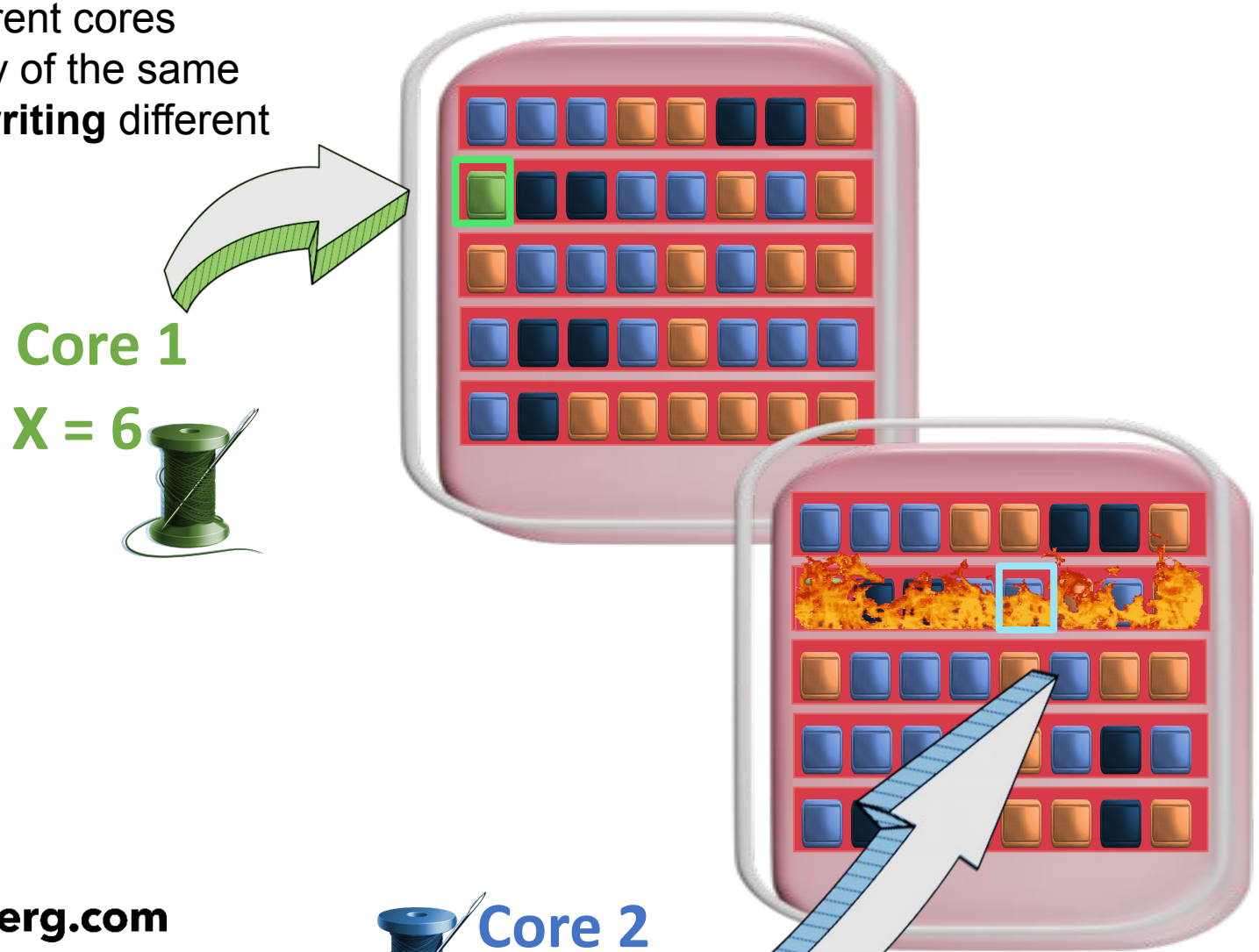
First Rule of being a Detective? Trust no one…



Source: pngarts (https://www.pngarts.com)

False Sharing

Threads on different cores accessing a copy of the same cache line, but **writing** different variables



Source: Are.na (https://www.are.na.com)

Source: Vecteezy (www.vecteezy.com)

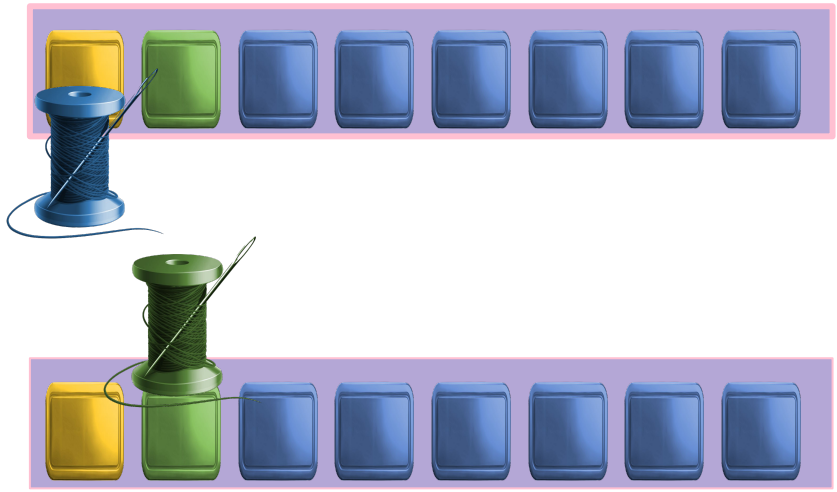
Source: pngarts(https://www.pngarts.com)

False Sharing will happen when you have

- Multiple cores
- Sharing the same cache line
- At least one core writes a variable on that shared cache line.

Common Scenarios for False Sharing

Counters/Flags: A common pattern where each thread has its own counter or flag, but these are laid out contiguously in memory.



Thread 1 Counter	Thread 2 Counter	Thread 3 Counter	Thread 4 Counter

A counter that will be accessed by multiple threads

```
struct Counter {
    volatile int value = 0;
};

std::array<Counter, num_threads> counters;
```

Thread 1 Counter	Thread 2 Counter	Thread 3 Counter	Thread 4 Counter

A counter that will be accessed by multiple threads

Compiler	Availability	Notes
GCC	✅ GCC 8+	Requires <new> and -std=c++17 or newer
Clang	✅ Clang 7+ with libc++ 7+	Must use libc++. Not guaranteed with libstdc++
MSVC	✅ VS 2017 15.7+	Provided in the standard library from Visual Studio 2017 Update 7

```
ze

std::array<PaddedCounter, num_threads> paddedCounters;
```

Removing false sharing improved performance by ~67.5%

Let’s try cachegrind!

valgrind --tool=cachegrind ./no_false_sharing
valgrind --tool=cachegrind ./false_sharing

Metric	No False Sharing	False Sharing	Difference / Observation
Time	2131 ms	1961 ms (faster)	Surprisingly, false sharing was faster here
D1 misses	14,912	14,909	Nearly identical
LLd misses	9,373	9,371	Nearly identical
LL misses (total)	11,597	11,591	Nearly identical

A detective is just as good as their tool

Cachegrind is a **simulator**.

It **doesn't simulate true multi-core hardware effects** like cache line contention and bus invalidation, which are the heart of false sharing problems.

P.S., it is super useful for simulating things in non-hardware/thread specific cases though.

How to fix false sharing?

Data Structure Reorganization

Reorder members within structs or classes to place frequently accessed/modified fields on different cache lines.

Padding & Alignment

Local Variables on the Thread

For temporary variables, local to only the thread. Each thread has a separate stack, so it prevents, doesn't prohibit false sharing.

Thread-Local Storage

Global/static storage, independent for each thread. Prevents, doesn't prohibit. However thread-local is bad in general imo.

TechAtBloomberg.com

Tools to Profile the Behavior of your Cache

Tool	Platform	Real-Time	Cache Metrics	Notes
perf	Linux	✅	✅	Most flexible CLI tool
Intel VTune	Linux/Win	✅	✅✅✅	Deepest insights (L1→LLC, TLB)
AMD uProf	Linux/Win	✅	✅	Best for AMD CPUs
Cachegrind	Linux/macOS	❌ (sim)	✅	Great for static analysis
Visual Studio Profiler	Windows	✅	✅ (limited)	Integrated with IDE
Instruments (with xcode)	macOS	✅	⚠️ Limited	Visual, but lacks deep hardware data
LIKWID	Linux	✅	✅	Lightweight and easy

In Summary - You've CACHED it!

Contig **A**void f **C**ache **H**eed th **E**fficien **D**ata access patterns are everything

Predictable, sequential access, with contiguous memory is the way forward!

Please trust other people…
but always verify things yourself to build understanding!

TechAtBloomberg.com

Bloomberg

Engineering

Some other great related talks!

- CppCon 2017: Chandler Carruth “Going Nowhere Faster”
- CppCon 2016: Timur Doumler “Want fast C++? Know your hardware!"
- CppCon 2014: Mike Acton "Data-Oriented Design and C++"
- C++ on Sea 2024: Björn Fahller “C++ Cache Friendly Data + Functional + Ranges = ❤️”
- Meeting C++ 2018: Jonathan Müller - Writing cache friendly C++

Hope your attention cached this talk ... time to get evicted!

Please write back to main memory!

Thank You!

linkedin.com/in/michellefae/

michellefae.github.io/

Special Thanks:
Chris Cotter, Dietmar Kühl, Sherry Sontag, Jess Winer, Alex Lo, & Bloomberg's C++ Guild for helping me rehearse my talk!

TechAtBloomberg.com

Bloomberg

Engineering