More Speed & Simplicity

Practical Data-Oriented Design in C++

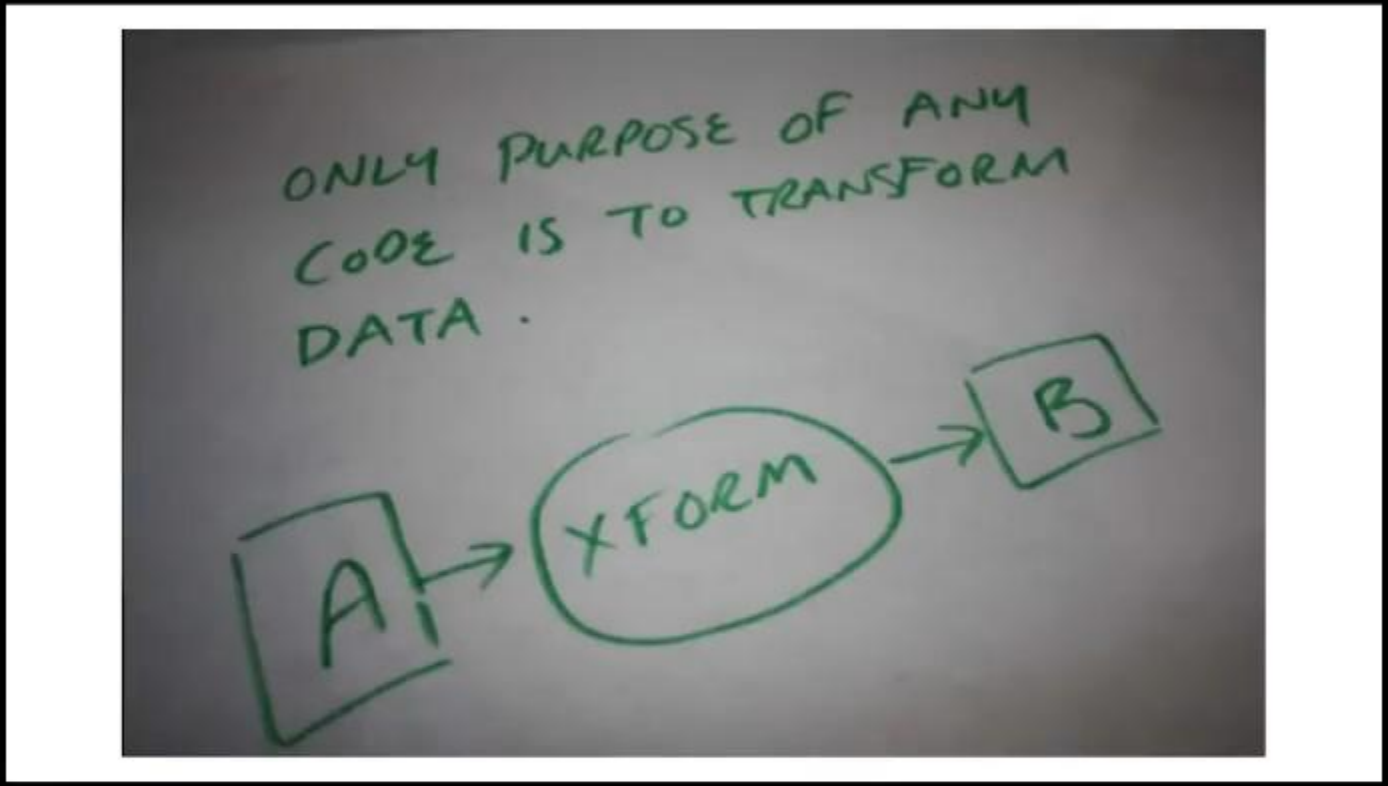
VITTORIO ROMEO

More Speed & Simplicity

Practical Data-Oriented Design in C++

Vittorio Romeo
https://romeo.training

cppcon



DATA-ORIENTED DESIGN AND C++

Mike Acton


cppcon

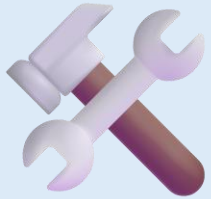
About **myself**

- C++ consultant, trainer, and mentor
  https://romeo.training
- 10 years of experience @ Bloomberg
  Microservices & market data analytics
  Technical training
- ISO C++ Standardization
  `std::function_ref`
- Open-source contributions
  SFML, SDL, libraries, games, virtual reality
- «Embracing Modern C++ Safely»
  J. Lakos, V. Romeo, R. Khlebnikov, A. Meredith

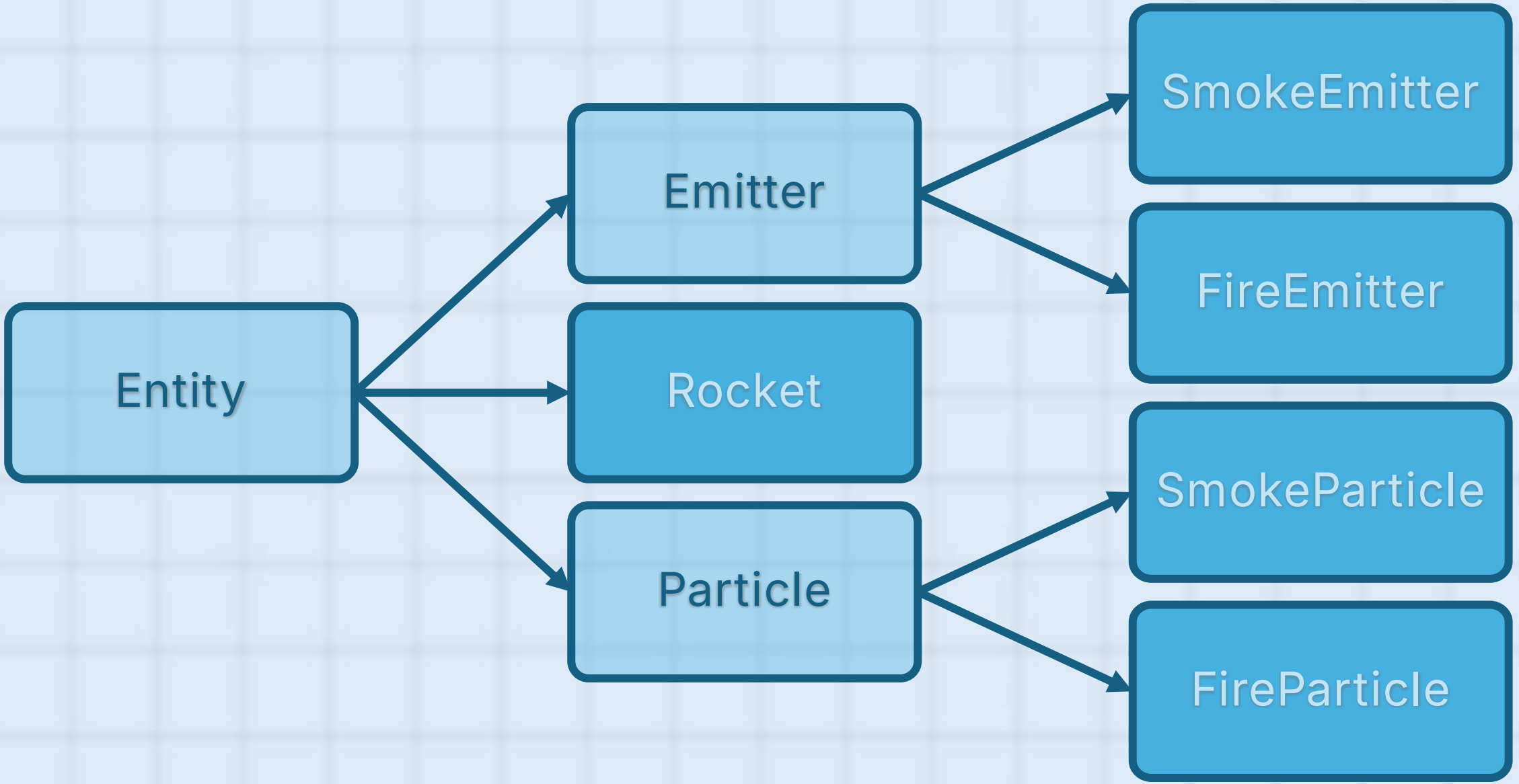
Today’s **goals**

- Discover a new mindset
  Learn how thinking *data-first* changes the way we design systems
- See the hardware for what it is
  Understand how modern CPUs and memory *really* work
- Measure the impact
  See how reorganizing data unlocks *dramatic* performance gains
- Appreciate side benefits
  Discover how DOD makes code *simpler* and more *maintainable*
- Find the right balance
  Recognize where OOP and C++ abstractions still shine

Time for a **live demo**!



Let’s get to **work**!


Entity
Emitter
Rocket
Particle
SmokeEmitter
FireEmitter
SmokeParticle
FireParticle

```
struct Entity
{

};
```

```
struct Entity
{
    virtual ~Entity() = default;
};
```

```
struct Entity
{
    virtual ~Entity() = default;

    virtual void update(float dt);

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    virtual ~Entity() = default;

    virtual void update(float dt);

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Vector2f
{
    float x;
    float y;

    // ... operator overloads ...
};
```

```
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{



};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {

    }
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);
    }
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;
    }
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};
```

```cpp
struct SmokeParticle : Particle
{
    void draw(sf::RenderTarget& rt) override
    {
        // ... draw using smoke texture ...
    }
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};



struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};



struct SmokeParticle : Particle
{
    void draw(sf::RenderTarget& rt) override
    {
        // ... draw using smoke texture ...
    }
};
```

```cpp
struct World
{
```

```cpp
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};



struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};



struct SmokeParticle : Particle
{
    void draw(sf::RenderTarget& rt) override
    {
        // ... draw using smoke texture ...
    }
};
```

```cpp
struct World
{
    std::vector<std::unique_ptr<Entity>> entities;

    void update(float dt)
    {
        for (auto& entity : entities)
            entity->update(dt);
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const auto& entity : entities)
            entity->draw(rt);
    }

    void cleanup()
    {
        std::erase_if(entities, [](const auto& entity) { return !entity->alive; });
    }
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};
```

```cpp
struct SmokeParticle : Particle
{
    void draw(sf::RenderTarget& rt) override
    {
        // ... draw using smoke texture ...
    }
};
```

```cpp
struct World
{
    std::vector<std::unique_ptr<Entity>> entities;

    void update(float dt)
    {
        for (auto& entity : entities)
            entity->update(dt);
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const auto& entity : entities)
            entity->draw(rt);
    }

    void cleanup()
    {
        std::erase_if(entities, [](const auto& entity) { return !entity->alive; });
    }
};
```

```cpp
struct Emitter : Entity
{




};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct World
{
    std::vector<std::unique_ptr<Entity>> entities;

    void update(float dt)
    {
        for (auto& entity : entities)
            entity->update(dt);
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const auto& entity : entities)
            entity->draw(rt);
    }

    void cleanup()
    {
        std::erase_if(entities, [](const auto& entity) { return !entity->alive; });
    }
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};
```

```cpp
struct SmokeParticle : Particle
{
    void draw(sf::RenderTarget& rt) override
    {
        // ... draw using smoke texture ...
    }
};
```

```cpp
struct Emitter : Entity
{
    float spawnTimer, spawnRate;

    virtual void spawnParticle() = 0;

    void update(float dt) override
    {
        // ... periodically call `spawnParticle` ...
    }
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};
```

```cpp
struct SmokeParticle : Particle
{
    void draw(sf::RenderTarget& rt) override
    {
        // ... draw using smoke texture ...
    }
};
```

```cpp
struct World
{
    std::vector<std::unique_ptr<Entity>> entities;

    void update(float dt)
    {
        for (auto& entity : entities)
            entity->update(dt);
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const auto& entity : entities)
            entity->draw(rt);
    }

    void cleanup()
    {
        std::erase_if(entities, [](const auto& entity) { return !entity->alive; });
    }
};
```

```cpp
struct Emitter : Entity
{
    float spawnTimer, spawnRate;

    virtual void spawnParticle() = 0;

    void update(float dt) override
    {
        // ... periodically call `spawnParticle` ...
    }
};
```

```cpp
struct SmokeEmitter : Emitter
{




};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct World
{
    std::vector<std::unique_ptr<Entity>> entities;

    void update(float dt)
    {
        for (auto& entity : entities)
            entity->update(dt);
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const auto& entity : entities)
            entity->draw(rt);
    }

    void cleanup()
    {
        std::erase_if(entities, [](const auto& entity) { return !entity->alive; });
    }
};
```

```cpp
struct Emitter : Entity
{
    float spawnTimer, spawnRate;

    virtual void spawnParticle() = 0;

    void update(float dt) override
    {
        // ... periodically call `spawnParticle` ...
    }
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};
```

```cpp
struct SmokeParticle : Particle
{
    void draw(sf::RenderTarget& rt) override
    {
        // ... draw using smoke texture ...
    }
};
```

```cpp
struct SmokeEmitter : Emitter
{
    void spawnParticle() override
    {
        auto p = std::make_unique<SmokeParticle>();
        // ... set particle data ...
        world->entities.push_back(std::move(p));
    }
};
```

```cpp
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

```cpp
struct Particle : Entity
{
    float scale, scaleRate;
    float opacity, opacityChange;
    float rotation, angularVelocity;

    void update(float dt) override
    {
        Entity::update(dt);

        scale += scaleRate * dt;
        opacity += opacityChange * dt;
        rotation += angularVelocity * dt;

        alive = opacity > 0.f;
    }
};
```

```cpp
struct SmokeParticle : Particle
{
    void draw(sf::RenderTarget& rt) override
    {
        // ... draw using smoke texture ...
    }
};
```

```cpp
struct World
{
    std::vector<std::unique_ptr<Entity>> entities;

    void update(float dt)
    {
        for (auto& entity : entities)
            entity->update(dt);
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const auto& entity : entities)
            entity->draw(rt);
    }

    void cleanup()
    {
        std::erase_if(entities, [](const auto& entity) { return !entity->alive; });
    }
};
```

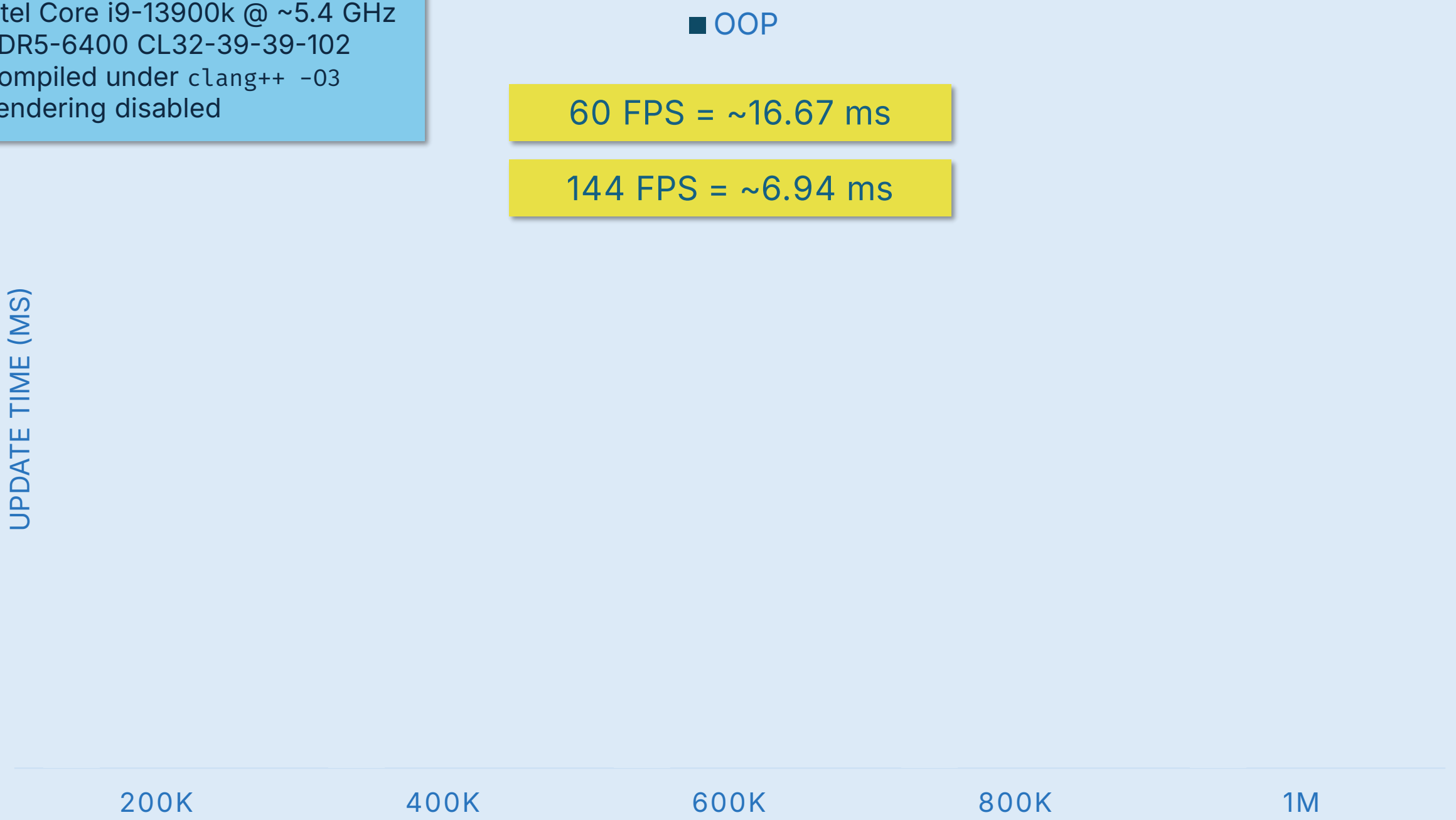
```cpp
struct Emitter : Entity
{
    float spawnTimer, spawnRate;

    virtual void spawnParticle() = 0;

    void update(float dt) override
    {
        // ... periodically call `spawnParticle` ...
    }
};
```

```cpp
struct SmokeEmitter : Emitter
{
    void spawnParticle() override
    {
        auto p = std::make_unique<SmokeParticle>();
        // ... set particle data ...
        world->entities.push_back(std::move(p));
    }
};
```

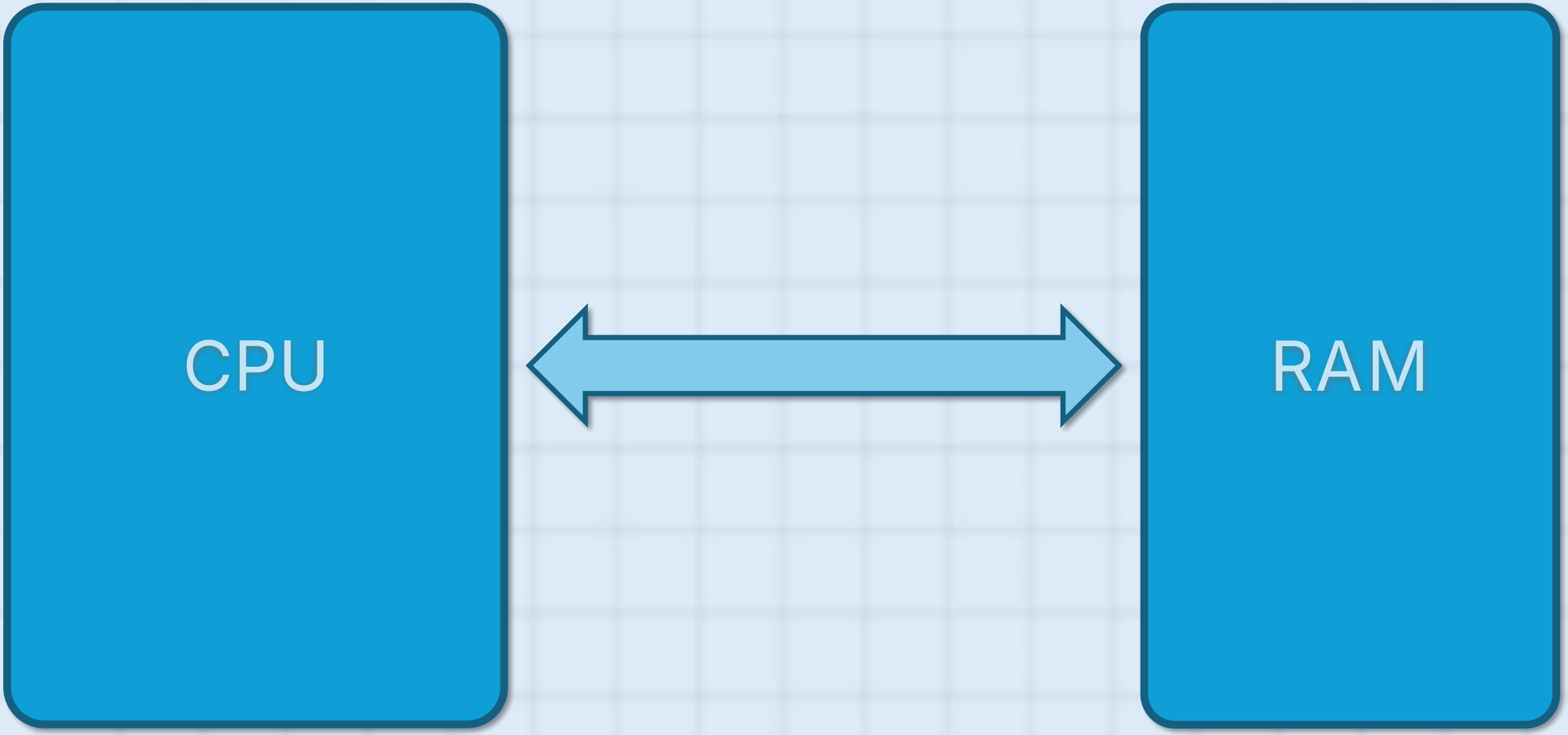
```cpp
struct Rocket : Entity
{
};
```

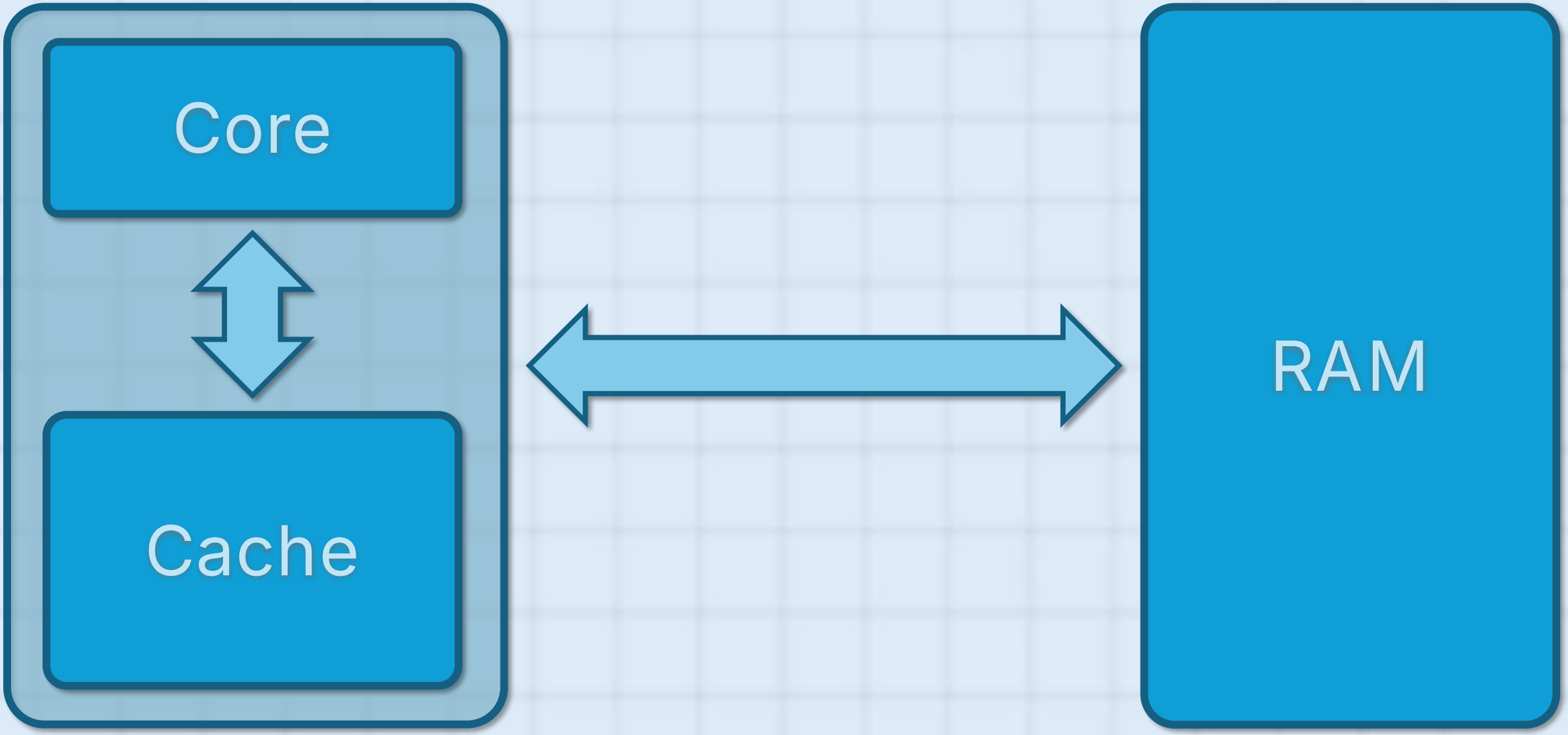
First round of **benchmarks**!

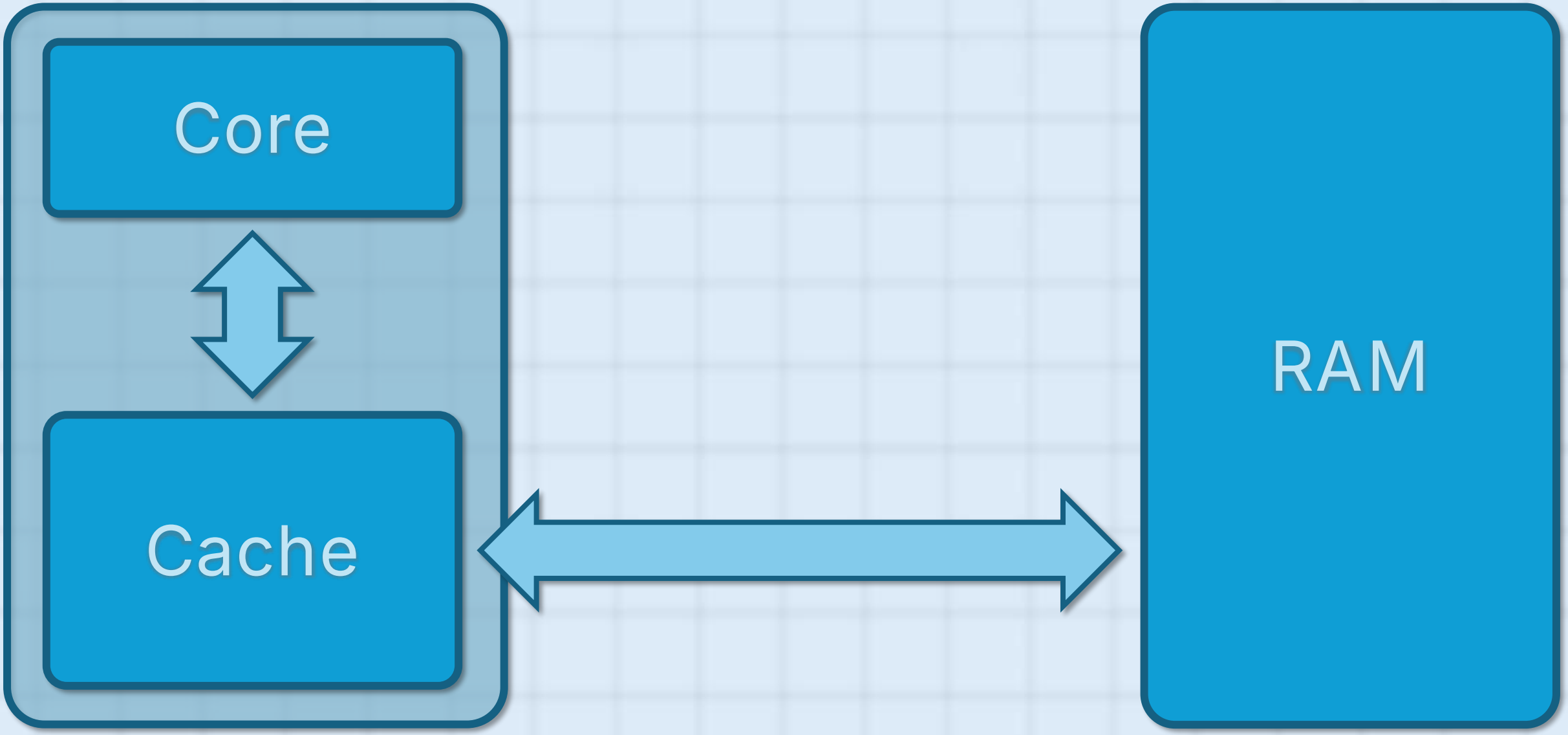
- Intel Core i9-13900k @ ~5.4 GHz
- DDR5-6400 CL32-39-39-102
- Compiled under `clang++ -O3`
- Rendering disabled

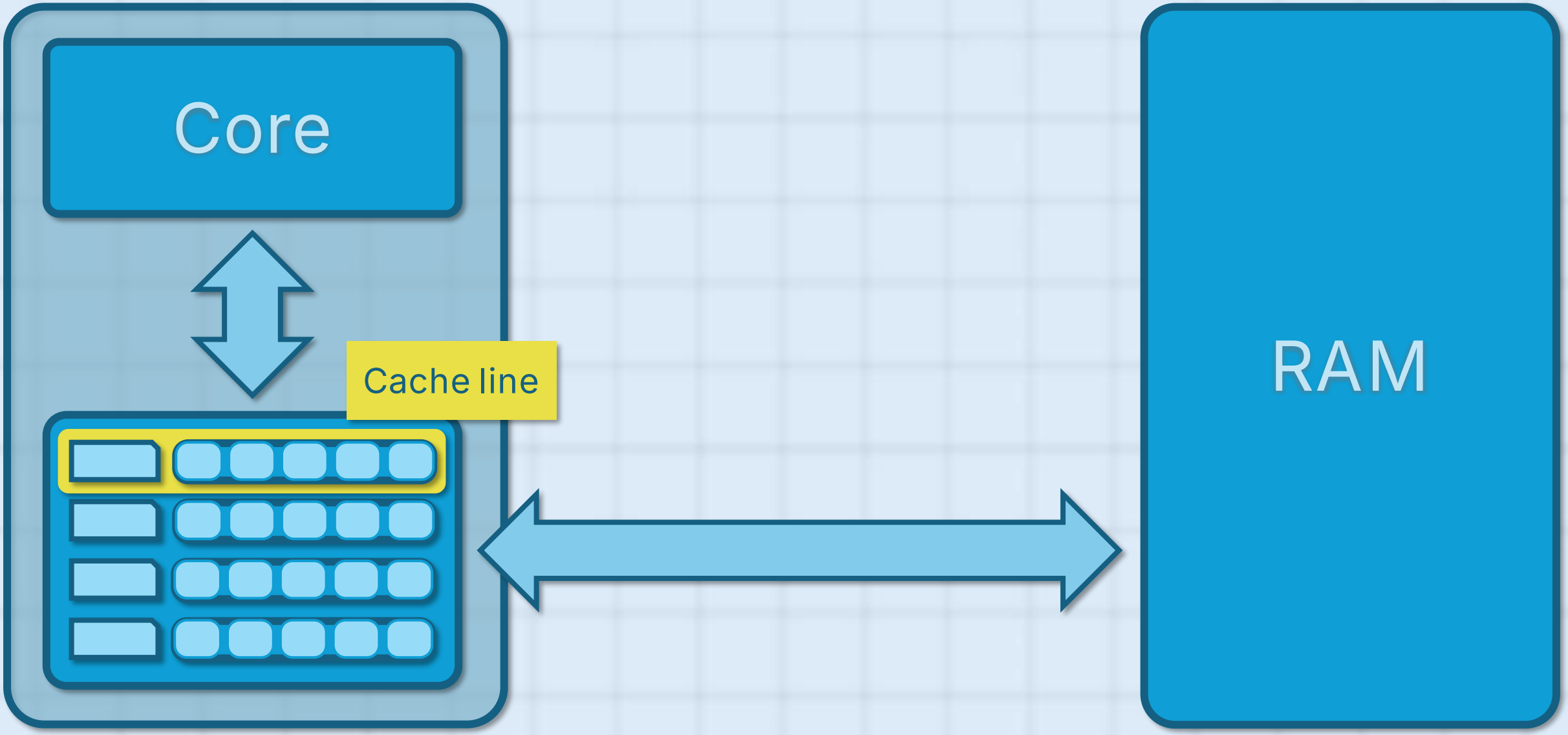


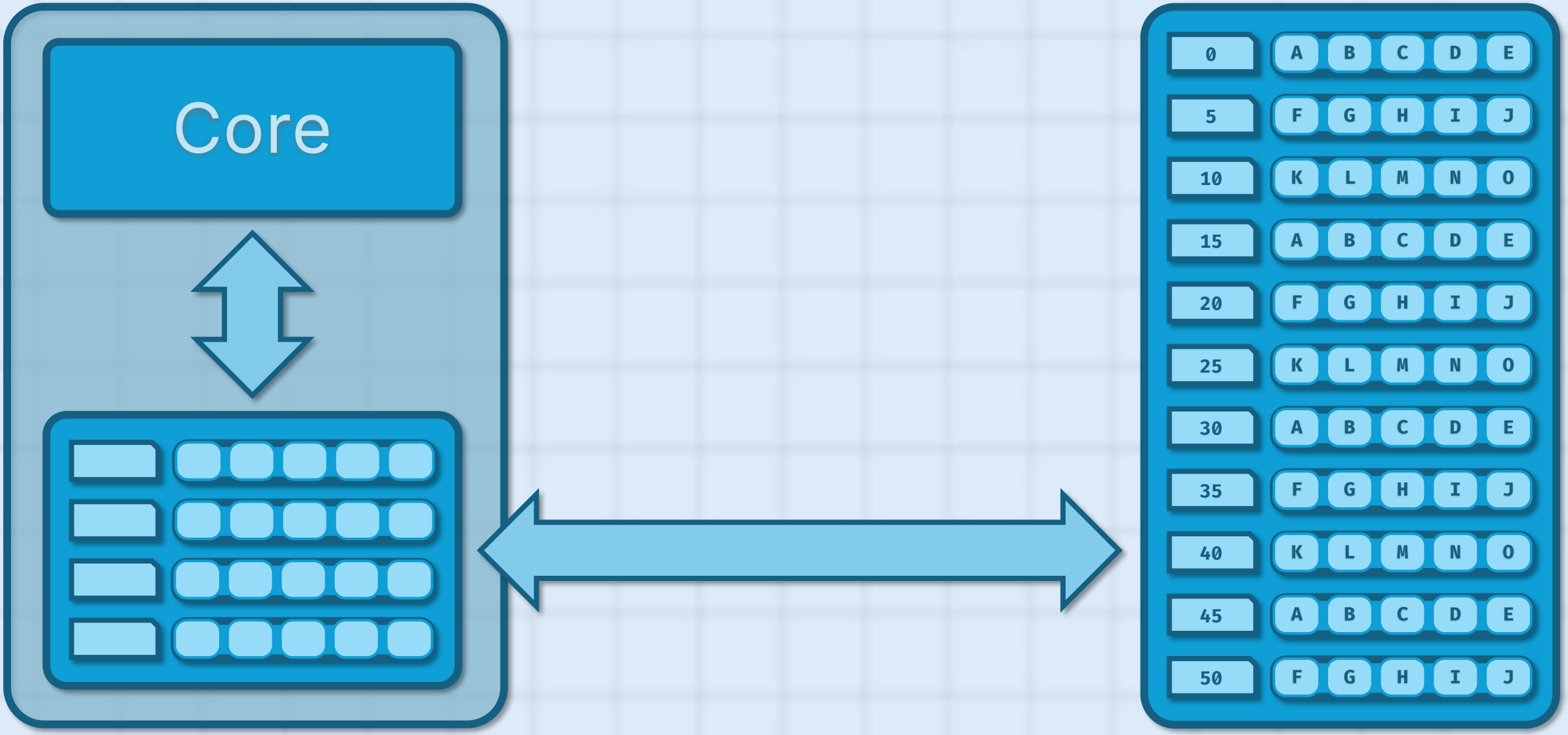
Time for a refresher on **memory**!


CPU
RAM


Core
Cache
RAM


Core
Cache
RAM

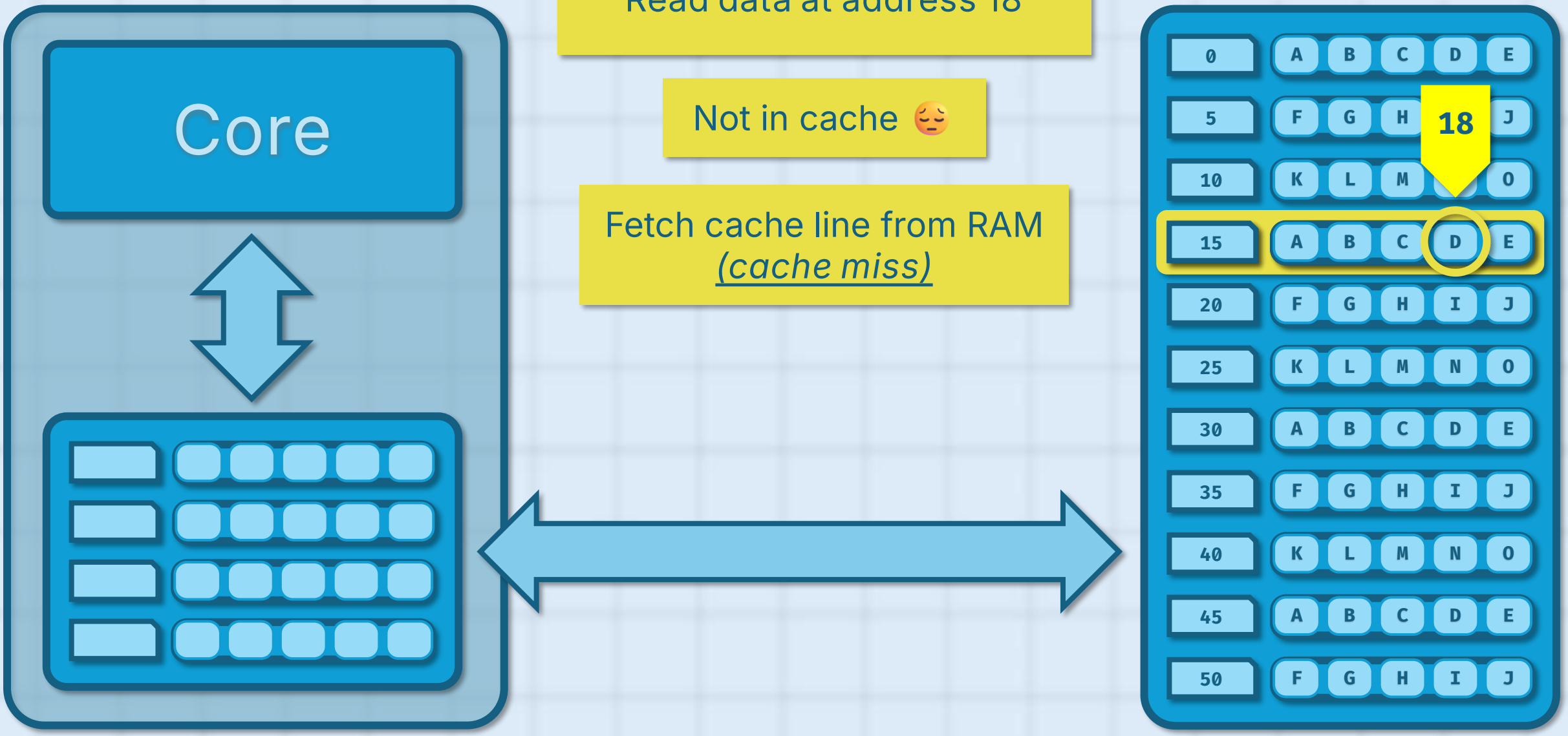

Core
Cache line
RAM

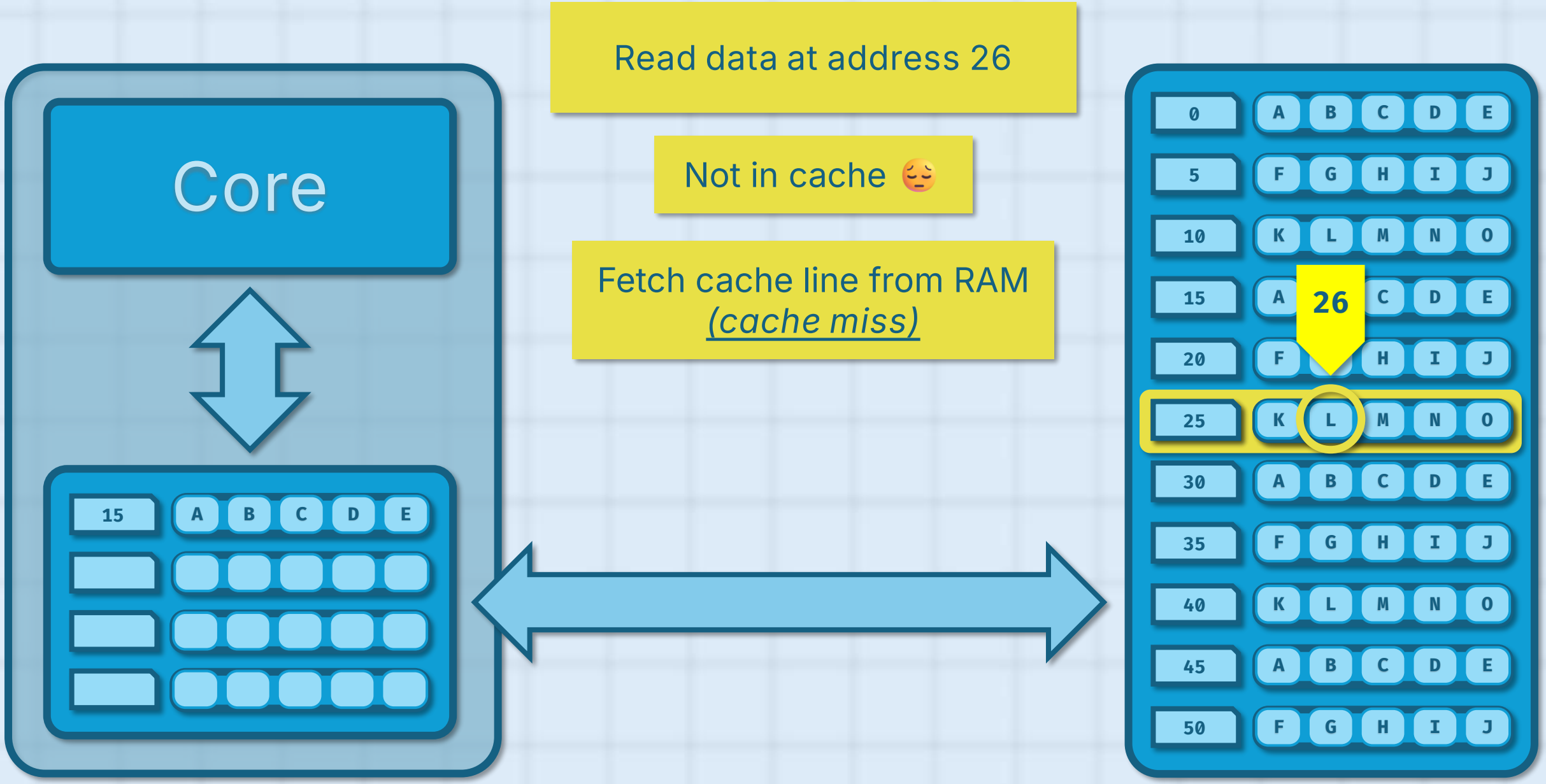

Core
0 A B C D E
5 F G H I J
10 K L M N O
15 A B C D E
20 F G H I J
25 K L M N O
30 A B C D E
35 F G H I J
40 K L M N O
45 A B C D E
50 F G H I J

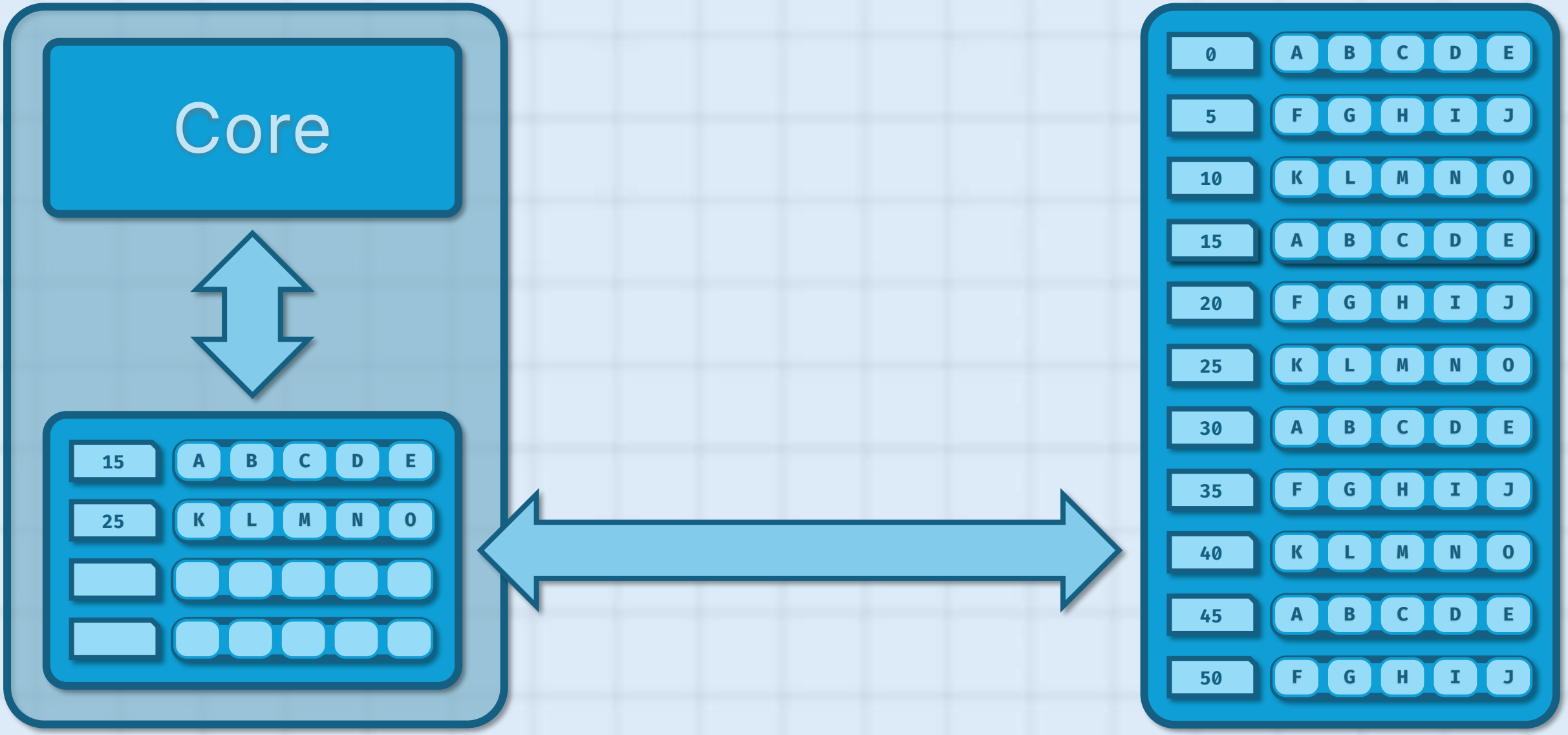
Read data at address 18

Not in cache 😔

Fetch cache line from RAM
(cache miss)



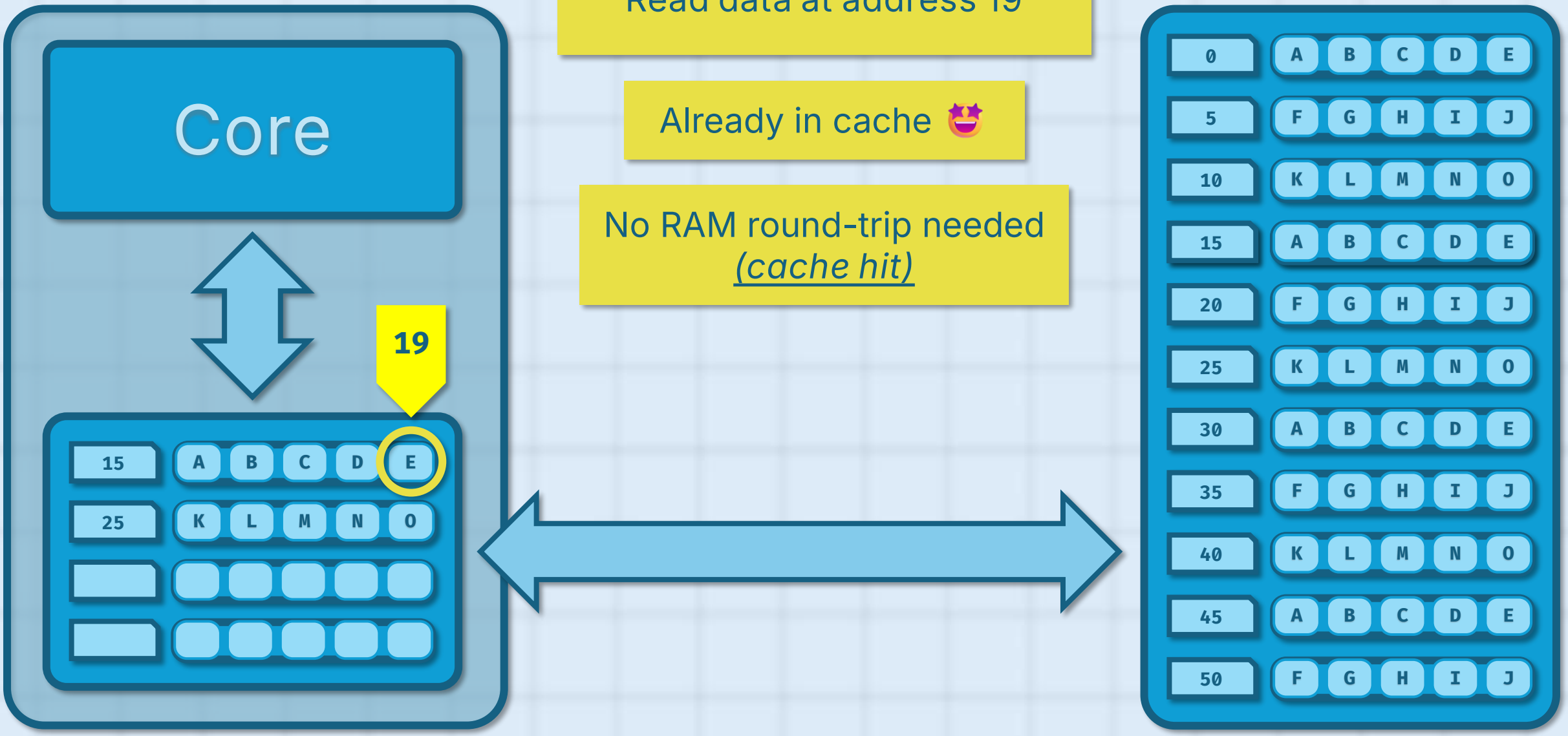

Read data at address 26
Not in cache 😔
Fetch cache line from RAM
(cache miss)
Core
15
A
B
C
D
E
0
5
10
15
20
25
30
35
40
45
50
F
G
H
I
J
K
L
M
N
O
26


Core
15
A B C D E
25
K L M N O
0
A B C D E
5
F G H I J
10
K L M N O
15
A B C D E
20
F G H I J
25
K L M N O
30
A B C D E
35
F G H I J
40
K L M N O
45
A B C D E
50
F G H I J

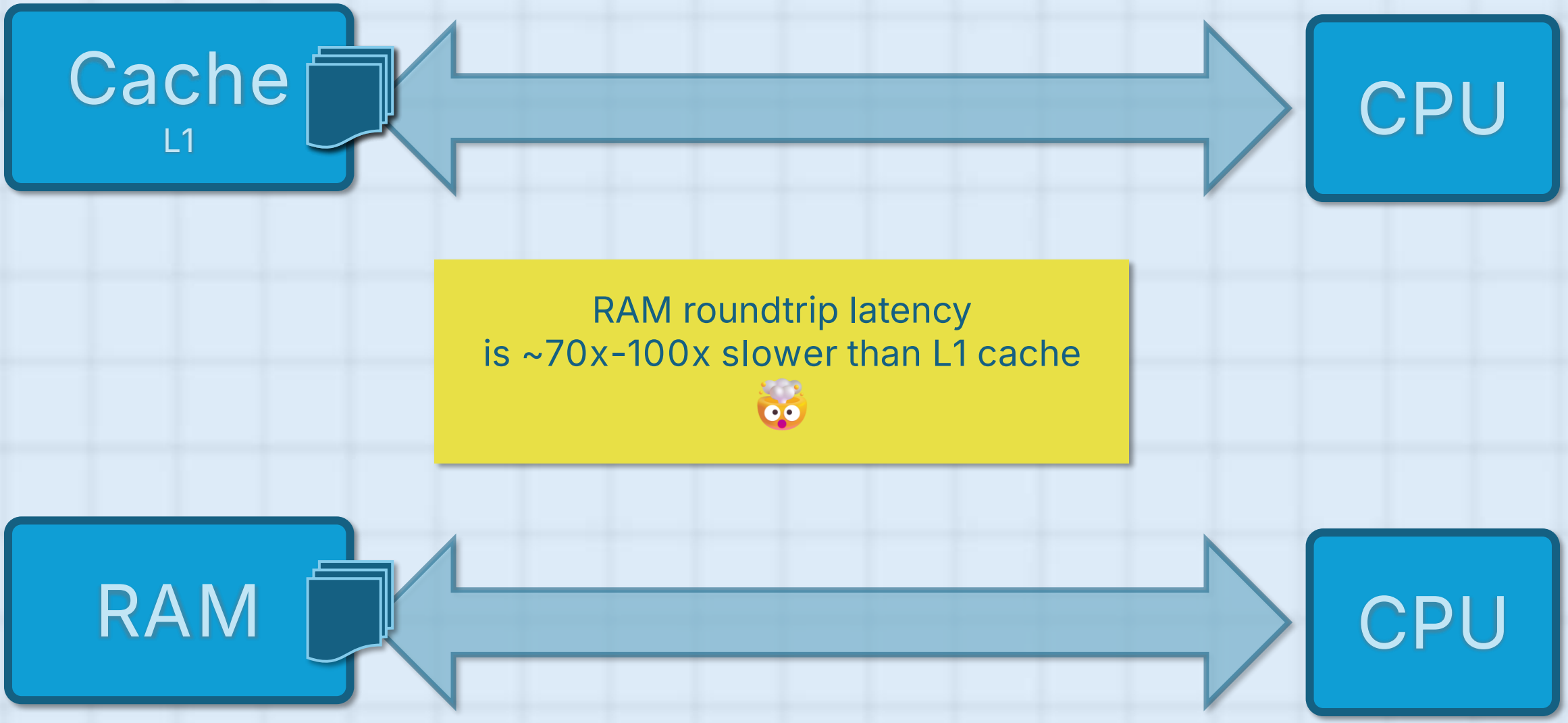
Read data at address 19

Already in cache 🤩

No RAM round-trip needed
(cache hit)



How **much** does this matter?


Cache
L1
CPU
RAM roundtrip latency
is ~70x-100x slower than L1 cache
RAM
CPU

What have we **learned**?

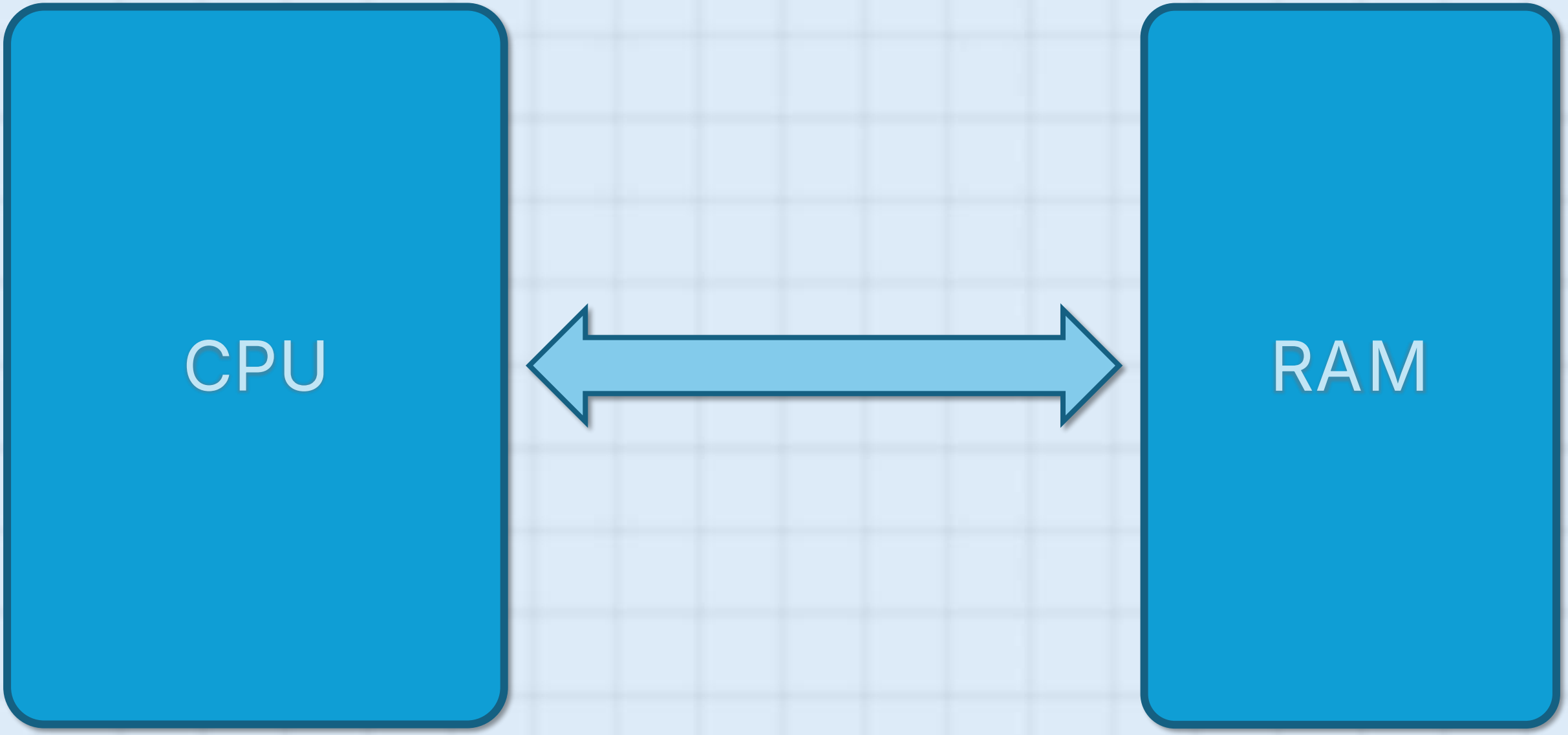
- Smallest unit of transferable memory: cache line
  Generally ~64B on modern desktop CPUs

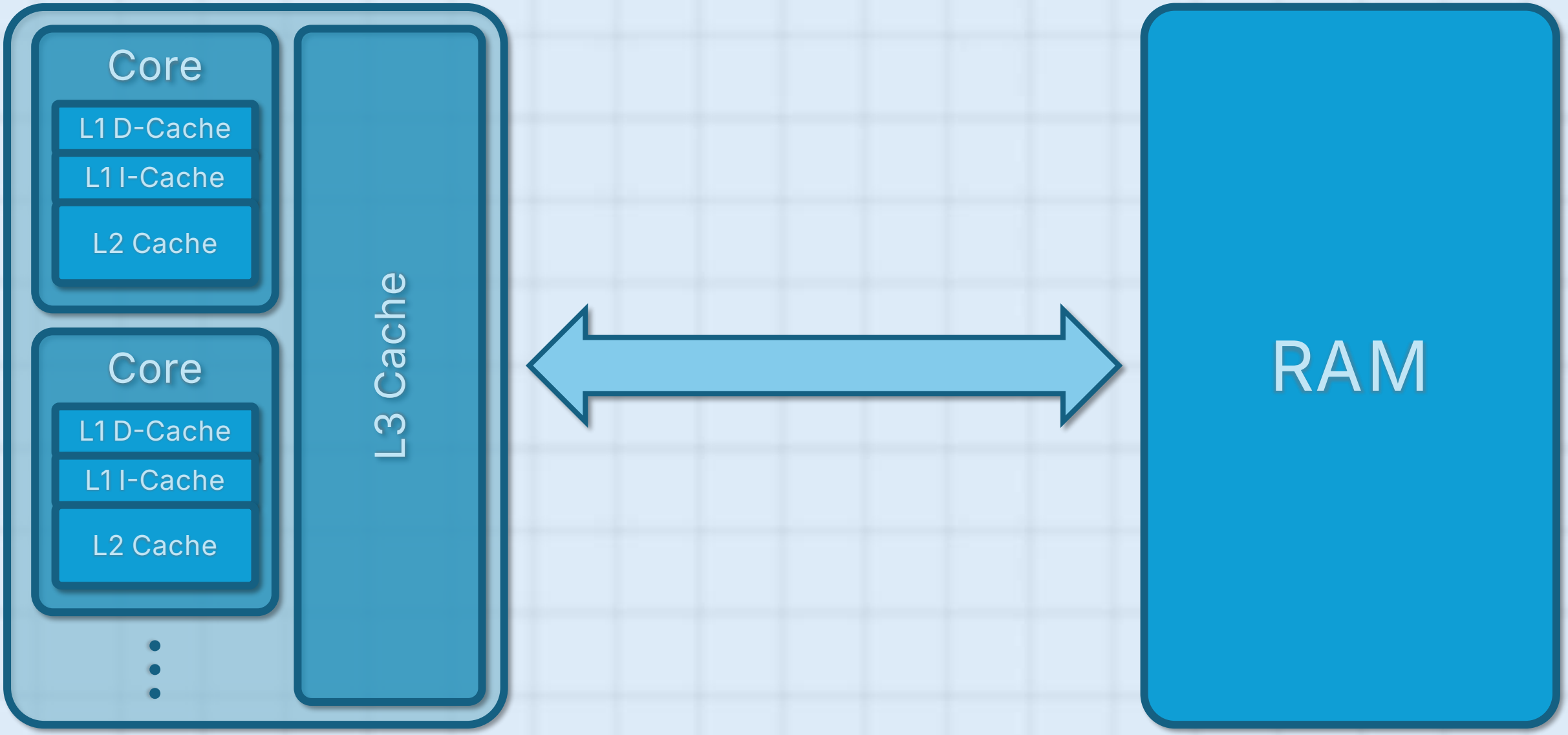
- All data must traverse the memory hierarchy
  CPU↔RAM latency can be ~100x slower than CPU↔L1 Cache

- Spatial locality of data *greatly* affects performance

💡 Store related data close together in memory
  Prefer flat/contiguous storage

💡 Access data in predictable patterns
  Helps CPU prefetch probably-useful cache lines


CPU
RAM


Core
L1 D-Cache
L1 I-Cache
L2 Cache
Core
L1 D-Cache
L1 I-Cache
L2 Cache
L3 Cache
RAM

CPU Caches and Why You Care

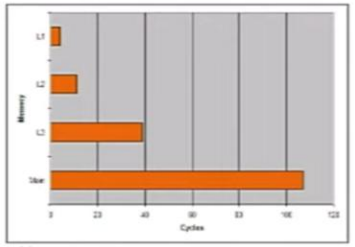
Scott Meyers
code::dive conference 2014

CPU Cache Characteristics

Caches are small.

- Assume 100MB program at runtime (code + data).
  - 8% fits in core-i79xx's L3 cache.
    - L3 cache shared by *every running process* (incl. OS).
  - 0.25% fits in each L2 cache.
  - 0.03% fits in each L1 cache.

Caches much faster than main memory.



- For Core i7-9xx:
  - L1 latency is 4 cycles.
  - L2 latency is 11 cycles.
  - L3 latency is 39 cycles.
  - Main memory latency is 107 cycles.
    - 27 times slower than L1!
    - 100% CPU utilization ⇒ >99% CPU idle time!

Why is our OOP implementation **slow**?

```
struct World
{
    std::vector<std::unique_ptr<Entity>> entities;

    void update(float dt)
    {
        for (auto& entity : entities)
            entity->update(dt);
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const auto& entity : entities)
            entity->draw(rt);
    }

    void cleanup()
    {
        std::erase_if(entities, [](const auto& entity) { return !entity->alive; });
    }
};
```

```
struct SmokeEmitter : Emitter
{
    void spawnParticle() override
    {
        auto p = std::make_unique<SmokeParticle>();
        // ... set particle data ...
        world->entities.push_back(std::move(p));
    }
};
```

DATA IS SCATTERED AROUND IN MEMORY

CACHE MISS PER ITERATION

VIRTUAL DISPATCH OVERHEAD

CACHE MISS PER ITERATION

VIRTUAL DISPATCH OVERHEAD

CACHE MISS PER ITERATION

FREQUENT DYNAMIC DEALLOCATION

FREQUENT DYNAMIC ALLOCATION

The object-oriented mindset

🤖 Model a world of autonomous objects

Self-contained agents with their own *identity* and *responsibilities*

📬 Communicate through messages

The main loop *asks* the entities to update or draw *themselves*

🔒 Hide data, expose behavior

The world doesn't care *how* a particle gets updated, only that it does

🔮 Plan for the unknown

Bet on an *interface* that allows us to extend functionality in the future

The data-oriented mindset

🔄 **Model a world of data transformation**

Code is a pipeline that transforms *data* from one state to another

📚 **Operate directly on batches of data**

The main loop is in *control*, directly manipulating data in bulk

👁️ **Expose data, centralize behavior**

Data is *transparent* and laid out for efficient processing, not hidden

⚡ **Plan for today**

Design for the problem at hand, prioritize *performance* and *simplicity*

Shifting your mindset

🧮 The *only* purpose of code is to transform data

Focus should be on the data's journey, not modeling abstract objects

🎯 Data is the *centerpiece*, not something to be hidden

Understand its shape, size, and access patterns

🛣️ Computers thrive on *simple*, *predictable* work

Feed them long, straight runs of contiguous data

🖥️ Design for the *machine*, not the *metaphor*

Effective solutions are aligned with the physical reality of hardware

First **optimization** pass

- Avoid individual heap allocations
  Get rid of `std::unique_ptr`
- Flatten the hierarchy
  Begone, inheritance!
- Decouple data from logic
  Entities are just data, the world deals with the behavior
- Store each type in its own contiguous container
  From one polymorphic vector to many concrete ones

```
struct Emitter
{




};
```

```
struct Particle
{




};
```

```
struct Rocket
{




};

enum class ParticleType
{
    Smoke,
    Fire
};
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};
```

```cpp
struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};


enum class ParticleType
{
    Smoke,
    Fire
};
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};


struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};


struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};
```

```cpp
struct World
{




};
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};
```

```cpp
struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};
```

```cpp
struct World
{
    std::vector<Particle>                particles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> emitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};
```

```cpp
struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};
```

```cpp
struct World
{
    std::vector<Particle>                particles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> emitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```cpp
void World::update(float dt)
{
    for (Particle& p : particles)
    {



    }



}
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};
```

```cpp
struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};
```

```cpp
struct World
{
    std::vector<Particle>               particles;
    std::vector<Rocket>                 rockets;
    std::vector<std::optional<Emitter>> emitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```cpp
void World::update(float dt)
{
    for (Particle& p : particles)
    {
        p.position += p.velocity * dt;
        p.velocity += p.acceleration * dt;
        p.scale += p.scaleRate * dt;
        p.opacity += p.opacityChange * dt;
        p.rotation += p.angularVelocity * dt;
    }

    for (std::optional<Emitter>& e : emitters)
    {









    }

}
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};
```

```cpp
struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};
```

```cpp
struct World
{
    std::vector<Particle>                particles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> emitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```cpp
void World::update(float dt)
{
    for (Particle& p : particles)
    {
        p.position += p.velocity * dt;
        p.velocity += p.acceleration * dt;
        p.scale += p.scaleRate * dt;
        p.opacity += p.opacityChange * dt;
        p.rotation += p.angularVelocity * dt;
    }

    for (std::optional<Emitter>& e : emitters)
    {
        if (!e.has_value())
            continue;

        e->position += e->velocity * dt;
        e->velocity += e->acceleration * dt;
        e->spawnTimer += e->spawnRate * dt;

        for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
            if (e->type == ParticleType::Smoke)
                particles.push_back({ ... });
            else if (e->type == ParticleType::Fire)
                particles.push_back({ ... });
    }

    for (Rocket& r : rockets)
    {

    }

}
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};

struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};
```

```cpp
void World::update(float dt)
{
    for (Particle& p : particles)
    {
        p.position += p.velocity * dt;
        p.velocity += p.acceleration * dt;
        p.scale += p.scaleRate * dt;
        p.opacity += p.opacityChange * dt;
        p.rotation += p.angularVelocity * dt;
    }

    for (std::optional<Emitter>& e : emitters)
    {
        if (!e.has_value())
            continue;

        e->position += e->velocity * dt;
        e->velocity += e->acceleration * dt;
        e->spawnTimer += e->spawnRate * dt;

        for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
            if (e->type == ParticleType::Smoke)
                particles.push_back({ ... });
            else if (e->type == ParticleType::Fire)
                particles.push_back({ ... });
    }

    for (Rocket& r : rockets)
    {
        r.position += r.velocity * dt;
        r.velocity += r.acceleration * dt;

        if (std::optional<Emitter>& se = emitters[r.smokeEmitterIdx])
            se->position = r.position - sf::Vector2f{12.f, 0.f};

        if (std::optional<Emitter>& fe = emitters[r.fireEmitterIdx])
            fe->position = r.position - sf::Vector2f{12.f, 0.f};
    }
}
```

```cpp
struct World
{
    std::vector<Particle>               particles;
    std::vector<Rocket>                 rockets;
    std::vector<std::optional<Emitter>> emitters;
```

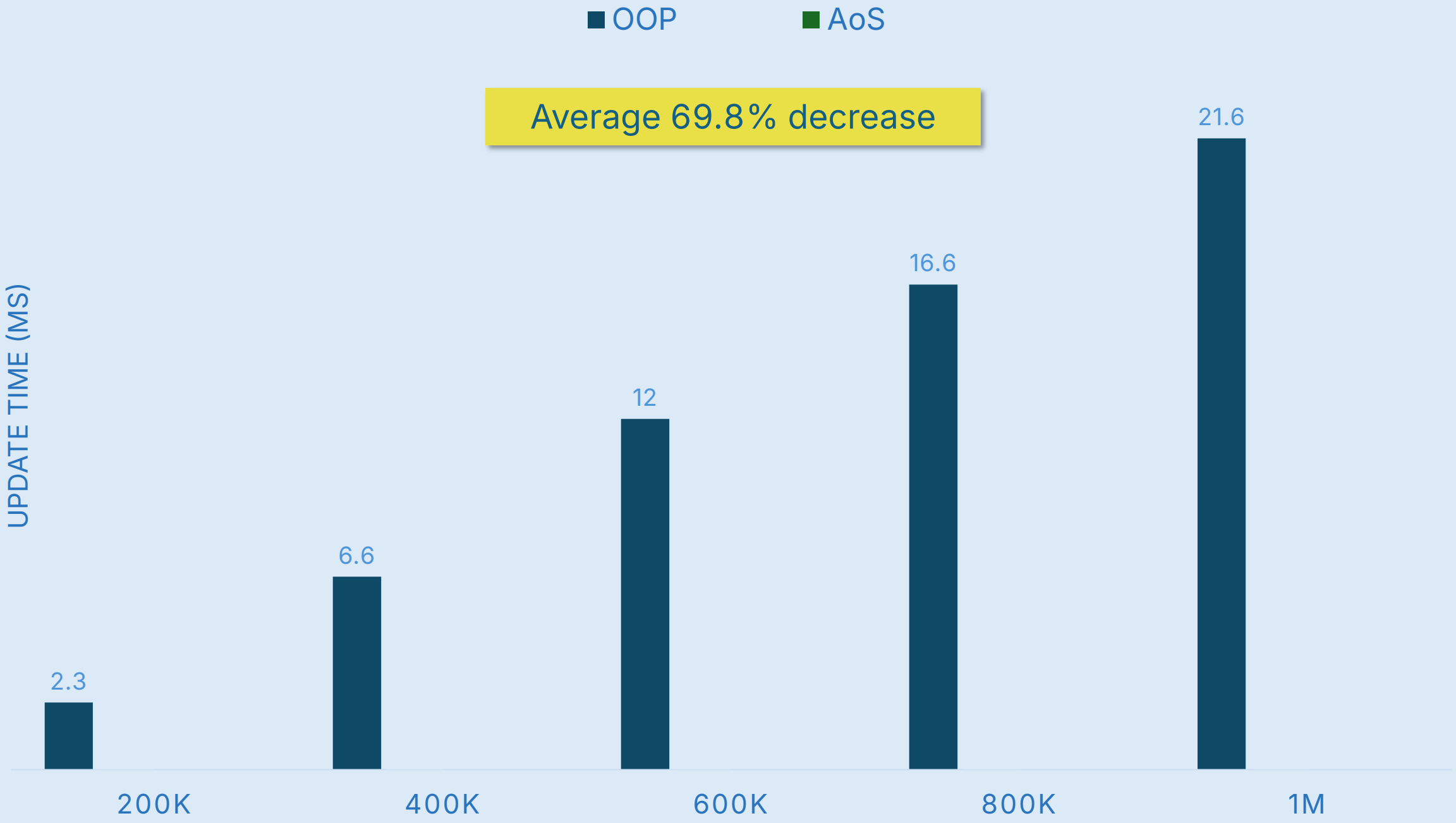
```cpp
std::size_t World::addEmitter(const Emitter& e)
{

}
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};

struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};


void World::update(float dt)
{
    for (Particle& p : particles)
    {
        p.position += p.velocity * dt;
        p.velocity += p.acceleration * dt;
        p.scale += p.scaleRate * dt;
        p.opacity += p.opacityChange * dt;
        p.rotation += p.angularVelocity * dt;
    }


    for (std::optional<Emitter>& e : emitters)
    {
        if (!e.has_value())
            continue;

        e->position += e->velocity * dt;
        e->velocity += e->acceleration * dt;
        e->spawnTimer += e->spawnRate * dt;

        for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
            if (e->type == ParticleType::Smoke)
                particles.push_back({ ... });
            else if (e->type == ParticleType::Fire)
                particles.push_back({ ... });
    }


    for (Rocket& r : rockets)
    {
        r.position += r.velocity * dt;
        r.velocity += r.acceleration * dt;

        if (std::optional<Emitter>& se = emitters[r.smokeEmitterIdx])
            se->position = r.position - sf::Vector2f{12.f, 0.f};

        if (std::optional<Emitter>& fe = emitters[r.fireEmitterIdx])
            fe->position = r.position - sf::Vector2f{12.f, 0.f};
    }
}

struct World
{
    std::vector<Particle>                particles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> emitters;
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};


std::size_t World::addEmitter(const Emitter& e)
{
    for (std::size_t i = 0u; i < emitters.size(); ++i)
        if (!emitters[i].has_value())
        {
            emitters[i].emplace(e);
            return i;
        }

    emitters.emplace_back(e);
    return emitters.size() - 1;
}
```

```cpp
void World::cleanup()
{



}
```

Time for the **2nd round**!


OOP
AoS
Average 69.8% decrease
UPDATE TIME (MS)
2.3
6.6
12
16.6
21.6
200K
400K
600K
800K
1M

It’s **not** only about performance

New requirement: Keep track of the number of rockets

OOP approach

```
for (const std::unique_ptr<Entity>& e : entities)
    if (dynamic_cast<Rocket*>(e.get()))
        ++numRockets;
```

Too slow!

```
struct Entity
{
    // ...
    virtual EntityType getType() const = 0;
};

for (const std::unique_ptr<Entity>& e : entities)
    if (e->getType() == EntityType::Rocket)
        ++numRockets;
```

Defeats the purpose of OOP!

```
struct Rocket : Entity
{
    // ...
    Rocket()  { ++world.numRockets; }
    ~Rocket() { --world.numRockets; }
};
```

More statefulness!
SRP violation!

```
struct World
{
    // ...
    void addRocket() { ++numRockets; }
    void cleanup()   { /* ... */ }
};
```

Specialized function for rockets!
More complexity due to bookkeeping!

New requirement: Keep track of the number of rockets

DOD approach

```
const std::size_t numRockets = world.rockets.size();
```

...that's it!

It’s also about **simplicity**

Code review: what is the cognitive overhead?

OOP approach

```
struct Entity
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    World& world;
    bool alive = true;

    virtual ~Entity() = default;

    virtual void update(float dt)
    {
        position += velocity * dt;
        velocity += acceleration * dt;
    }

    virtual void draw(sf::RenderTarget&);
};
```

Extra state + back-reference!

Tight coupling with rendering system!

```
struct SmokeEmitter : Emitter
{
    void spawnParticle() override
    {
        auto p = std::make_unique<SmokeParticle>();
        // ... set particle data ...
        world->entities.push_back(std::move(p));
    }
};
```

Affects the outside world!

```
struct World
{
    std::vector<std::unique_ptr<Entity>> entities;

    void update(float dt)
    {
        for (auto& entity : entities)
            entity->update(dt);
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const auto& entity : entities)
            entity->draw(rt);
    }

    void cleanup()
    {
        std::erase_if(entities, [](const auto& entity) { return !entity->alive; });
    }
};
```

Could do anything!

Code review: what is the cognitive overhead?

DOD approach

```
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};

struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};

struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};
```

It's just data!
Decoupled from logic.
WYSIWYG!

```
struct World
{
    std::vector<std::optional<Emitter>> emitters;
    std::vector<Particle>               particles;
    std::vector<Rocket>                 rockets;

    void update(const float dt)
    {
        for (Particle& p : particles) { /* ... */ }
        for (std::optional<Emitter>& e : emitters) { /* ... */ }
        for (Rocket& r : rockets) { /* ... */ }
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const Particle& p : particles) { /* ... */ }
        for (const Rocket& r : rockets) { /* ... */ }
    }
};
```

Data is not hidden.
Stage-based approach.
All logic is here!
Relationships are visible.
Loose coupling with rendering system.

And also about **new opportunities**

New requirement: serialization

OOP approach

```
struct Entity
{
    // ...

    virtual void serialize(std::ostream& os) const { /* ... */ }
    virtual void deserialize(std::istream& is) const { /* ... */ }
};
```

Too tight coupling!
Locked into a specific format.
Dependency on iostreams.

```
struct Visitor
{
    virtual void visit(Particle&) = 0;
    virtual void visit(Emitter&) = 0;
    virtual void visit(Rocket&) = 0;
};
```

Must list all types!

```
struct Entity
{
    virtual void accept(Visitor& visitor) = 0;
};
```

Double dispatch overhead!

```
struct SerializationVisitor : Visitor
{
    std::ostream& os;
    virtual void visit(Particle&) override;
    virtual void visit(Emitter&) override;
    virtual void visit(Rocket&) override;
};
```

Additional statefulness.

Must keep in sync.

```
struct Rocket : Entity
{
    SmokeEmitter* smokeEmitter = nullptr;
    FireEmitter*  fireEmitter  = nullptr;
};
```

Uh-oh...

Does it have to be so complicated?

New requirement: serialization

DOD approach

```
void serialize(const World& w, std::ostream& os)
{



}
```

New requirement: serialization

DOD approach

```
void serialize(const World& w, std::ostream& os)
{
    for (const auto& r : w.rockets)   { /* ... */ }
    for (const auto& p : w.particles) { /* ... */ }
    for (const auto& e : w.emitters)  { /* ... */ }
}
```

A single function!

New requirement: serialization

Opportunities

The entire state of the program is just data
- No reliance on pointers or addresses

Saving/loading the state is trivial to implement
- Better testability, debuggability, and tooling

Networking also becomes easier
- Can send snapshots of the entire state
- State updates can be sent as deltas

What about **multithreading**?

New requirement: multithreaded particle update

OOP approach

```
struct World
{
    std::vector<std::unique_ptr<Entity>> entities;

    void update(float dt)
    {
        for (auto& entity : entities)
            entity->update(dt);
    }

    void draw(sf::RenderTarget& rt)
    {
        for (const auto& entity : entities)
            entity->draw(rt);
    }

    void cleanup()
    {
        std::erase_if(entities, [](const auto& entity) { return !entity->alive; });
    }
};

struct SmokeEmitter : Emitter
{
    void spawnParticle() override
    {
        auto p = std::make_unique<SmokeParticle>();
        // ... set particle data ...
        world->entities.push_back(std::move(p));
    }
};
```

Could do anything!

Creates new entities during update!

Not trivially parallelizable!

```
struct Rocket : Entity
{
    SmokeEmitter* smokeEmitter = nullptr;
    FireEmitter*  fireEmitter  = nullptr;

    void update(float dt) override
    {
        Entity::update(dt);

        smokeEmitter->position = position - {12.f, 0.f};
        fireEmitter->position  = position - {12.f, 0.f};

        if (position.x > 1080.f + 64.f)
        {
            alive = false;
            smokeEmitter->alive = false;
            fireEmitter->alive  = false;
        }
    }
};
```

Write to linked entities!

Write to linked entities!

New requirement: multithreaded particle update

DOD approach

```
void World::update(float dt)
{
    for (Particle& p : particles)
    {
        p.position += p.velocity * dt;
        p.velocity += p.acceleration * dt;
        p.scale += p.scaleRate * dt;
        p.opacity += p.opacityChange * dt;
        p.rotation += p.angularVelocity * dt;
    }
```

Trivially parallelizable!

Data-oriented architectures provide many **side benefits**!

Can we go even **further**?

Second **optimization** pass

- Avoid branching in hot loops
  - Group the data beforehand
- Reduce the size of common types
  - Try to fit more data in cache
- No other major changes
  - Stick to the Array-of-Structures (AoS) layout for now

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;

    ParticleType type;
};
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;

    ParticleType type;
};
```

```cpp
struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};


enum class ParticleType
{
    Smoke,
    Fire
};
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;
};
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;
};
```

```cpp
struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::size_t smokeEmitterIdx;
    std::size_t fireEmitterIdx;
};
```

```cpp
struct Emitter
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float spawnTimer;
    float spawnRate;
};
```

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;
};
```

```cpp
struct Rocket
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    std::uint16_t smokeEmitterIdx;
    std::uint16_t fireEmitterIdx;
};
```

```
struct World
{
    std::vector<Particle>                particles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> emitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```
struct World
{
    std::vector<Particle>                smokeParticles;
    std::vector<Particle>                fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> emitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```
struct World
{
    std::vector<Particle>                smokeParticles;
    std::vector<Particle>                fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(auto& targetVec, const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```
struct World
{
    std::vector<Particle>                smokeParticles;
    std::vector<Particle>                fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(auto& targetVec, const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```cpp
struct World
{
    std::vector<Particle>                smokeParticles;
    std::vector<Particle>                fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(auto& targetVec, const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```cpp
void World::update(float dt)
{
    auto updateParticle = [&](Particle& p)
    {
        p.position += p.velocity * dt;
        p.velocity += p.acceleration * dt;
        p.scale += p.scaleRate * dt;
        p.opacity += p.opacityChange * dt;
        p.rotation += p.angularVelocity * dt;
    };

    for (Particle& p : smokeParticles) updateParticle(p);
    for (Particle& p : fireParticles)  updateParticle(p);









}
```

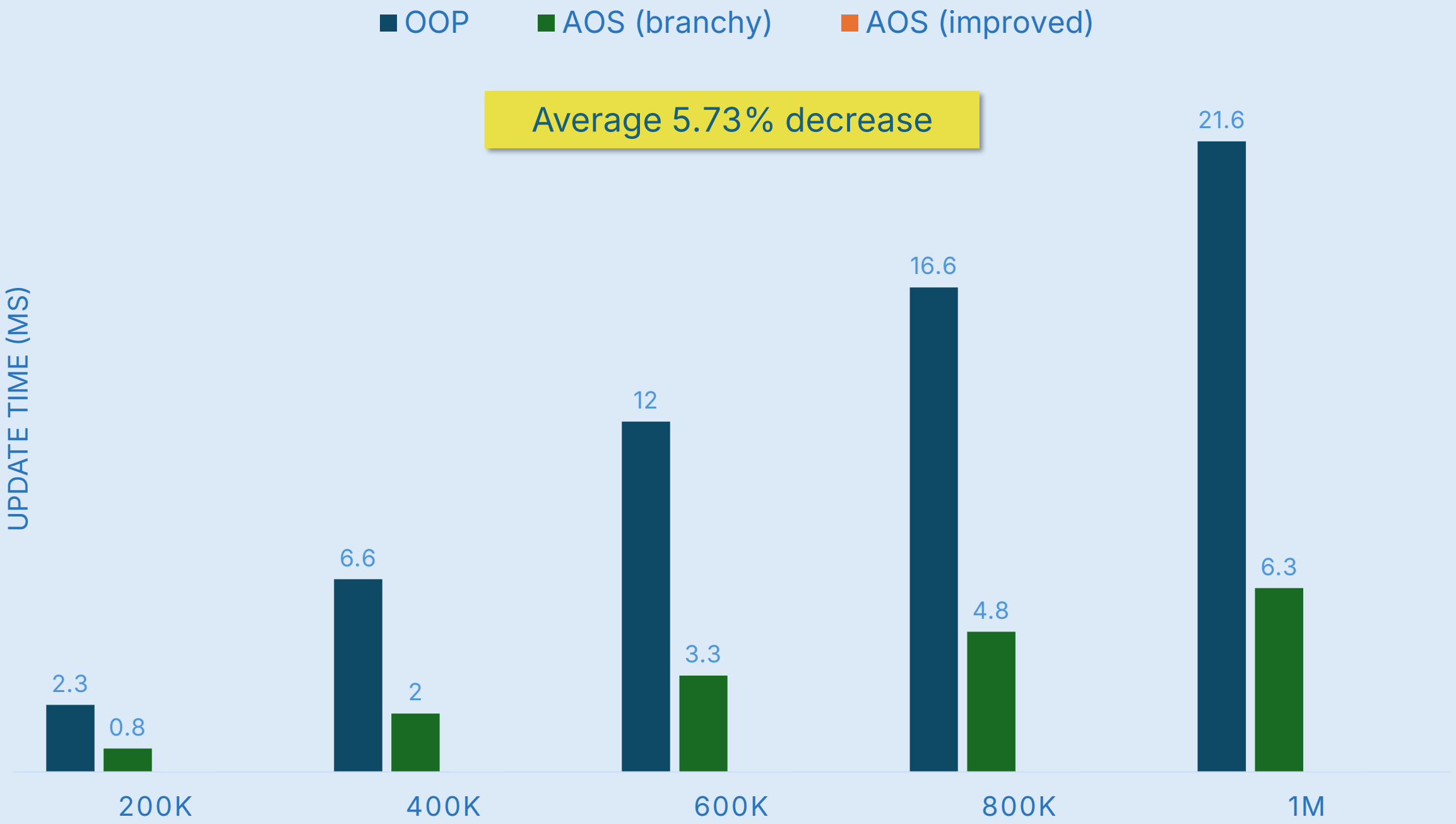
```
struct World
{
    std::vector<Particle>               smokeParticles;
    std::vector<Particle>               fireParticles;
    std::vector<Rocket>                 rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(auto& targetVec, const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```
void World::update(float dt)
{
        auto updateParticle = [&](Particle& p)
        {
            p.position += p.velocity * dt;
            p.velocity += p.acceleration * dt;
            p.scale += p.scaleRate * dt;
            p.opacity += p.opacityChange * dt;
            p.rotation += p.angularVelocity * dt;
        };

        for (Particle& p : smokeParticles) updateParticle(p);
        for (Particle& p : fireParticles)  updateParticle(p);



    auto updateEmitter = [&](std::optional<Emitter>& e, auto&& fSpawn)
    {
        if (!e.hasValue())
            return;

        e->position += e->velocity * dt;
        e->velocity += e->acceleration * dt;
        e->spawnTimer += e->spawnRate * dt;

        for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
            fSpawn();
    };









}
```

```cpp
struct World
{
    std::vector<Particle>                smokeParticles;
    std::vector<Particle>                fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    void addRocket(const Rocket& r);
    std::size_t addEmitter(auto& targetVec, const Emitter& e);

    void update(float dt);
    void draw(sf::RenderTarget& rt);
    void cleanup();
};
```

```cpp
void World::update(float dt)
{
        auto updateParticle = [&](Particle& p)
        {
            p.position += p.velocity * dt;
            p.velocity += p.acceleration * dt;
            p.scale += p.scaleRate * dt;
            p.opacity += p.opacityChange * dt;
            p.rotation += p.angularVelocity * dt;
        };

        for (Particle& p : smokeParticles) updateParticle(p);
        for (Particle& p : fireParticles)  updateParticle(p);



        auto updateEmitter = [&](std::optional<Emitter>& e, auto&& fSpawn)
        {
            if (!e.hasValue())
                return;

            e->position += e->velocity * dt;
            e->velocity += e->acceleration * dt;
            e->spawnTimer += e->spawnRate * dt;

            for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
                fSpawn();
        };


         for (std::optional<Emitter>& e : smokeEmitters)
             updateEmitter(e, [&]{ smokeParticles.push_back({/* ... */}); });

         for (std::optional<Emitter>& e : fireEmitters)
             updateEmitter(e, [&]{ fireParticles.push_back({/* ... */}); });



}
```

Time for the **3rd round**!


OOP
AOS (branchy)
AOS (improved)
Average 5.73% decrease
UPDATE TIME (MS)
2.3
0.8
6.6
2
12
3.3
16.6
4.8
21.6
6.3
200K
400K
600K
800K
1M

We are still far from our **full potential**

Third **optimization** pass

- Completely change data layout for particles
  - Migrate to Structure-of-Arrays (SoA)

```
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;
};
```

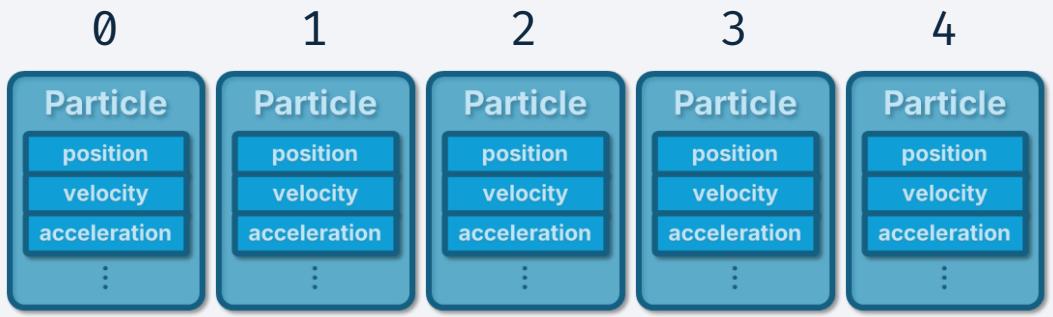
```
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;
};
```

```
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;
};
```

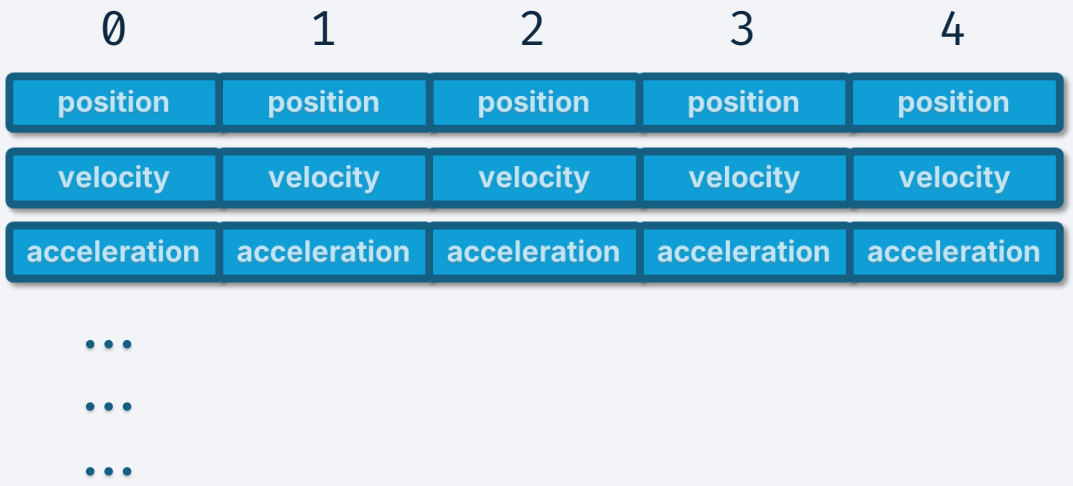
```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;
};
```

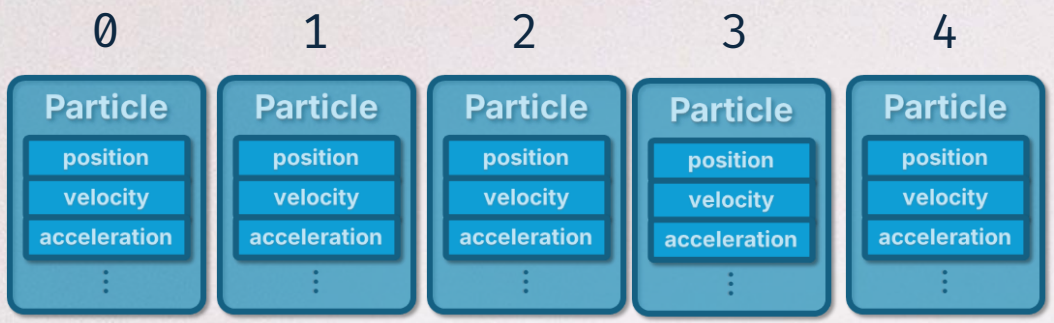
```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;
};
```

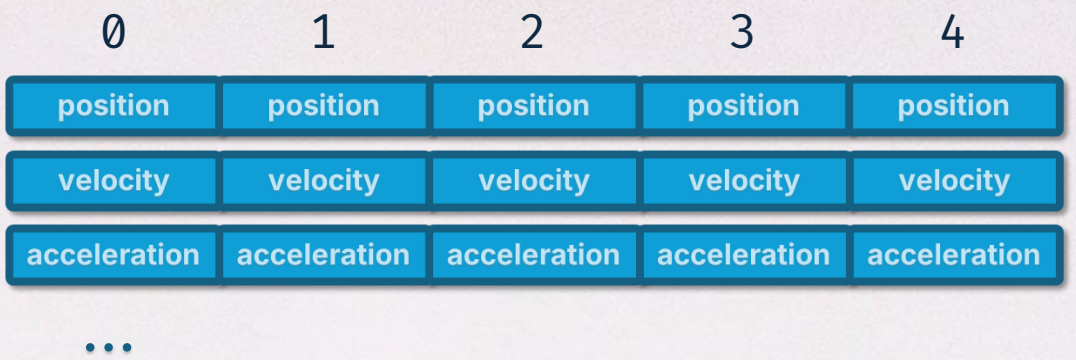
```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;
};
```




```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;
};
```







Array-of-Structures (AOS)

Explicit Entities

Each object in the array is a complete entity

Internal padding

Compiler might add padding between fields

Whole-object loading

Entire object is loaded to access any field

Resists vectorization

Field data is scattered, making SIMD hard

Structure-of-Arrays (SOA)

Implicit Entities

Entity: conceptual slice across the arrays

No internal padding

Every byte is useful, no cache space waste

Surgical field loading

Only needed fields are loaded in cache

Ideal for vectorization

Field data is contiguous, making SIMD trivial

```
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;
};
```

```cpp
struct ParticleSoA
{
        std::vector<sf::Vector2f> positions;
        std::vector<sf::Vector2f> velocities;
        std::vector<sf::Vector2f> accelerations;

        std::vector<float> scales;
        std::vector<float> opacities;
        std::vector<float> rotations;

        std::vector<float> scaleRates;
        std::vector<float> opacityChanges;
        std::vector<float> angularVelocities;




};
```

```
struct ParticleSoA
{
        std::vector<sf::Vector2f> positions;
        std::vector<sf::Vector2f> velocities;
        std::vector<sf::Vector2f> accelerations;

        std::vector<float> scales;
        std::vector<float> opacities;
        std::vector<float> rotations;

        std::vector<float> scaleRates;
        std::vector<float> opacityChanges;
        std::vector<float> angularVelocities;


    void forAllVectors(auto&& f)
    {









    }
};
```

```
struct ParticleSoA
{
        std::vector<sf::Vector2f> positions;
        std::vector<sf::Vector2f> velocities;
        std::vector<sf::Vector2f> accelerations;

        std::vector<float> scales;
        std::vector<float> opacities;
        std::vector<float> rotations;

        std::vector<float> scaleRates;
        std::vector<float> opacityChanges;
        std::vector<float> angularVelocities;


    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};
```

```
struct ParticleSoA
{
        std::vector<sf::Vector2f> positions;
        std::vector<sf::Vector2f> velocities;
        std::vector<sf::Vector2f> accelerations;

        std::vector<float> scales;
        std::vector<float> opacities;
        std::vector<float> rotations;

        std::vector<float> scaleRates;
        std::vector<float> opacityChanges;
        std::vector<float> angularVelocities;


    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};
```

```
struct ParticleSoA
{
        std::vector<sf::Vector2f> positions;
        std::vector<sf::Vector2f> velocities;
        std::vector<sf::Vector2f> accelerations;

        std::vector<float> scales;
        std::vector<float> opacities;
        std::vector<float> rotations;

        std::vector<float> scaleRates;
        std::vector<float> opacityChanges;
        std::vector<float> angularVelocities;

};
```

```
void forAllVectors(auto&& f)
{
    f(positions);
    f(velocities);
    f(accelerations);

    f(scales);
    f(opacities);
    f(rotations);

    f(scaleRates);
    f(opacityChanges);
    f(angularVelocities);
}
```

```
struct ParticleSoA
{
        std::vector<sf::Vector2f> positions;
        std::vector<sf::Vector2f> velocities;
        std::vector<sf::Vector2f> accelerations;

        std::vector<float> scales;
        std::vector<float> opacities;
        std::vector<float> rotations;

        std::vector<float> scaleRates;
        std::vector<float> opacityChanges;
        std::vector<float> angularVelocities;

};
```

```
void forAllVectors(auto&& f)
{
    f(positions);
    f(velocities);
    f(accelerations);

    f(scales);
    f(opacities);
    f(rotations);

    f(scaleRates);
    f(opacityChanges);
    f(angularVelocities);
}
```

```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;

    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};
```

```cpp
struct World
{
    std::vector<Particle>                smokeParticles;
    std::vector<Particle>                fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    // ...
};
```

```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;

    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};
```

```cpp
struct World
{
    ParticleSoA                          smokeParticles;
    ParticleSoA                          fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>>  smokeEmitters;
    std::vector<std::optional<Emitter>>  fireEmitters;

    // ...
};
```

```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;

    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};
```

```cpp
struct World
{
    ParticleSoA                          smokeParticles;
    ParticleSoA                          fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>>  smokeEmitters;
    std::vector<std::optional<Emitter>>  fireEmitters;

    // ...
};
```

```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;

    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};

struct World
{
    ParticleSoA                          smokeParticles;
    ParticleSoA                          fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>>> smokeEmitters;
    std::vector<std::optional<Emitter>>> fireEmitters;

    // ...
};
```

```cpp
void World::update(float dt)
{
    auto updateParticles = [&](ParticleSoA& soa)
    {

    };

    updateParticles(smokeParticles);
    updateParticles(fireParticles);
```

```cpp
struct ParticleSoA
{
        std::vector<sf::Vector2f> positions;
        std::vector<sf::Vector2f> velocities;
        std::vector<sf::Vector2f> accelerations;

        std::vector<float> scales;
        std::vector<float> opacities;
        std::vector<float> rotations;

        std::vector<float> scaleRates;
        std::vector<float> opacityChanges;
        std::vector<float> angularVelocities;

    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};

struct World
{
    ParticleSoA                          smokeParticles;
    ParticleSoA                          fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    // ...
};
```

```cpp
void World::update(float dt)
{
    auto updateParticles = [&](ParticleSoA& soa)
    {
        const auto nParticles = soa.positions.size();

        for (std::size_t i = 0u; i < nParticles; ++i)
        {
            soa.velocities[i] += soa.accelerations[i] * dt;
            soa.positions[i] += soa.velocities[i] * dt;
            soa.scales[i] += soa.scaleRates[i] * dt;
            soa.opacities[i] += soa.opacityChanges[i] * dt;
            soa.rotations[i] += soa.angularVelocities[i] * dt;
        }
    };

    updateParticles(smokeParticles);
    updateParticles(fireParticles);
```

```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;

    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};

struct World
{
    ParticleSoA                          smokeParticles;
    ParticleSoA                          fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    // ...
};
```

```cpp
void World::update(float dt)
{
    auto updateParticles = [&](ParticleSoA& soa)
    {
        const auto nParticles = soa.positions.size();

        for (std::size_t i = 0u; i < nParticles; ++i)
        {
            soa.velocities[i] += soa.accelerations[i] * dt;
            soa.positions[i] += soa.velocities[i] * dt;
            soa.scales[i] += soa.scaleRates[i] * dt;
            soa.opacities[i] += soa.opacityChanges[i] * dt;
            soa.rotations[i] += soa.angularVelocities[i] * dt;
        }
    };

    updateParticles(smokeParticles);
    updateParticles(fireParticles);

    for (std::optional<Emitter>& e : smokeEmitters)
    {

    }
```

```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;

    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};

struct World
{
    ParticleSoA                            smokeParticles;
    ParticleSoA                            fireParticles;
    std::vector<Rocket>                    rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    // ...
};
```

```cpp
void World::update(float dt)
{
    auto updateParticles = [&](ParticleSoA& soa)
    {
        const auto nParticles = soa.positions.size();

        for (std::size_t i = 0u; i < nParticles; ++i)
        {
            soa.velocities[i] += soa.accelerations[i] * dt;
            soa.positions[i] += soa.velocities[i] * dt;
            soa.scales[i] += soa.scaleRates[i] * dt;
            soa.opacities[i] += soa.opacityChanges[i] * dt;
            soa.rotations[i] += soa.angularVelocities[i] * dt;
        }
    };

    updateParticles(smokeParticles);
    updateParticles(fireParticles);

    for (std::optional<Emitter>& e : smokeEmitters)
    {
        if (!e.has_value())
            continue;

        e->position += e->velocity * dt;
        e->velocity += e->acceleration * dt;
        e->spawnTimer += e->spawnRate * dt;

        for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
        {
            smokeParticles.positions.push_back(...);
            smokeParticles.velocities.push_back(...);
            // ... all other vectors ...
            smokeParticles.angularVelocities.push_back(...);
        }
    }
```

```
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;

    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};

struct World
{
    ParticleSoA                          smokeParticles;
    ParticleSoA                          fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>>  smokeEmitters;
    std::vector<std::optional<Emitter>>  fireEmitters;

    // ...
};
```

```
void World::update(float dt)
{
    auto updateParticles = [&](ParticleSoA& soa)
    {
        const auto nParticles = soa.positions.size();

        for (std::size_t i = 0u; i < nParticles; ++i)
        {
            soa.velocities[i] += soa.accelerations[i] * dt;
            soa.positions[i] += soa.velocities[i] * dt;
            soa.scales[i] += soa.scaleRates[i] * dt;
            soa.opacities[i] += soa.opacityChanges[i] * dt;
            soa.rotations[i] += soa.angularVelocities[i] * dt;
        }
    };

    updateParticles(smokeParticles);
    updateParticles(fireParticles);

    for (std::optional<Emitter>& e : smokeEmitters)
    {
        if (!e.has_value())
            continue;

        e->position += e->velocity * dt;
        e->velocity += e->acceleration * dt;
        e->spawnTimer += e->spawnRate * dt;

        for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
        {
            smokeParticles.positions.push_back(...);
            smokeParticles.velocities.push_back(...);
            // ... all other vectors ...
            smokeParticles.angularVelocities.push_back(...);
        }
    }

}
```

• • •

```
struct ParticleSoA
{
        std::vector<sf::Vector2f> positions;
        std::vector<sf::Vector2f> velocities;
        std::vector<sf::Vector2f> accelerations;

        std::vector<float> scales;
        std::vector<float> opacities;
        std::vector<float> rotations;

        std::vector<float> scaleRates;
        std::vector<float> opacityChanges;
        std::vector<float> angularVelocities;

    void forAllVectors(auto&& f)
    {
        f(positions);
        f(velocities);
        f(accelerations);

        f(scales);
        f(opacities);
        f(rotations);

        f(scaleRates);
        f(opacityChanges);
        f(angularVelocities);
    }
};

struct World
{
    ParticleSoA                          smokeParticles;
    ParticleSoA                          fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    // ...
};
```

```
void World::update(float dt)
{
        auto updateParticles = [&](ParticleSoA& soa)
        {
            const auto nParticles = soa.positions.size();

            for (std::size_t i = 0u; i < nParticles; ++i)
            {
                soa.velocities[i] += soa.accelerations[i] * dt;
                soa.positions[i] += soa.velocities[i] * dt;
                soa.scales[i] += soa.scaleRates[i] * dt;
                soa.opacities[i] += soa.opacityChanges[i] * dt;
                soa.rotations[i] += soa.angularVelocities[i] * dt;
            }
        };

        updateParticles(smokeParticles);
        updateParticles(fireParticles);


        for (std::optional<Emitter>& e : smokeEmitters)
        {
            if (!e.has_value())
                continue;

            e->position += e->velocity * dt;
            e->velocity += e->acceleration * dt;
            e->spawnTimer += e->spawnRate * dt;

            for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
            {
                smokeParticles.positions.push_back(...);
                smokeParticles.velocities.push_back(...);
                // ... all other vectors ...
                smokeParticles.angularVelocities.push_back(...);
            }
        }

}
```

• • •

```
struct ParticleSoA
{
        std::vector<sf::Vector2f> positions;
        std::vector<sf::Vector2f> velocities;
        std::vector<sf::Vector2f> accelerations;

        std::vector<float> scales;
        std::vector<float> opacities;
        std::vector<float> rotations;

        std::vector<float> scaleRates;
        std::vector<float> opacityChanges;
        std::vector<float> angularVelocities;
};
```

```
void forAllVectors(auto&& f)
{
    f(positions);
    f(velocities);
    f(accelerations);

    f(scales);
    f(opacities);
    f(rotations);

    f(scaleRates);
    f(opacityChanges);
    f(angularVelocities);
}
```

```
struct World
{
    ParticleSoA                            smokeParticles;
    ParticleSoA                            fireParticles;
    std::vector<Rocket>                    rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    // ...
};
```

```
void World::update(float dt)
{
    auto updateParticles = [&](ParticleSoA& soa)
    {
        const auto nParticles = soa.positions.size();

        for (std::size_t i = 0u; i < nParticles; ++i)
        {
            soa.velocities[i] += soa.accelerations[i] * dt;
            soa.positions[i] += soa.velocities[i] * dt;
            soa.scales[i] += soa.scaleRates[i] * dt;
            soa.opacities[i] += soa.opacityChanges[i] * dt;
            soa.rotations[i] += soa.angularVelocities[i] * dt;
        }
    };

    updateParticles(smokeParticles);
    updateParticles(fireParticles);

    for (std::optional<Emitter>& e : smokeEmitters)
    {
        if (!e.has_value())
            continue;

        e->position += e->velocity * dt;
        e->velocity += e->acceleration * dt;
        e->spawnTimer += e->spawnRate * dt;

        for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
        {
            smokeParticles.positions.push_back(...);
            smokeParticles.velocities.push_back(...);
            // ... all other vectors ...
            smokeParticles.angularVelocities.push_back(...);
        }
    }

    ...

}
```

```
void World::cleanup()
{




}
```

```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;
};
```

```cpp
void forAllVectors(auto&& f)
{
    f(positions);
    f(velocities);
    f(accelerations);

    f(scales);
    f(opacities);
    f(rotations);

    f(scaleRates);
    f(opacityChanges);
    f(angularVelocities);
}
```

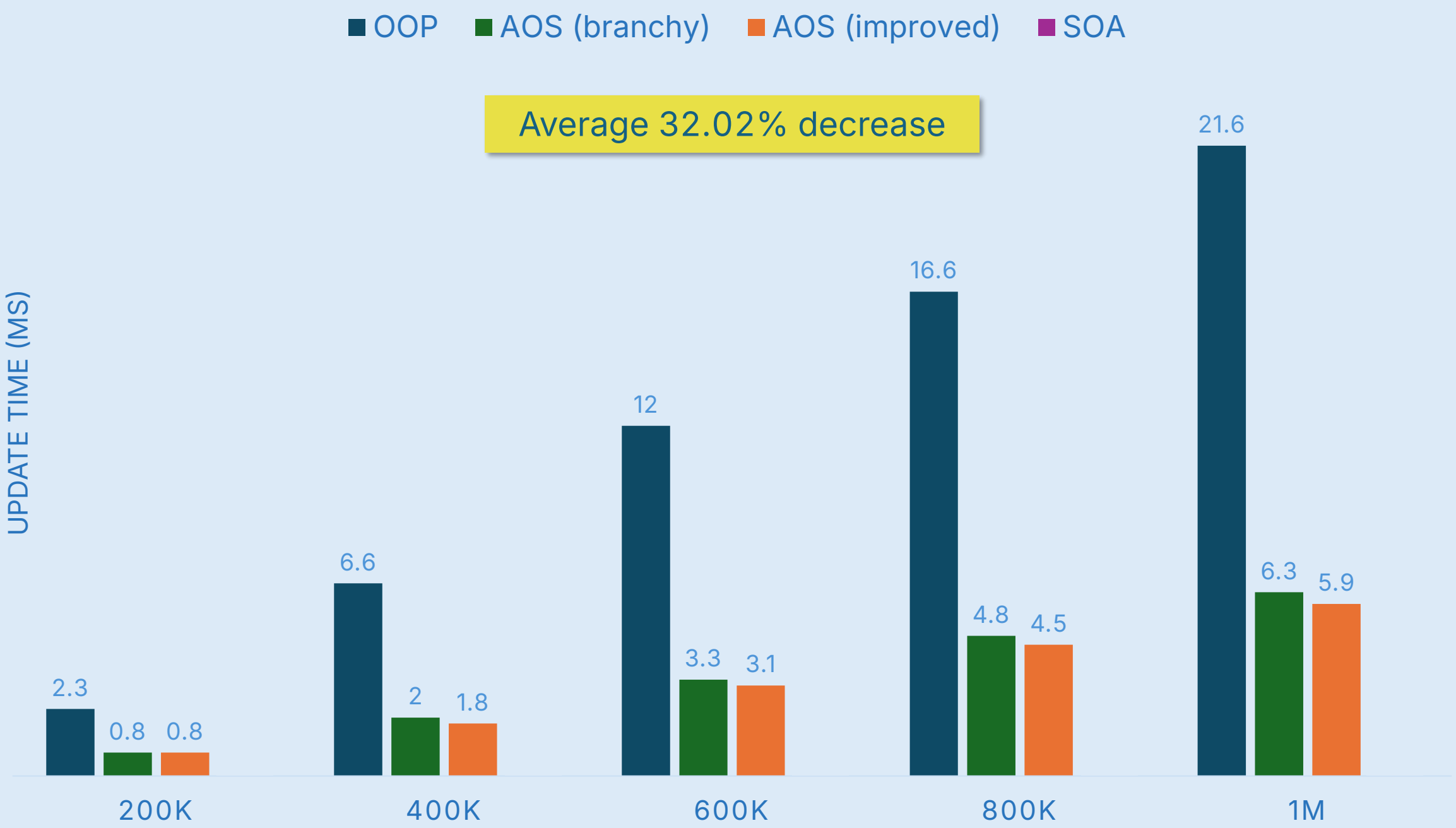
```cpp
struct World
{
    ParticleSoA                          smokeParticles;
    ParticleSoA                          fireParticles;
    std::vector<Rocket>                  rockets;
    std::vector<std::optional<Emitter>> smokeEmitters;
    std::vector<std::optional<Emitter>> fireEmitters;

    // ...
};
```

```cpp
void World::update(float dt)
{
    auto updateParticles = [&](ParticleSoA& soa)
    {
        const auto nParticles = soa.positions.size();

        for (std::size_t i = 0u; i < nParticles; ++i)
        {
            soa.velocities[i] += soa.accelerations[i] * dt;
            soa.positions[i] += soa.velocities[i] * dt;
            soa.scales[i] += soa.scaleRates[i] * dt;
            soa.opacities[i] += soa.opacityChanges[i] * dt;
            soa.rotations[i] += soa.angularVelocities[i] * dt;
        }
    };

    updateParticles(smokeParticles);
    updateParticles(fireParticles);

    for (std::optional<Emitter>& e : smokeEmitters)
    {
        if (!e.has_value())
            continue;

        e->position += e->velocity * dt;
        e->velocity += e->acceleration * dt;
        e->spawnTimer += e->spawnRate * dt;

        for (; e->spawnTimer >= 1.f; e->spawnTimer -= 1.f)
        {
            smokeParticles.positions.push_back(...);
            smokeParticles.velocities.push_back(...);
            // ... all other vectors ...
            smokeParticles.angularVelocities.push_back(...);
        }
    }

    ...

}
```

```cpp
void World::cleanup()
{
    auto hasNegativeOpacity =
        [](const ParticleSoA& soa, std::size_t i)
        {
            return soa.opacities[i] <= 0.f;
        };

    soaEraseIf(smokeParticles, hasNegativeOpacity);
    soaEraseIf(fireParticles, hasNegativeOpacity);

    // ...
}
```

```cpp
void soaEraseIf(ParticleSoA& soa, auto&& predicate)
{

}
```

Time for the **last round**!


OOP
AOS (branchy)
AOS (improved)
SOA
Average 32.02% decrease
UPDATE TIME (MS)
2.3
0.8
0.8
6.6
2
1.8
12
3.3
3.1
16.6
4.8
4.5
21.6
6.3
5.9
200K
400K
600K
800K
1M

Why is SoA much faster?

Array-of-Structures (AoS)

```
# ...
vpsllq  ymm5, ymm3, 4
vpsllq  ymm6, ymm3, 5
vpaddq  ymm5, ymm6, ymm5
vgatherqps xmm8 {k1}, dword ptr [rdi + ymm6 + 8]
vgatherqps xmm9 {k2}, dword ptr [rdi + ymm5 + 8]
vinsertf128 ymm8, ymm9, xmm8, 1
vfmadd231ps ymm7, ymm8, ymm1
[illegible] [rdi + ymm5] {k1}, xmm7
```

Fused Multiply-Add

Gather-Scatter Overhead

Structure-of-Arrays (SoA)

```
# ...
vbroadcastss ymm1, xmm0
vmovups ymm2, ymmword ptr [rdx + 4*rbp]
vmovups ymm3, ymmword ptr [rdi + 4*rbp]
vfmadd231ps ymm3, ymm2, ymm1
[illegible] [rdi + 4*rbp], ymm3
vmovups ymm3, ymmword ptr [rcx + 4*rbp]
vmovups ymm4, ymmword ptr [rsi + 4*rbp]
vfmadd231ps ymm4, ymm3, ymm1
```

Fused Multiply-Add

Other possible factors

- Straddled particle data on cache line boundaries
- Prefetcher has an harder time with strided access
- Register/pressure spills due to addresses in AoS

Was simplicity **sacrificed**?

- C++ has no native support for SoA
  - Language & Standard Library are built around AoS
- Functional patterns help
  - Local lambdas and higher order functions reduce SoA repetition
- Can we do better?
  - Time to reflect...

```cpp
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;
};
```

```cpp
struct ParticleSoA
{
    std::vector<sf::Vector2f> positions;
    std::vector<sf::Vector2f> velocities;
    std::vector<sf::Vector2f> accelerations;

    std::vector<float> scales;
    std::vector<float> opacities;
    std::vector<float> rotations;

    std::vector<float> scaleRates;
    std::vector<float> opacityChanges;
    std::vector<float> angularVelocities;
};
```

SoA<Particle> particles;

MANUAL TRANSFORMATION...

```
struct Particle
{
    sf::Vector2f position;
    sf::Vector2f velocity;
    sf::Vector2f acceleration;

    float scale;
    float opacity;
    float rotation;

    float scaleRate;
    float opacityChange;
    float angularVelocity;
};

SoA<Particle> particles;
```

```
particles.pushBack({
    .position     = /* ... */,
    .velocity     = /* ... */,
    .acceleration = /* ... */,

    .scale    = /* ... */,
    .opacity  = /* ... */,
    .rotation = /* ... */,

    .scaleRate       = /* ... */,
    .opacityChange   = /* ... */,
    .angularVelocity = /* ... */,
});
```

```
particles.withAll(
    [&](sf::Vector2f&       position,
        sf::Vector2f&       velocity,
        const sf::Vector2f& acceleration,
        float&              scale,
        float&              opacity,
        float&              rotation,
        const float         scaleRate,
        const float         opacityChange,
        const float         angularVelocity)
{
    velocity += acceleration * dt;
    position += velocity * dt;
    scale += scaleRate * dt;
    opacity += opacityChange * dt;
    rotation += angularVelocity * dt;
});
```

ALL OF THIS IS VALID C++17 !

```
particles.with<&Particle::scale, &Particle::scaleChange>(
    [&](float& scale, const float scaleChange)
{
    scale += scaleChange * dt;
});
```

```
particles.with<&Particle::scale, &Particle::scaleChange>(
    [&](float& scale, const float scaleChange)
{
    scale += scaleChange * dt;
});
```

WITH C++26 REFLECTION...

```
particles.with([&](float& scale, const float scaleChange)
{
    scale += scaleChange * dt;
});
```

SoA is **not** always the answer

- DOD is *not* SoA by default
  - DOD = designing architectures with major focus on data layout
- Let access patterns choose the layout
  - Which fields are frequently accessed together?
  - Are there any “cold” fields?
  - What is the SIMD lane size on this platform?
- Use abstractions to experiment easily
  - Switch between different layouts at compile-time
  - C++26 reflection will make this easier!

DOD is **not** always the answer

- OOP principles work well at a coarser granularity
  - E.g. OOP `ParticleManager` API with DOD implementation
  - Swapping strategies via interfaces is straightforward
- Clear boundaries and SRP can boost collaboration
  - E.g. in larger teams and with novice developers
  - Invariants and access control lead to safer code
- Polymorphism enables plugin systems
  - E.g. loading addon libraries at run-time
- Hybrid approach
  - OOP as the *shell*, DOD as the *engine*

Final **takeaways**

- Design for performance from the start
  - If speed is a *requirement*, it must be an *architectural* priority
  - Profiling is still key to find the real bottlenecks, afterwards
- Flatten and shrink your data
  - Prefer `std::vector<T>` to `std::vector<std::unique_ptr<T>>`
  - Minimize the size of structs used in hot loops
  - Steer away from polymorphism and inheritance trees
- Group data by usage
  - Split mixed-type containers into homogeneous ones to avoid branching
  - Make `bool`/`enum` flags *implicit* by moving data into different containers
- Consider SoA for bulk processing
  - Unleash SIMD and minimize memory bandwidth overhead
  - Be mindful of the added complexity/verbosity cost

Data drives design

Target the machine

It’s a spectrum, not dogma

Embrace Modern C++

Pragmatism wins

Think **data-first**

Your code will be ***faster*** and ***simpler***

Thank you! 🙏

https://linktr.ee/vittorioromeo