Cache-Friendly C++

JONATHAN MÜLLER

Over two decades of experience building a market-leading C++ add-in for Microsoft Office

- think-cell is the gold standard for creating professional PowerPoint presentations.
- Founded in 2002, we now have 1,200,000+ users at over 30,000 companies.
- Our software is built entirely in C++ and fully integrated into Microsoft Office.
- We are a member of Standard C++ Foundation, run the Berlin C++ Meetup and talk at conferences.
- We are driven by excellence, setting ambitious goals and fostering the growth of exceptional talent.

We're hiring: think-cell.com/career/dev

Disclaimer

think-cell

This isn’t a talk on how to properly benchmark!

- All benchmarks are deliberately written to get a particular result.
- All benchmarks are executed on a machine where it demonstrates that result.
- The effects are real, the results are true, but the benchmark is gamed to achieve them.

Disclaimer

think-cell

This isn't a talk on how to properly benchmark!

- All benchmarks are deliberately written to get a particular result.
- All benchmarks are executed on a machine where it demonstrates that result.
- The effects are real, the results are true, but the benchmark is gamed to achieve them.

Always benchmark yourself on your target hardware to make optimization decisions.

Motivation

How to write a set?

think-cell

1. `std::set`: Binary search tree
   - `O(log n)` lookup: tree traversal
   - `O(log n)` insertion: tree traversal
   - `O(n log n)` bulk insert: n insertions

2. `std::unordered_set`: Hash table
   - `O(1)` lookup: hash index
   - `O(1)` insertion: hash index
   - `O(n)` bulk insert: n insertions

How to write a set?

think-cell

1. `std::set`: Binary search tree
   - `O(log n)` lookup: tree traversal
   - `O(log n)` insertion: tree traversal
   - `O(n log n)` bulk insert: n insertions

2. `std::unordered_set`: Hash table
   - `O(1)` lookup: hash index
   - `O(1)` insertion: hash index
   - `O(n)` bulk insert: n insertions

3. Unsorted `std::vector`: Linear search array
   - `O(n)` lookup: linear search
   - `O(1)` insertion: `.push_back()`
   - `O(n)` bulk insert: n insertions

How to write a set?

think-cell

1. `std::set`: Binary search tree
   - `O(log n)` lookup: tree traversal
   - `O(log n)` insertion: tree traversal
   - `O(n log n)` bulk insert: n insertions

2. `std::unordered_set`: Hash table
   - `O(1)` lookup: hash index
   - `O(1)` insertion: hash index
   - `O(n)` bulk insert: n insertions

3. Unsorted `std::vector`: Linear search array
   - `O(n)` lookup: linear search
   - `O(1)` insertion: `.push_back()`
   - `O(n)` bulk insert: n insertions

4. Sorted `std::vector`: Binary search array
   - `O(log n)` lookup: binary search
   - `O(n)` insertion: binary search + `.insert()`
   - `O(n log n)` bulk insert: append + `std::sort()`

Let's benchmark

think-cell

1. Setup: Fill set with `N` random integers
2. Lookup: Repeatedly lookup random integers

Let's benchmark

think-cell

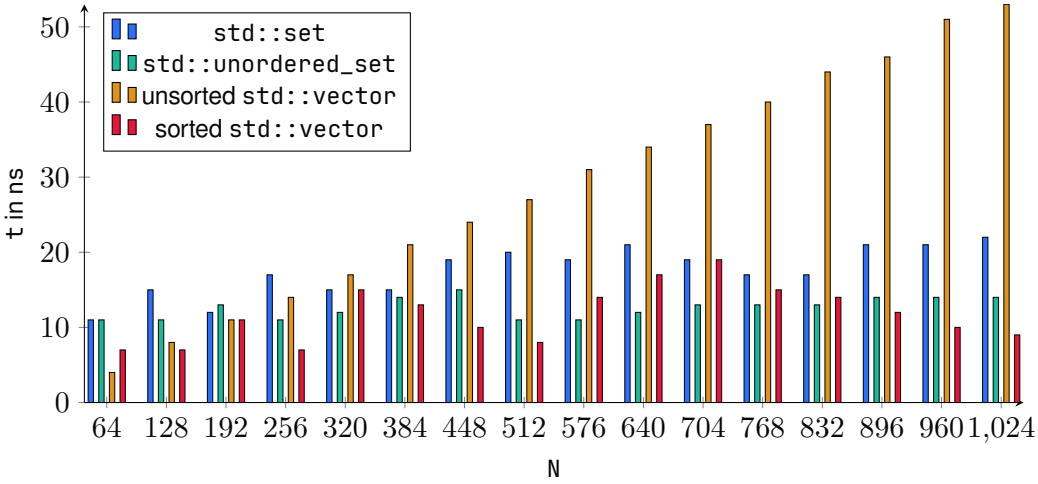
1. Setup: Fill set with `N` random integers
2. Lookup: Repeatedly lookup random integers

Expectations:

1. `std::unordered_set` with `O(1)` lookup
2. `std::set` and sorted `std::vector` with `O(log n)` lookup
3. Unsorted `std::vector` with `O(n)` lookup

Sorted `std::vector` is consistently faster than `std::unordered_map`!

think-cell



CPU caches

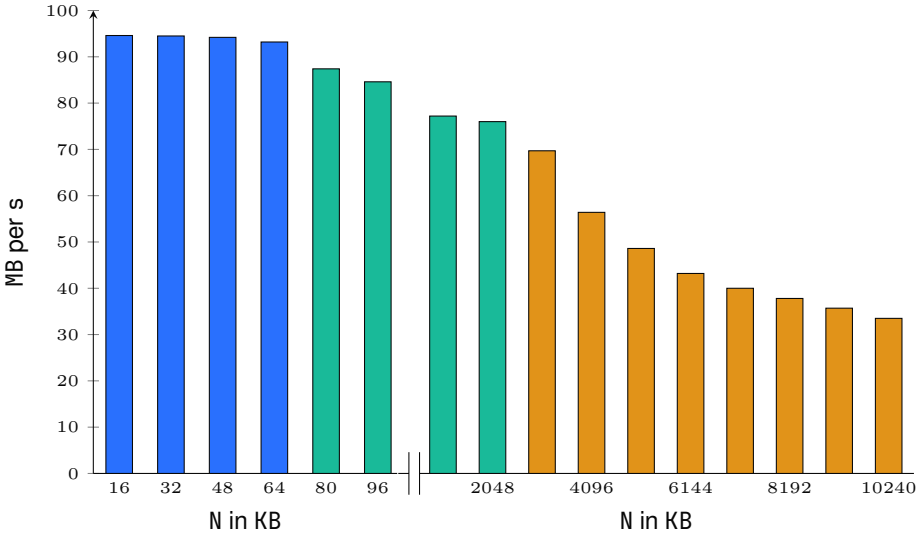
How fast is *random* memory access?

think-cell

```cpp
void bm_memory_access(benchmark::State& state)
{
    std::default_random_engine engine;
    auto bytes = state.range(0) * 1024;

    std::vector<unsigned> data(bytes / sizeof(unsigned));
    stdr::generate(data, engine);

    std::uniform_int_distribution<std::size_t> dist(0, data.size() - 1);
    for (auto _ : state) {
        benchmark::DoNotOptimize(++data[dist(engine)]);
    }

    state.SetBytesProcessed(state.iterations() * data.size());
}
```

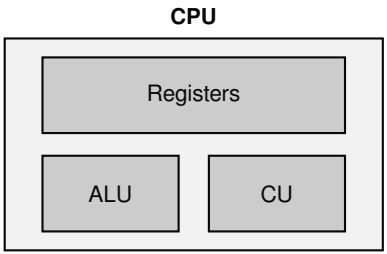
Memory access is slower the more data you have?!

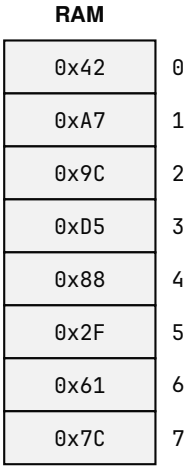
think-cell



Naive memory access

think-cell

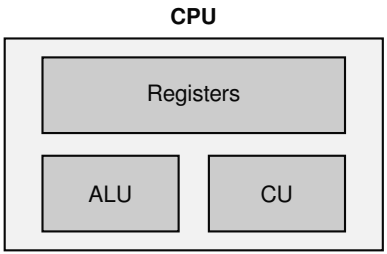


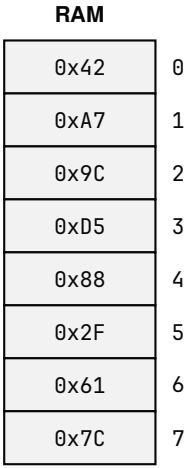



```
load R1, [2]
load R2, [5]
load R3, [2]
```

Naive memory access

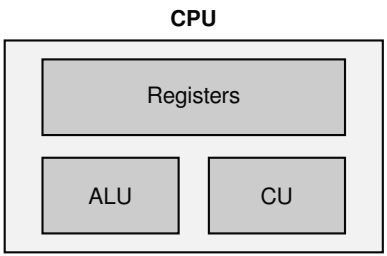
think-cell

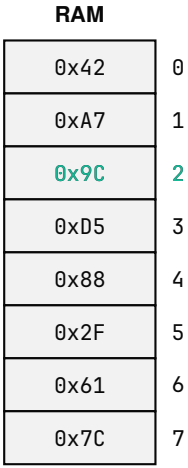





```
▸ load R1, [2]
  load R2, [5]
  load R3, [2]
```

Naive memory access

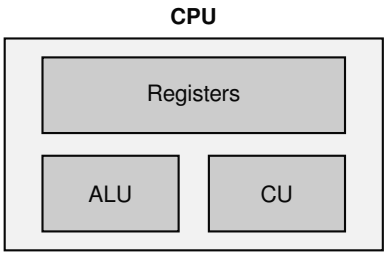




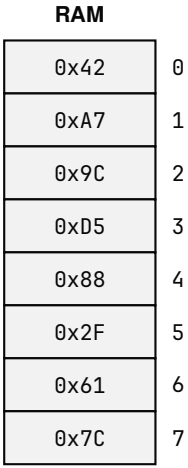

```
load R1, [2]
load R2, [5]
load R3, [2]
```

Naive memory access

think-cell

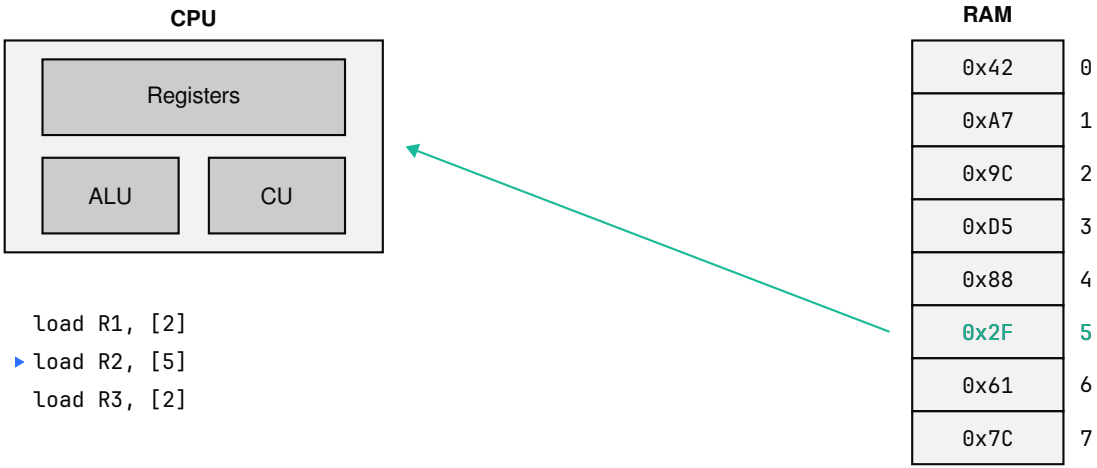



```
  load R1, [2]
▸ load R2, [5]
  load R3, [2]
```



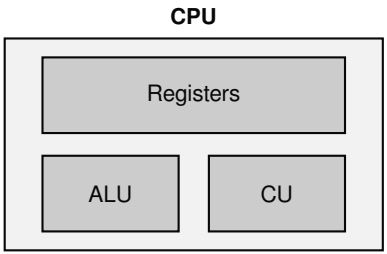
Naive memory access

think-cell

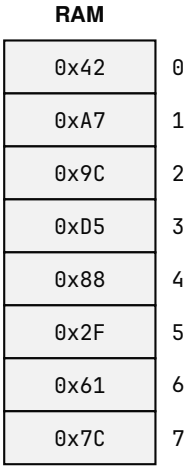


Naive memory access

think-cell

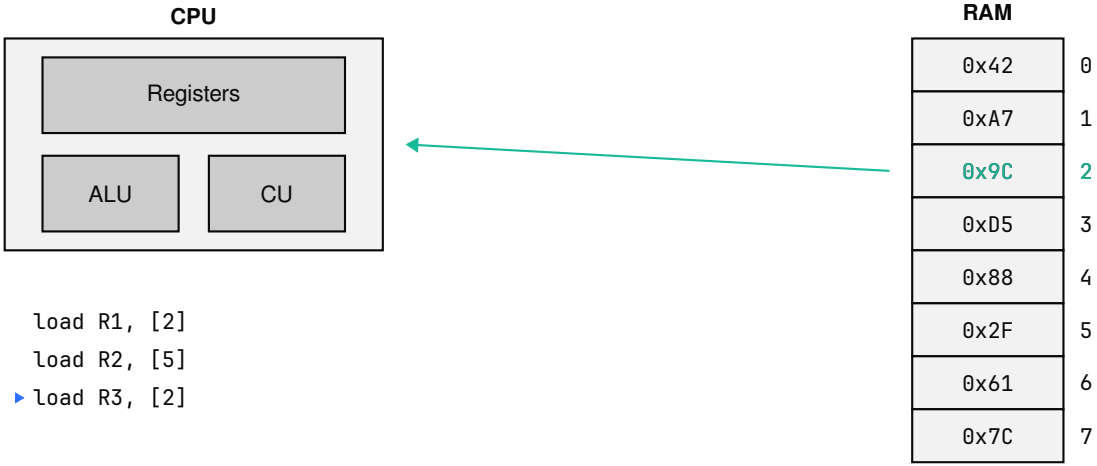



```
  load R1, [2]
  load R2, [5]
▸ load R3, [2]
```



Naive memory access

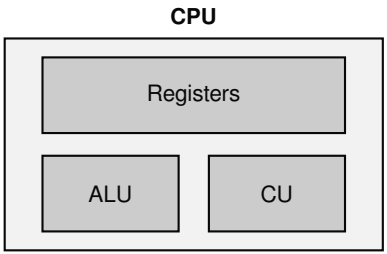
think-cell

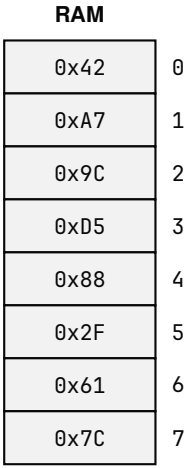



```
load R1, [2]
load R2, [5]
load R3, [2]
```

Naive memory access

think-cell






```
load R1, [2]
load R2, [5]
load R3, [2]
```

Problem: RAM access is *slow*.

Speed of accessing main memory

think-cell

Apple M1 (2020): [1]

- RAM: `68.3 GB/s`

[1] https://en.wikipedia.org/wiki/Apple_M1

Speed of accessing main memory

think-cell

Apple M1 (2020): [1]

- RAM: `68.3 GB/s`
- Compute: `2.6 TFLOPS` → `10650 GB/s`

[1] https://en.wikipedia.org/wiki/Apple_M1

Speed of accessing main memory

think-cell

Apple M1 (2020): [1]

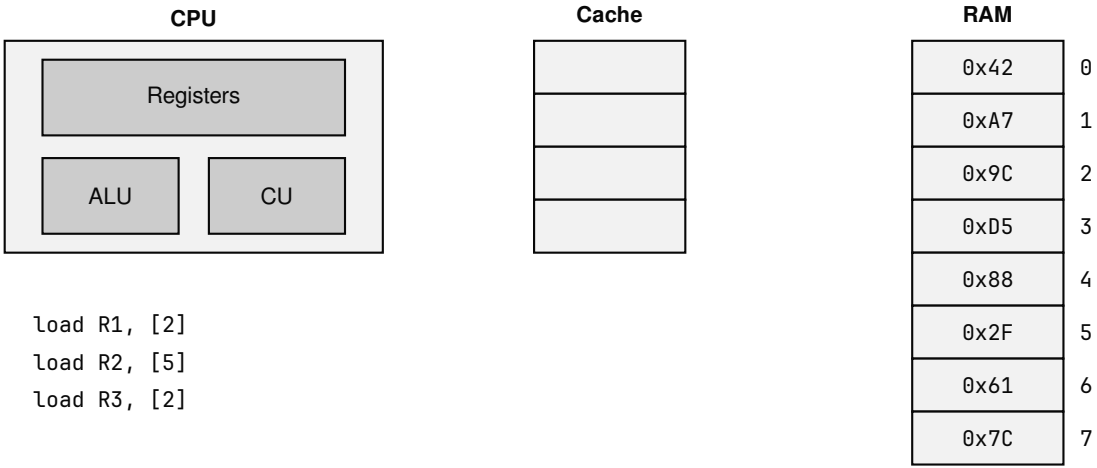
- RAM: `68.3 GB/s`
- Compute: `2.6 TFLOPS` → `10650 GB/s`

Main memory access is 100 times slower than compute!

[1] https://en.wikipedia.org/wiki/Apple_M1

Memory access with caching

think-cell

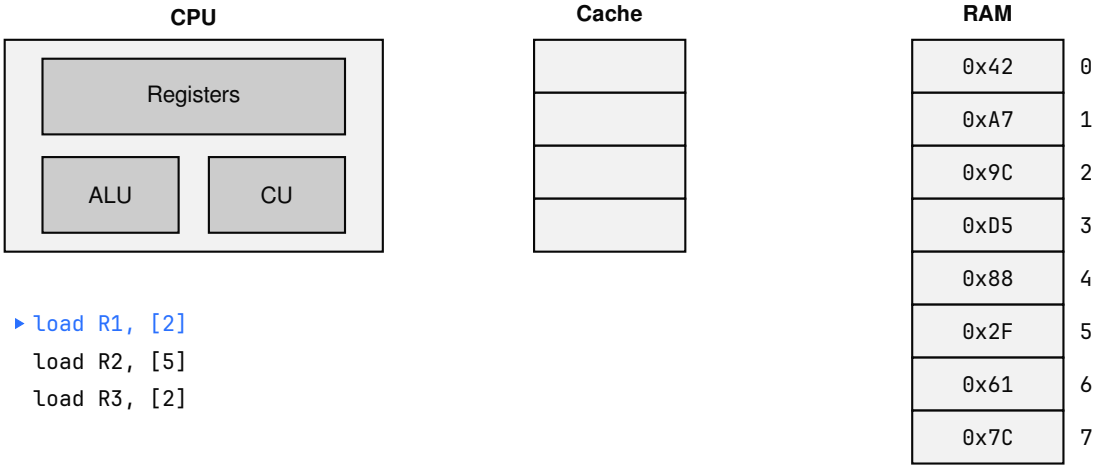



```
load R1, [2]
load R2, [5]
load R3, [2]
```

Solution: Smaller, faster memory close to the CPU serving as cache.

Memory access with caching

think-cell

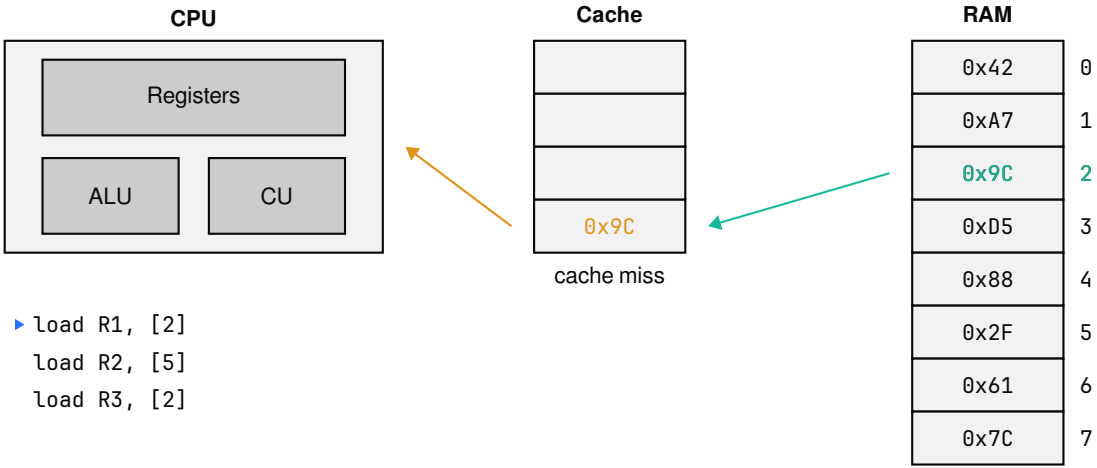



```
▸ load R1, [2]
  load R2, [5]
  load R3, [2]
```

Solution: Smaller, faster memory close to the CPU serving as cache.

Memory access with caching

think-cell

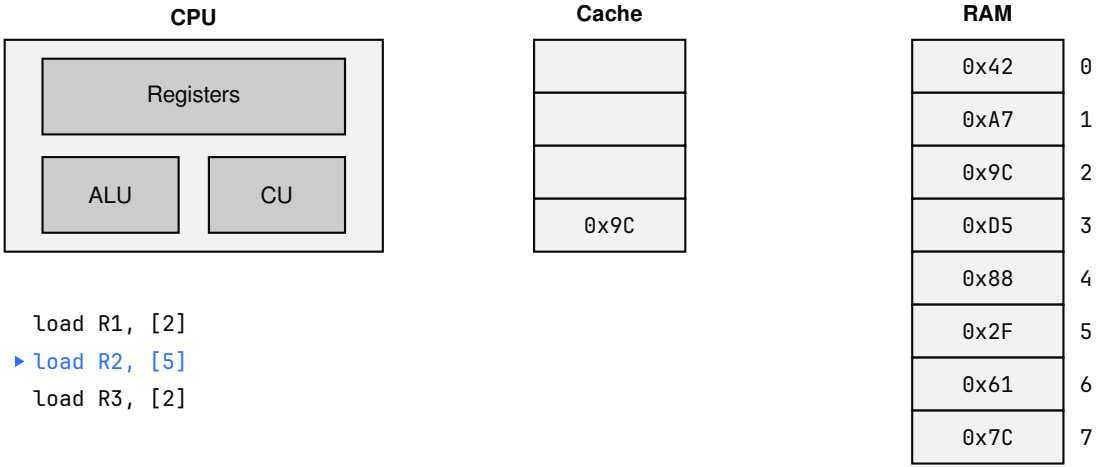



```
load R1, [2]
load R2, [5]
load R3, [2]
```

Solution: Smaller, faster memory close to the CPU serving as cache.

Memory access with caching

think-cell

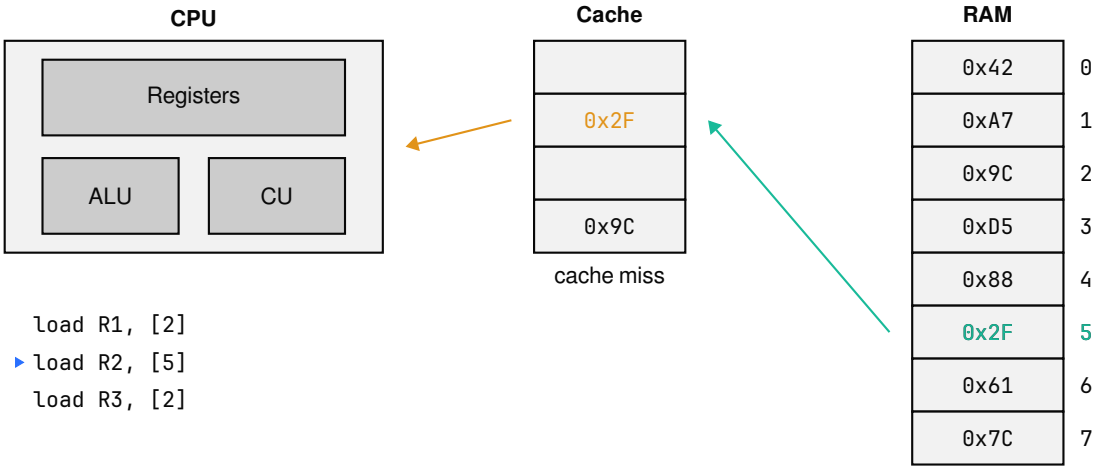



```
  load R1, [2]
▸ load R2, [5]
  load R3, [2]
```

Solution: Smaller, faster memory close to the CPU serving as cache.

Memory access with caching

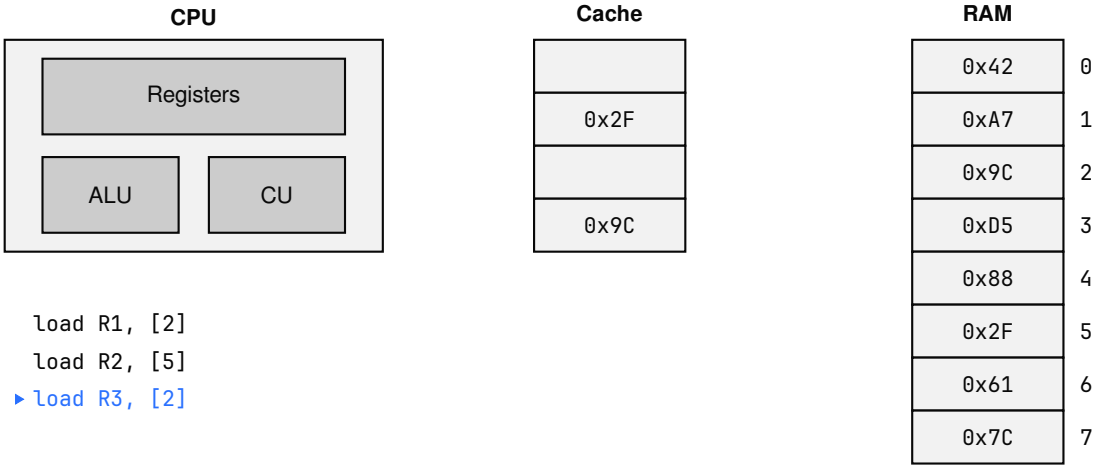
think-cell



Solution: Smaller, faster memory close to the CPU serving as cache.

Memory access with caching

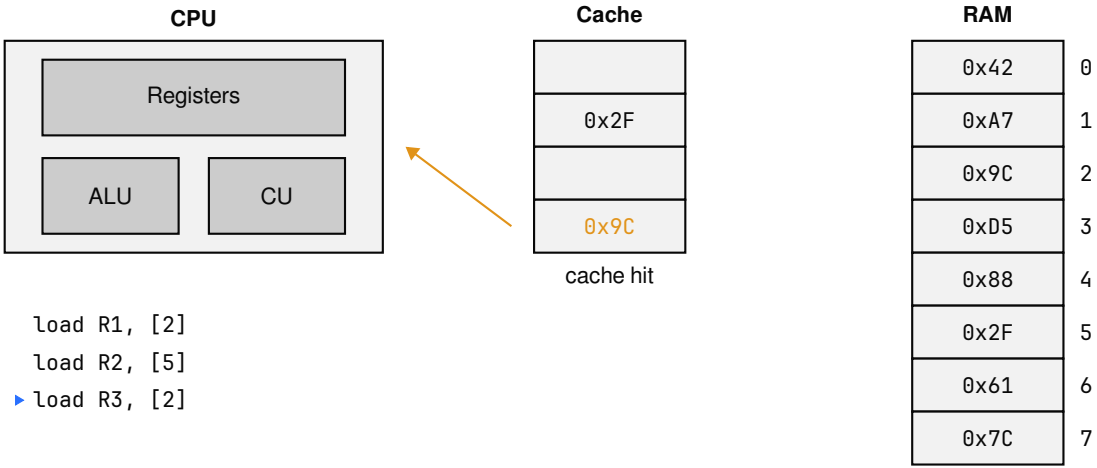
think-cell



Solution: Smaller, faster memory close to the CPU serving as cache.

Memory access with caching

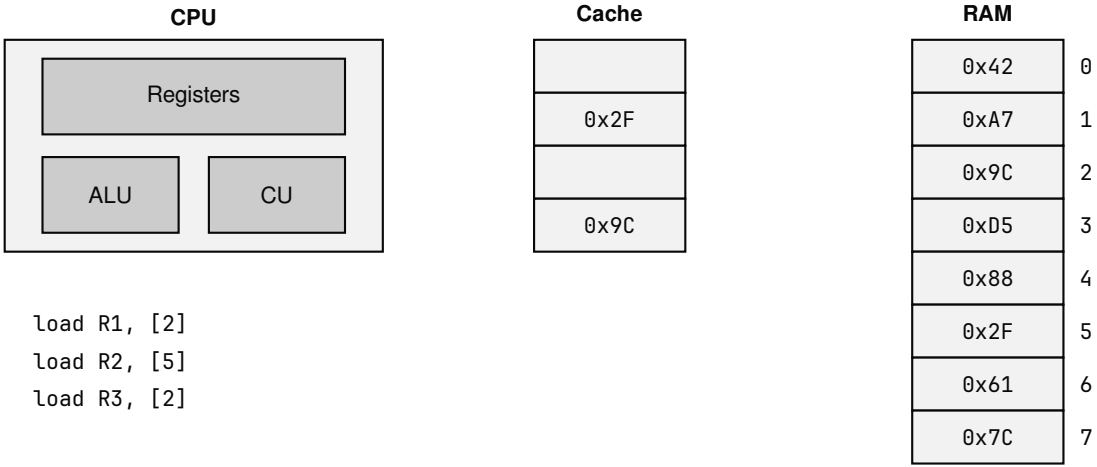
think-cell



Solution: Smaller, faster memory close to the CPU serving as cache.

Memory access with caching

think-cell

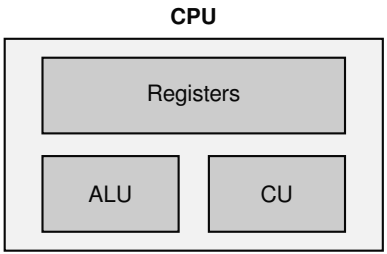


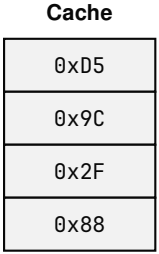

```
load R1, [2]
load R2, [5]
load R3, [2]
```

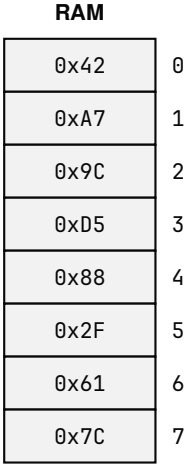
Solution: Smaller, faster memory close to the CPU serving as cache.

Unfortunately, caches are small

think-cell



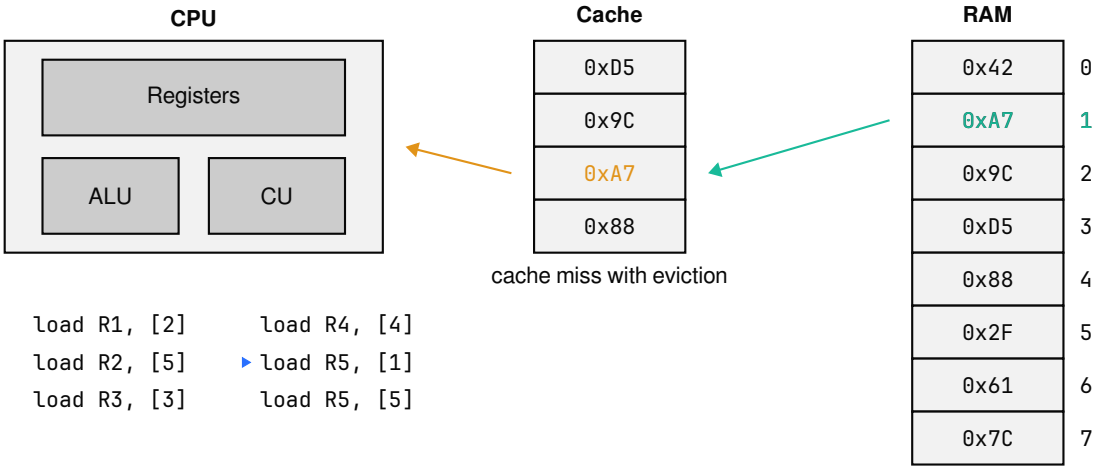





```
load R1, [2]      load R4, [4]
load R2, [5]    ▶ load R5, [1]
load R3, [3]      load R5, [5]
```

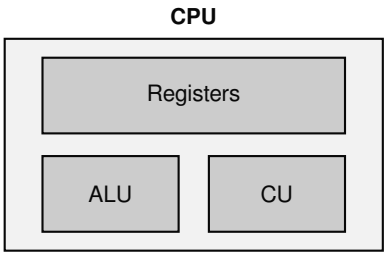
Unfortunately, caches are small

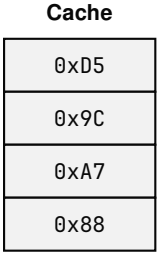
think-cell

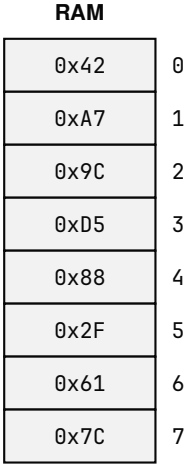


Unfortunately, caches are small

think-cell

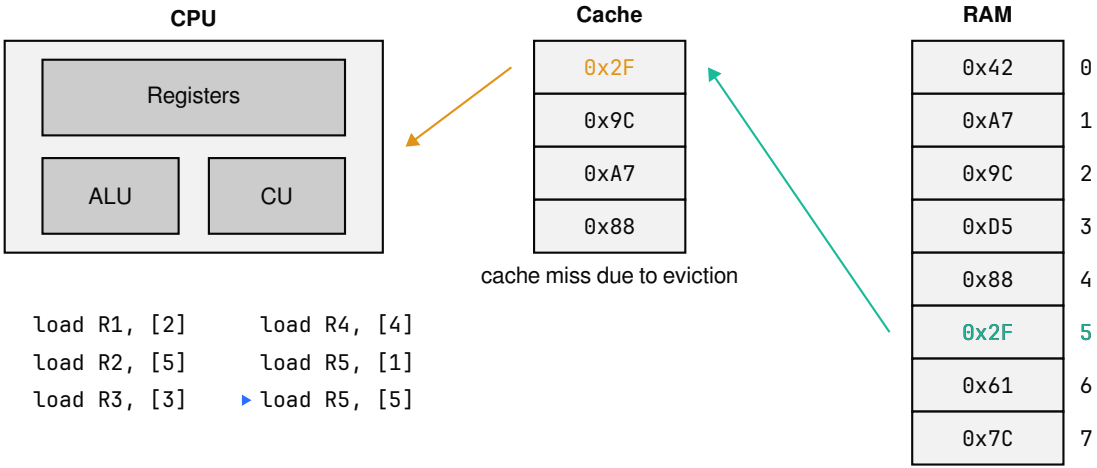







```
load R1, [2]      load R4, [4]
load R2, [5]      load R5, [1]
load R3, [3]    ▸ load R5, [5]
```

Unfortunately, caches are small

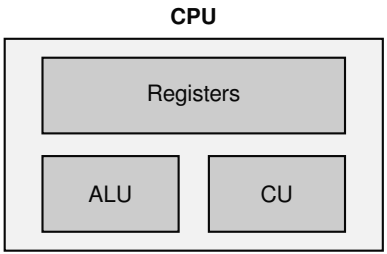


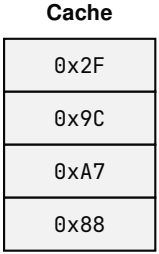
cache miss due to eviction

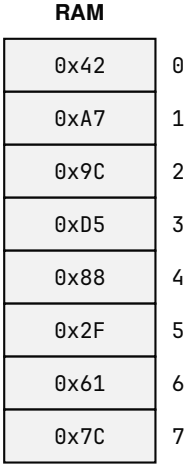
```
load R1, [2]      load R4, [4]
load R2, [5]      load R5, [1]
load R3, [3]    ▸ load R5, [5]
```

Unfortunately, caches are small

think-cell








```
load R1, [2]      load R4, [4]
load R2, [5]      load R5, [1]
load R3, [3]      load R5, [5]
```

Solution: Hierarchy of caches.

Cache hierarchy

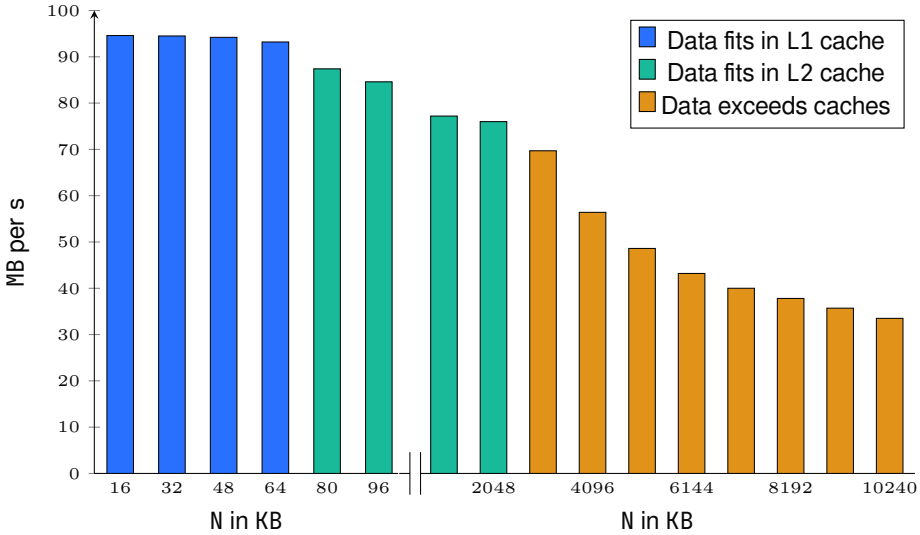
think-cell

Apple M1 (2020): [2]

L1	64-128 KB per core @ 3 cycles
L2	8-12 MB shared @ 18 cycles
L3	8 MB shared with GPU @ 18 cycles + 10-15 ns
RAM	16 GB @ 18 cycles + 100 ns

[2] https://www.7-cpu.com/cpu/Apple_M1.html

Memory access slows down when data exceeds cache sizes



What happens when a cached address is written?

think-cell

Write-through cache Write to cache and main memory

Write-back cache Write to main memory upon eviction

What happens when a cached address is written?

think-cell

Write-through cache Write to cache and main memory
Write-back cache Write to main memory upon eviction

CPU caches are usually write-back.

Takeaways

think-cell

- Main memory access is slow

Takeaways

think-cell

- Main memory access is slow
- Therefore: main memory access is avoided using faster **caches**

Takeaways

think-cell

- Main memory access is slow
- Therefore: main memory access is avoided using faster **caches**
- But: caches are **small**

Takeaways

- Main memory access is slow
- Therefore: main memory access is avoided using faster **caches**
- But: caches are **small**

Ensure as much data as possible fits into the cache!

Efficiently using cache space

Computing the average age of a group of people

think-cell

```
enum class Age : int {};
```

```
Age average_age(const std::vector<Age>& ages)
{
    int total = 0;
    for (Age age : ages) {
        total += static_cast<int>(age);
    }
    return static_cast<Age>(total / ages.size());
}
```

Optimizing for cache-friendliness

think-cell

Ensure as much data as possible fits into the cache!

think-cell

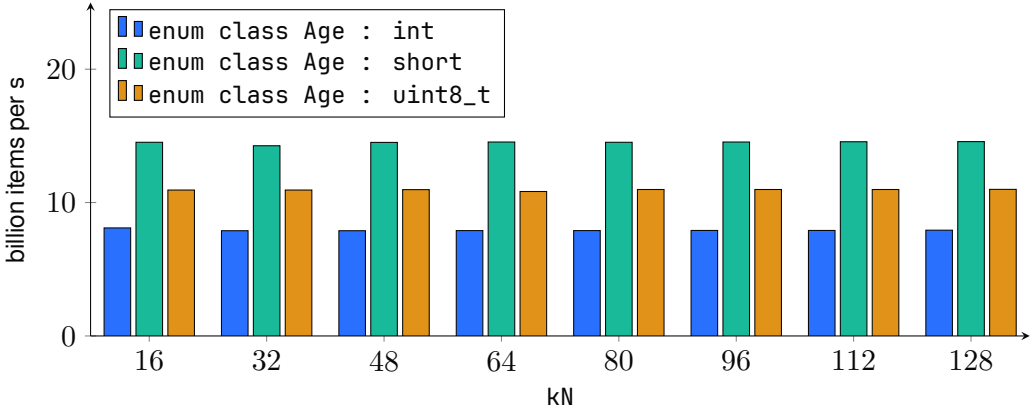
Ensure as much data as possible fits into the cache!

```
enum class Age : int       4 bytes   ages -2G to +2G
enum class Age : short     2 bytes   ages -32k to +32k
enum class Age : uint8_t   1 byte    ages 0 to 255
```

Benchmark results

think-cell

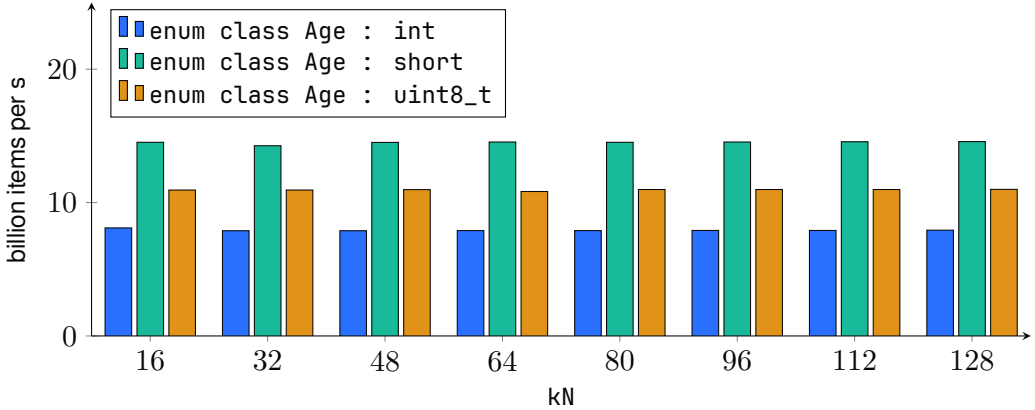
Averaging random ages between 0 and 100.



Benchmark results

think-cell

Averaging random ages between 0 and 100.



Smaller size of `uint8_t` offset by slower byte operations.

Benchmarks are important

think-cell

Always measure before and after optimizations!

Optimize type size: Pick smaller primitive types

think-cell

- `signed char` or `short` instead of `int`
- `float` instead of `double`
- specify underlying type of enumerations

Interlude: `char` vs. `signed char`

think-cell

Three distinct types:

- `signed char` (“`int8_t`”)
- `unsigned char` (“`uint8_t`”).
- `char` (“character”): either `signed` or `unsigned`, yet distinct

```
// both assertions hold!
static_assert(!std::is_same_v<char, signed char>);
static_assert(!std::is_same_v<char, unsigned char>);
```

Interlude: `char` vs. `signed char`

think-cell

Three distinct types:

- `signed char` (“`int8_t`”)
- `unsigned char` (“`uint8_t`”).
- `char` (“character”): either `signed` or `unsigned`, yet distinct

```
// both assertions hold!
static_assert(!std::is_same_v<char, signed char>);
static_assert(!std::is_same_v<char, unsigned char>);
```

Furthermore:

- `char8_t` as a distinct unsigned 8 bit type for UTF-8 text
- `std::byte` as a distinct type 8 bit type for raw memory access

Benchmarking one byte types

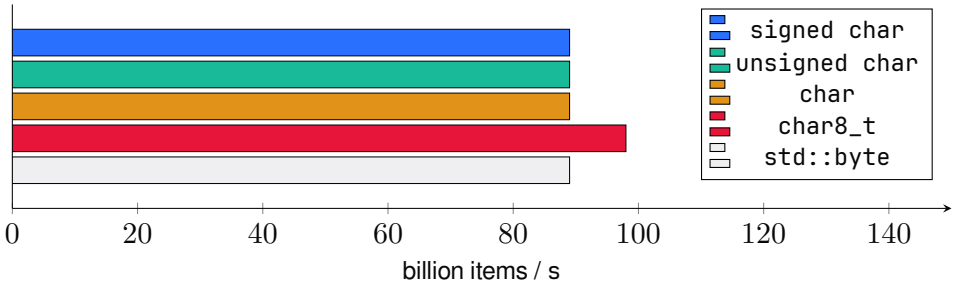
think-cell

```cpp
template <typename T> requires (sizeof(T) == 1)
void increment(std::vector<T>& data)
{
    for (std::size_t i = 0; i < data.size(); ++i)
        data[i] = static_cast<T>(int(data[i]) + 1);
}
```

Benchmarking one byte types

think-cell

```cpp
template <typename T> requires (sizeof(T) == 1)
void increment(std::vector<T>& data)
{
    for (std::size_t i = 0; i < data.size(); ++i)
        data[i] = static_cast<T>(int(data[i]) + 1);
}
```



Interlude: Strict aliasing

think-cell

C++26, basic.lval/11:

> *An object of dynamic type* T_{obj} *is type-accessible through a glvalue of type* T_{ref} *if* T_{ref} *is similar to:*
>
> - T_{obj},
> - *a type that is the* `signed` *or* `unsigned` *type corresponding to* T_{obj}, *or*
> - ***a*** `char`***,*** `unsigned char`***, or*** `std::byte` ***type***.
>
> *If a program attempts to access the stored value of an object through a glvalue through which it is not type-accessible, the behavior is undefined.*

Interlude: Strict aliasing

think-cell

C++26, basic.lval/11:

An object of dynamic type T_{obj} *is type-accessible through a glvalue of type* T_{ref} *if* T_{ref} *is similar to:*

- T_{obj},
- *a type that is the* `signed` *or* `unsigned` *type corresponding to* T_{obj}, *or*
- ***a* `char`, `unsigned char`, *or* `std::byte` *type***.

If a program attempts to access the stored value of an object through a glvalue through which it is not type-accessible, the behavior is undefined.

C11, 6.5/7:

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

- *a type compatible with the effective type of the object,*
- *[…], or*
- ***a character type***.

Optimizations in the presence of aliasing pointers

think-cell

```cpp
template <typename T> requires (sizeof(T) == 1)
void increment(std::vector<T>& data)
{
    for (std::size_t i = 0; i < data.size(); ++i)
        data[i] = static_cast<T>(int(data[i]) + 1);
}
```

- If T is non-aliasing: modification of `data[i]` can only modify other T objects.
- If T is aliasing: modification of `data[i]` may modify arbitrary other objects, including `std::vector<T> data` (!).

Benchmarking one byte types: Fixed

think-cell

```cpp
template <typename T> requires (sizeof(T) == 1)
void increment(std::vector<T>& data)
{
    auto size = data.size();
    for (std::size_t i = 0; i < size; ++i)
        data[i] = static_cast<T>(int(value) + 1);
}
```

Benchmarking one byte types: Fixed

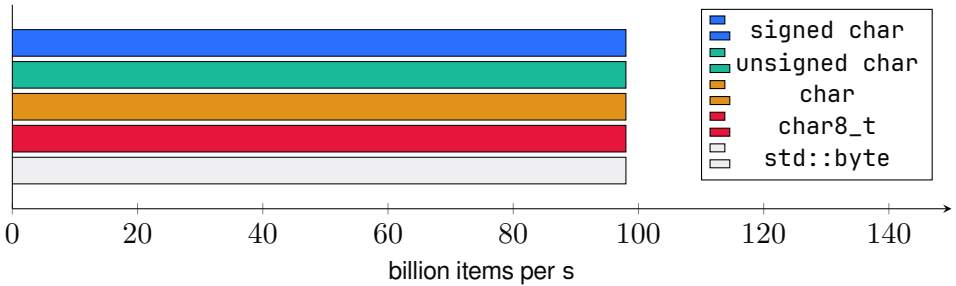
think-cell

```cpp
template <typename T> requires (sizeof(T) == 1)
void increment(std::vector<T>& data)
{
    for (auto& value : data)
        value = static_cast<T>(int(value) + 1);
}
```

Benchmarking one byte types: Fixed

think-cell

```cpp
template <typename T> requires (sizeof(T) == 1)
void increment(std::vector<T>& data)
{
    for (auto& value : data)
        value = static_cast<T>(int(value) + 1);
}
```



Optimize type size: Bitfields

think-cell

```
enum class State : uint8_t
{
    A, B, C
};
```

```
struct Widget
{
    bool is_enabled;
    bool is_visible;
    State state;
};
```

```
sizeof(Widget) == 3
```

Optimize type size: Bitfields

think-cell

```
enum class State : uint8_t
{
    A, B, C
};
```

```
struct Widget
{
    bool is_enabled : 1;
    bool is_visible : 1;
    State state : 2;
};
```

`sizeof(Widget) == 1` with bits to spare!

Optimize type size: Bitfields

think-cell

```cpp
enum class State : uint8_t
{
    A, B, C
};
```

```cpp
struct Widget
{
    bool is_enabled : 1;
    bool is_visible : 1;
    State state : 2;
};
```

`sizeof(Widget) == 1` with bits to spare!

Beware: Accessing bitfields may be slower → **benchmark**!

Optimize type size: Re-ordering members

think-cell

```
struct Message
{
    uint8_t type;
    uint32_t length;
    unsigned char* data;
    uint16_t checksum;
};
```

Optimize type size: Re-ordering members

think-cell

```
struct Message
{
    uint8_t type;
    uint32_t length;
    unsigned char* data;
    uint16_t checksum;
};
```

```
sizeof(uint8_t) + sizeof(uint32_t) + sizeof(unsigned char*) +
sizeof(uint16_t) == 15
```

Optimize type size: Re-ordering members

think-cell

```
struct Message
{
    uint8_t type;
    uint32_t length;
    unsigned char* data;
    uint16_t checksum;
};
```

```
sizeof(uint8_t) + sizeof(uint32_t) + sizeof(unsigned char*) +
sizeof(uint16_t) == 15
```

```
sizeof(Message) == 24
```

Interlude: Alignment

think-cell

basic.align/1

> *Object types have alignment requirements which place restrictions on the addresses at which an object of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated.*

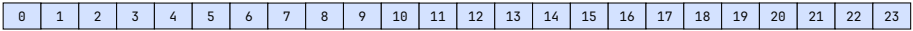
Interlude: Alignment

think-cell

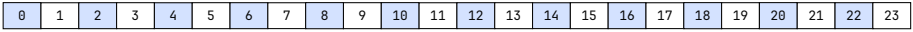
basic.align/1

> *Object types have alignment requirements which place restrictions on the addresses at which an object of that type may be allocated. An alignment is an implementation-defined integer value representing the number of bytes between successive addresses at which a given object can be allocated.*

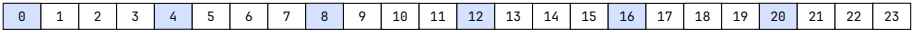
`alignof(uint8_t) == 1`



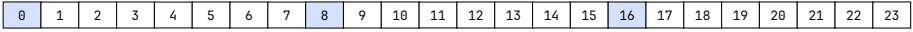
`alignof(uint16_t) == 2`



`alignof(uint32_t) == 4`



`alignof(uint64_t) == 8`



Interlude: Padding

think-cell

Members of a `struct` need to be aligned properly:

Struct alignment Maximum alignment of any member of the `struct`.

Interlude: Padding

think-cell

Members of a `struct` need to be aligned properly:

Struct alignment Maximum alignment of any member of the `struct`.

Internal padding Padding bytes between members to ensure proper alignment of the next member.

Interlude: Padding

think-cell

Members of a `struct` need to be aligned properly:

Struct alignment Maximum alignment of any member of the `struct`.

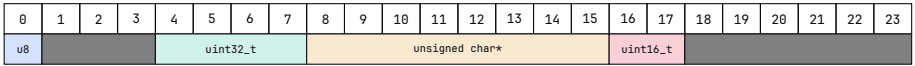
Internal padding Padding bytes between members to ensure proper alignment of the next member.

Tail padding Padding bytes at the end of a `struct` to ensure proper alignment of arrays of that `struct`.

Optimize type size: Re-ordering members

think-cell

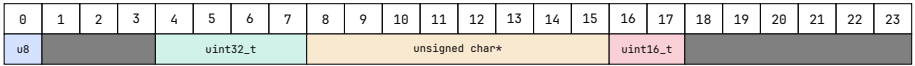
```
struct Message { uint8_t; uint32_t; unsigned char*; uint16_t; };
```



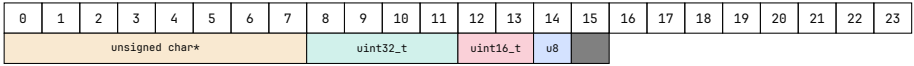
Optimize type size: Re-ordering members

think-cell

```
struct Message { uint8_t; uint32_t; unsigned char*; uint16_t; };
```




```
struct Message { unsigned char*; uint32_t; uint16_t; uint8_t; };
```



Optimize type size: Pointer bits

think-cell

Observation: `T*` always points to a multiple of `alignof(T)`.

Optimize type size: Pointer bits

think-cell

Observation: `T*` always points to a multiple of `alignof(T)`.

Therefore: The lower `log2(alignof(T))` bits of `T*` are always zero!

Optimize type size: Pointer bits

think-cell

Observation: `T*` always points to a multiple of `alignof(T)`.

Therefore: The lower `log2(alignof(T))` bits of `T*` are always zero!

```
class ContainerOrText
{
    bool _is_container;
    void* _ptr;

public:
    bool is_container() const { return _is_container; }
    Container* as_container() const { return (Container*)_ptr; }
    Text* as_text() const { return (Text*)_ptr; }
};
```

```
sizeof(ContainerOrText) == 16
```

Optimize type size: Pointer bits

think-cell

Observation: `T*` always points to a multiple of `alignof(T)`.

Therefore: The lower `log2(alignof(T))` bits of `T*` are always zero!

```
class ContainerOrText
{
    std::uintptr_t _impl;

public:
    bool is_container() const { return (_impl & 0b0) == 0; }
    Container* as_container() const { return (Container*)(_impl); }
    Text* as_text() const { return (Text*)(_impl & ~uintptr_t(0b0)); }
};
```

```
sizeof(ContainerOrText) == 8
```

Takeaways

think-cell

If types are smaller, more fit into the cache.

- `short` instead of `int`, `float` instead of `double`
- specify underlying type of enumerations, consider bitfields
- reorder members of `structs` to minimize padding
- consider tricks such as leveraging pointer alignment bits

Beware: Access of data might be slower → **benchmark**!

CPU prefetching and cache lines

Wait, why do smaller types help actually?

think-cell

```
Age average_age(const std::vector<Age>& ages)
{
    int total = 0;
    for (Age age : ages) {
        total += static_cast<int>(age);
    }
    return static_cast<Age>(total / ages.size());
}
```

Why did a smaller type make it faster? We only access everything once.

Wait, why do smaller types help actually?

think-cell

```
Age average_age(const std::vector<Age>& ages)
{
    int total = 0;
    for (Age age : ages) {
        total += static_cast<int>(age);
    }
    return static_cast<Age>(total / ages.size());
}
```

Why did a smaller type make it faster? We only access everything once.

And why was `std::vector` faster than `std::unordered_map` in the motivation?

Benchmarking memory access patterns

think-cell

Random access:

```
benchmark::DoNotOptimize(++data[dist(engine)]);
```

Sequential forward access:

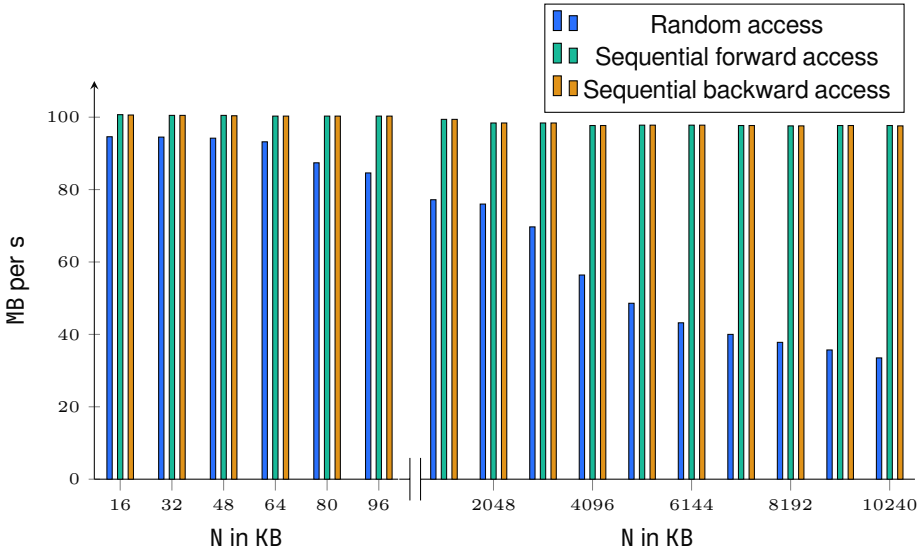
```
for (std::size_t idx = 0; idx < data.size(); ++idx)
{
    benchmark::DoNotOptimize(dist(engine));
    benchmark::DoNotOptimize(++data[idx]);
}
```

Sequential backward access:

```
for (std::size_t idx = data.size(); idx > 0; --idx)
{
    benchmark::DoNotOptimize(dist(engine));
    benchmark::DoNotOptimize(++data[idx - 1]);
}
```

Structured memory access is consistently fast!

think-cell



CPU cache prefetching

think-cell

Insight: When accessing address x, usually you also access x+1, x+2, …

CPU cache prefetching

think-cell

Insight: When accessing address `x`, usually you also access `x+1`, `x+2`, …

Prefetching: Pre-emptively load those addresses into the cache as well.

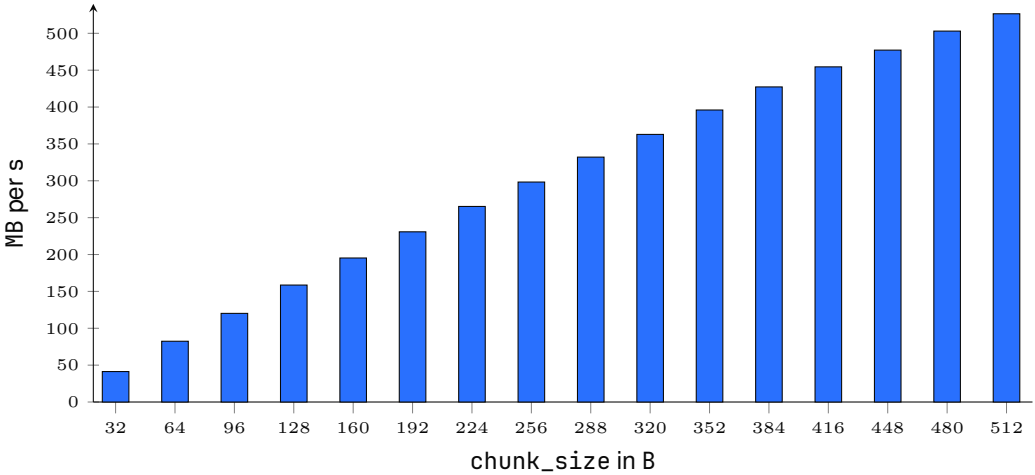
Chunked memory access

think-cell

```cpp
std::vector<std::size_t> indices(chunk_size);
std::iota(indices.begin(), indices.end(), 0);

std::uniform_int_distribution<std::size_t> dist(0, N - chunk_size - 1);
for (auto _ : state)
{
    state.PauseTiming();
    std::shuffle(indices.begin(), indices.end(), engine);
    state.ResumeTiming();

    auto start = dist(engine);
    for (auto i : indices)
        sum += data[start + i];
    benchmark::DoNotOptimize(sum);
}
```

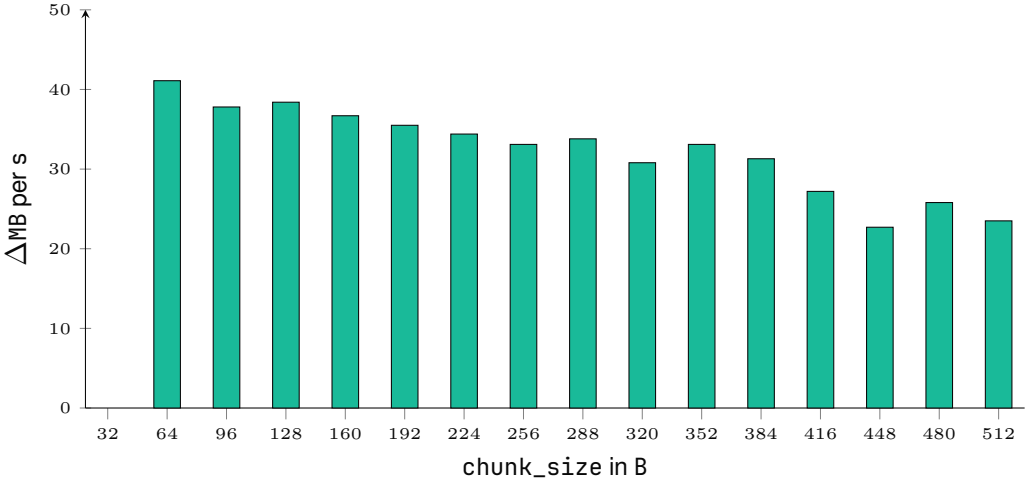
Bigger chunks are faster

think-cell



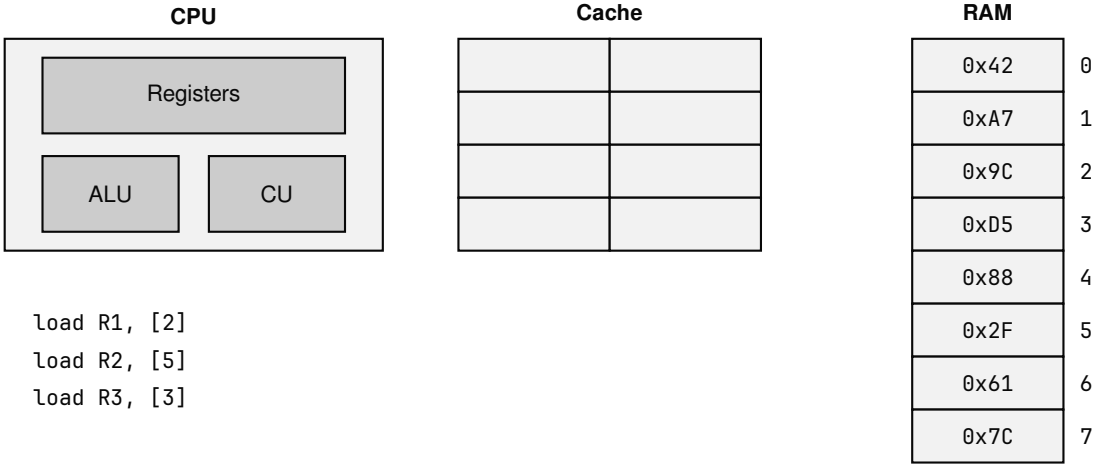
Bigger chunks are faster

think-cell



CPU cache lines

think-cell

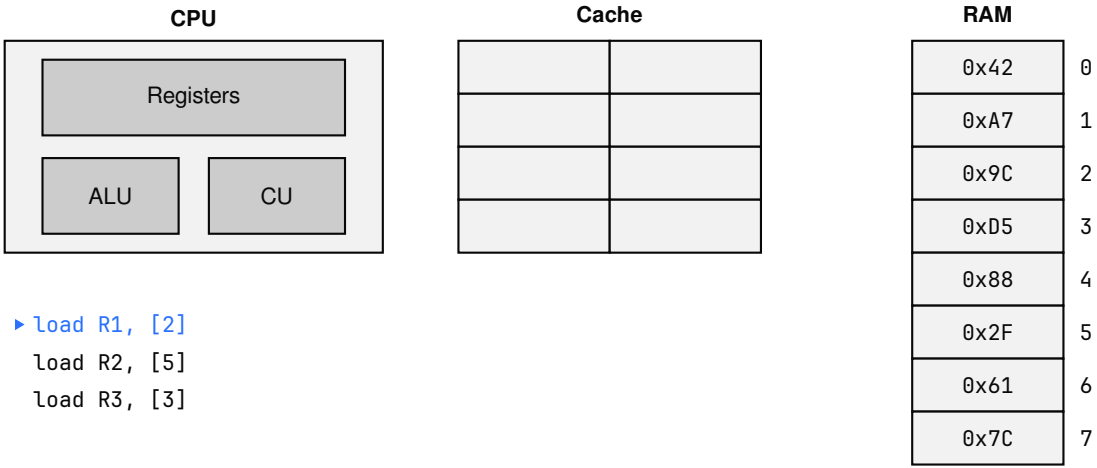



```
load R1, [2]
load R2, [5]
load R3, [3]
```

Cache line: Fetch contiguous byte range (64–128 bytes) from RAM into the cache.

CPU cache lines

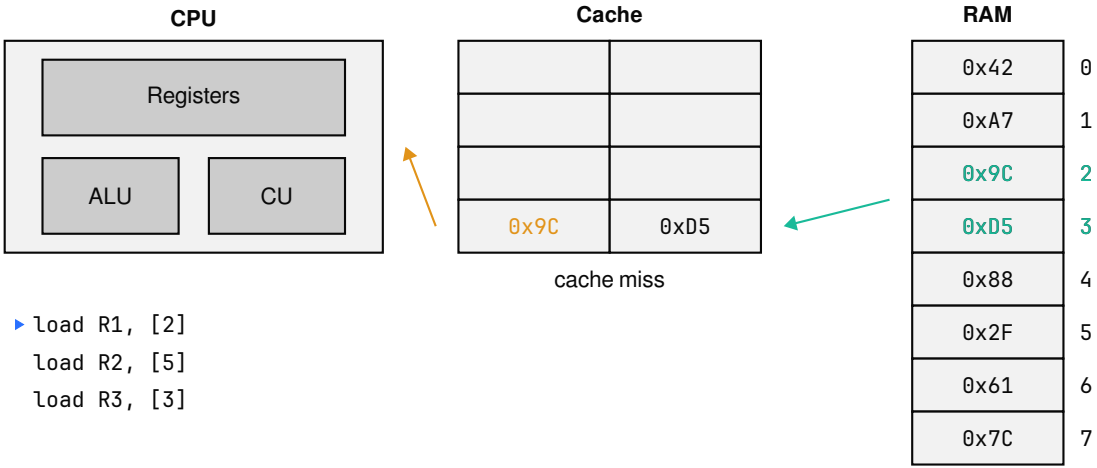
think-cell



Cache line: Fetch contiguous byte range (64–128 bytes) from RAM into the cache.

CPU cache lines

think-cell

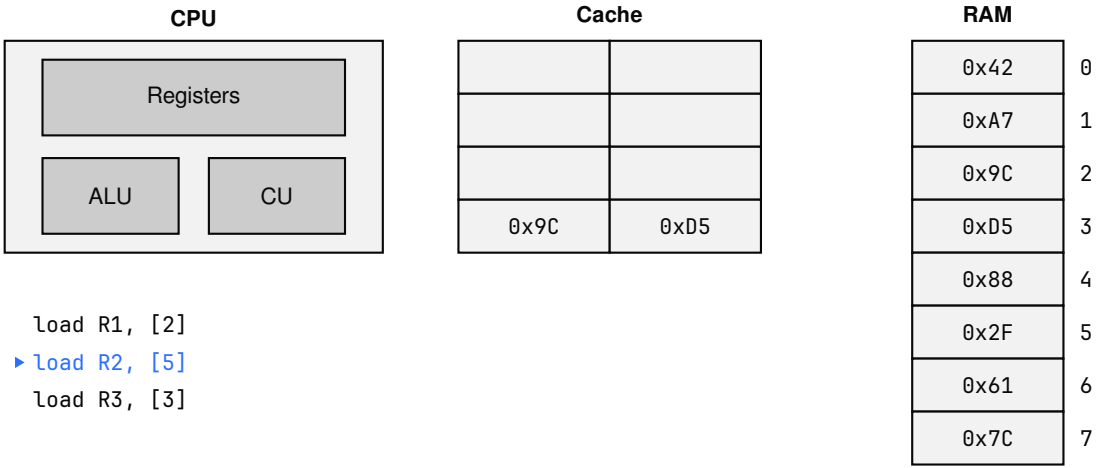



```
load R1, [2]
load R2, [5]
load R3, [3]
```

Cache line: Fetch contiguous byte range (64–128 bytes) from RAM into the cache.

CPU cache lines

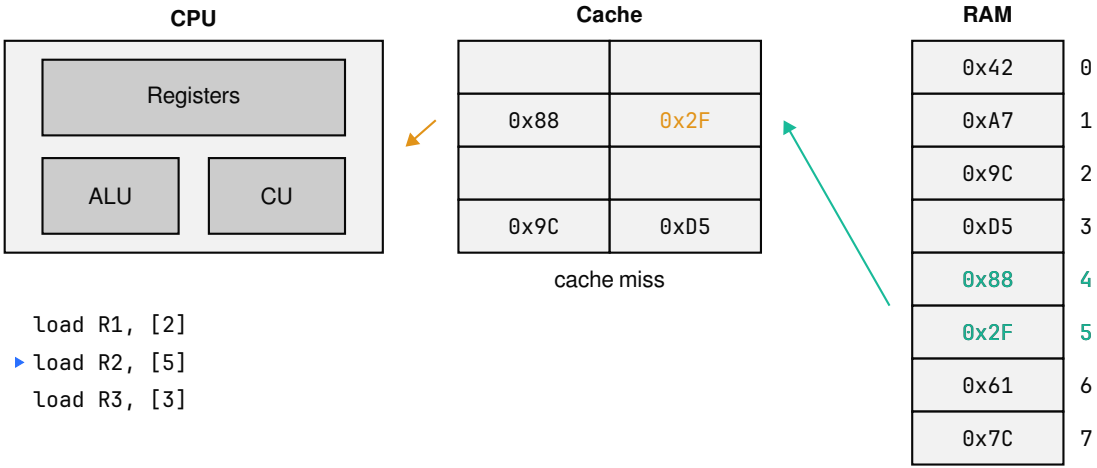
think-cell



Cache line: Fetch contiguous byte range (64–128 bytes) from RAM into the cache.

CPU cache lines

think-cell

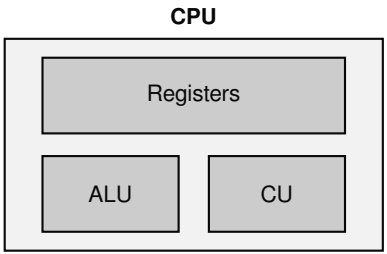


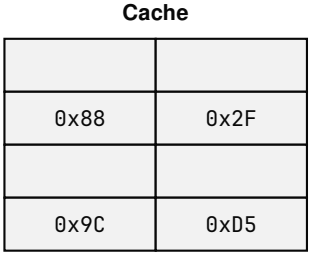

```
  load R1, [2]
▸ load R2, [5]
  load R3, [3]
```

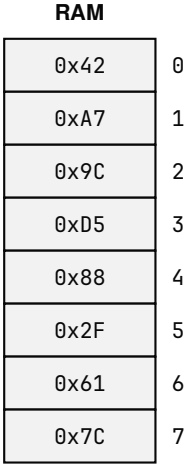
Cache line: Fetch contiguous byte range (64–128 bytes) from RAM into the cache.

CPU cache lines

think-cell



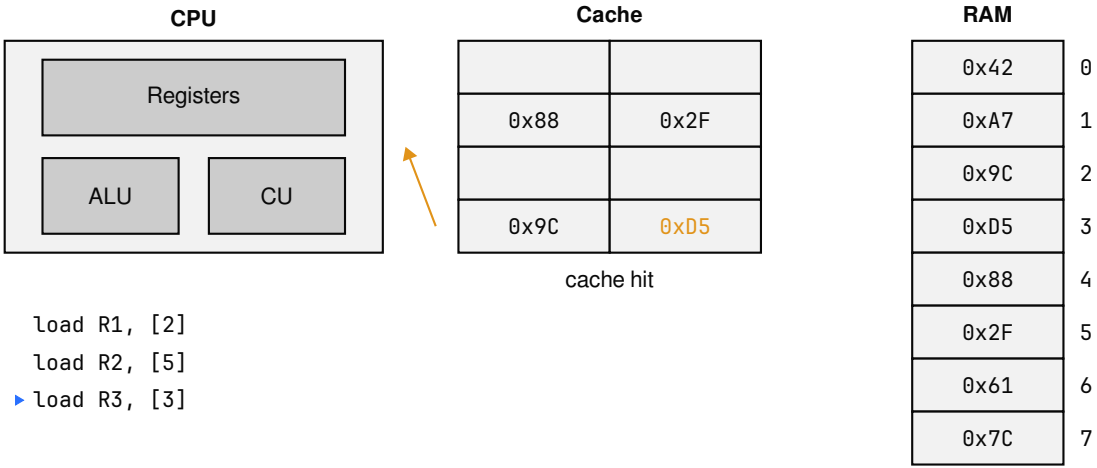





```
  load R1, [2]
  load R2, [5]
▸ load R3, [3]
```

Cache line: Fetch contiguous byte range (64–128 bytes) from RAM into the cache.

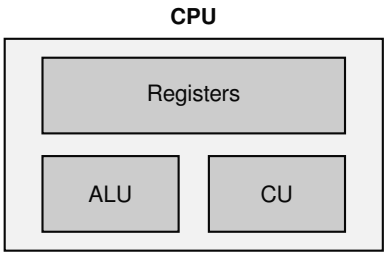
CPU cache lines

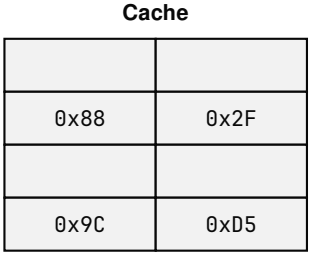


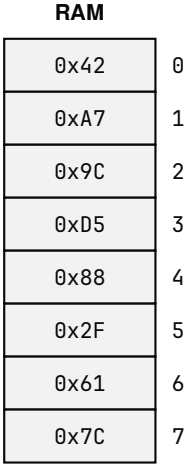
Cache line: Fetch contiguous byte range (64–128 bytes) from RAM into the cache.

CPU cache lines

think-cell








```
load R1, [2]
load R2, [5]
load R3, [3]
```

Cache line: Fetch contiguous byte range (64–128 bytes) from RAM into the cache.

Takeaways

think-cell

- CPU wants to minimize RAM access.

Takeaways

think-cell

- CPU wants to minimize RAM access.
- Therefore, CPU fetches entire **cache line** around access.

Takeaways

think-cell

- CPU wants to minimize RAM access.
- Therefore, CPU fetches entire **cache line** around access.
- Also, CPU detects simple access patterns and **prefetches** relevant memory in the background.

Takeaways

think-cell

- CPU wants to minimize RAM access.
- Therefore, CPU fetches entire **cache line** around access.
- Also, CPU detects simple access patterns and **prefetches** relevant memory in the background.

Ensure you have a predictable memory access pattern with high locality!

Use cache-friendly containers

How to leverage prefetching and cache lines?

think-cell

Predictable access:

- Avoid pointer chasing → sequential memory access

High locality:

- Minimize size of types → more fit into a cache line
- Move uninteresting data away → less waste in cache line

`std::vector<T>` is cache-friendly

think-cell

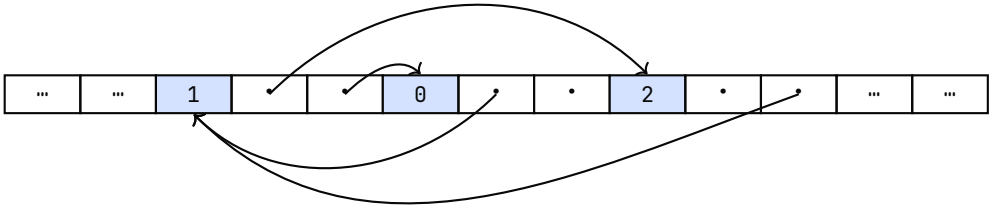
```
T* ptr;
std::size_t size;
std::size_t capacity;
```

So is `std::array<T, N>`, `std::inplace_vector<T, N>`, …

std::list<T> is not cache-friendly

think-cell

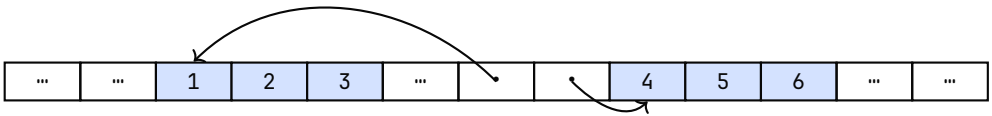
```
struct Node {
    T value;
    Node* next;
    Node* prev;
};
Node* head;
Node* tail;
```



`std::deque<T>` can be cache-friendly

think-cell

```
using Block = std::array<T, BLOCK_SIZE>;
std::vector<Block*> blocks;
```

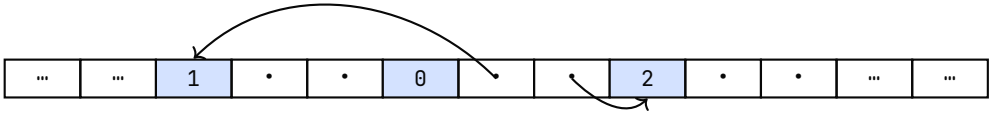


Note: `BLOCK_SIZE` has to be large enough!

`std::map<K, V>`/`std::set<T>` are not cache-friendly

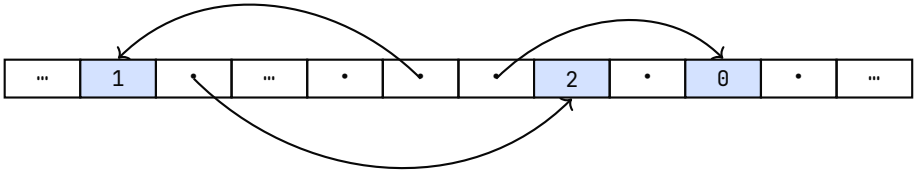
think-cell

```
struct Node {
    T key_value;
    Node* left;
    Node* right;
};
Node* root;
```



`std::unordered_{map<K, V>,set<T>}` are not cache-friendly | think-cell

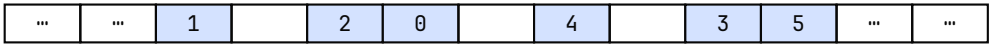
```
struct Node {
    T key_value;
    Node* next;
};
std::vector<Node*> buckets;
```



Hash tables with open addressing are cache-friendly

think-cell

```
T* ptr;
std::size_t size;
std::size_t capacity;
```



Takeaways

think-cell

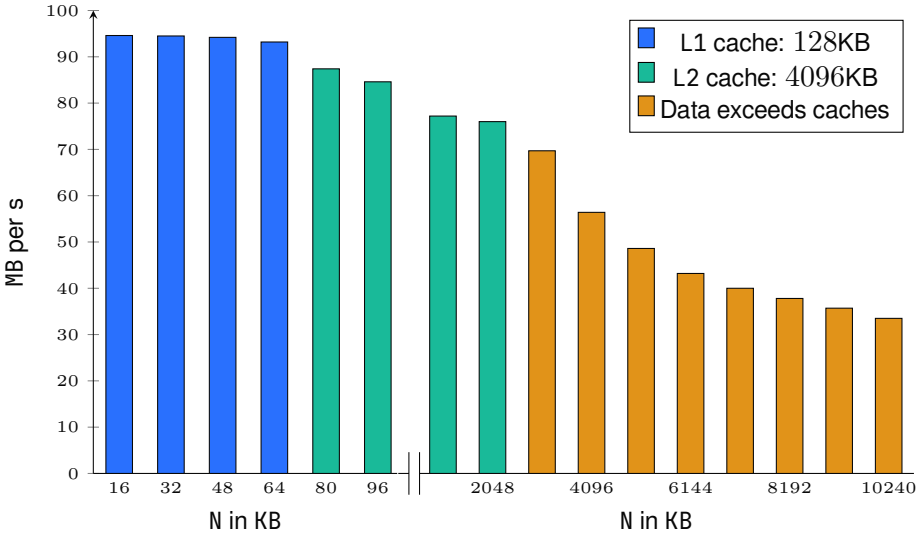
Prefer containers that use big contiguous chunks of memory.

- `std::vector<T>`
- Some implementations of `std::deque<T>`
- Hash tables with open addressing

Instruction caches

Measured cache size does not match actual cache size

think-cell



Code is data too!

think-cell

- CPU instructions are stored in memory
- Main memory access is slow
- Therefore: main memory access is avoid using faster caches

Code is data too!

think-cell

- CPU instructions are stored in memory
- Main memory access is slow
- Therefore: main memory access is avoid using faster caches

- L1 cache: usually separate for instructions and data
- L2 cache: usually unified
- Therefore: measured L2 cache size was smaller

Do sum then increments or increments then sum?

think-cell

Sum then increments

```
void sum_then_increments(const std::vector<unsigned>& data)
{
    auto count = 0;
    for (auto d : data)
    {
        count += d;
        if (d == 0) // rarely true
        {
            REPEAT64(benchmark::DoNotOptimize(++count));
        }
    }
    benchmark::DoNotOptimize(count);
}
```

Do sum then increments or increments then sum?

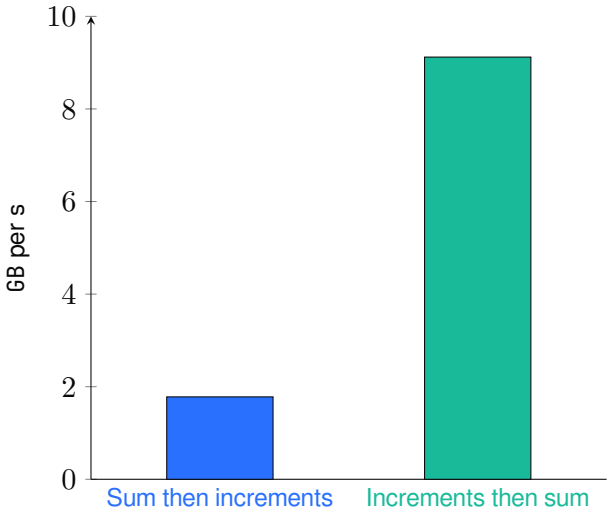
think-cell

Increments then sum

```
void increments_then_sum(const std::vector<unsigned>& data)
{
    auto count = 0;
    for (auto d : data)
    {
        if (d == 0) // rarely true
        {
            REPEAT64(benchmark::DoNotOptimize(++count));
        }
        count += d;
    }
    benchmark::DoNotOptimize(count);
}
```

Apparently you want to do increments then sum?!

think-cell



Takeways

think-cell

Cache-friendly data access patterns

- Access data in linear order
- Avoid pointer chasing
- Minimize the size of data

Cache-friendly code patterns:

- Avoid long branches
- Avoid indirect jumps
- Minimize the size of hot code

Takeways

think-cell

Cache-friendly data access patterns

- Access data in linear order
- Avoid pointer chasing
- Minimize the size of data

Cache-friendly code patterns:

- Avoid long branches
- Avoid indirect jumps
- Minimize the size of hot code

Design your program by keeping caches in mind!

Data-oriented design

OOP vs. DOD

think-cell

Object-oriented programming:

- Focused on objects: domain-specific abstractions
- Smart objects encapsulate data and behavior
- Heterogeneous collections of base classes

Data-oriented programming:

- Focused on algorithms: data transformations
- Algorithms operate on dumb data without behavior
- Multiple homogeneous collections of different records

OOP: Records of people

think-cell

Approach: Design the objects first.

OOP: Records of people

think-cell

Approach: Design the objects first.

```
class Person
{
public:
    std::string_view name() const;
    Age age() const;
};

using People = std::vector<Person>;
```

OOP: Records of people

think-cell

Approach: Design the objects first.

```
class Person
{
public:
    std::string_view name() const;
    Age age() const;
};

using People = std::vector<Person>;
```

```
Age average_age(const People& people);
```

OOP: Records of people

think-cell

Approach: Design the objects first.

```
class Person
{
public:
    std::string_view name() const;
    Age age() const;
};

using People = std::vector<Person>;
```

```
Age average_age(const People& people);
```

```
std::string_view name_of_oldest_person(const People& people);
```

DOD: Records of people

think-cell

Approach: Design the algorithms first.

DOD: Records of people

think-cell

Approach: Design the algorithms first.

```
Age average_age(???);
```

DOD: Records of people

think-cell

Approach: Design the algorithms first.

```
Age average_age(const std::vector<Age>& ages);
```

DOD: Records of people

think-cell

Approach: Design the algorithms first.

```
Age average_age(const std::vector<Age>& ages);
```

```
std::string_view name_of_oldest_person(???);
```

DOD: Records of people

think-cell

Approach: Design the algorithms first.

```
Age average_age(const std::vector<Age>& ages);
```

```
std::size_t index_of_oldest_person(const std::vector<Age>& ages);

std::string_view name_of_oldest_person(
    const std::vector<std::string_view>& names,
    const std::vector<Age>& ages
)
{
    auto index = index_of_oldest_person(ages);
    return names[index];
}
```

Two different data layout approaches

think-cell

Array of Structures (AoS):

```
struct Person
{
  std::string name;
  Age age;
};

using People = std::vector<Person>;
```

Two different data layout approaches

think-cell

Array of Structures (AoS):

```
struct Person
{
  std::string name;
  Age age;
};

using People = std::vector<Person>;
```

Structure of Arrays (SoA):

```
enum class Person : std::size_t {};

struct People
{
  std::vector<std::string> names;
  std::vector<Age> ages;
};
```

Two different data layout approaches

think-cell

Array of Structures (AoS):

```
struct Person
{
  std::string name;
  Age age;
};

using People = std::vector<Person>;
```

Structure of Arrays (SoA):

```
enum class Person : std::size_t {};

struct People
{
  std::vector<std::string> names;
  std::vector<Age> ages;
};
```

OOP Array of structures

DOD Whatever works for the algorithm!

Memory layout of AoS and SoA

think-cell

Array of Structures

Efficient access to individual rows.

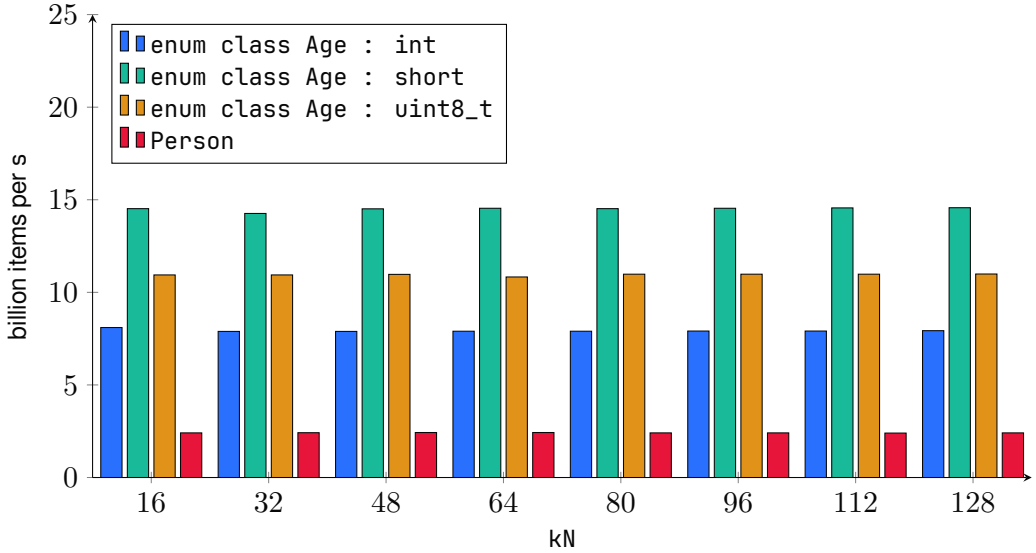
Structure of Arrays

Efficient access to individual columns.

{Illustration ignores padding and size differences.}

SoA doesn't waste space in the cache line

think-cell



OOP: Shapes

Approach: Design a class hierarchy.

```
struct Shape
{
    virtual double area() const = 0;
};

struct Circle : Shape
{
    double area() const override;
};

struct Square : Shape
{
    double area() const override;
};
```

OOP: Shapes

think-cell

Approach: Design a class hierarchy.

```
double total_area(const std::vector<std::unique_ptr<Shape>>& shapes)
{
    double total = 0.0;
    for (const auto& shape : shapes)
        total += shape->area();
    return total;
}
```

DOD: Shapes

think-cell

Approach: Design the algorithms first.

```
double total_area(???);
```

DOD: Shapes

think-cell

Approach: Design the algorithms first.

```
double total_area(
    const std::vector<std::variant<Circle, Square>>& shapes
);
```

DOD: Shapes

think-cell

Approach: Design the algorithms first.

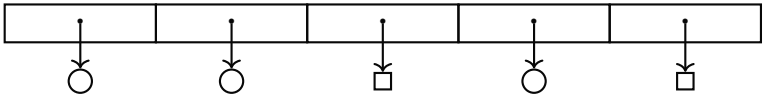
```
double total_area(
    const std::vector<Circle>& circles,
    const std::vector<Square>& squares
);
```

Representation of heterogeneous data

think-cell

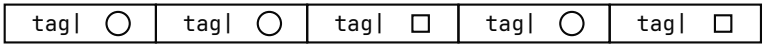
Boxed

```
std::vector<std::unique_ptr<Shape>>
```



Tagged union

```
std::vector<std::variant<Circle, Square>>
```

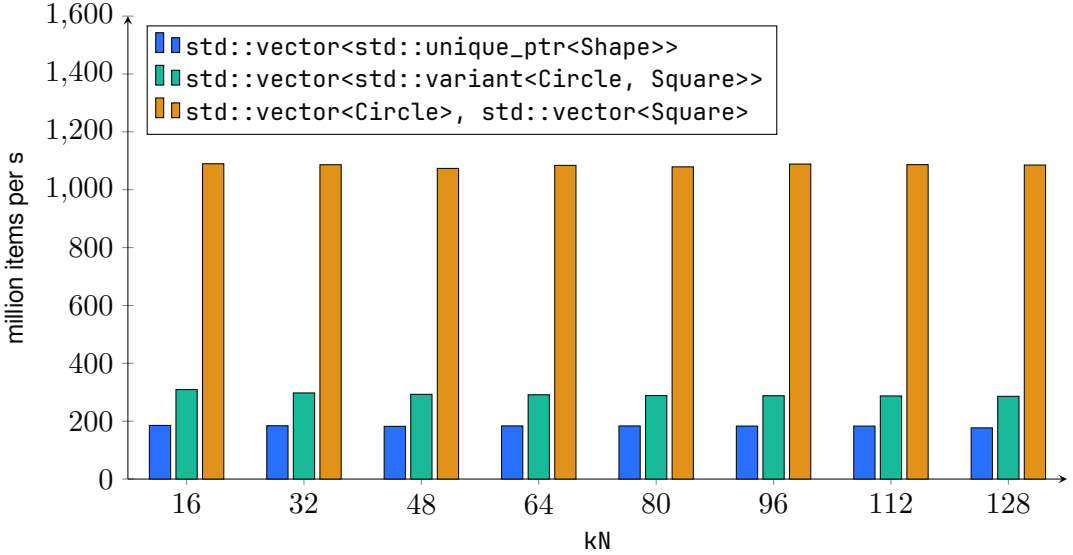


Multiple homogeneous arrays

```
std::vector<Circle>, std::vector<Square>
```

Multiple homogeneous arrays are significantly faster

think-cell



Takeaways

think-cell

Consider data-oriented design:

- Design around algorithms that transform data
- Write functions that operate on `N` elements, not single ones
- Use structure of arrays to group relevant data
- Use multiple homogeneous arrays for heterogeneous data

Caches in multi-core processors

Parallel fold pattern

think-cell

```
std::vector<T> local_results(pool.thread_count());
pool.run([&](std::size_t thread_idx) {
    local_results[thread_idx] = fold(work.slice(thread_idx));
});
T global_result = fold(local_results);
```

Silly parallel fibonacci example

think-cell

```
void accumulate_fib(unsigned& result, unsigned n)
{
    if (n < 2)
    {
        result += n;
    }
    else
    {
        accumulate_fib(result, n - 1);
        accumulate_fib(result, n - 2);
    }
}
```

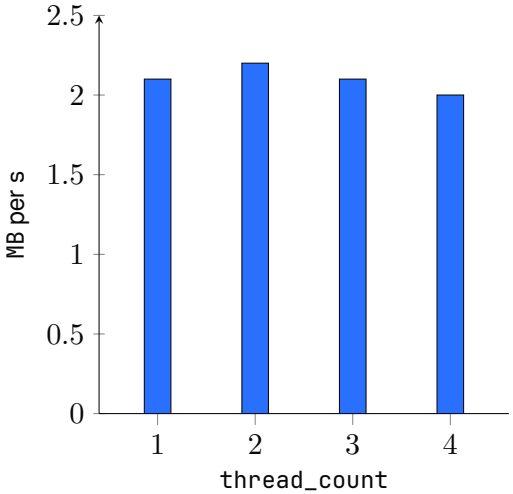
Silly parallel fibonacci example

think-cell

```
std::vector<unsigned> results(thread_count);
for (auto _ : state)
{
    pool.run([&](std::size_t thread_id) {
        results[thread_id] = 0;
        for (auto i = 0; i != 1024 / thread_count; ++i)
            accumulate_fib(results[thread_id], i % 20);
    });
    benchmark::DoNotOptimize(results);
}
```

It doesn't scale at all!

think-cell



Caches in multi-core processors

think-cell

- Each core has its own L1 cache

Caches in multi-core processors

think-cell

- Each core has its own L1 cache
- Therefore, modifications invalidate caches for all cores

Caches in multi-core processors

think-cell

- Each core has its own L1 cache
- Therefore, modifications invalidate caches for all cores
- However, caches operate in granularity of **cache lines**!

False sharing

think-cell

Thread 0 writes:

```
// invalidates cache line, ...
results[0] = 0;
```

Thread 1 reads:

```
// ... therefore, cache miss!
auto value = results[1];
```

False sharing

think-cell

Thread 0 writes:

```
// invalidates cache line, ...
results[0] = 0;
```

Thread 1 reads:

```
// ... therefore, cache miss!
auto value = results[1];
```

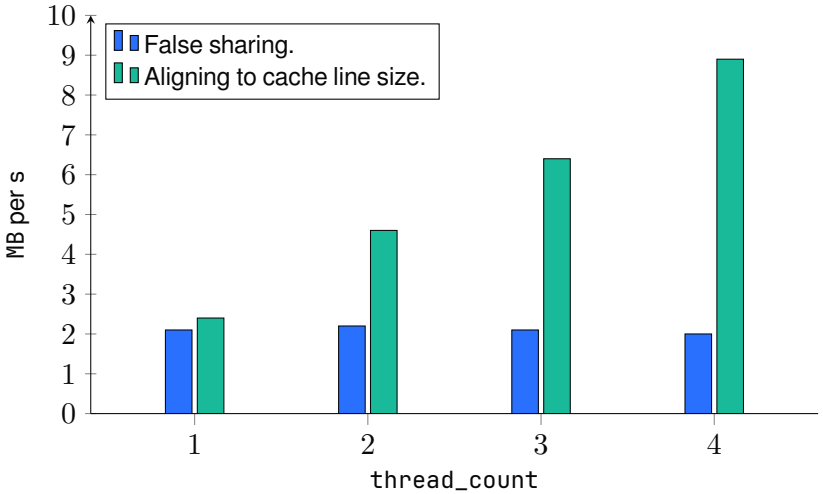
False sharing: Cache invalidation due to modifications of unrelated data in the same cache line.

Preventing false sharing by aligning to cache line size

think-cell

```
constexpr auto alignment
  = std::hardware_destructive_interference_size / sizeof(unsigned);
std::vector<unsigned> results(thread_count * alignment);
for (auto _ : state)
{
    pool.run([&](std::size_t thread_id) {
        results[thread_id * alignment] = 0;
        for (auto i = 0; i != 1024 / thread_count; ++i)
            accumulate_fib(results[thread_id * alignment], i % 20);
    });
    benchmark::DoNotOptimize(results);
}
```

Now it scales!



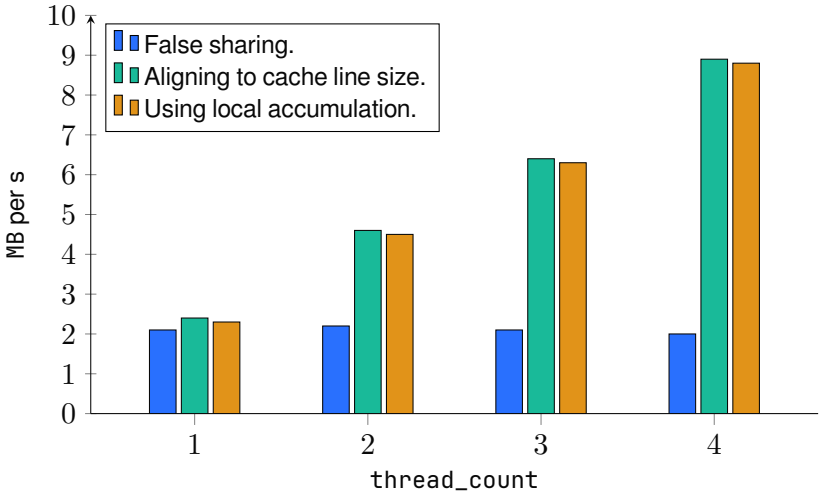
Preventing false sharing using local accumulation

think-cell

```
std::vector<unsigned> results(thread_count);
for (auto _ : state)
{
    pool.run([&](std::size_t thread_id) {
        auto local = 0u;
        for (auto i = 0; i != 1024 / thread_count; ++i)
            accumulate_fib(local, i % 20);
        results[thread_id] = local;
    });
    benchmark::DoNotOptimize(results);
}
```

Both methods work

think-cell



Takeaways

False sharing: Cache invalidation due to modifications of unrelated data in the same cache line.

Keep data used by different threads in different cache lines!

(And data used by the same thread together in the same cache line)

Conclusion

CPU caches

think-cell

Cache smaller, faster memory used to avoid accessing slower main memory

CPU caches

think-cell

Cache	smaller, faster memory used to avoid accessing slower main memory
Cache line	smallest unit of data transferred between main memory and caches

CPU caches

think-cell

Cache	smaller, faster memory used to avoid accessing slower main memory
Cache line	smallest unit of data transferred between main memory and caches
Prefetcher	predict memory access and pre-emptively load data into caches

CPU caches

think-cell

Cache	smaller, faster memory used to avoid accessing slower main memory
Cache line	smallest unit of data transferred between main memory and caches
Prefetcher	predict memory access and pre-emptively load data into caches
False sharing	cache invalidation due to modifications of unrelated data in the same cache line.

CPU caches

think-cell

Cache	smaller, faster memory used to avoid accessing slower main memory
Cache line	smallest unit of data transferred between main memory and caches
Prefetcher	predict memory access and pre-emptively load data into caches
False sharing	cache invalidation due to modifications of unrelated data in the same cache line.

Remember: Code lives in memory too!

CPU caches

think-cell

Cache	smaller, faster memory used to avoid accessing slower main memory
Cache line	smallest unit of data transferred between main memory and caches
Prefetcher	predict memory access and pre-emptively load data into caches
False sharing	cache invalidation due to modifications of unrelated data in the same cache line.

More reading: What Every Programmer Should Know About Memory

Leverage CPU caches

think-cell

Cache-friendly data access patterns

- Access data in linear order
- Avoid pointer chasing
- Minimize the size of data

Cache-friendly code patterns:

- Avoid long branches
- Avoid indirect jumps
- Minimize the size of hot code

Also: Keep data used by different threads in different cache lines!

Write cache-friendly C++!

But benchmark to make sure you're actually optimizing.