

Cache-Friendly C++

Jonathan Müller — @foonathan | Budapest C++ | 2025-10-21

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.

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?

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

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

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

3. Unsorted `std::vector`: Linear search array

- $O(n)$ lookup: linear search
- $O(1)$ insertion: `.push_back()`
- $O(n)$ 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

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

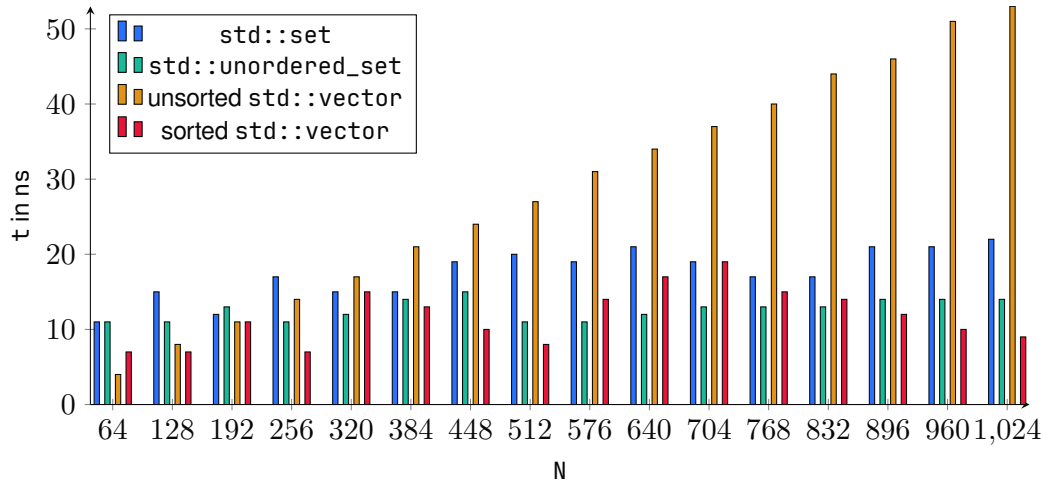
1. Setup: Fill set with N random integers
2. Lookup: Repeatedly lookup random integers

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`!



CPU caches

How fast is *random* memory access?

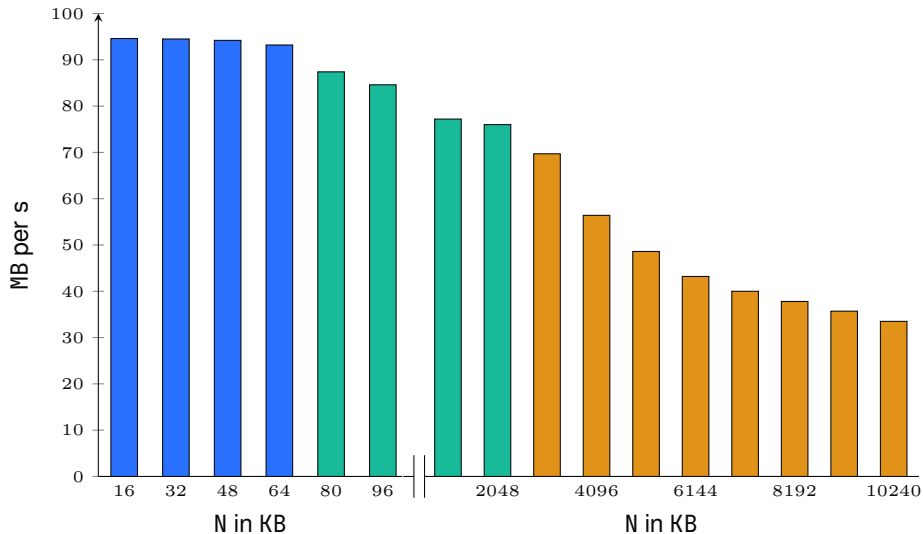
```
void bm_memory_access(benchmark::State& state)
{
    std::default_random_engine engine;
    auto bytes = state.range(0) * 1024;

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

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

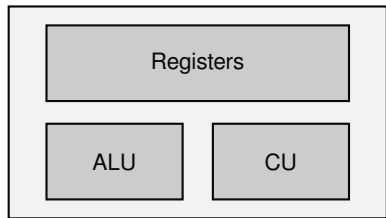
    state.SetBytesProcessed(state.iterations() * data.size());
}
```

Memory access is slower the more data you have?!



Naive memory access

CPU



load R1, [2]

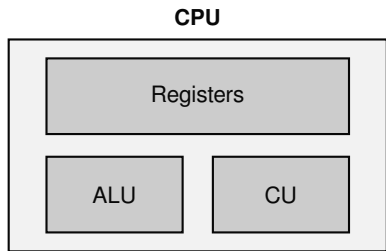
load R2, [5]

load R3, [2]

RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

Naive memory access

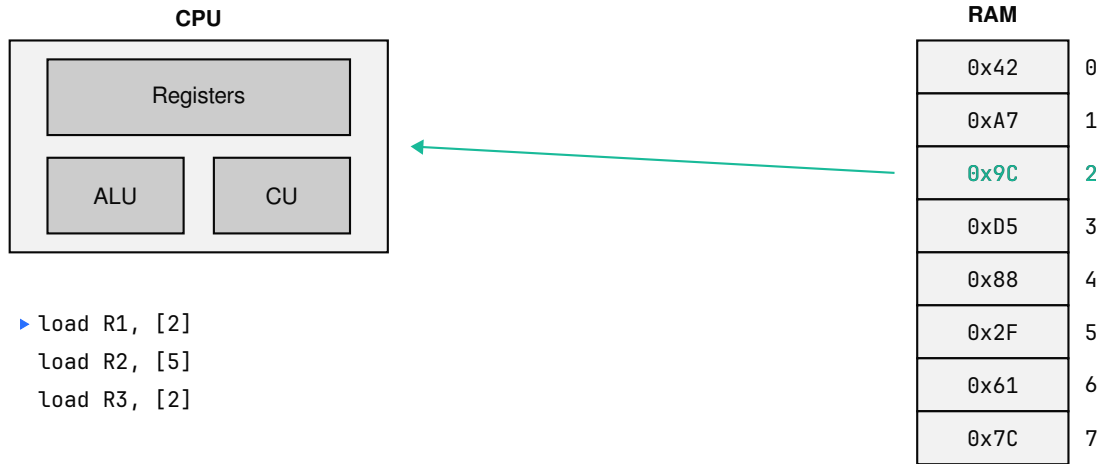


```
► load R1, [2]  
  load R2, [5]  
  load R3, [2]
```

RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

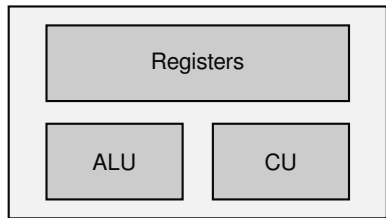
Naive memory access



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

Naive memory access

CPU



load R1, [2]

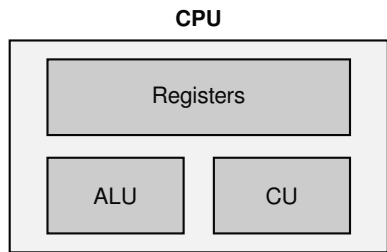
► load R2, [5]

load R3, [2]

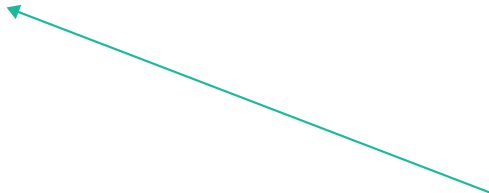
RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

Naive memory access



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

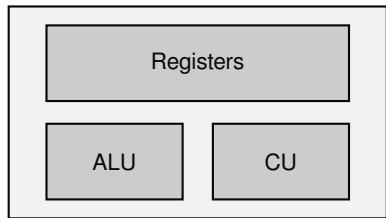


RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

Naive memory access

CPU



load R1, [2]

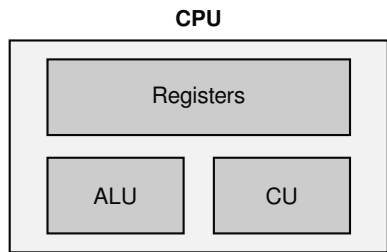
load R2, [5]

▶ load R3, [2]

RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

Naive memory access



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

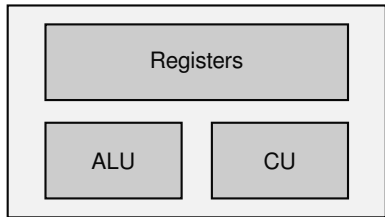


RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

Naive memory access

CPU



```
load R1, [2]
```

```
load R2, [5]
```

```
load R3, [2]
```

RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

Problem: RAM access is *slow*.

Apple M1 (2020): ¹

- RAM: 68.3 GB/s

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

Apple M1 (2020): ¹

- RAM: 68.3 GB/s
- Compute: 2.6 TFLOPS $\rightarrow$ 10650 GB/s

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

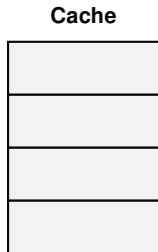
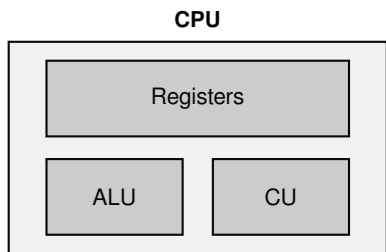
Apple M1 (2020): ¹

- RAM: 68.3 GB/s
- Compute: 2.6 TFLOPS $\rightarrow$ 10650 GB/s

Main memory access is 100 times slower than compute!

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

Memory access with caching



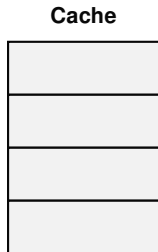
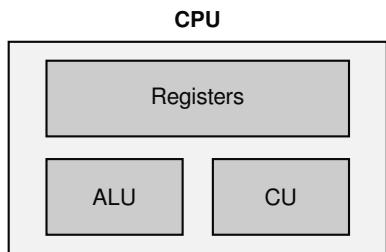
RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

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

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

Memory access with caching



RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

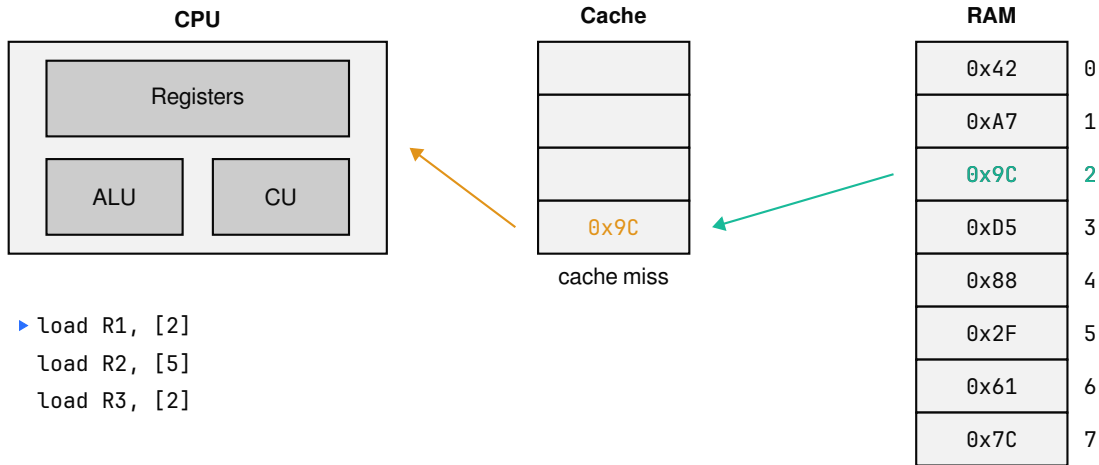
► `load R1, [2]`

`load R2, [5]`

`load R3, [2]`

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

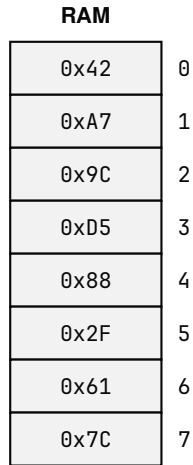
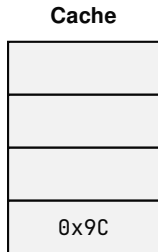
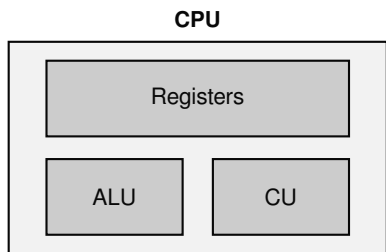
Memory access with caching



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

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

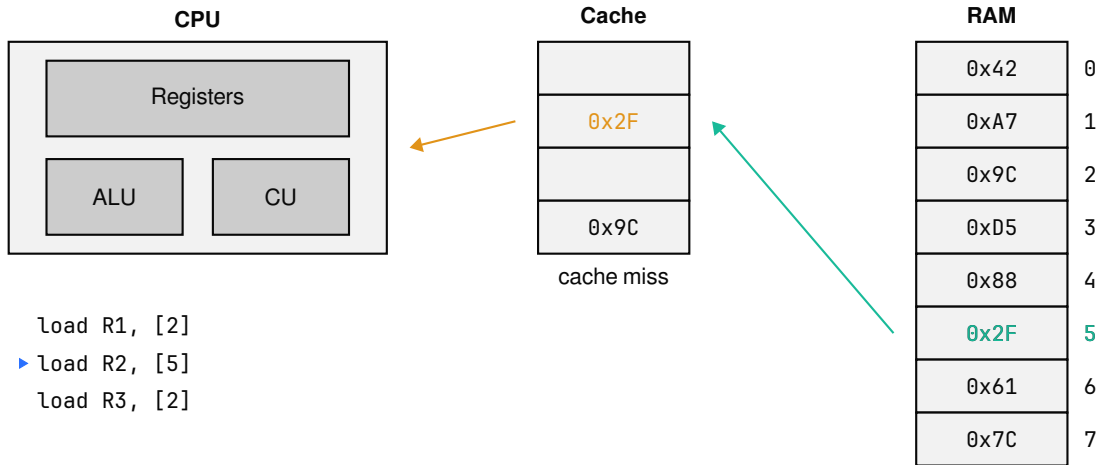
Memory access with caching



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

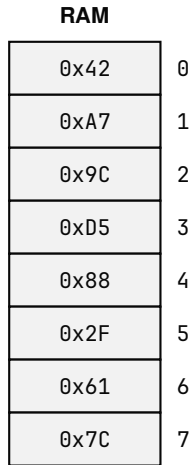
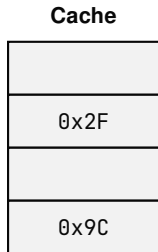
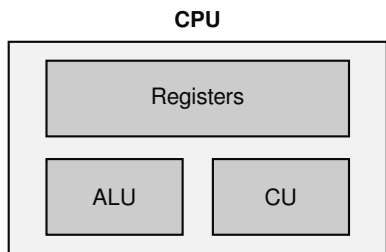
Solution: Smaller, faster memory close to the CPU serving as cache.

Memory access with caching



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

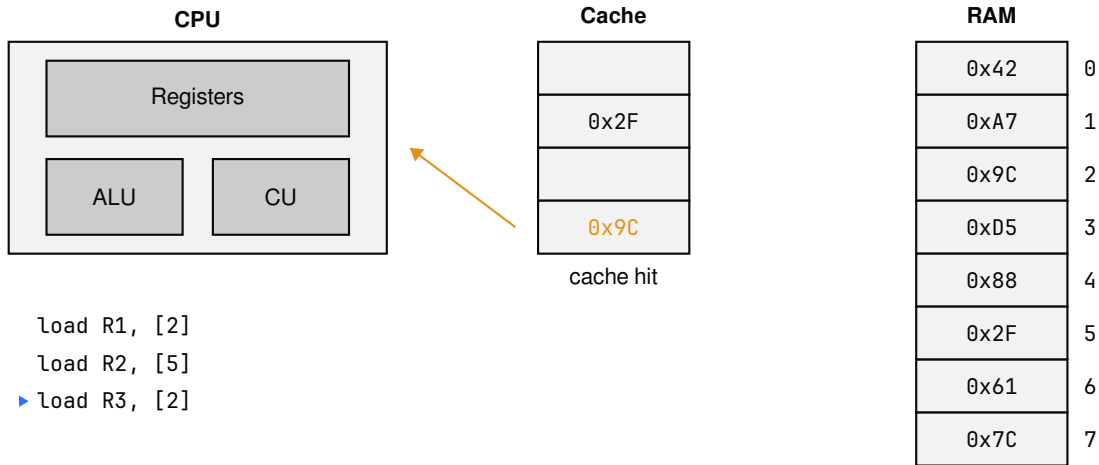
Memory access with caching



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

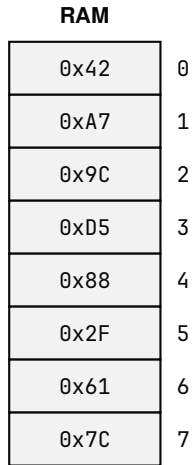
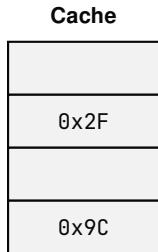
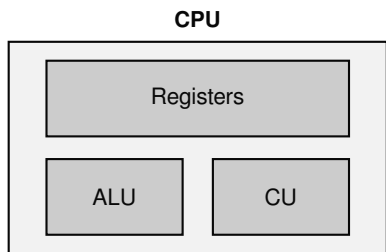
Solution: Smaller, faster memory close to the CPU serving as cache.

Memory access with caching



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

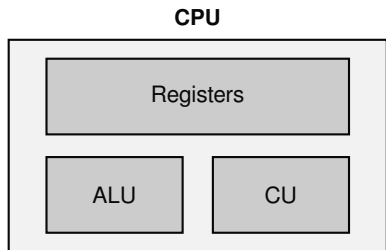
Memory access with caching



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

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

Unfortunately, caches are small



Cache

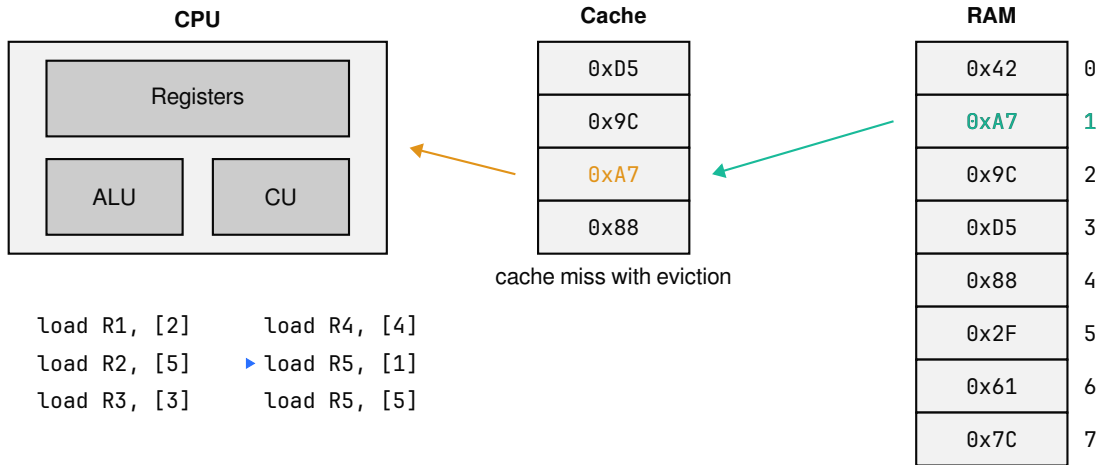
0xD5
0x9C
0x2F
0x88

RAM

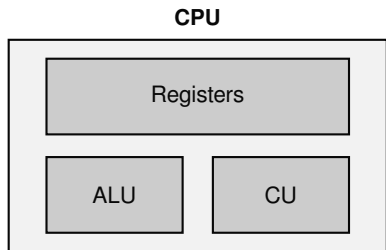
0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

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

Unfortunately, caches are small



Unfortunately, caches are small



Cache

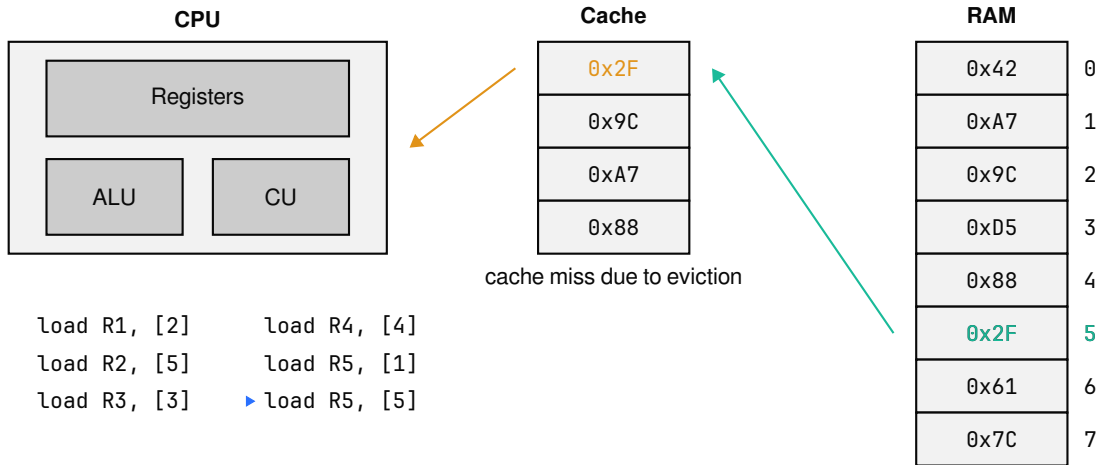
0xD5
0x9C
0xA7
0x88

RAM

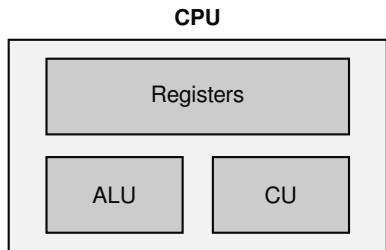
0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

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

Unfortunately, caches are small



Unfortunately, caches are small



Cache

0x2F
0x9C
0xA7
0x88

RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

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

Solution: Hierarchy of caches.

Apple M1 (2020): ²

L1 64-128 KB per core @ 3 cycles

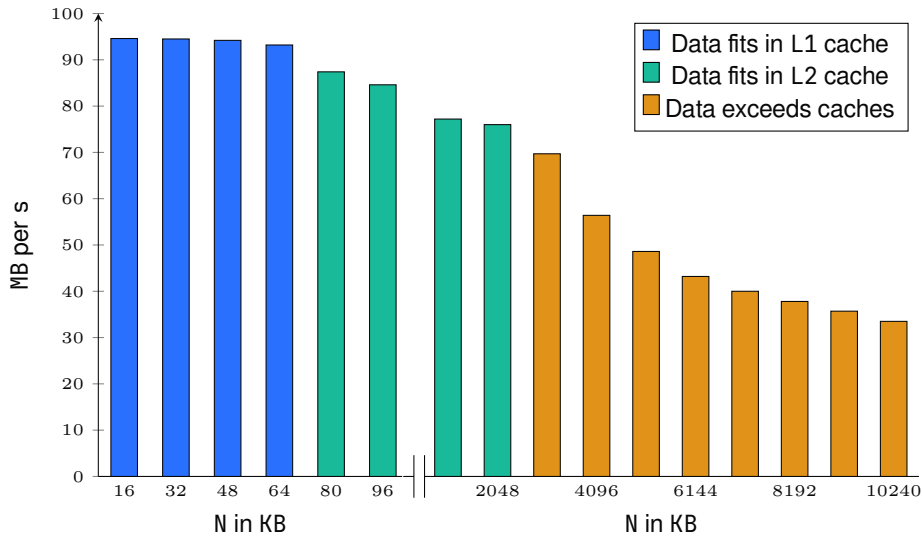
L2 8-12 MB shared @ 18 cycles

L3 8 MB shared with GPU @ 18 cycles + 10-15 ns

RAM 16 GB @ 18 cycles + 100 ns

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

Memory access slows down when data exceeds cache



- Main memory access is slow

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

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

- 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

```
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());
}
```

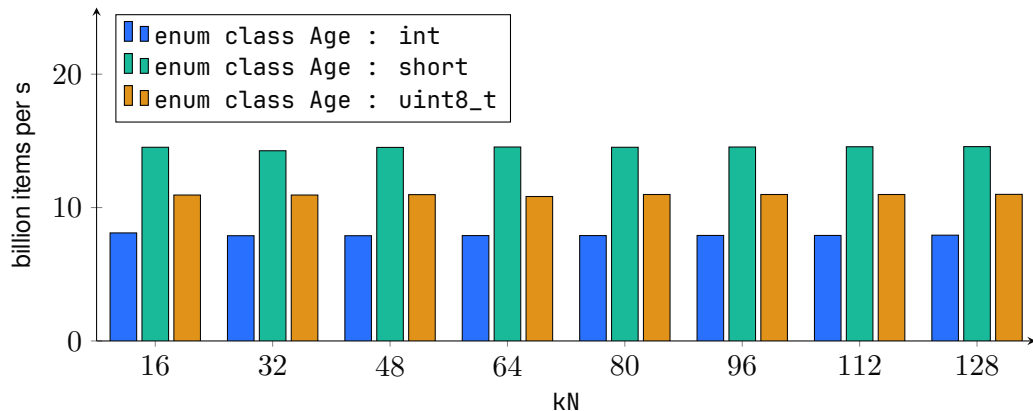
Ensure as much data as possible fits into the cache!

Ensure as much data as possible fits into the cache!

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

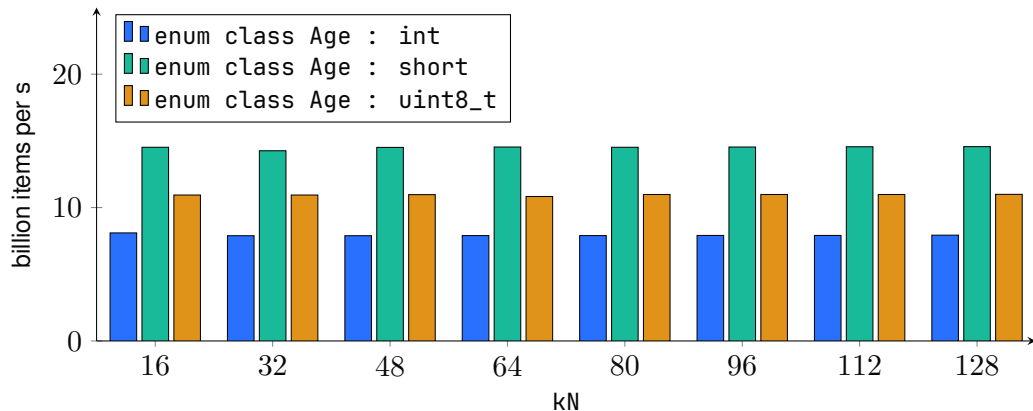
Benchmark results

Averaging random ages between 0 and 100.



Benchmark results

Averaging random ages between 0 and 100.



Smaller size of uint8_t offset by slower byte operations.

Always measure before and after optimizations!

Optimize type size: Pick smaller primitive types

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

Optimize type size: Bitfields

```
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

```
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!

```
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!

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

Optimize type size: Re-ordering members

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

Optimize type size: Re-ordering members

```
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

```
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
```

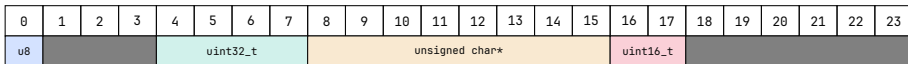
Optimize type size: Re-ordering members

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

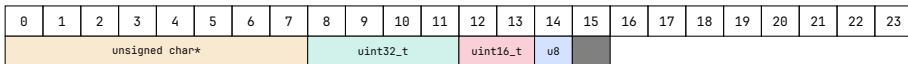
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
u8				uint32_t				unsigned char*								uint16_t							

Optimize type size: Re-ordering members

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



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



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

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

CPU prefetching and cache lines

Wait, why do smaller types help actually?

```
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?

```
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?

Random access:

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

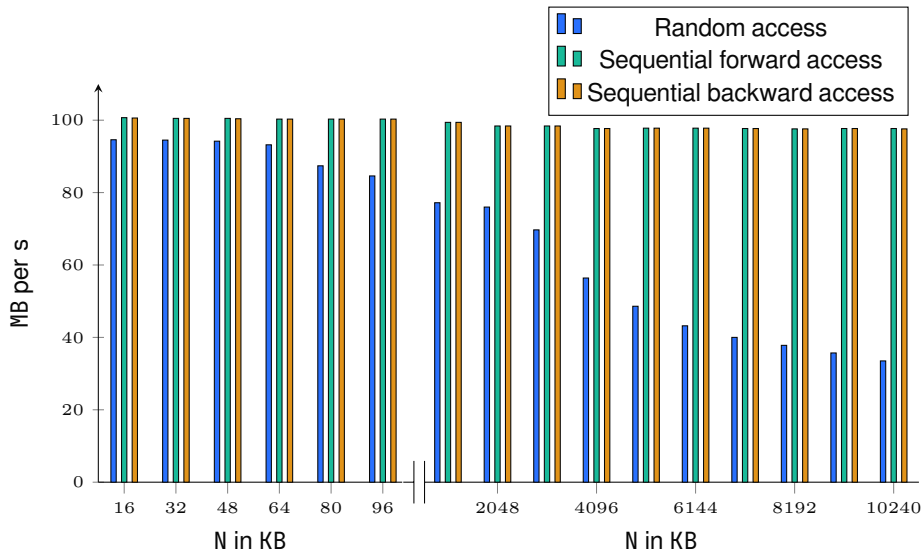
Sequential forward access:

```
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!

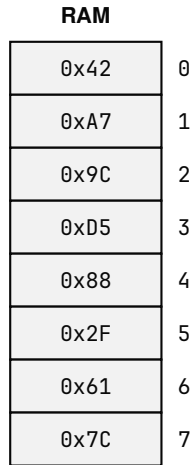
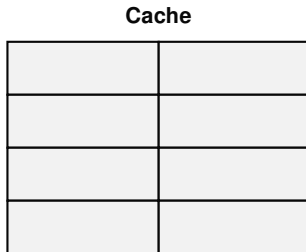
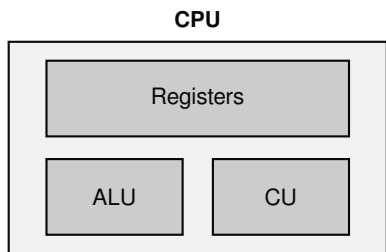


Insight: When accessing address x , usually you also access $x+1$, $x+2$, ...

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

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

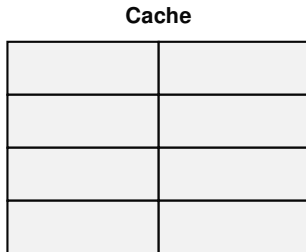
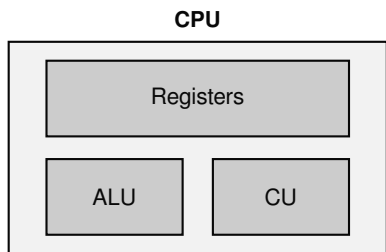
CPU cache lines



```
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



RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

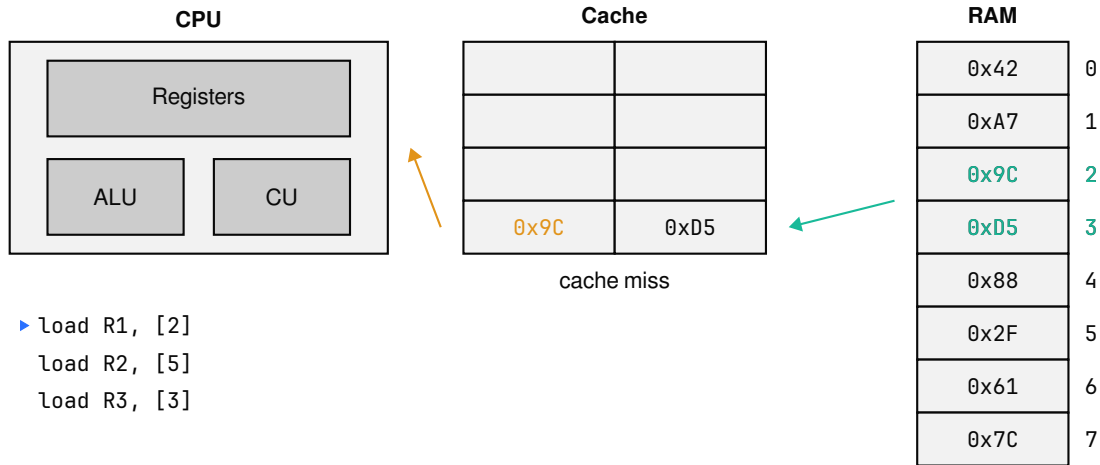
► `load R1, [2]`

`load R2, [5]`

`load R3, [3]`

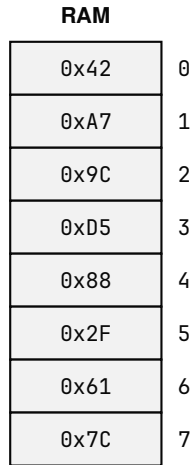
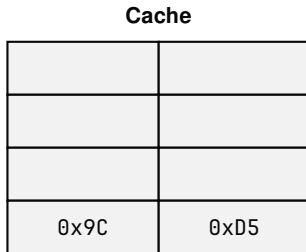
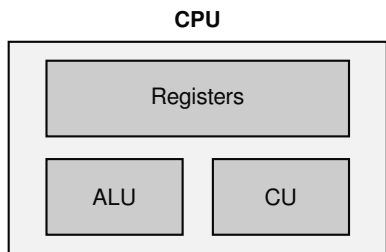
Cache line: Fetch contiguous byte range (64-128 bytes) from RAM into the cache.

CPU cache lines



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

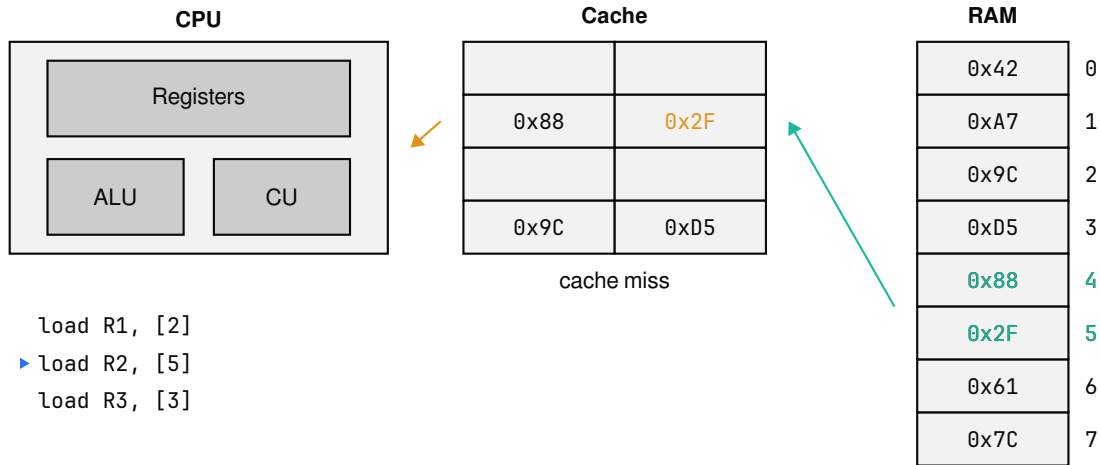
CPU cache lines



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

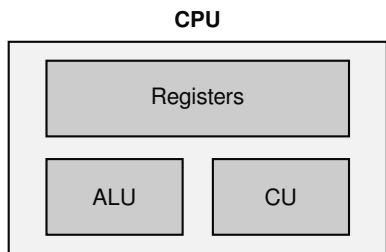
Cache line: Fetch contiguous byte range (64-128 bytes) from RAM into the cache.

CPU cache lines



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

CPU cache lines



Cache

0x88	0x2F
0x9C	0xD5

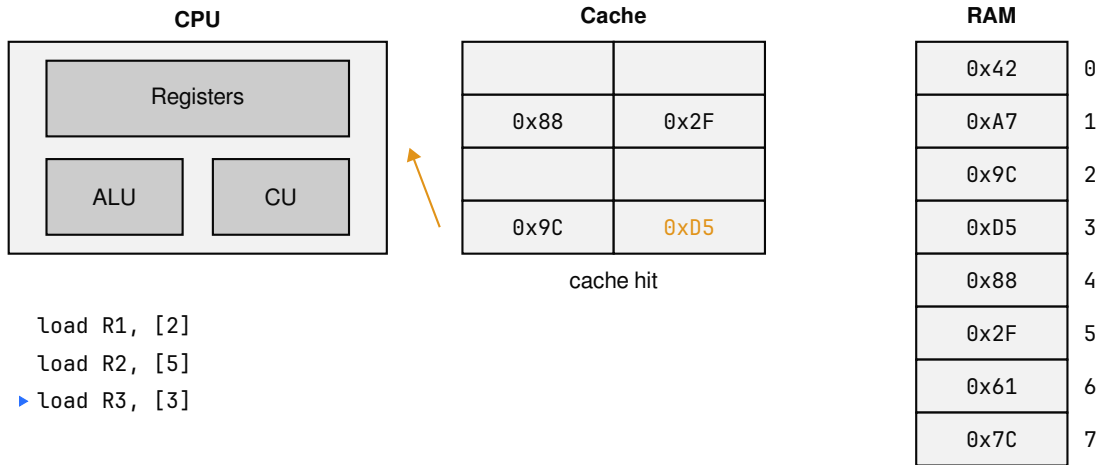
RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

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

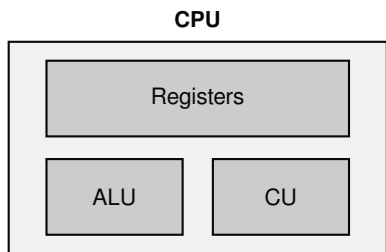
Cache line: Fetch contiguous byte range (64-128 bytes) from RAM into the cache.

CPU cache lines



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

CPU cache lines



Cache

0x88	0x2F
0x9C	0xD5

RAM

0x42	0
0xA7	1
0x9C	2
0xD5	3
0x88	4
0x2F	5
0x61	6
0x7C	7

load R1, [2]

load R2, [5]

load R3, [3]

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

- CPU wants to minimize RAM access.

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

- 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.

- 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?

Predictable access:

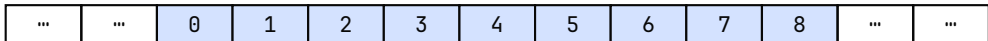
- Avoid pointer chasing → sequential memory access

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

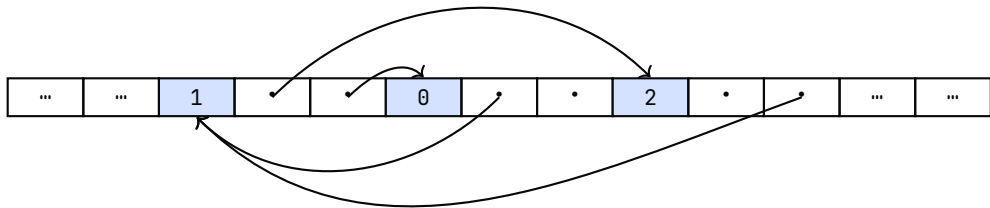
```
T* ptr;  
std::size_t size;  
std::size_t capacity;
```



So is std::array<T, N>, std::inplace_vector<T, N>, ...

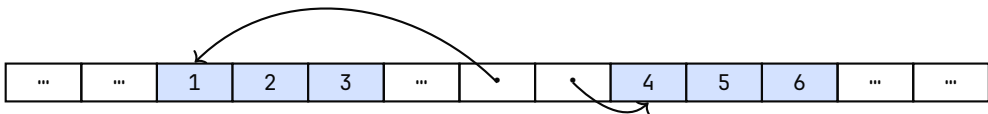
std::list<T> is not cache-friendly

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



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

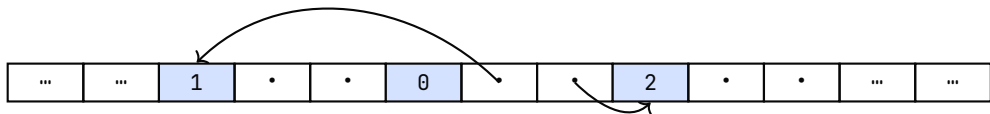
```
using Block = std::array<T, BLOCK_SIZE>;  
std::vector<Block*> blocks;
```



Note: `BLOCK_SIZE` has to be large enough!

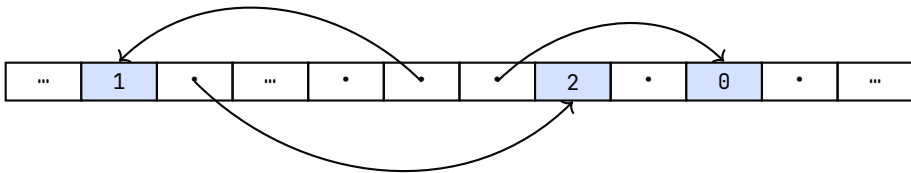
`std::map<K, V>/std::set<T>` are not cache-friendly

```
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

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

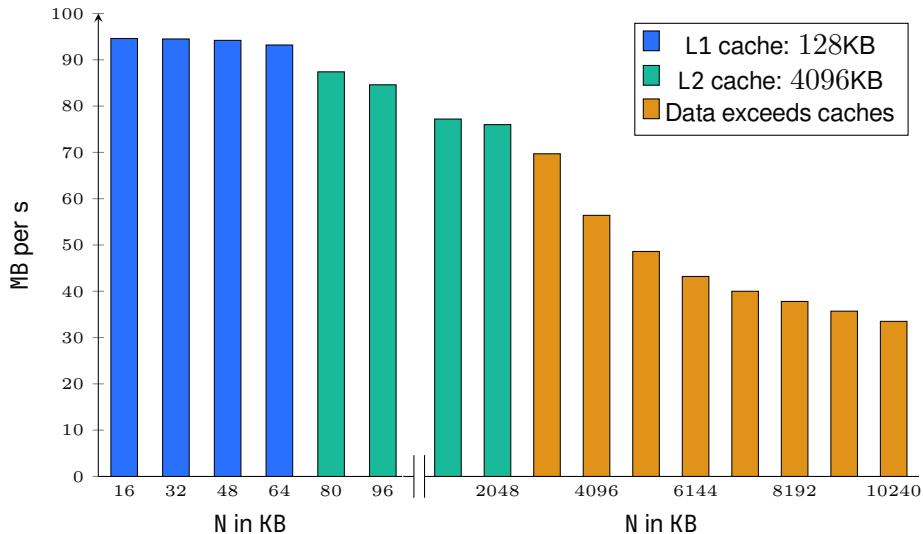
...	...	1		2	0		4		3	5	...	...
-----	-----	---	--	---	---	--	---	--	---	---	-----	-----

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



Code is data too!

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

- 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

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

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

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

Approach: Design the objects first.

Approach: Design the objects first.

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

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

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);
```

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

Approach: Design the algorithms first.

DOD: Records of people

Approach: Design the algorithms first.

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

DOD: Records of people

Approach: Design the algorithms first.

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

DOD: Records of people

Approach: Design the algorithms first.

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

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

DOD: Records of people

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

Array of Structures (AoS):

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

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

Two different data layout approaches

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

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!

Array of Structures

name	age	name	age	name	age	name	age	name	age
------	-----	------	-----	------	-----	------	-----	------	-----

Efficient access to individual rows.

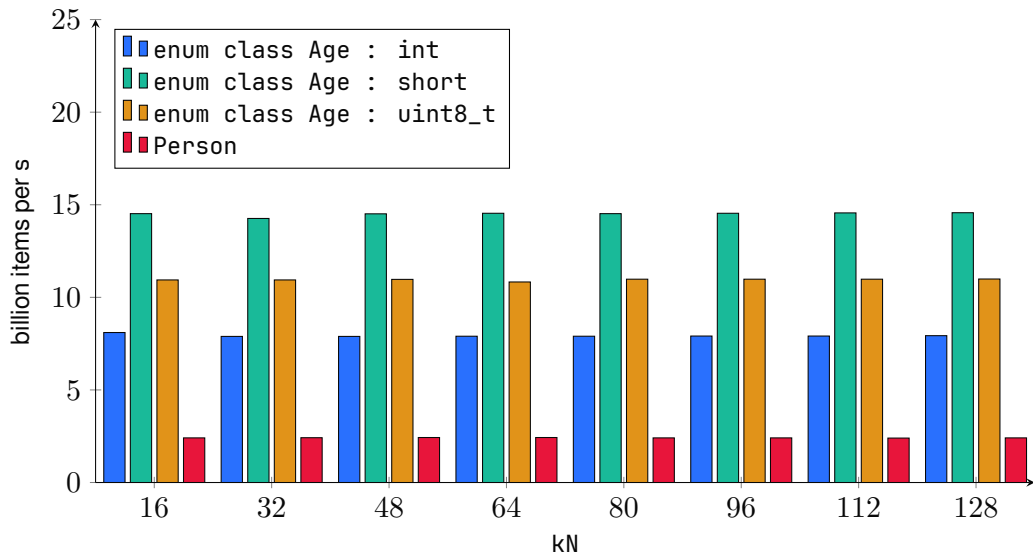
Structure of Arrays

name	name	name	name	name	age	age	age	age	age
------	------	------	------	------	-----	-----	-----	-----	-----

Efficient access to individual columns.

{Illustration ignores padding and size differences.}

SoA doesn't waste space in the cache line



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

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;
}
```

Approach: Design the algorithms first.

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

Approach: Design the algorithms first.

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

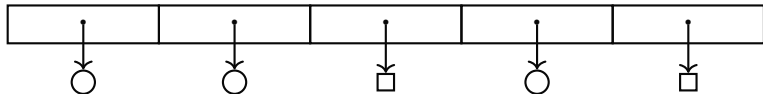
Approach: Design the algorithms first.

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

Representation of heterogeneous data

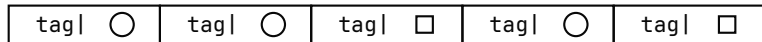
Boxed

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



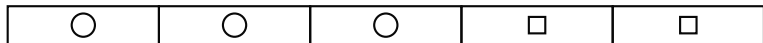
Tagged union

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

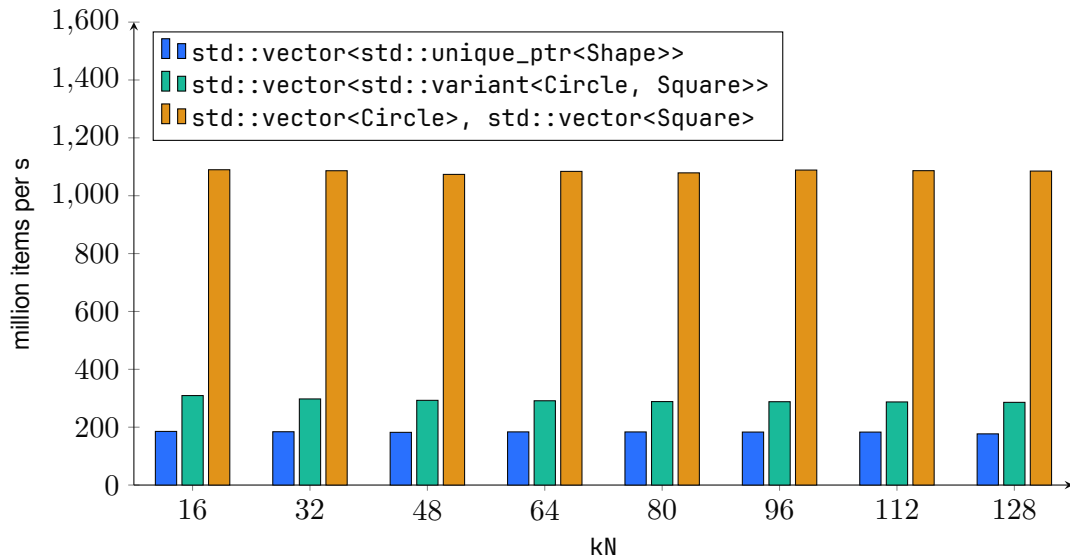


Multiple homogeneous arrays

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



Multiple homogeneous arrays are significantly faster



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

```
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);
```

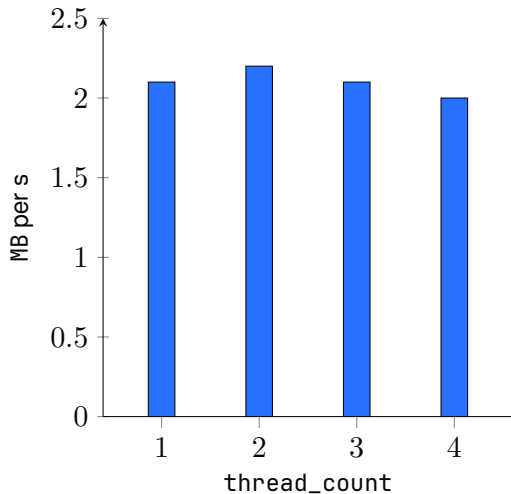
Silly parallel fibonacci example

```
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

```
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!



- Each core has its own L1 cache

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

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

Thread 0 writes:

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

Thread 1 reads:

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

Thread 0 writes:

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

Thread 1 reads:

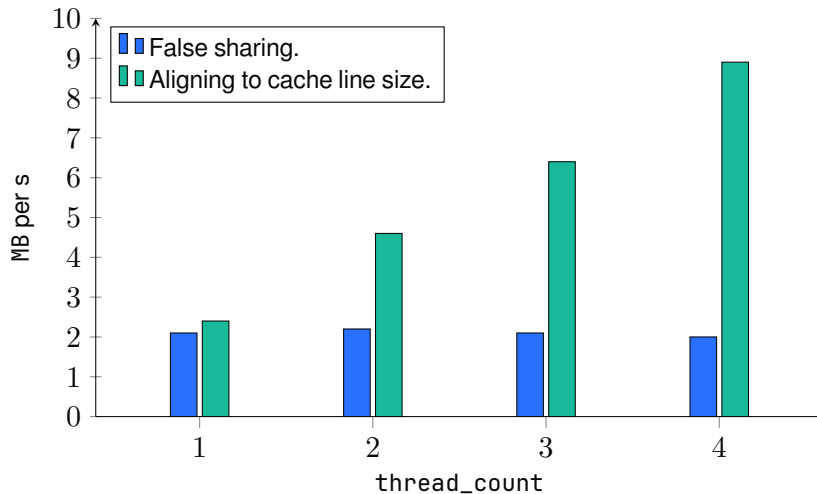
```
// ... therefore, cache miss!  
auto value = results[1];
```

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

Preventing false sharing by aligning to cache line size

```
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);
}
```

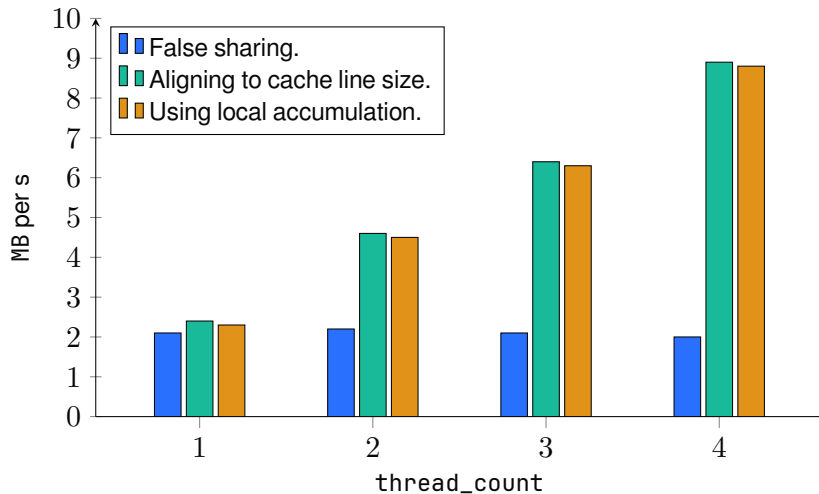
Now it scales!



Preventing false sharing using local accumulation

```
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



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

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

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

Cache line smallest unit of data transferred between main memory and caches

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

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.

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!

- 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](#)

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.