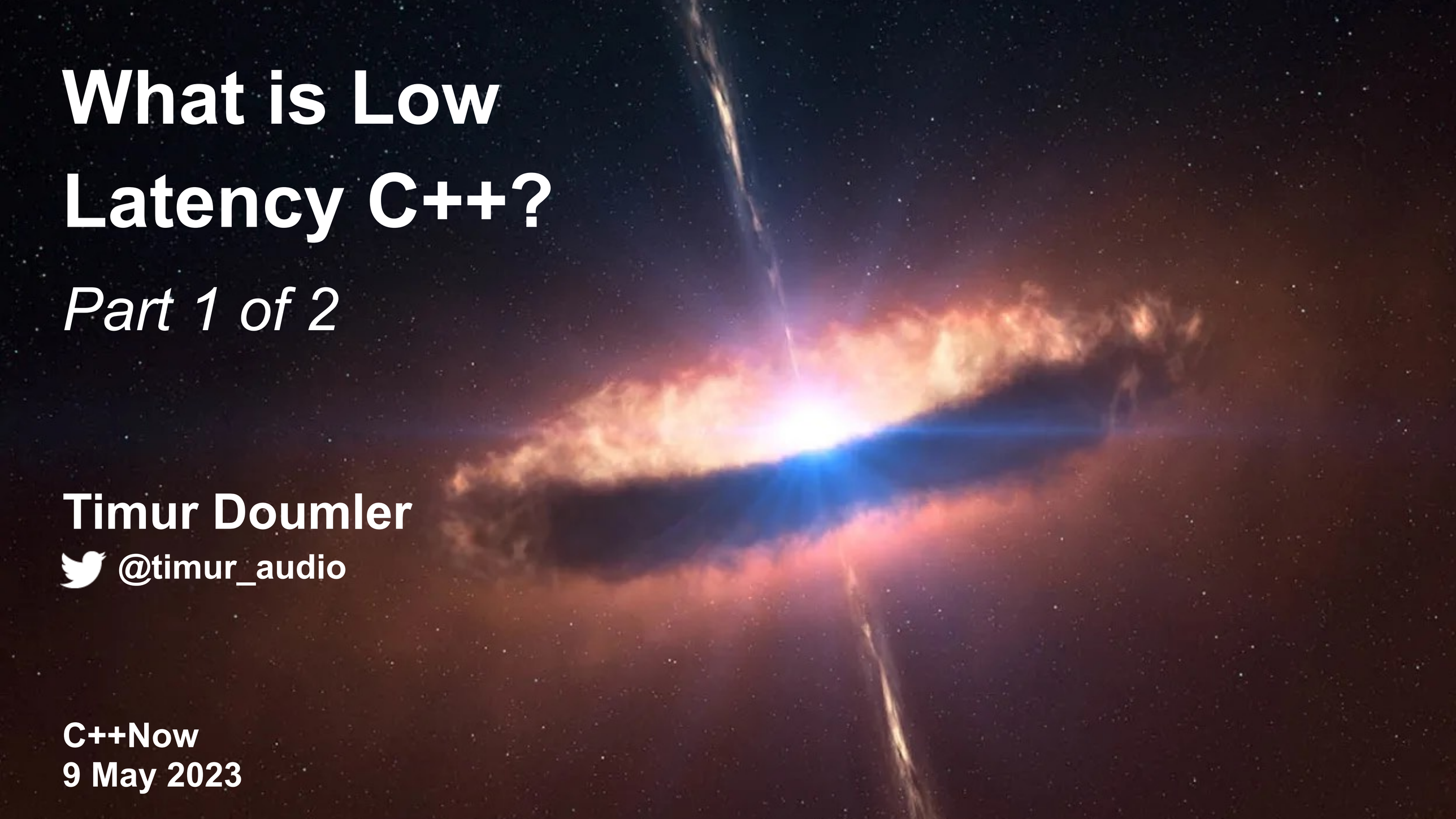


What is Low Latency C++?

Part 1 of 2

Timur Doumler
@timur_audio

C++Now
9 May 2023

JET BRAINS

CLion

A cross-platform IDE for C and C++ developers

ReSharper C++

Visual Studio Extension for C++ developers

Rider

Smart C++ and Blueprints support for crafting the best games

CppCast

```
auto CppCast = pod_cast<C++>("http://cppcast.com");
```


CppCast

The first podcast for C++ developers, by C++ developers!

With hosts Timur Doumler and Phil Nash.

Every two weeks, or so, we sit down with guests from the C++ community to discuss the latest news and what they have been up to.

Latest episode

AI Infrastructure

Episode 359, published Friday, 28 Apr 2023

Pro Tools File Edit View Track Clip Event AudioSuite Options Setup Window Avid Link Help

Mix: Home Again_The Spirit

TRACKS

Drum 1X
Drums
BosMicro
Kik
Crash Kik
Sub Drop
OpenHts
HH1
HH2
Snaps
RvCymbl
DrumFX1
DrumFX2
Drum1
[▼Drm1]
1 Drums.cm
2 Bass
Guitar
3 Keys
4 Synth
5 MFX
6 LDV
LDVcm02
7 LDV.Flat
8 LDVC.Flt
9 LDVV.cm
LDVc101
10 LDVC.cm
11 [▼LDV](L
12 LDV FX
13 BGV
14 BGV FX
15 [▼MSTR](
16 ⬇VCA(⬇V
17 Mix
18 DA10
19 VRT12MST
Out 1-2
20 Mic
21 HmAgn102
22 HmAgR_M

Memory Loc...

#	Name
1	Intro A
2	Intro B
3	V1
4	PC
5	C1
6	Re-Intro
7	V2
8	PC2
9	C2
10	C2b

GROUPS

! <ALL>
b Test
a ocaibwev
c Master Groups
d Stem Send
g Mix VCAs ([▼MST
h ⬇ VCA (⬇ VCA)
i Drum VCA
j Bass VCA ([▼Bass
k Synth1 VCA ([▼Sy
l Keys VCA ([▼Keys
m LDV
n BG V

System Usage
CPU Total 23%
Activity

Track: LDV / The Spirit
Preset: <factory default>
Auto: BYPASS / SAFE / COMPARE / Native
Preset: init
UI Scale: 100%

-5.9 -5.9
rider
gain
input
compression
focus
character
high
low
eq
stereo
mix
doubler
filter
48%
room plate
reverb
-1.4 -1.4
gain
output
THE SPIRIT

HOME AGAIN
BY LYRA BROWN

cradle

My low-latency/real-time programming talks so far

- *C++ in the Audio Industry (CppCon 2015)*
- *C++ in the Audio Industry, Episode II: Floating Atomics (JUCE Summit 2015)*
- *Want fast C++? Know your hardware! (CppCon 2016)*
- *Lock-free programming with modern C++ (ACCU 2016)*
- *Using locks in real-time audio programming, safely (ADC 2020)*
- *Real-time programming with the C++ standard library (CppCon 2021)*
- *A lock-free atomic shared_ptr (C++Now 2022)*
- *Thread synchronisation in real-time audio processing with RCU (ADC 2022)*

Low latency programming domains

- Video games
- Audio processing
- High-frequency trading (HFT)
- Embedded
  - Automotive
  - Aeronautics
  - Robotics
  - Medical
  - ...

This talk is not about...

- Embedded systems
- Mainframes
- Exotic architectures
- GPUs
- Heterogeneous computing
- Safety & security
- Language design
- C++ successor languages

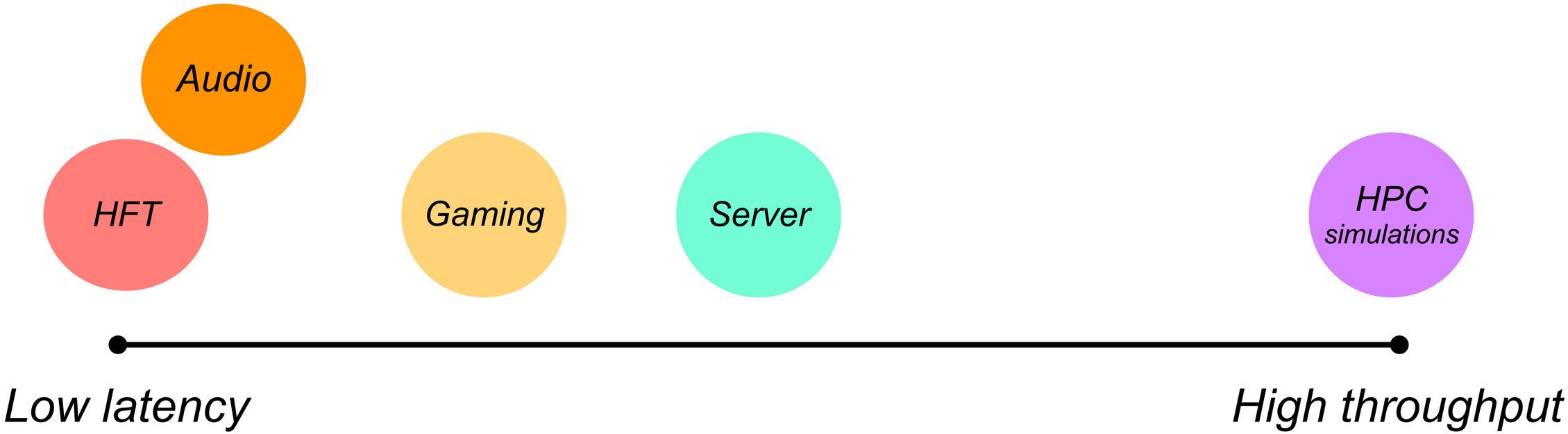
"Low Latency"?

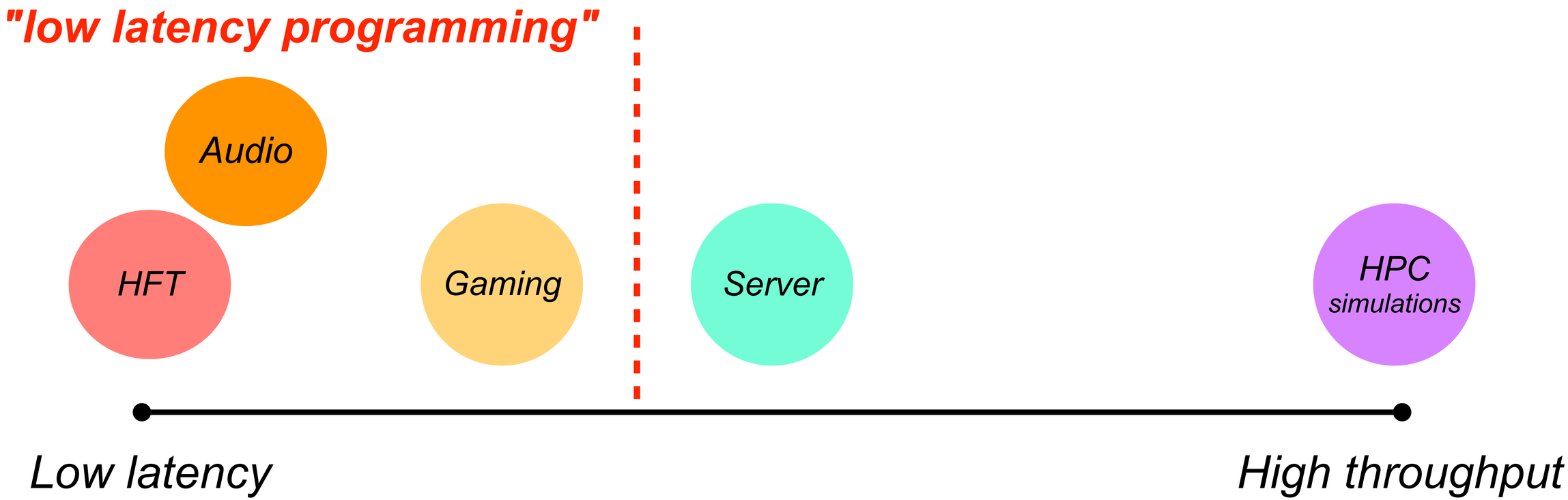
"Performance"?

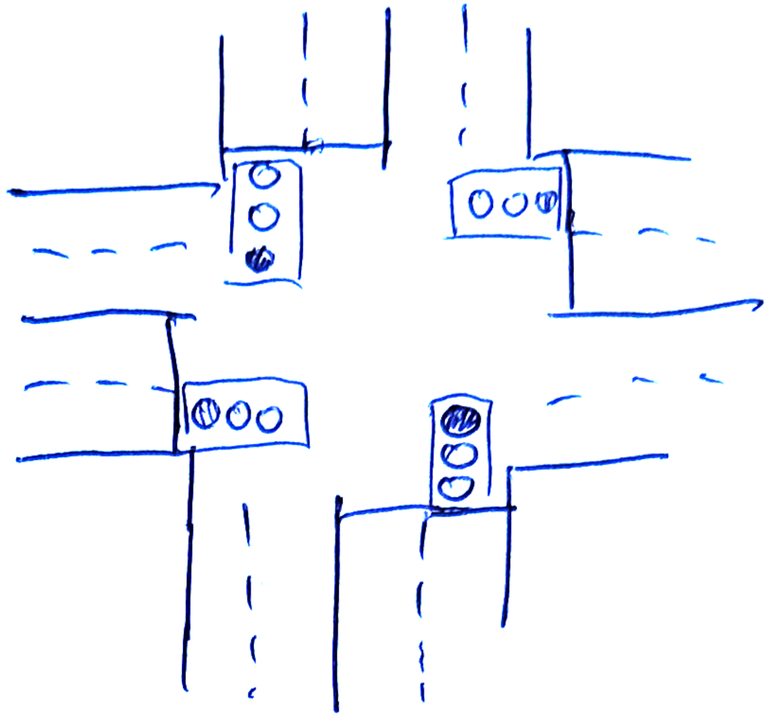

latency
throughput

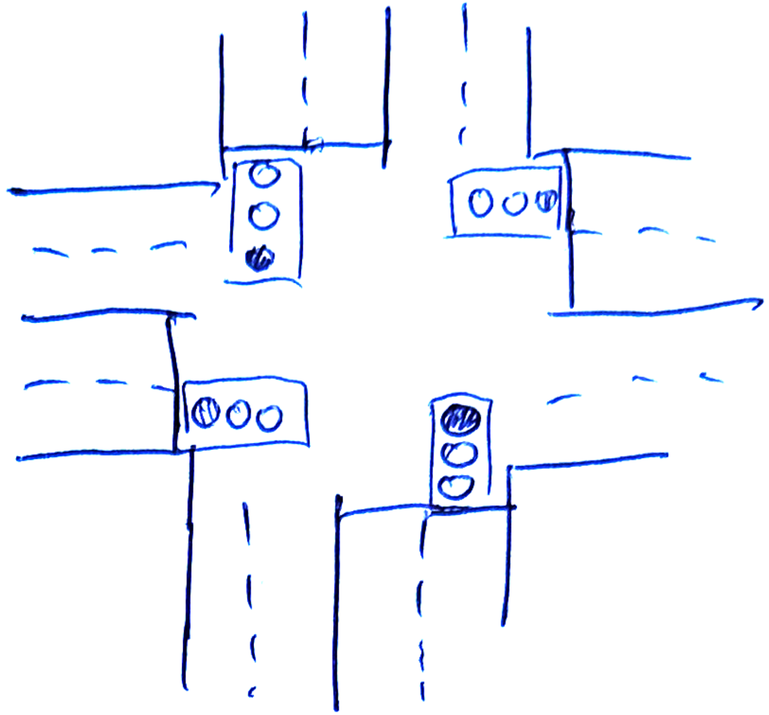
Low latency

High throughput

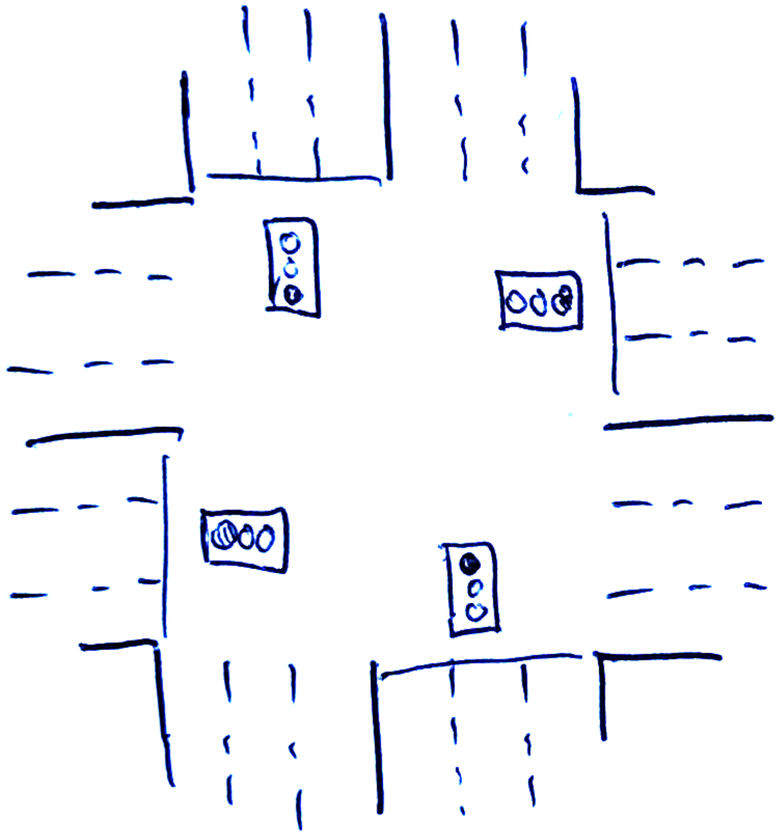

Audio
HFT
Gaming
Server
HPC
simulations
Low latency
High throughput

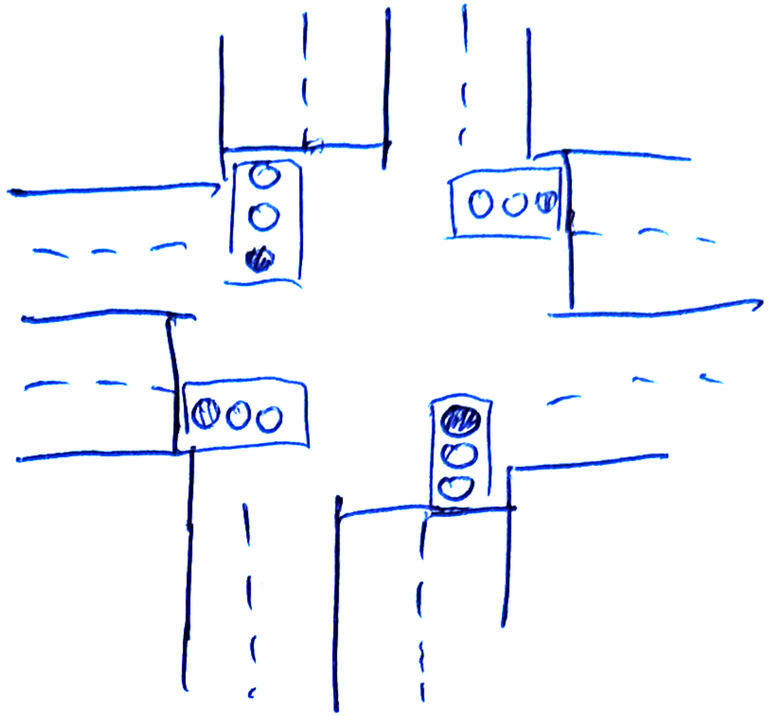

"low latency programming"
Audio
HFT
Gaming
Server
HPC
simulations
Low latency
High throughput



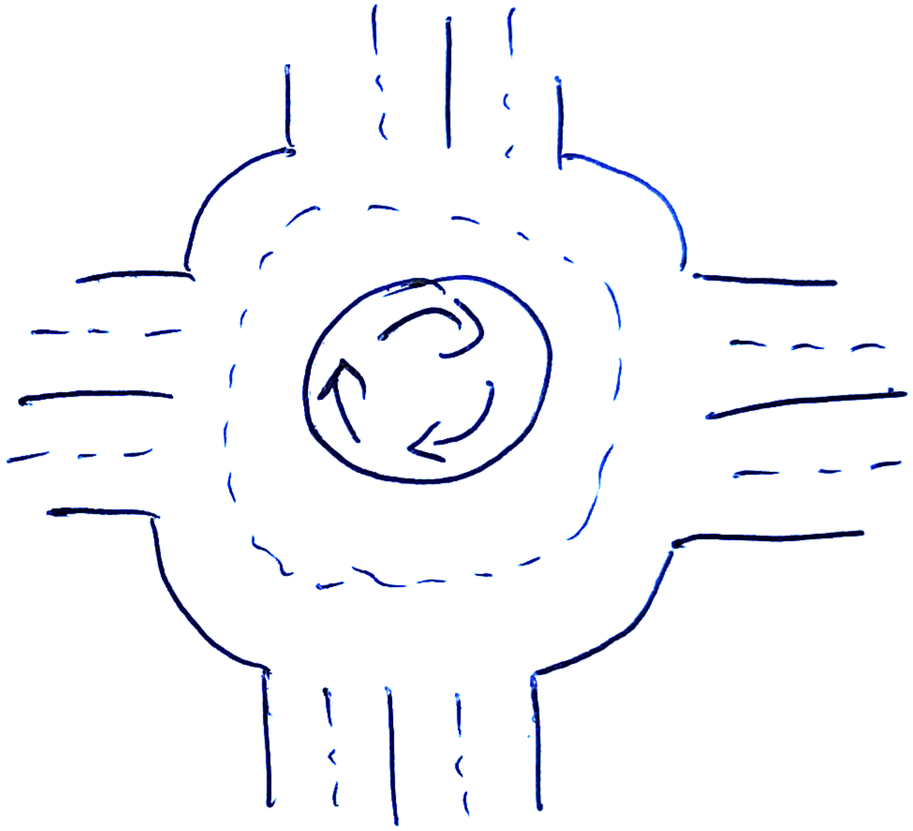


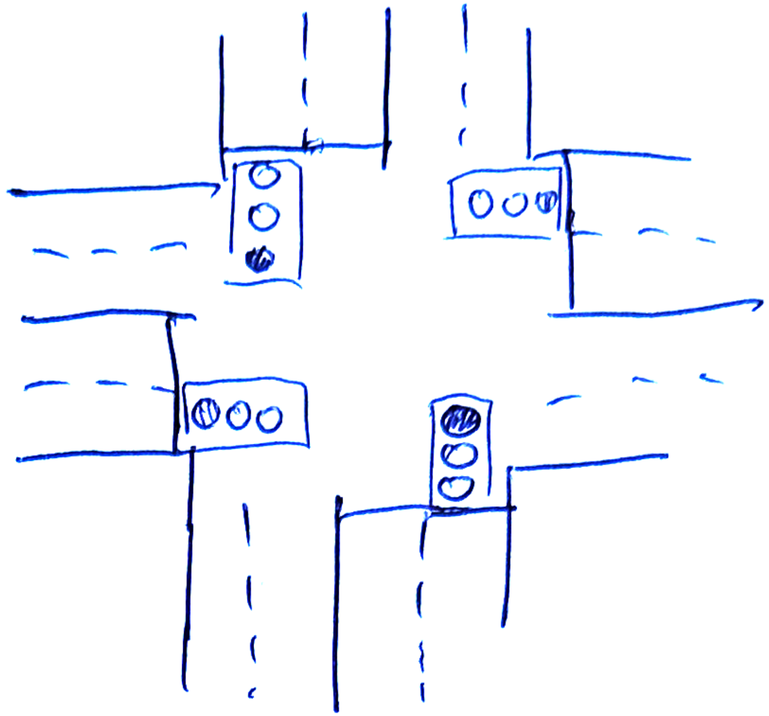
optimising
for high throughput



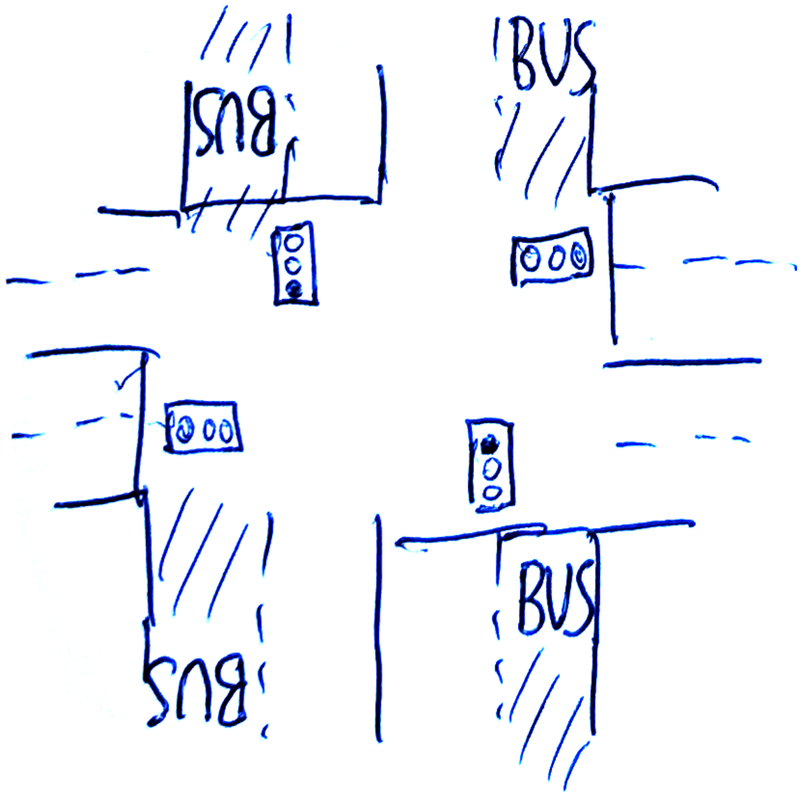


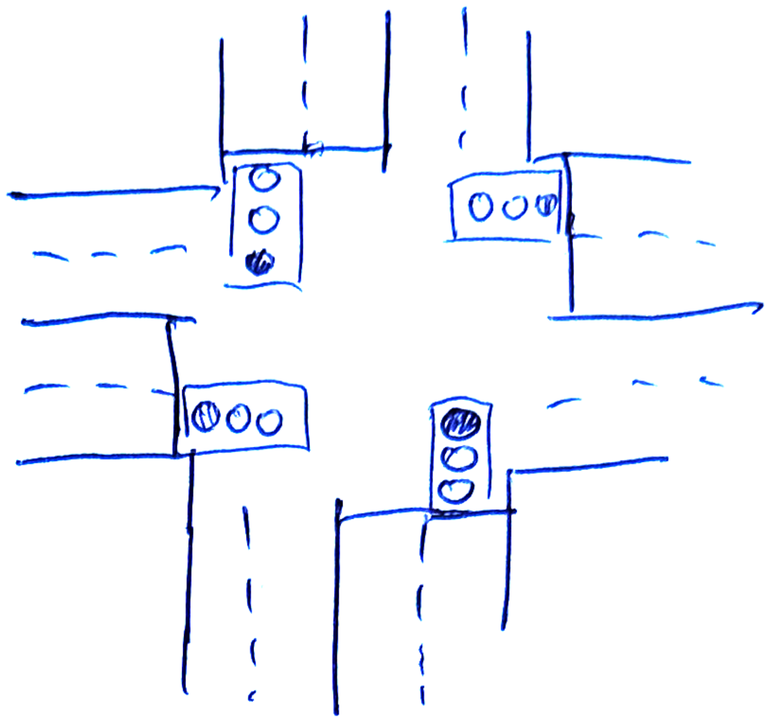
optimising
for high throughput
and low latency?



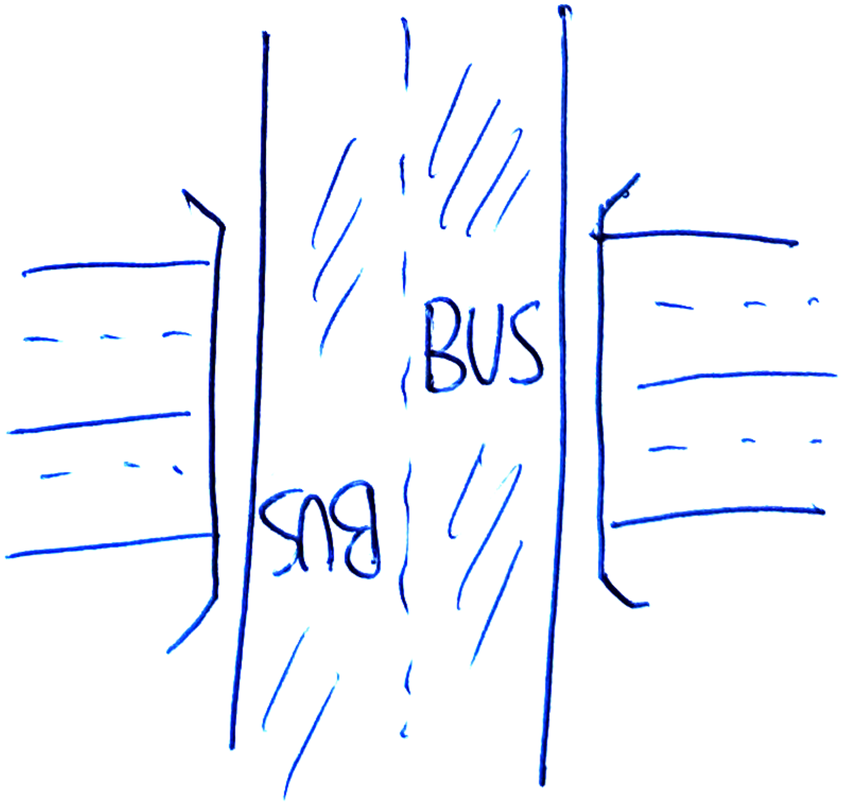


optimising
for low latency?



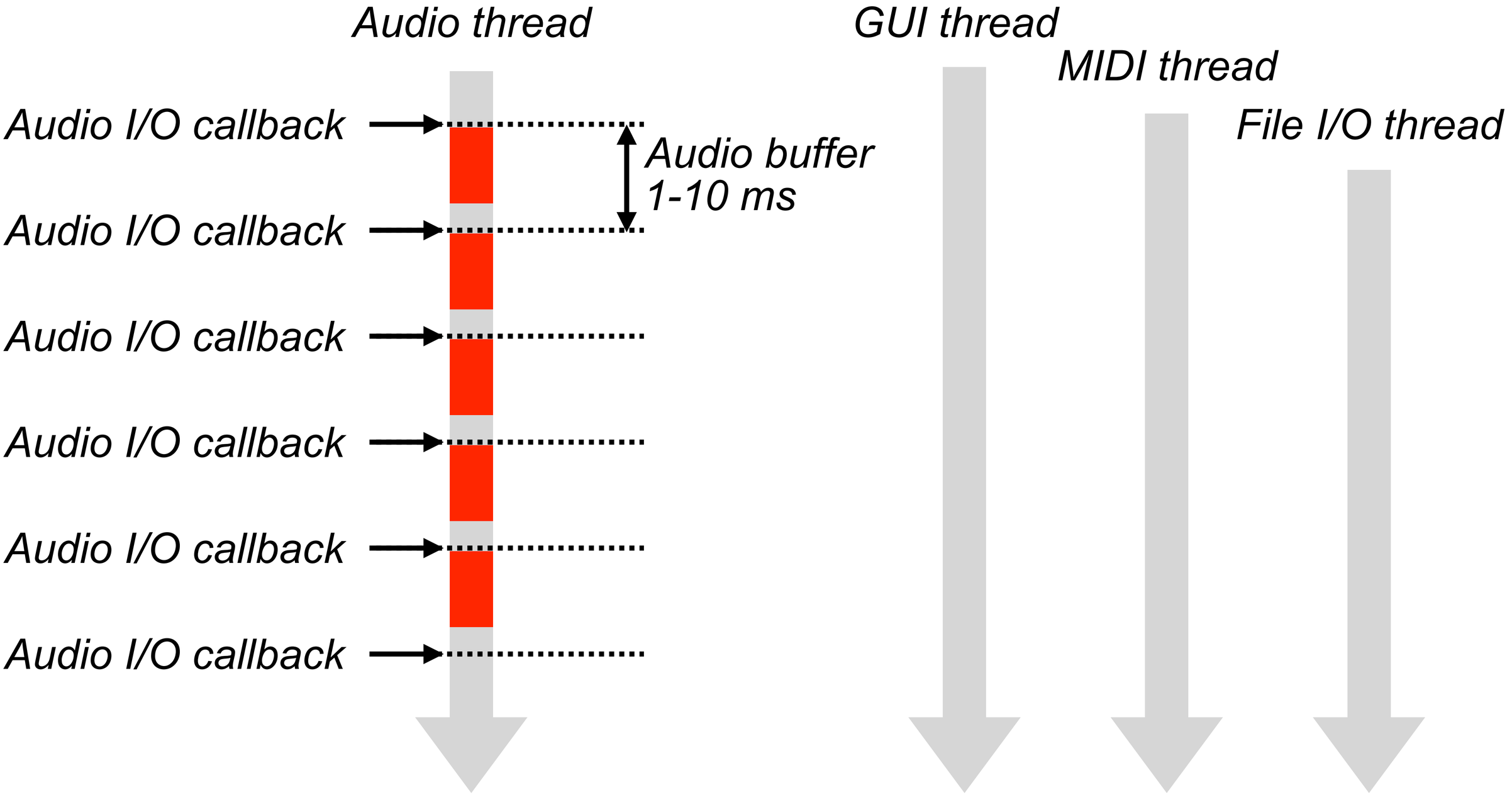


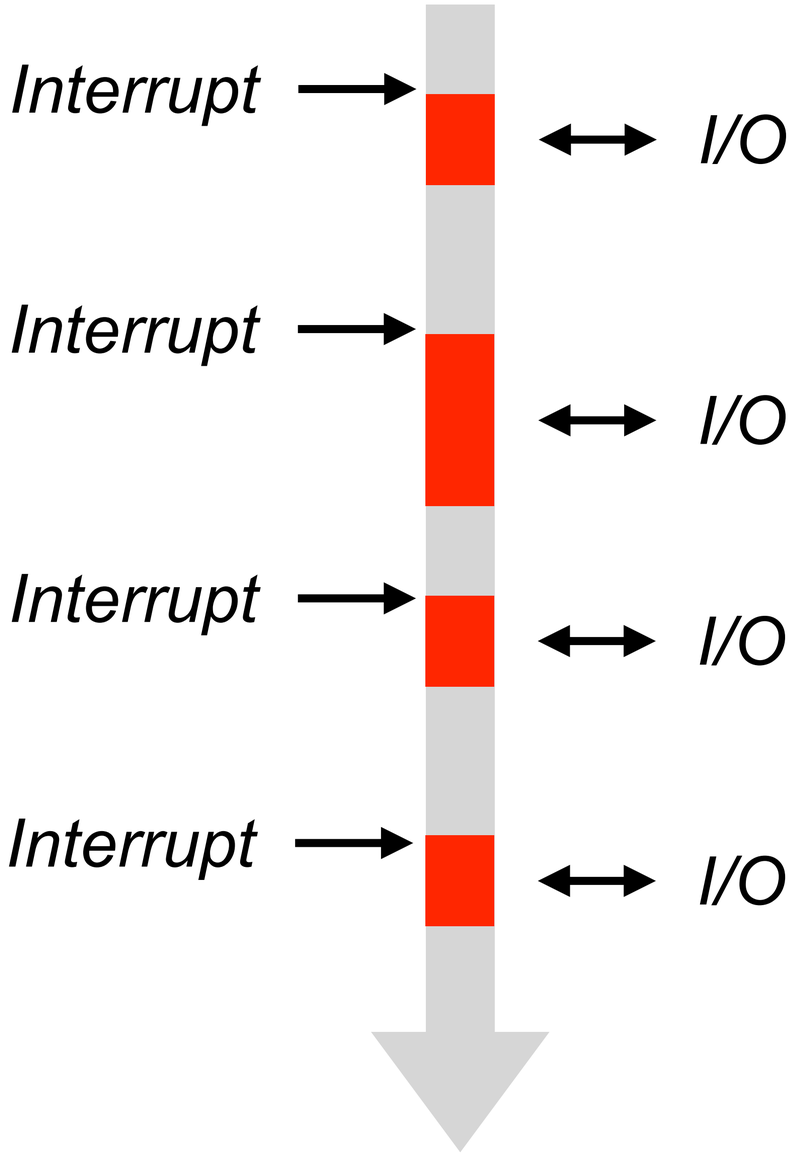
optimising
for low latency!

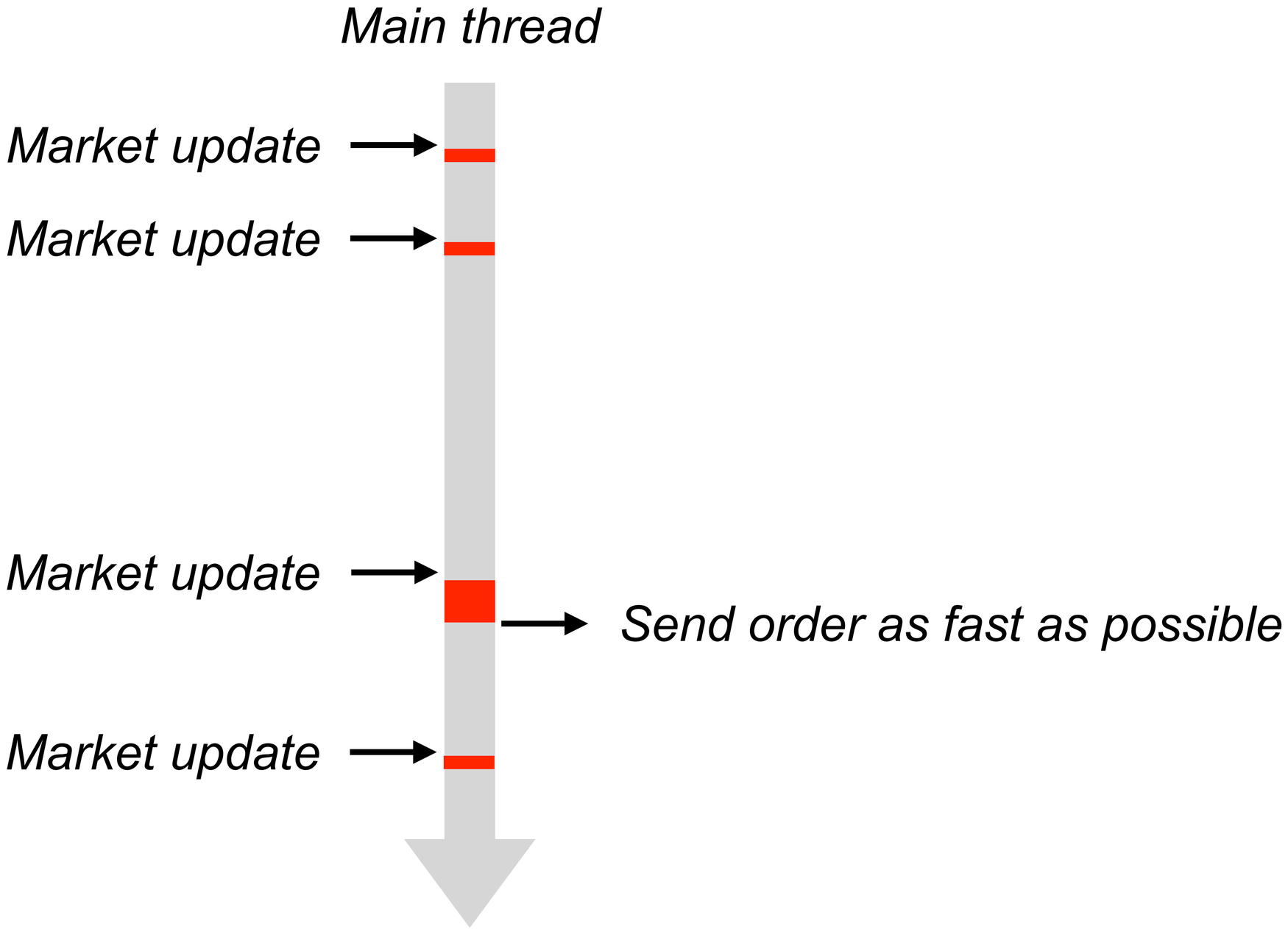


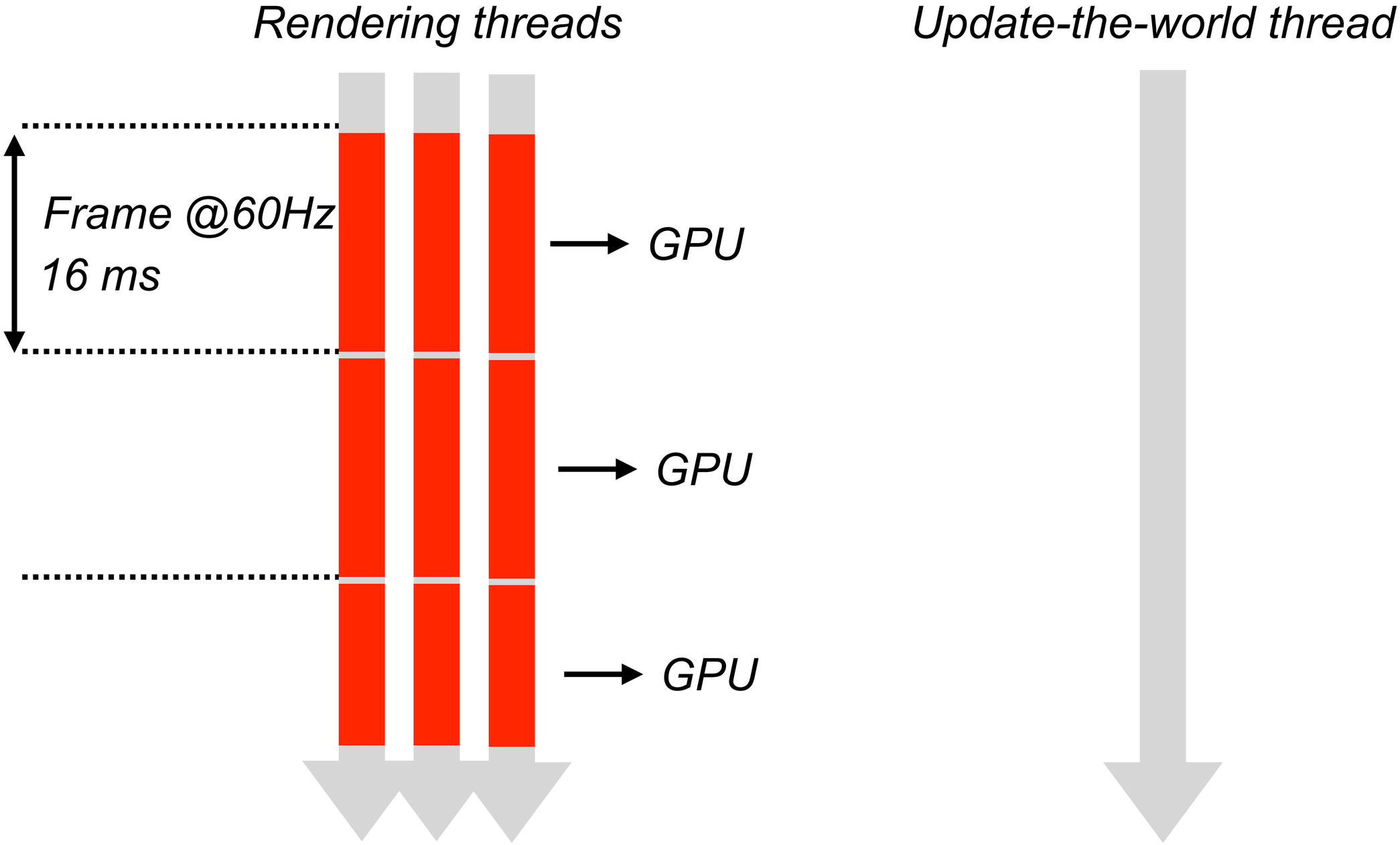

Smart & Final

"Hot path"

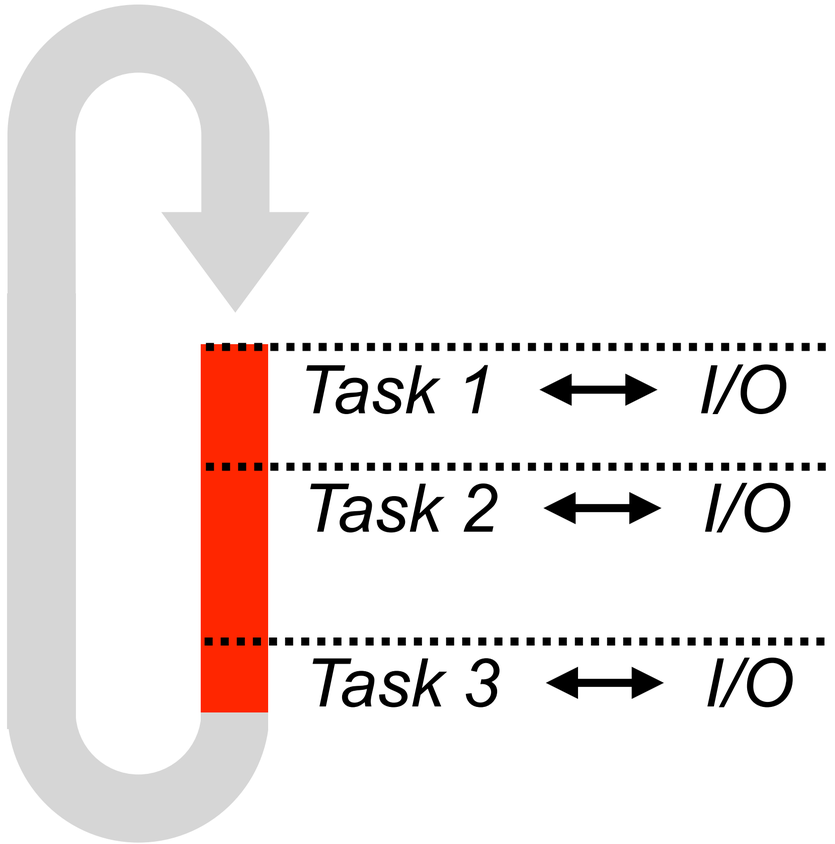

Audio thread
GUI thread
MIDI thread
File I/O thread
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio I/O callback
Audio buffer
1-10 ms

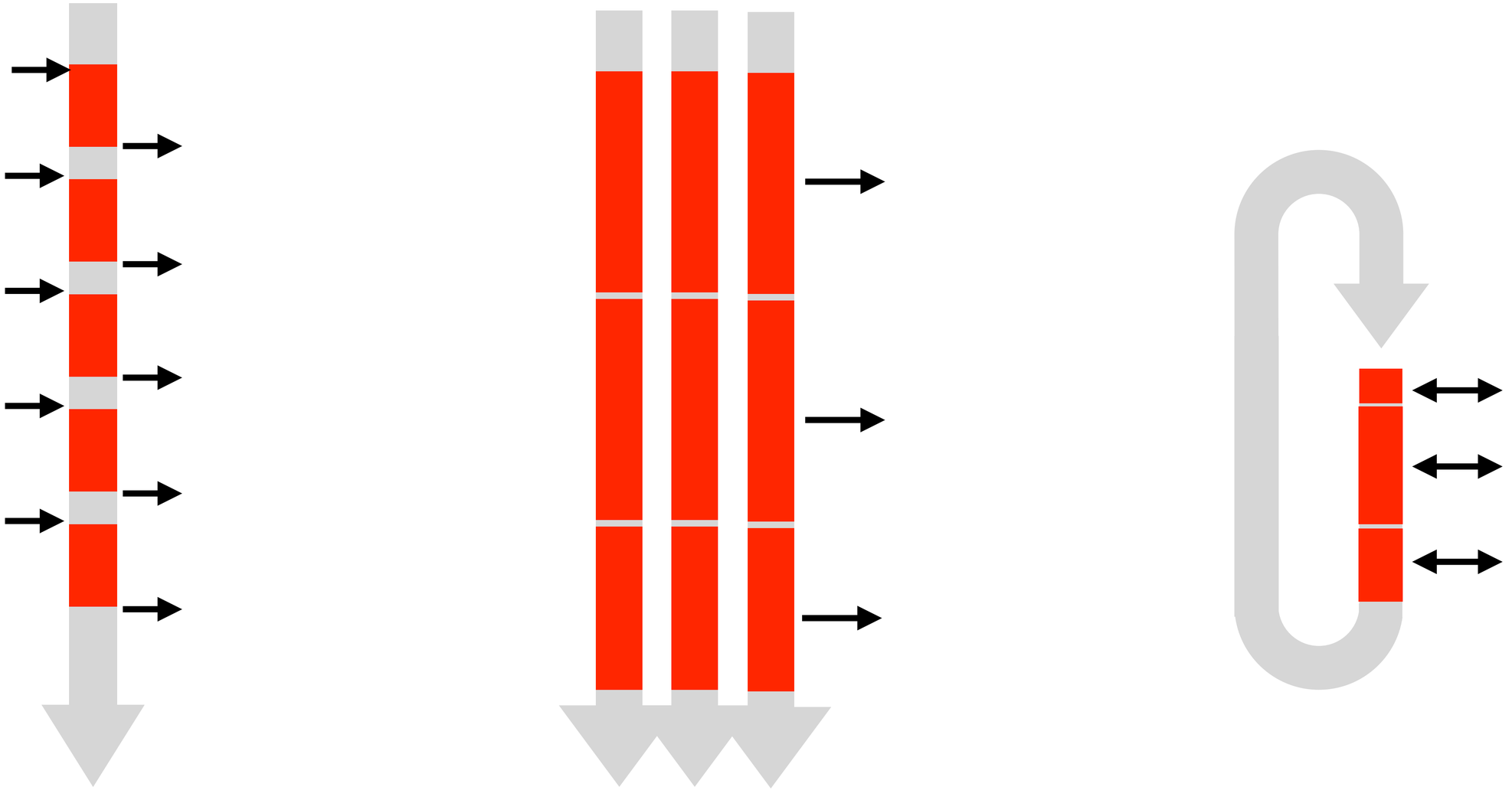

Interrupt
I/O
Interrupt
I/O
Interrupt
I/O
Interrupt
I/O


Main thread
Market update
Market update
Market update
Send order as fast as possible
Market update


Rendering threads
Update-the-world thread
Frame @60Hz
16 ms
GPU
GPU
GPU

"Superloop"





"Real-time"?

“Real-time”

In order to be considered correct,

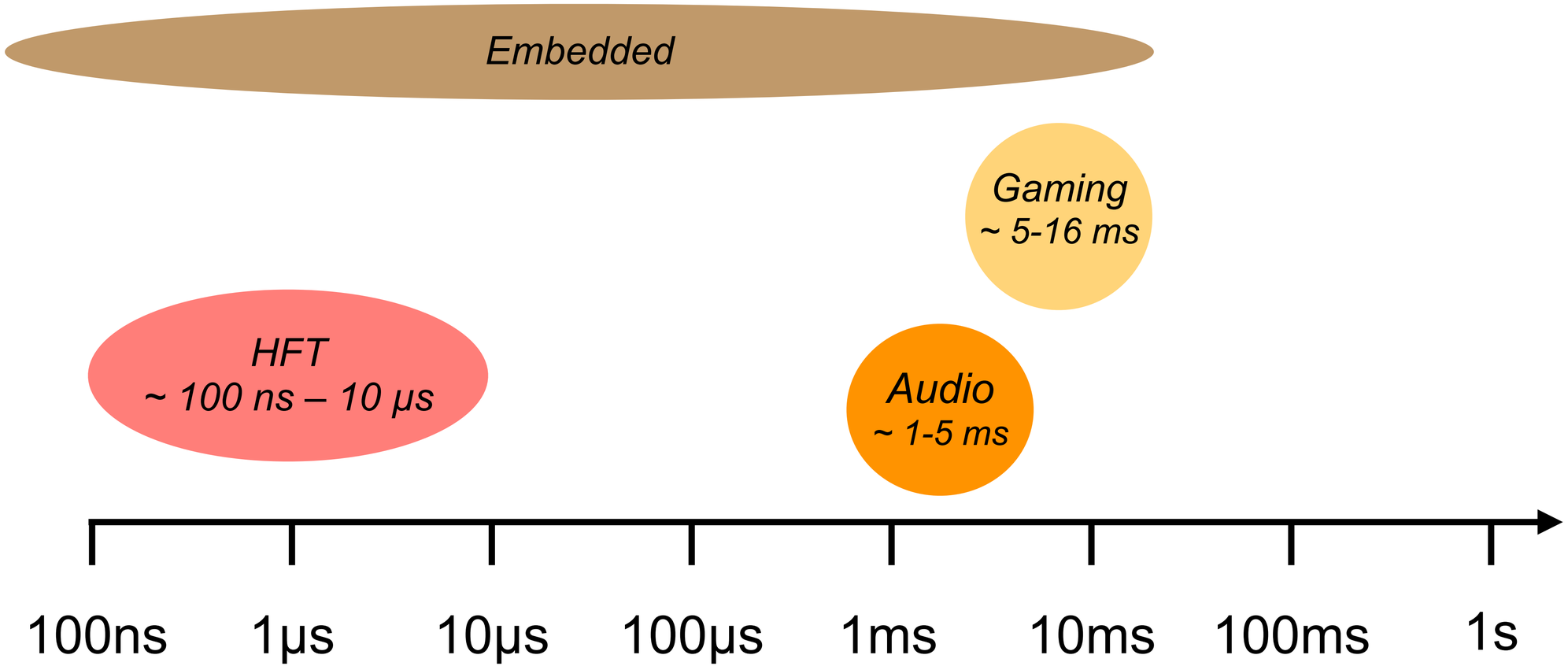
not only does the program have to

produce the correct result,

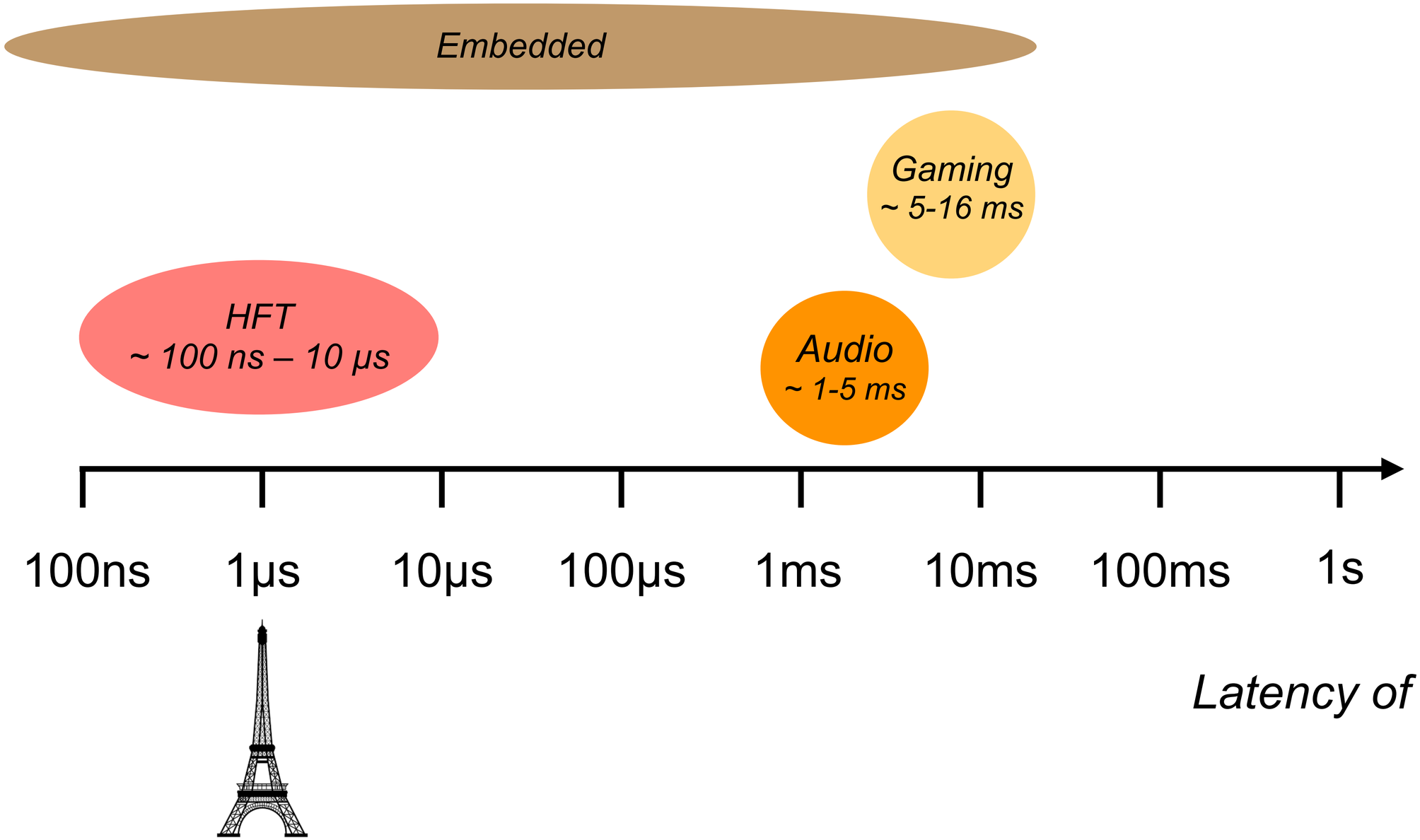
but it also has to produce it

within a certain amount of time.

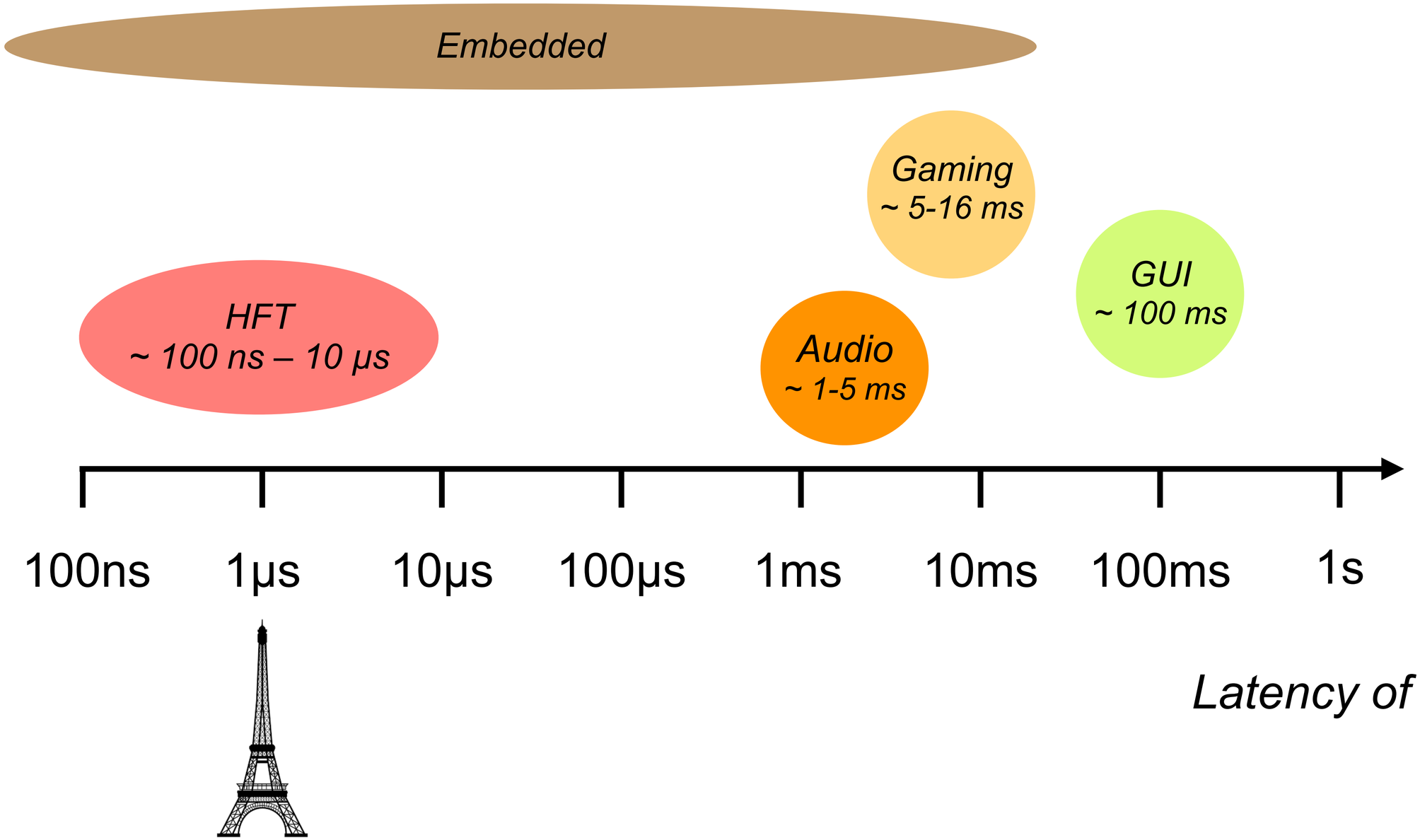
How long is the deadline?



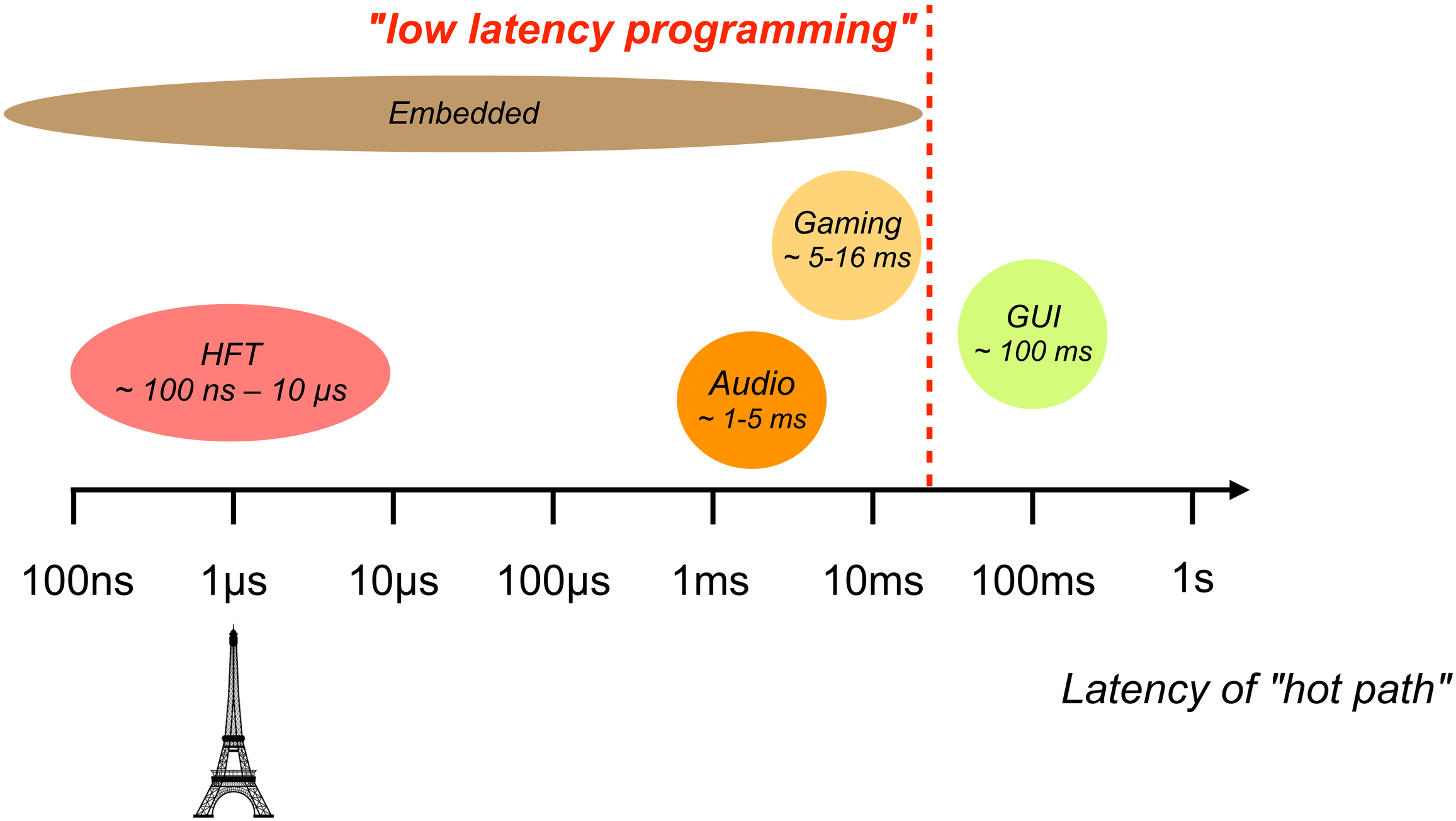
Latency of "hot path"



Latency of "hot path"



Latency of "hot path"



Latency of "hot path"

Consequences of "hot path" missing deadline

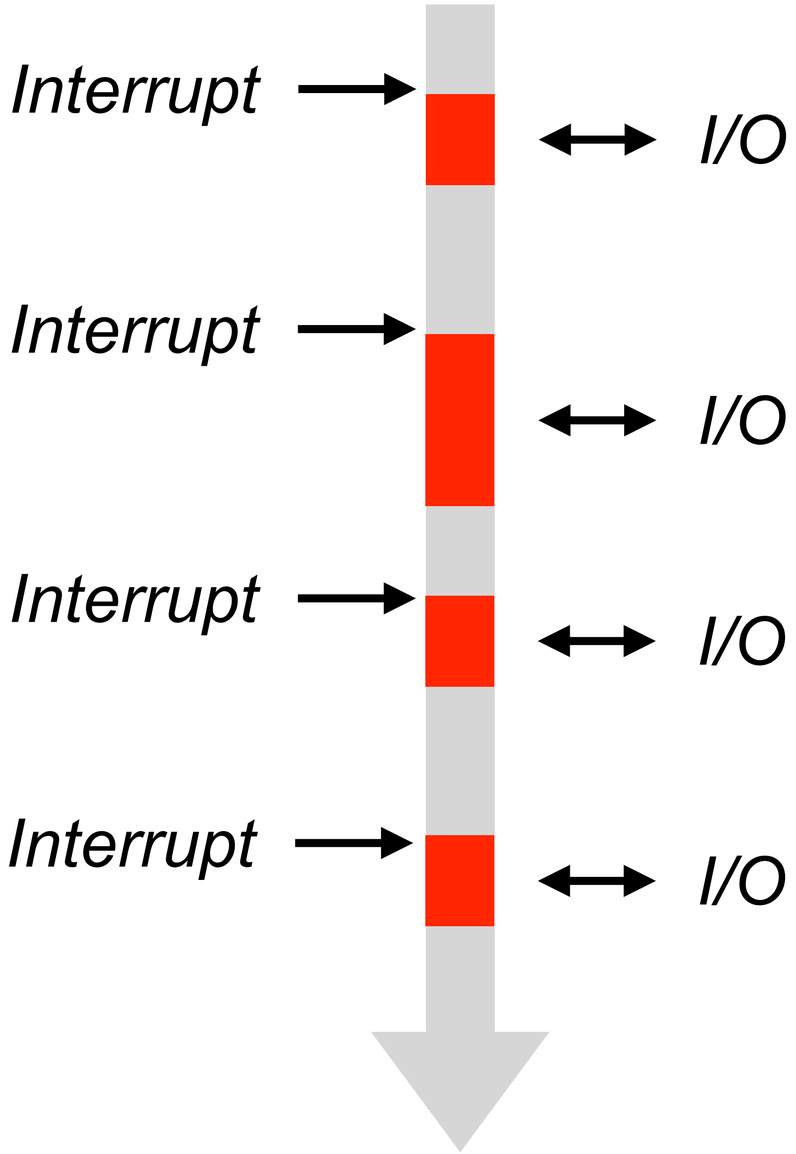
- Gaming: dropped video frames
- Audio: audible clicks/glitches
- HFT: lost money (potentially a lot of money)
- Embedded (some use cases): risk for human life

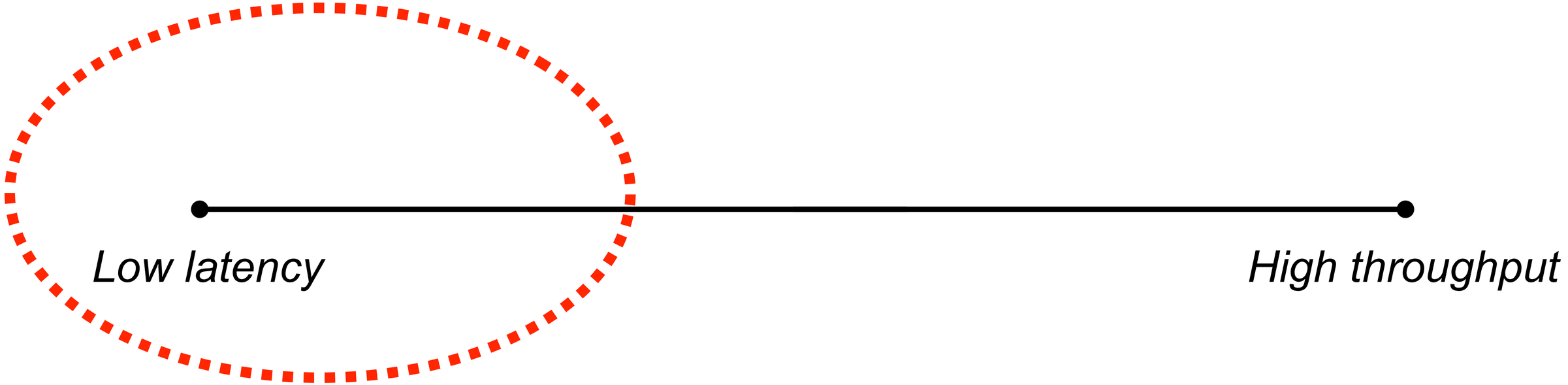
- **"Hard real time"**
  – *Any missed deadline is considered a system failure*

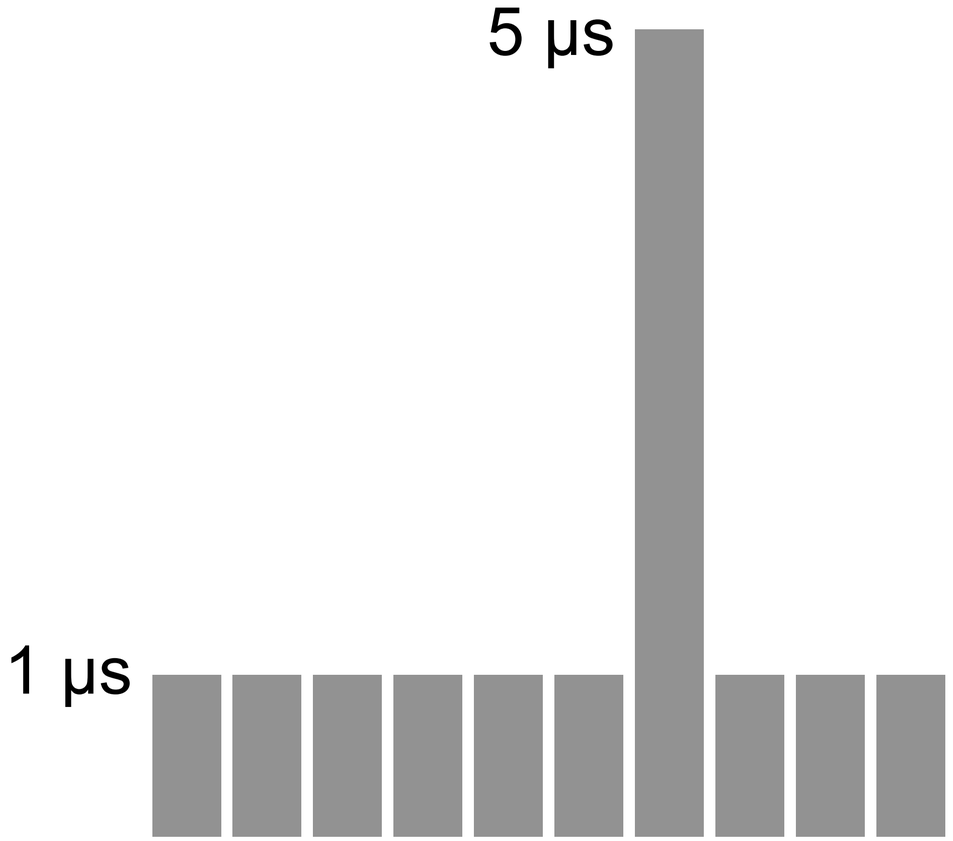
- **"Firm real time"**
  – *The system might survive missed deadlines, so long as they are adequately spaced, however the value of the task's completion drops to zero or becomes impossible.*

- **"Soft real time"**
  – *The system can tolerate missed deadlines; as long as tasks are timely executed, their results continue to have value. The value of completed tasks decreases past the deadline.*

"Jitter"


Interrupt
I/O
Interrupt
I/O
Interrupt
I/O
Interrupt
I/O

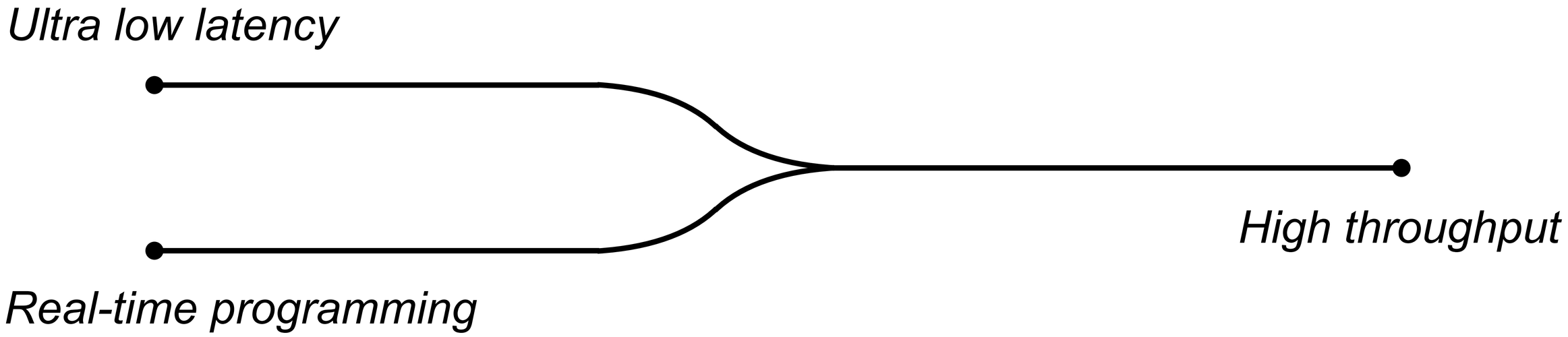

Low latency
High throughput



Average: 1.4 µs

Average: 1.8 µs

Low latency *High throughput*


Ultra low latency
High throughput
Real-time programming

"Efficiency"

“Efficiency”

To program efficiently is to reduce the amount of work that the program has to perform to complete a certain task.

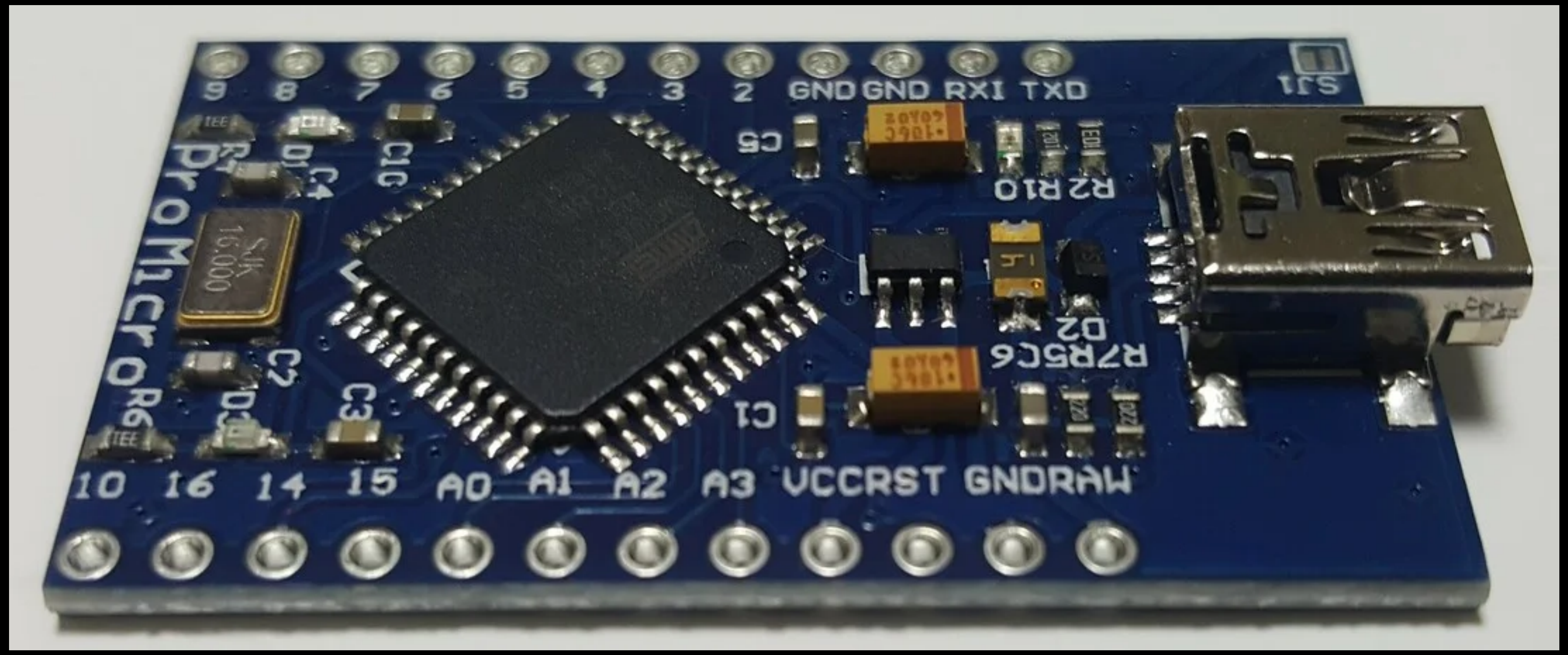
Low latency *High throughput*

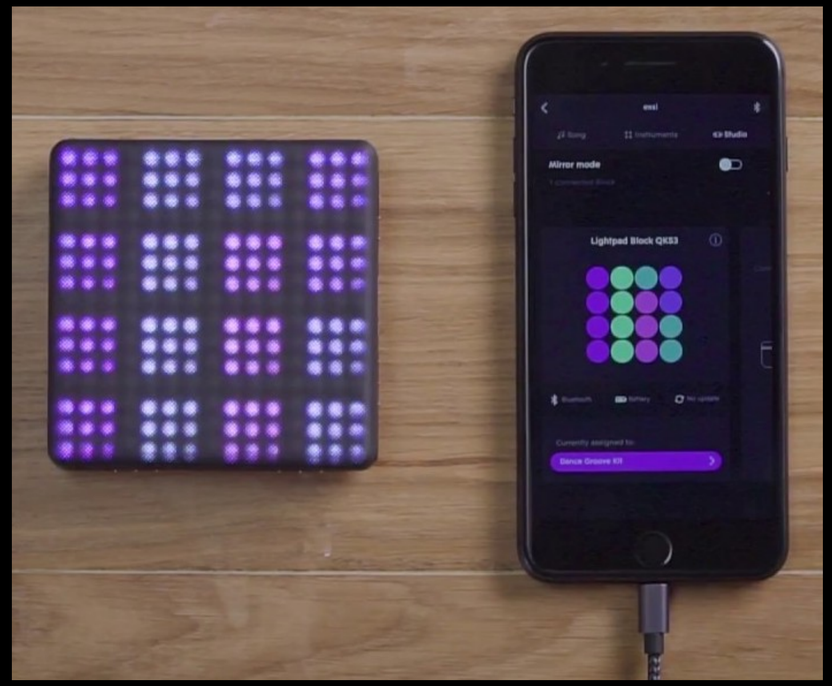
Low latency *Efficiency* *High throughput*

Low latency *Efficiency* *High throughput*

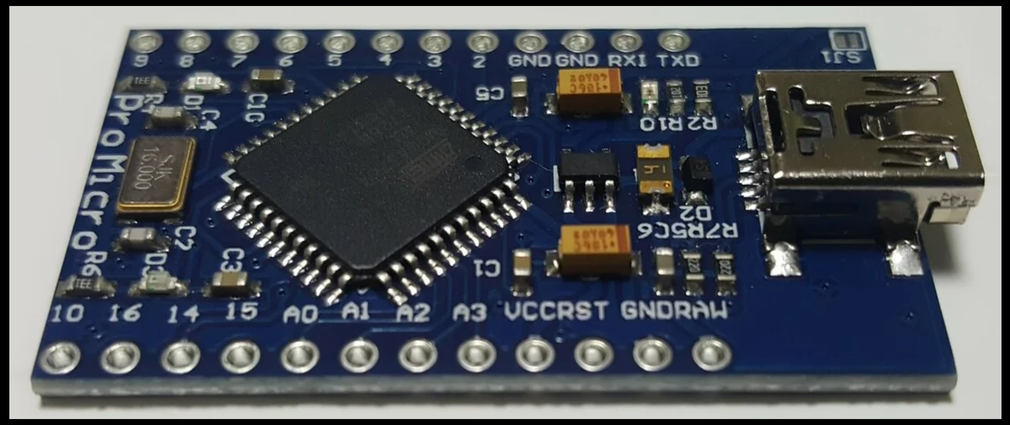
Low latency *Efficiency* *High throughput*

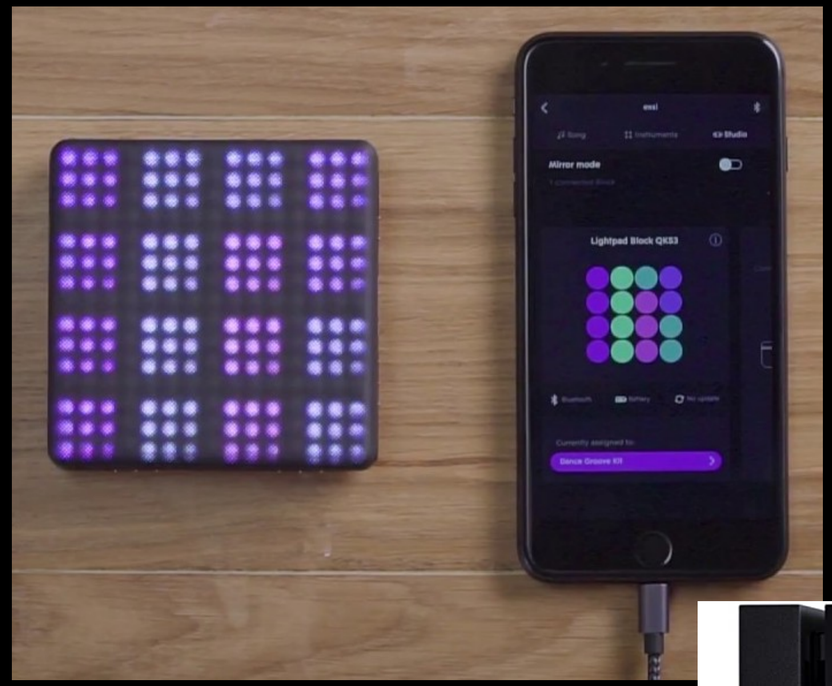
Low latency *Efficiency* *High throughput*


9 8 7 6 5 4 3 2 GND GND RXI TXD
ProMicro
10 16 14 15 A0 A1 A2 A3 VCC RST GND RAW

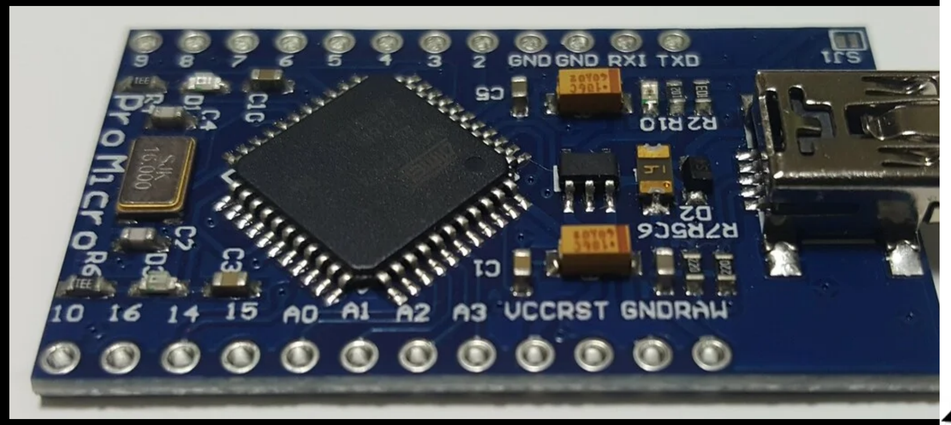

Mirror mode
Lightpad Block QK33
Dance Groove Kit


KOMPLETE

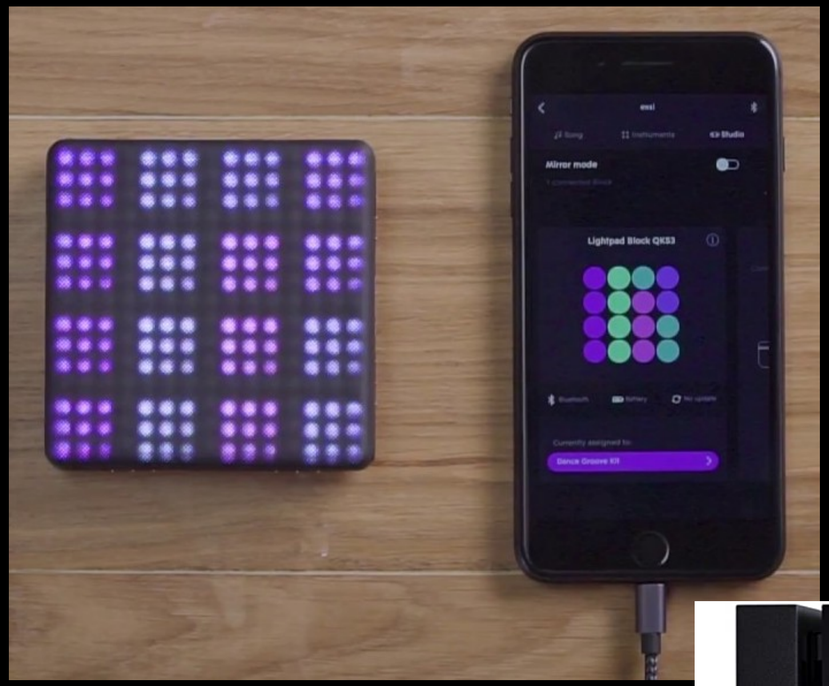

GND GND RXI TXD
10 16 14 15 A0 A1 A2 A3 VCCRST GNDRAW


Mirror mode
Lightpad Block QK33
Dance Groove Kit


KOMPLETE

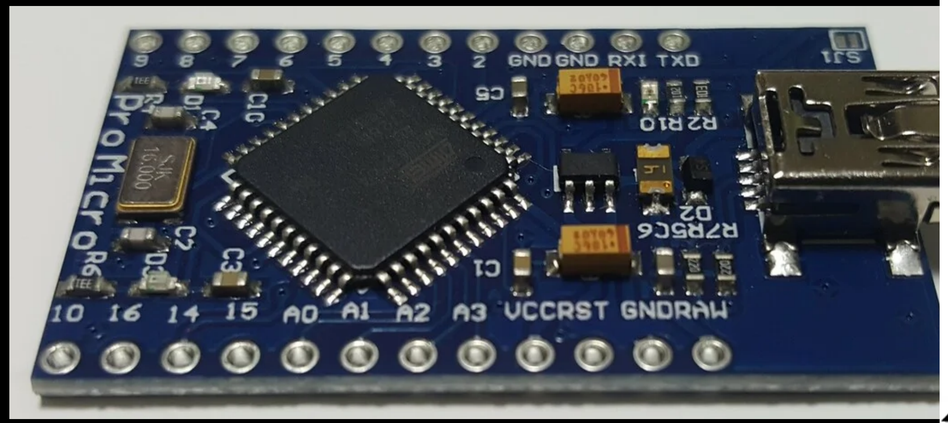

GND GND RXI TXD
ProMicro
10 16 14 15 A0 A1 A2 A3 VCCRST GNDRAW


SONY
PS4


Mirror mode
Lightpad Block QK33
Dance Groove Kit


KOMPLETE


SONY
PS4


GND GND RXI TXD
ProMicro
10 16 14 15 A0 A1 A2 A3 VCCRST GNDRAW

"Hot path"

"Deadline"

Why use C++ for low-latency programming?

- Manual memory management

  You can program in such a way as to avoid allocations/deallocations and blocking in the hot path

.

Why use C++ for low-latency programming?

- Manual memory management

  You can program in such a way as to avoid allocations/deallocations and blocking in the hot path

- Scalable zero-cost abstractions

  e.g. algorithms, templates

-

Why use C++ for low-latency programming?

- Manual memory management

  You can program in such a way as to avoid allocations/deallocations and blocking in the hot path

- Scalable zero-cost abstractions

  e.g. algorithms, templates

- Huge body of existing libraries and frameworks

  e.g. JUCE (audio), Unreal Engine (gaming)

C++ techniques for low-latency programming

C++ techniques for low-latency programming

- Two categories:
  - Efficient programming
  - Programming for deterministic execution time

- Most crucial thing: measuring!

C++ techniques for low-latency programming

- Two categories:
  - Efficient programming
  - Programming for deterministic execution time

- **Most crucial thing: measuring!**

Measuring!

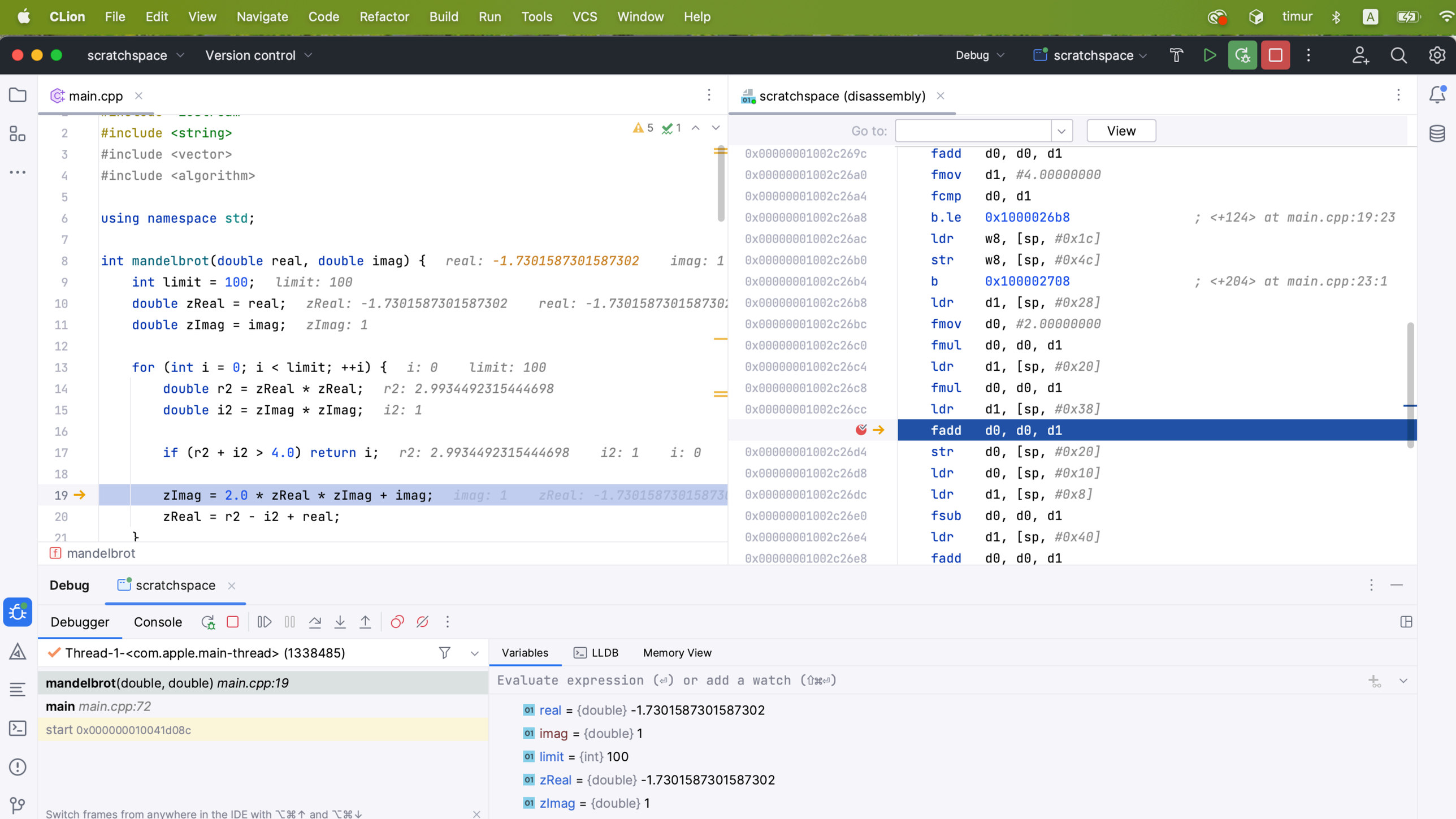
- You don't know whether something is efficient, fast, etc.
  unless you measure it
- Results often surprising/counterintuitive
  - *Fedor Pikus: The Art of Writing Efficient Programs (Packt Publishing, 2021)*
  - *Dave Rowland: Optimising a real-time audio processing library (CppOnSea 2023)*
  - *C++ Weekly, Episode 274: Why Is My Pair 310x Faster Than `std::pair`?*

Measuring!

- Profiling
  - gprof, perf, VTune, DTrace, valgrind, Optick...
  - sampling profiling vs. instrumentation profiling
    - *Mathieu Ropert: "The Basics of Profiling" (CppCon 2021)*
    - *Chandler Carruth: "Going Nowhere Faster" (CppCon 2017)*
- Performance analysis
  - Measure cache misses (perf stat, cachegrind)
  - Measure branch mispredicts, syscalls, allocations... (various platform-specific tools)
  - Measure performance of machine instructions on a particular CPU (llvm-mca)

Measuring!

- Inspect generated assembly (fewer instructions != faster code!)
  - Compiler explorer
  - CLion Disassembly on Demand


CLion File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
timur
scratchspace Version control
Debug scratchspace
main.cpp
scratchspace (disassembly)
Go to: View
#include <string>
#include <vector>
#include <algorithm>
using namespace std;
int mandelbrot(double real, double imag) { real: -1.7301587301587302 imag: 1
int limit = 100; limit: 100
double zReal = real; zReal: -1.7301587301587302 real: -1.730158730158730
double zImag = imag; zImag: 1
for (int i = 0; i < limit; ++i) { i: 0 limit: 100
double r2 = zReal * zReal; r2: 2.9934492315444698
double i2 = zImag * zImag; i2: 1
if (r2 + i2 > 4.0) return i; r2: 2.9934492315444698 i2: 1 i: 0
zImag = 2.0 * zReal * zImag + imag; imag: 1 zReal: -1.730158730158730
zReal = r2 - i2 + real;
}
mandelbrot
0x000000001002c269c fadd d0, d0, d1
0x000000001002c26a0 fmov d1, #4.00000000
0x000000001002c26a4 fcmp d0, d1
0x000000001002c26a8 b.le 0x1000026b8 ; <+124> at main.cpp:19:23
0x000000001002c26ac ldr w8, [sp, #0x1c]
0x000000001002c26b0 str w8, [sp, #0x4c]
0x000000001002c26b4 b 0x100002708 ; <+204> at main.cpp:23:1
0x000000001002c26b8 ldr d1, [sp, #0x28]
0x000000001002c26bc fmov d0, #2.00000000
0x000000001002c26c0 fmul d0, d0, d1
0x000000001002c26c4 ldr d1, [sp, #0x20]
0x000000001002c26c8 fmul d0, d0, d1
0x000000001002c26cc ldr d1, [sp, #0x38]
fadd d0, d0, d1
0x000000001002c26d4 str d0, [sp, #0x20]
0x000000001002c26d8 ldr d0, [sp, #0x10]
0x000000001002c26dc ldr d1, [sp, #0x8]
0x000000001002c26e0 fsub d0, d0, d1
0x000000001002c26e4 ldr d1, [sp, #0x40]
0x000000001002c26e8 fadd d0, d0, d1
Debug scratchspace
Debugger Console
Thread-1-<com.apple.main-thread> (1338485)
Variables LLDB Memory View
Evaluate expression (⏎) or add a watch (⇧⌘⏎)
mandelbrot(double, double) main.cpp:19
main main.cpp:72
start 0x000000010041d08c
real = {double} -1.7301587301587302
imag = {double} 1
limit = {int} 100
zReal = {double} -1.7301587301587302
zImag = {double} 1
Switch frames from anywhere in the IDE with ⌥⌘↑ and ⌥⌘↓

Measuring!

- Inspect generated assembly (fewer instructions != faster code!)
  - Compiler explorer
  - CLion Disassembly on Demand
- Benchmarking
  - Google Benchmark, gperftools, Nonius...
  - Quick-bench
  - actual code vs. microbenchmarks

Microbenchmarks are tricky

- Warm the cache
- Randomise the heap
- Measure release build with same compiler flags
- But optimisations change what code you are measuring!
    - A lot of stuff can "constexpr away" in microbenchmark but not in production code

C++ techniques for low-latency programming

- Two categories:
  - Efficient programming
  - Programming for deterministic execution time

- Most crucial thing: measuring!

C++ techniques for low-latency programming

- Two categories:
  - **Efficient programming**
  - Programming for deterministic execution time

- Most crucial thing: measuring!

Writing efficient code requires...

- Knowledge of the programming language
- Knowledge of the libraries used
- Knowledge of the compiler
  - Optimiser
  - ABI (Itanium, Microsoft, ...)
- Knowledge of the hardware architecture
  - CPU architecture (Instruction set, pipeline, SIMD...)
  - Cache hierarchy (Registers, L1/2/3 cache)
  - Prefetcher, translation lookaside buffer
  - Branch predictor, branch target buffer

Writing efficient code requires...

- Knowledge of the programming language
- Knowledge of the libraries used
- Knowledge of the compiler
  - Optimiser
  - ABI (Itanium, Microsoft, ...)
- Knowledge of the hardware architecture
  - CPU architecture (Instruction set, pipeline, SIMD...)
  - Cache hierarchy (Registers, L1/2/3 cache)
  - Prefetcher, translation lookaside buffer
  - Branch predictor, branch target buffer

Avoid unnecessary work

- Avoid unnecessary copies
- Avoid unnecessary function calls / indirections
- Make as many decisions as possible at compile time
- ...

Avoid unnecessary work

- Avoid unnecessary copies
- Avoid unnecessary function calls / indirections
- Make as many decisions as possible at compile time
- ...

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [&prefix](const std::string& s) {
        return s == prefix + "bar";
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [&prefix](const std::string& s) {
        return s == prefix + "bar";
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [&prefix](const std::string& s) {
        return s == prefix + "bar";
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [&prefix](const std::string& s) {
        return s == prefix + "bar";
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [str = prefix + "bar"](const std::string& s) {
        return s == str;
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

```
const std::vector<std::string> vs = {“apple", "orange", "foobar", "lemon"};
const std::string prefix = "foo";

auto result = std::find_if(
    vs.begin(), vs.end(),
    [str = prefix + "bar"](const std::string& s) {
        return s == str;
    });

if (result != vs.end())
    std::cout << prefix << "-something found!\n";
```

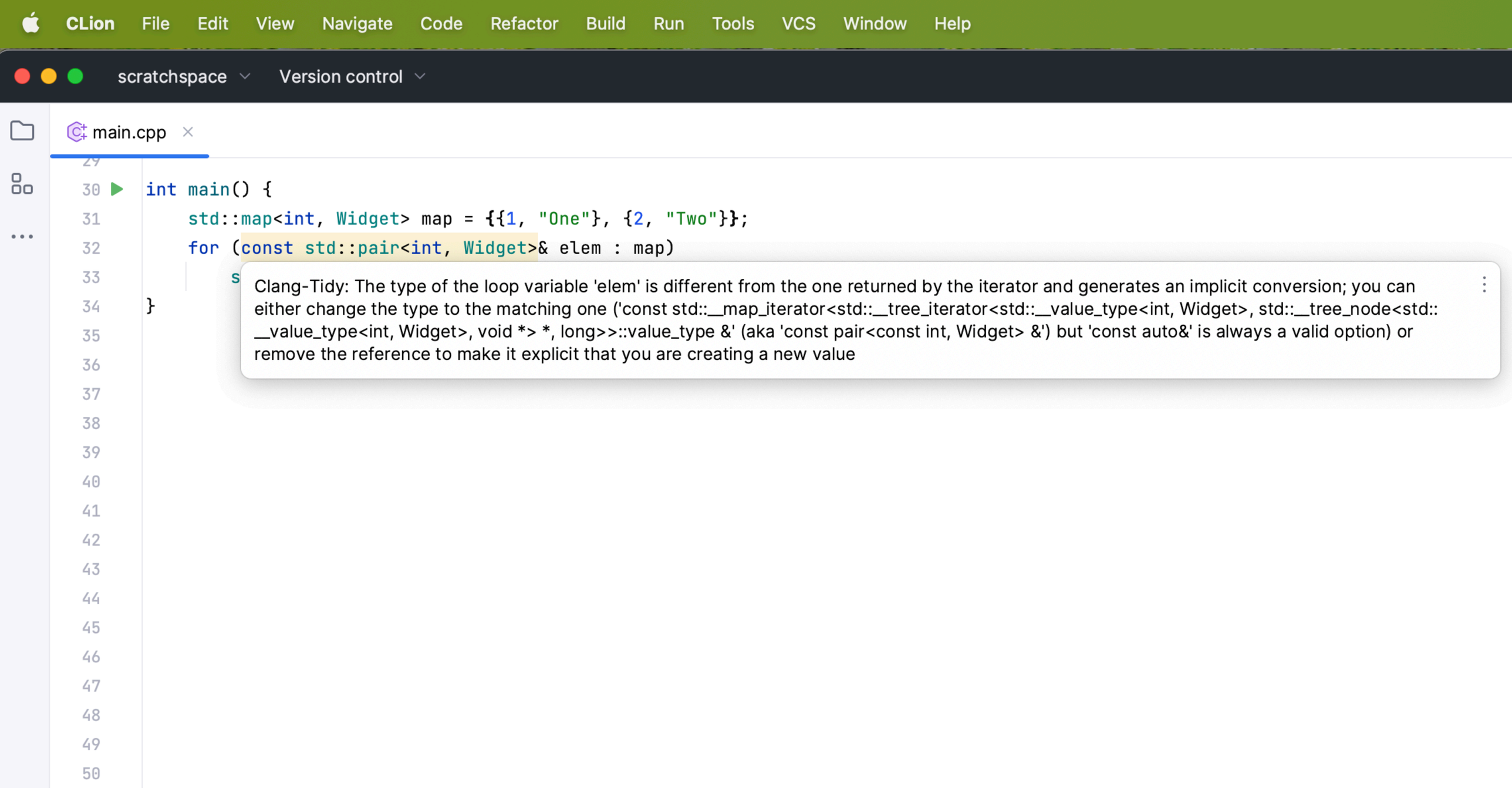
```
std::vector<std::string> v = { /* ... */ };
for (auto s : v)
    do_something(s);
```

```
std::vector<std::string> v = { /* ... */ };
for (auto s : v)
    do_something(s);
```

```
std::vector<std::string> v = { /* ... */ };
for (const auto& s : v)
    do_something(s);
```

```
int main() {
    std::map<int, Widget> map = {{1, "One"}, {2, "Two"}};
    for (const std::pair<int, Widget>& elem : map)
        std::cout << elem.second.str << '\n';
}
```

```
int main() {
    std::map<int, Widget> map = {{1, "One"}, {2, "Two"}};
    for (const std::pair<int, Widget>& elem : map)
        std::cout << elem.second.str << '\n';
}
```


CLion File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
scratchspace Version control
main.cpp
int main() {
std::map<int, Widget> map = {{1, "One"}, {2, "Two"}};
for (const std::pair<int, Widget>& elem : map)
}
Clang-Tidy: The type of the loop variable 'elem' is different from the one returned by the iterator and generates an implicit conversion; you can either change the type to the matching one ('const std::__map_iterator<std::__tree_iterator<std::__value_type<int, Widget>, std::__tree_node<std::__value_type<int, Widget>, void *> *, long>>::value_type &' (aka 'const pair<const int, Widget> &') but 'const auto&' is always a valid option) or remove the reference to make it explicit that you are creating a new value

Avoid unnecessary work

- Avoid unnecessary copies
- Avoid unnecessary function calls / indirections
  - inline functions
  - prefer std::variant / CRTP / "deducing `this`" (since C++23) over virtual functions

Avoid unnecessary work

- Avoid unnecessary copies
- Avoid unnecessary function calls / indirections
  - inline functions
  - prefer std::variant / CRTP / "deducing `this`" (since C++23) over virtual functions
- Make as many decisions as possible at compile time
  - `constexpr` all the things
  - Template metaprogramming
  - Generate lookup tables at compile time

Use algorithms optimised for speed

- Use efficient mathematical operations
  - Fast approximations

```
float Q_rsqrt(float number)
{
    int i;
    float x2, y;
    const float threehalfs = 1.5f;

    x2 = number * 0.5f;
    y  = number;

    i  = *(int*) &y;
    i  = 0x5f3759df - (i >> 1);

    y  = *(float *) &i;
    y  = y * (threehalfs - ( x2 * y * y ));

    return y;
}
```

```
float Q_rsqrt(float number)
{
    int i;
    float x2, y;
    const float threehalfs = 1.5f;

    x2 = number * 0.5f;
    y  = number;

    i  = *(int*) &y;
    i  = 0x5f3759df - (i >> 1);

    y  = *(float *) &i;
    y  = y * (threehalfs - ( x2 * y * y ));

    return y;
}
```

```
constexpr float Q_rsqrt(float y)
{
    constexpr auto threehalfs = 1.5f;
    constexpr auto x2 = y * 0.5f;

    auto i = std::bit_cast<int>(y);
    i  = 0x5f3759df - (i >> 1);

    y = std::bit_cast<float>(i);
    y  = y * (threehalfs - ( x2 * y * y ));

    return y;
}
```

```
constexpr float Q_rsqrt(float y)
{
    constexpr auto threehalfs = 1.5f;
    constexpr auto x2 = y * 0.5f;

    auto i = std::bit_cast<int>(y);
    i  = 0x5f3759df - (i >> 1);

    y = std::bit_cast<float>(i);
    y  = y * (threehalfs - ( x2 * y * y ));

    return y;
}
```

Timur Doumler: "Type punning in modern C++" (2019)

Low-level bit manipulation in C++

- Object lifetime rules
- Aliasing rules
- Alignment rules
- Object representations
- Value representations
- `std::bit_cast` (since C++20)
- Implicit-lifetime types (since C++20)
- `std::start_lifetime_as` (since C++23)

Use algorithms optimised for speed

- Use efficient mathematical operations
  - Fast approximations
  - Use powers of two for sizes

    (compiler can replace division/mod with bit shifts)
  - Lookup tables
  - Many other techniques

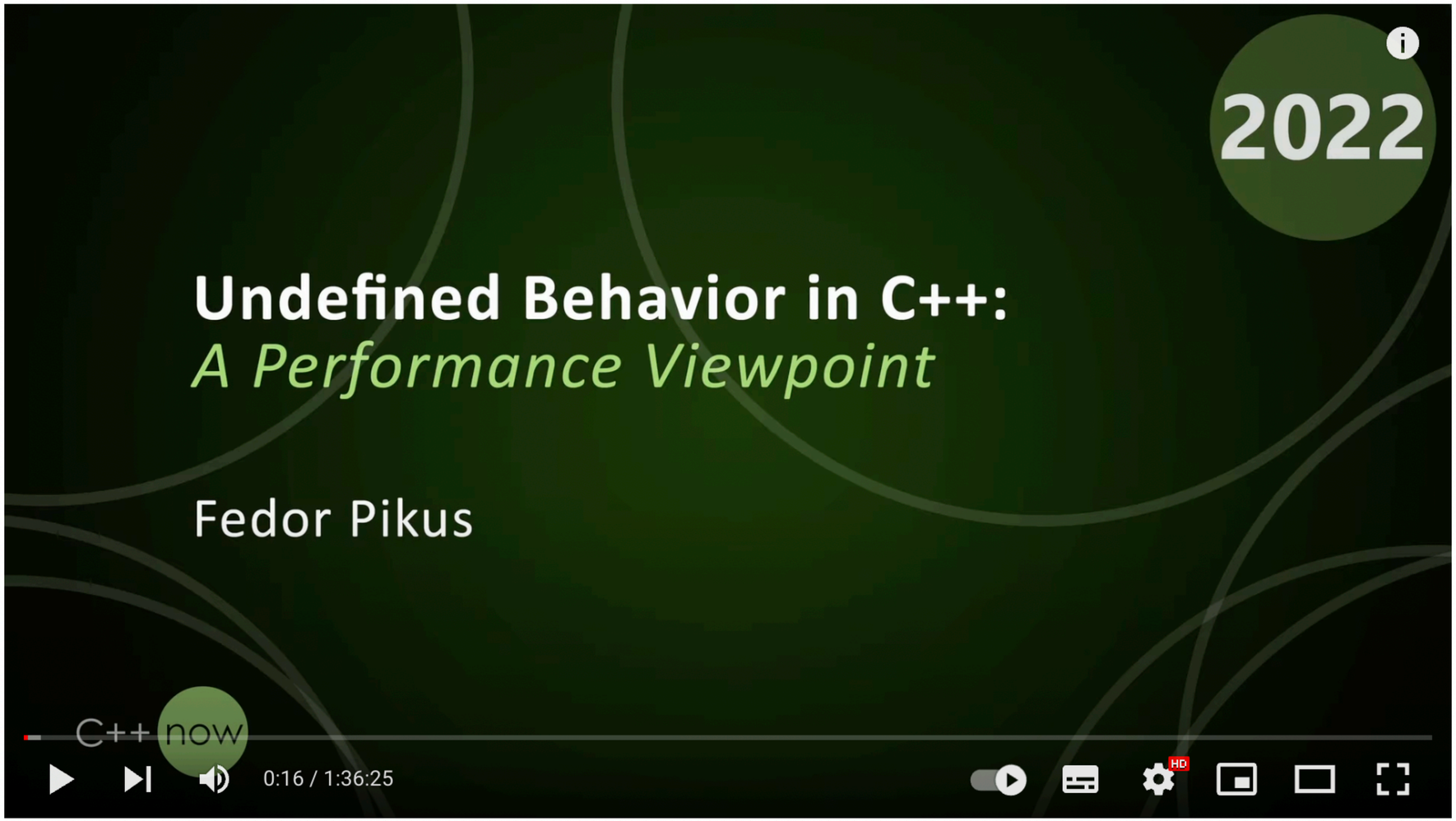
Know your optimiser

Know your optimiser

The optimiser & undefined behaviour

- memory-related UB
  - type system violation
  - out-of-bounds
  - lifetime violation (dangling pointers, use-after-free)
  - uninitialised variables
- data races
- signed integer overflow
- infinite loops with no side effects

The optimiser & undefined behaviour



The optimiser & undefined behaviour

- `std::assume_aligned` *(C++20)*
- `[[assume]]` *(C++23)*

Portable assumptions

Timur Doumler (papers@timur.audio)

Document #: P1774R8
Date: 2022-06-14
Project: Programming Language C++
Audience: Core Working Group

Abstract

We propose a standard facility providing the semantics of existing compiler built-ins such as `__builtin_assume` (Clang) and `__assume` (MSVC, ICC). It gives the programmer a way to allow the compiler to assume that a given C++ expression is true, without evaluating it, and to optimise based on this assumption. This is very useful for high-performance and low-latency applications in order to generate both faster and smaller code.

```
int divide_by_32(int x) {
    return x/32;
}
```

```
int divide_by_32(int x) {
    return x/32;
}
```

```
mov eax, edi
sar eax, 31
shr eax, 27
add eax, edi
sar eax, 5
ret
```

```
int divide_by_32(int x) {
    return x/32;
}
```

```
mov eax, edi
sar eax, 31
shr eax, 27
add eax, edi
sar eax, 5
ret
```

```
int divide_by_32(int x) {
    [[assume(x >= 0)]];
    return x/32;
}
```

```
mov eax, edi
shr eax, 5
ret
```

```
void limiter(float* data, size_t size) {
    for (size_t i = 0; i < size; ++i {
        data[i] = std::clamp(data[i], -1.0f, 1.0f);
    }
}
```

```
void limiter(float* data, size_t size) {
    [[assume(size > 0)]];
    [[assume(size % 32 == 0)]];

    for (size_t i = 0; i < size; ++i {
        [[assume(std::isfinite(data[i]))]];
        data[i] = std::clamp(data[i], -1.0f, 1.0f);
    }
}
```

```
void limiter(float* data, size_t size) {
    [[assume(size > 0)]];
    [[assume(size % 32 == 0)]];

    for (size_t i = 0; i < size; ++i {
        [[assume(std::isfinite(data[i]))]];
        data[i] = std::clamp(data[i], -1.0f, 1.0f);
    }
}
```

https://wg21.link/p1774

Document: P2646R0
Date: 2022-OCT-15
Project: Programming Language C++
Audience: EWG
Reply-to: Parsa Amini: me@parsaamini.net
Joshua Berne: jberne4@bloomberg.net
John Lakos: jlakos@bloomberg.net

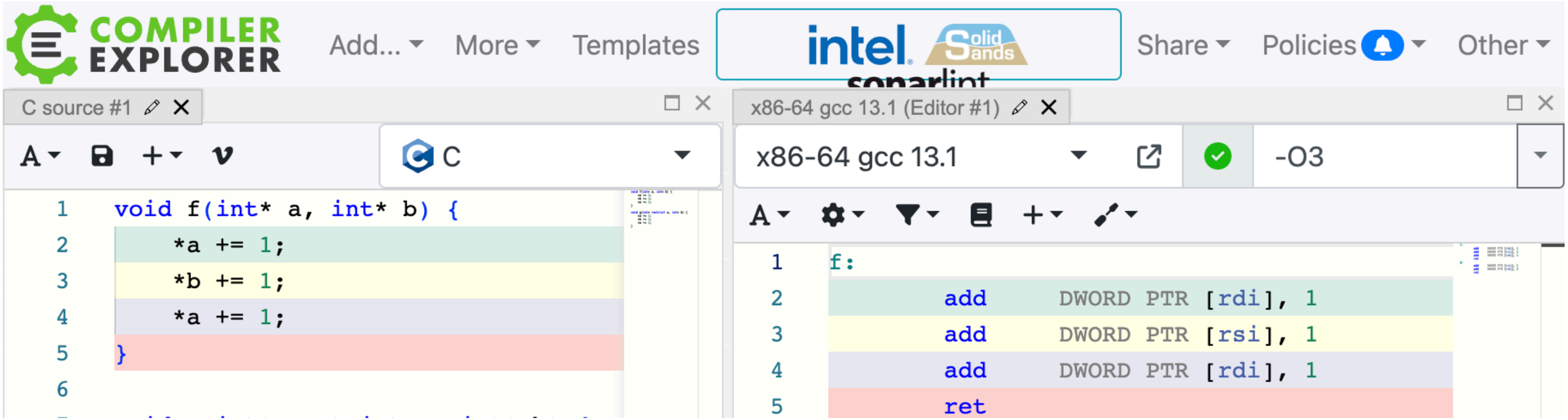
Explicit Assumption Syntax *Can* Reduce *Run Time*

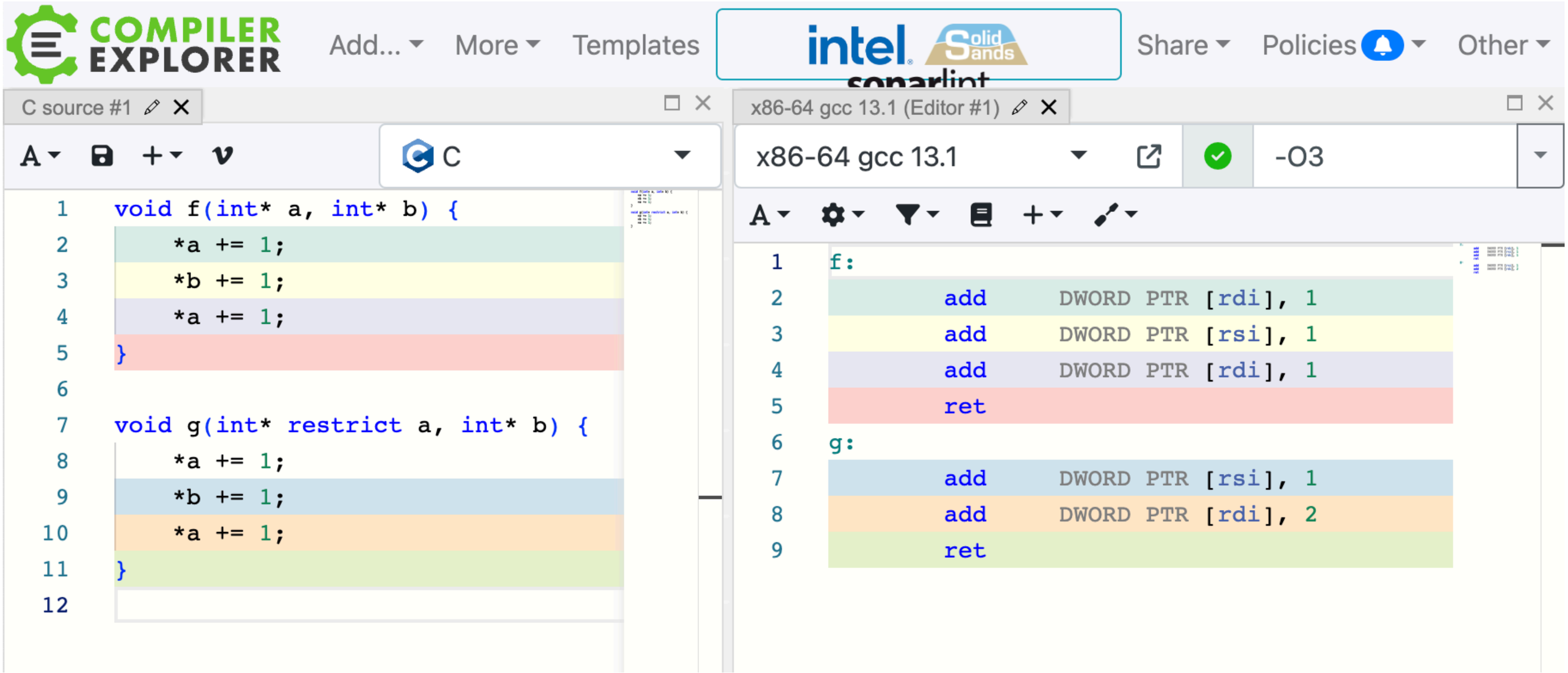
Abstract

Many compilers provide platform-specific *assumption* syntax, such as `__builtin_assume` in Clang or idiomatic use of `__builtin_unreachable()` in GCC. This augmented syntax can then indicate to the compiler that it is allowed but not required to assume that some condition — typically a Boolean-valued expression — is always true. Recently, after due consideration, the `[[assume]]` attribute was formally adopted into the C++ working draft (P1774R8) to provide a facility for expressing such assumptions *portably* in source code. As is well known and easily demonstrated, the use of such *compiler-accessible* assumption constructs can noticeably affect compile times as well as object-code and overall program sizes. On the other hand, some members of the C++ Standards Committee have suggested (wrongly) that modern compilers and CPUs conspire to realize essentially all runtime performance benefits available on modern architectures, thereby obviating use of explicit assumption constructs in source code.

The optimiser & undefined behaviour

- `std::assume_aligned` *(C++20)*
- `[[assume]]` *(C++23)*


COMPILER EXPLORER
Add...
More
Templates
intel
Solid Sands
sonarlint
Share
Policies
Other
C source #1
C
x86-64 gcc 13.1 (Editor #1)
x86-64 gcc 13.1
-O3
1 void f(int* a, int* b) {
2 *a += 1;
3 *b += 1;
4 *a += 1;
5 }
6
1 f:
2 add DWORD PTR [rdi], 1
3 add DWORD PTR [rsi], 1
4 add DWORD PTR [rdi], 1
5 ret


COMPILER EXPLORER
Add...
More
Templates
intel
Solid Sands
sonarlint
Share
Policies
Other
C source #1
C
x86-64 gcc 13.1 (Editor #1)
x86-64 gcc 13.1
-O3
1 void f(int* a, int* b) {
2 *a += 1;
3 *b += 1;
4 *a += 1;
5 }
6
7 void g(int* restrict a, int* b) {
8 *a += 1;
9 *b += 1;
10 *a += 1;
11 }
12
1 f:
2 add DWORD PTR [rdi], 1
3 add DWORD PTR [rsi], 1
4 add DWORD PTR [rdi], 1
5 ret
6 g:
7 add DWORD PTR [rsi], 1
8 add DWORD PTR [rdi], 2
9 ret

The optimiser & undefined behaviour

- `std::assume_aligned` *(C++20)*
- `[[assume]]` *(C++23)*
- **`[[noalias]]` *(C++26 ??)***

Towards support for attributes in C++ (Revision 6)

Jens Maurer, Michael Wong
jens.maurer@gmx.net
michaelw@ca.ibm.com

Document number: N2761
Date: 2008-09-18
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 6

General Attributes for C++

1 Overview

The idea is to be able to annotate some entities in C++ with additional information. Currently, there is no means to do that short of inventing a new keyword and augmenting the grammar accordingly, thereby reserving yet another name of the user's namespace. This proposal will survey existing industry practice for extending the C++ syntax, and presents a general means for such annotations, including its integration into the C++ grammar. Specific attributes are not introduced in this proposal. It does not obviate the ability to add or overload keywords where appropriate, but it does reduce such need and add an ability to extend the language. This proposal will allow many C++0x proposals to move forward. A draft form of this proposal was presented in Oxford and received

Good choices in attributes include:

- align(unsigned int)
- pure (promise that a function always returns the same value)
- probably(unsigned int) (hint for if, switch, ...)
    - if [[probably(true)]] (i == 42) { ... }
- noreturn (the function never returns)
- deprecated (functions)
- noalias (promises no other path to the object)
- unused (parameter name)
- final on virtual function declaration and on a class
- not_hiding (name of function does not hide something in a base class)
- register (if we had a time machine)
- owner (a pointer is owned and it is the owner's duty to delete it)

Bad choices in attributes include:

- C99 restrict (affects the type system)
- huge (really long long type, e.g. 256bits)
- C++ const

The optimiser & undefined behaviour

- `std::assume_aligned` *(C++20)*
- `[[assume]]` *(C++23)*
- `[[noalias]]` *(C++26 ??)*
- **`[[unsequenced]], [[reproducible]]` *(C23, C++26 ??)***

Unsequenced functions

Étienne Alepins (Thales Canada) and Jens Gustedt (INRIA France)

org:	ISO/IEC JCT1/SC22/WG14	document:	N2887
target:	IS 9899:2023	version:	4
date:	2021-12-31	license:	CC BY

1. Revision history

Paper number	Title	Changes	
N2477	Const functions	Initial version	
N2539	Unsequenced functions	Supersedes N2477	WG14 poll: 15-0-1
			new wording (extracted from N2522)
			no application to the C library
N2825	Unsequenced functions v3	Supersedes N2539	
			no attribute verification imposed
			support for function pointers
			optional text for inclusion of lambdas
N2887	Unsequenced functions v4	Supersedes N2825	refactoring of the properties
			regroup properties in general text
			attach properties to evaluations instead of syntax
			add a sentence to the wording for composite types
			editorial adjustments are collected in a note to the editors at the end
			emphasize on the relationship with existing implementations
			withdraw the special treatment of `call_once`

```
[[reproducible]] ≈ [[gnu::pure]]
[[unsequenced]] ≈ [[gnu::const]]
```

Sub-properties of reproducible & unsequenced

- **stateless:** function that does not define mutable static or thread-local objects (nor do functions that are called by it)
- **effectless:** function that does not have observable side effects
- **idempotent:** repeated evaluation gives the same result (hence may read global state)
- **independent:** does not depend on other state than the arguments or constants (hence may write to globals)

Sub-properties of reproducible & unsequenced

- **stateless:** function that does not define mutable static or thread-local objects (nor do functions that are called by it)
- **effectless:** function that does not have observable side effects
- **idempotent:** repeated evaluation gives the same result (hence may read global state)
- **independent:** does not depend on other state than the arguments or constants (hence may write to globals)
- **reproducible:** effectless and idempotent
- **unsequenced**: stateless, effectless, idempotent, and independent

```
double cos(double x);

if ((cos(angle1) > cos(angle2)) || (cos(angle1) > cos(angle3))) {
    ...
}
```

```
double cos(double x) [[unsequenced]];

if ((cos(angle1) > cos(angle2)) || (cos(angle1) > cos(angle3))) {
    ...
}
```

```
double cos(double x) [[unsequenced]];

if ((cos(angle1) > cos(angle2)) || (cos(angle1) > cos(angle3))) {
    ...
}
```

Writing efficient code requires...

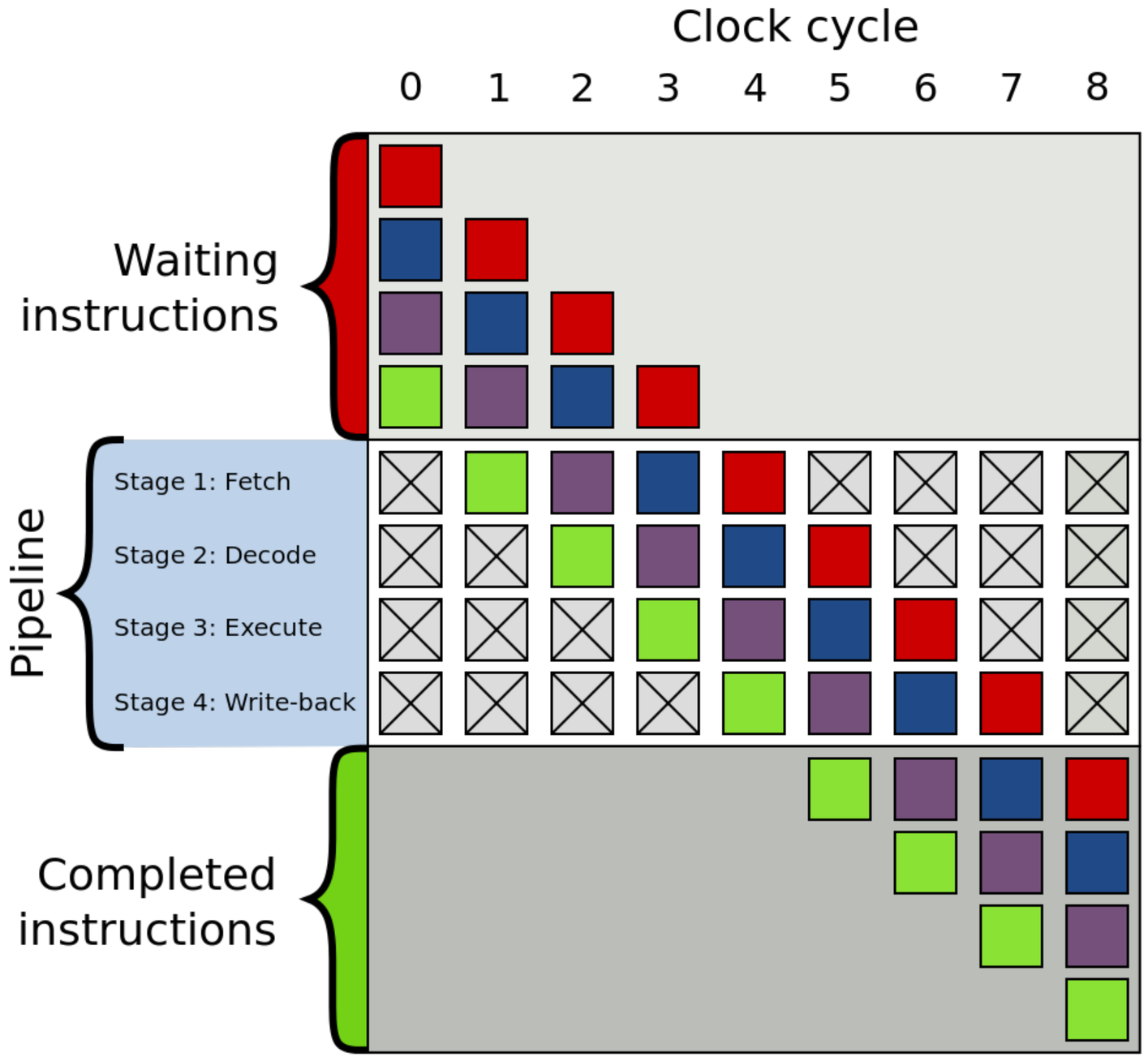
- Knowledge of the programming language
- Knowledge of the libraries used
- Knowledge of the compiler
  - Optimiser
  - ABI (Itanium, Microsoft, ...)
- Knowledge of the hardware architecture
  - CPU architecture (Instruction set, pipeline, SIMD...)
  - Cache hierarchy (Registers, L1/2/3 cache)
  - Prefetcher, translation lookaside buffer
  - Branch predictor, branch target buffer

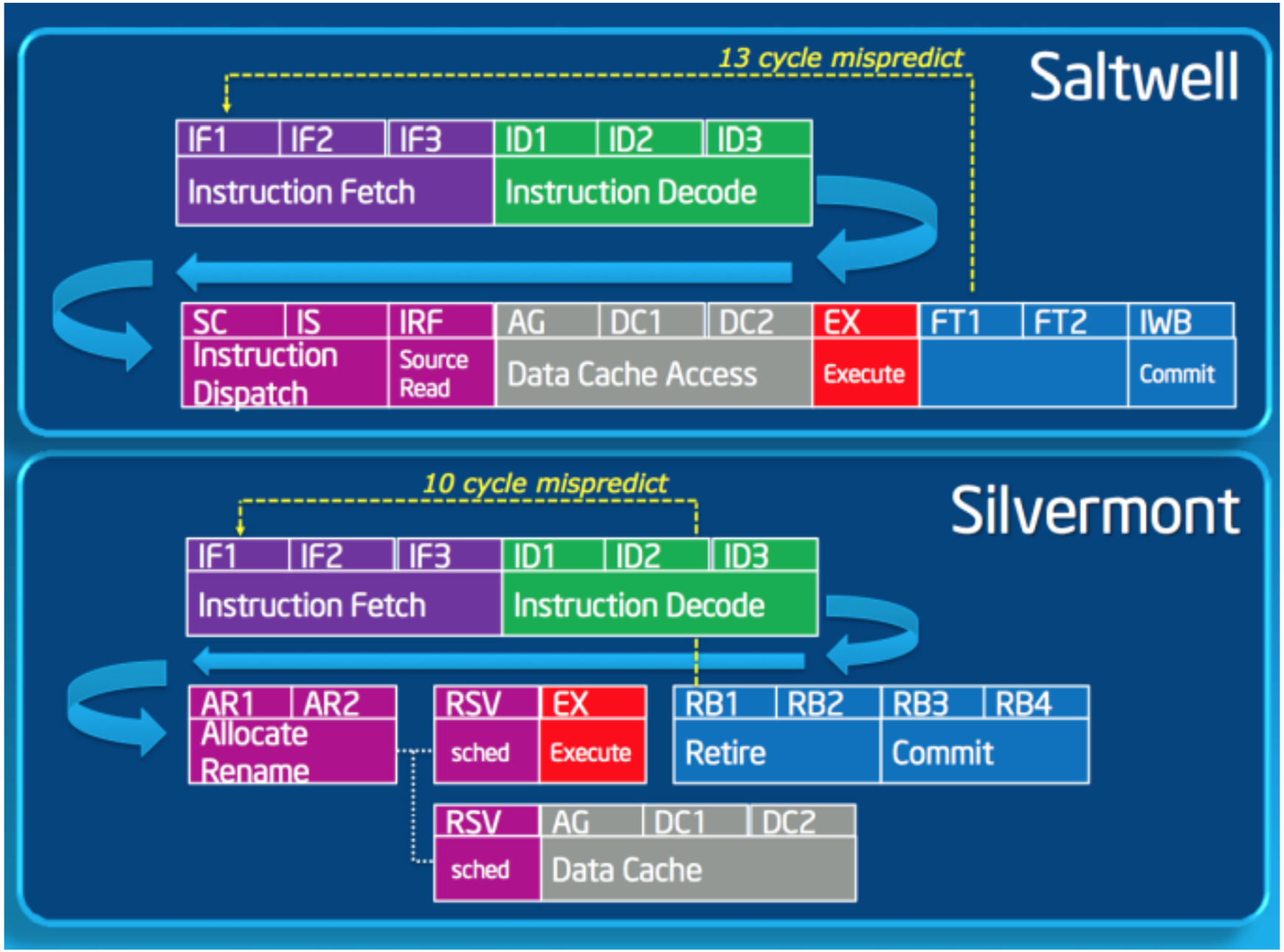
Writing efficient code requires...

- Knowledge of the programming language
- Knowledge of the libraries used
- Knowledge of the compiler
  - Optimiser
  - ABI (Itanium, Microsoft, ...)
- Knowledge of the hardware architecture
  - CPU architecture (Instruction set, pipeline, SIMD...)
  - Cache hierarchy (Registers, L1/2/3 cache)
  - Prefetcher, translation lookaside buffer
  - Branch predictor, branch target buffer

Knowledge of the hardware architecture

- x86, x86_64: *Agner Fog's optimisation manuals*
- ARM: ???


Clock cycle
0
1
2
3
4
5
6
7
8
Waiting instructions
Pipeline
Stage 1: Fetch
Stage 2: Decode
Stage 3: Execute
Stage 4: Write-back
Completed instructions


Saltwell
13 cycle mispredict
IF1
IF2
IF3
ID1
ID2
ID3
Instruction Fetch
Instruction Decode
SC
IS
IRF
AG
DC1
DC2
EX
FT1
FT2
IWB
Instruction Dispatch
Source Read
Data Cache Access
Execute
Commit
Silvermont
10 cycle mispredict
IF1
IF2
IF3
ID1
ID2
ID3
Instruction Fetch
Instruction Decode
AR1
AR2
Allocate Rename
RSV
EX
sched
Execute
RB1
RB2
RB3
RB4
Retire
Commit
RSV
AG
DC1
DC2
sched
Data Cache

CPU pipeline hazards

- Branch hazard
- Data hazard
- Hardware hazard

CPU pipeline hazards

- **Branch hazard**
- Data hazard
- Hardware hazard

Avoid branch mispredicts

```
std::vector<float> v (32768);

std::generate (v.begin(), v.end(),
               [] { return (rand() % 2) ? 1 : -1; });

return std::count_if (v.begin(), v.end(),
                      [] (float x) { return x > 0; });
```

Avoid branch mispredicts

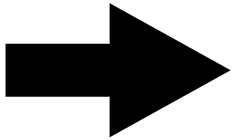
```
std::vector<float> v (32768);

std::generate (v.begin(), v.end(),
               [] { return (rand() % 2) ? 1 : -1; });

std::sort (v.begin(), v.end());

return std::count_if (v.begin(), v.end(),
                      [] (float x) { return x > 0; });
```

Branchless programming

Branchless programming

Branchless programming

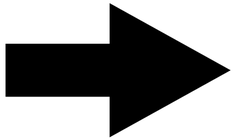
```
if (x || y) do_this();
else do_that();
```



```
if (bool(x) + bool(y)) do_this();
else do_that();
```

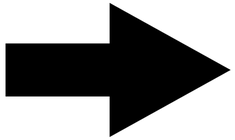
Branchless programming

```
if (x || y) do_this();
else do_that();
```



```
if (bool(x) + bool(y)) do_this();
else do_that();
```

```
sum += cond ? expr1 : expr2;
```



```
bool expr[2] = { expr1, expr2 };
sum += expr[bool(cond)];
```

[[likely]] and [[unlikely]]

```
auto order = get_order();
if (order.has_value()) {
    process_order(order.get_value());
}
else {
    fallback();
}
```

[[likely]] and [[unlikely]]

```
auto order = get_order();
if (order.has_value()) [[likely]] {
    process_order(order.get_value());
}
else {
    fallback();
}
```

[[likely]] and [[unlikely]]

- Does not affect branch predictor!
- Can affect code layout
- Have various of pitfalls
  - *Aaron Ballman: "Don’t use the [[likely]] or [[unlikely]] attributes"*
  - *Amir Kirsh & Tomer Vromen:"C++20 Likely and Unlikely:*
    A Journey Through Branch Prediction and Compiler Optimizations" (2022)

CPU pipeline hazards

- Branch hazard
- **Data hazard**
- Hardware hazard

Avoid data hazards

```
unsigned atoui (const char* c)
{
    unsigned result = 0;

    while (*c >= '0' && *c <= '9')
        result = result * 10 + (*c - '0');

    return result;
}
```

Avoid data hazards

```
unsigned atoui (const char* c)
{
    unsigned result = 0;

    while (*c >= '0' && *c <= '9')
        result = result * 10 + (*c - '0');

    return result;
}
```

Avoid data hazards

```
unsigned atoui (const char* c)
{
    unsigned result = 0;

    while (*c >= '0' && *c <= '9')
        result = result * 10 + (*c - '0');

    return result;
}
```

Andrei Alexandrescu: "Writing fast code" (2015)

CPU pipeline hazards

- Branch hazard
- Data hazard
- **Hardware hazard**

CPU pipeline hazards

- Branch hazard
- Data hazard
- **Hardware hazard**
  - Limited amount of adders/shifters

CPU pipeline hazards

- Branch hazard
- Data hazard
- **Hardware hazard**
  - Limited amount of adders/shifters
  - Limited amount of load/store units

    (latest Intel CPUs: 3 loads + 2 stores per cycle)

    → sometimes you can replace loads by shifts

    → increase bandwidth of every load using SIMD

CPU pipeline hazards

- Branch hazard
- Data hazard
- **Hardware hazard**
  - Limited amount of adders/shifters
  - Limited amount of load/store units
    (latest Intel CPUs: 3 loads + 2 stores per cycle)
    → sometimes you can replace loads by shifts
    → increase bandwidth of every load using SIMD
  - Hardware bugs

1607

I was looking for the fastest way to `popcount` large arrays of data. I encountered a *very weird* effect: Changing the loop variable from `unsigned` to `uint64_t` made the performance drop by 50% on my PC.

The Benchmark

```
uint64_t count,duration;
chrono::time_point<chrono::system_clock> startP,endP;
{
    startP = chrono::system_clock::now();
    count = 0;
    for( unsigned k = 0; k < 10000; k++){
        // Tight unrolled loop with unsigned
        for (unsigned i=0; i<size/8; i+=4) {
            count += _mm_popcnt_u64(buffer[i]);
            count += _mm_popcnt_u64(buffer[i+1]);
            count += _mm_popcnt_u64(buffer[i+2]);
            count += _mm_popcnt_u64(buffer[i+3]);
        }
    }
```

Culprit: False Data Dependency (and the compiler isn't even aware of it)

1723

On Sandy/Ivy Bridge and Haswell processors, the instruction:

```
popcnt  src, dest
```

appears to have a false dependency on the destination register `dest`. Even though the instruction only writes to it, the instruction will wait until `dest` is ready before executing. This false dependency is (now) documented by Intel as erratum HSD146 (Haswell) and SKL029 (Skylake)

Skylake fixed this for `lzcnt` and `tzcnt`.
Cannon Lake (and Ice Lake) fixed this for `popcnt`.
`bsf`/`bsr` have a true output dependency: output unmodified for input=0. (But no way to take advantage of that with intrinsics - only AMD documents it and compilers don't expose it.)

(Yes, these instructions all run on the same execution unit).

This dependency doesn't just hold up the 4 `popcnt`s from a single loop iteration. It can carry across loop iterations making it impossible for the processor to parallelize different loop iterations.

SIMD

- "Single instruction, multiple data"
  - CPU-specific: MMX, SSE 1/2/3/4, AVX, AVX2, AVX-512, AMX, NEON, SVE…
- How to use?
  - Auto-vectorisation
  - Explicit vectorisation using SIMD libraries
    - Google Highway, xsimd, vectorclass, eve, std::simd proposal for C++26 (P1928)
      → Jeff Garland: *"SIMD Libraries in C++" (CppNow 2023)*
  - Writing intrinsics
  - Writing assembly
- SWAR ("SIMD Within A Register")

Autovectorisation

```
void mult(float* x, int size, float factor) {
  for (int i = 0; i < size; ++i)
    x[i] *= factor;
}
```

```
1  void mult(float* x, int size, float factor) {
2    for (int i = 0; i < size; ++i)
3      x[i] *= factor;
4  }
```

A ▾ Output... ▾ Filter... ▾ Libraries + Add new... ▾ Add tool... ▾

```
1  mult(float*, int, float):
2          mov     rcx, rdi
3          test    esi, esi
4          jle     .L1
5          lea     eax, [rsi-1]
6          cmp     eax, 2
7          jbe     .L8
8          mov     edx, esi
9          movaps  xmm2, xmm0
10         mov     rax, rdi
11         shr     edx, 2
12         shufps  xmm2, xmm2, 0
13         sal     rdx, 4
14         add     rdx, rdi
15 .L4:
16         movups  xmm1, XMMWORD PTR [rax]
17         add     rax, 16
18         mulps   xmm1, xmm2
19         movups  XMMWORD PTR [rax-16], xmm1
20         cmp     rdx, rax
21         jne     .L4
22         mov     eax, esi
23         and     eax, -4
24         mov     edx, eax
25         cmp     esi, eax
26         je      .L17
27 .L3:
28         sub     esi, edx
29         cmp     esi, 1
30         je      .L6
31         lea     rdx, [rcx+rdx*4]
32         movaps  xmm2, xmm0
33         movq    xmm1, QWORD PTR [rdx]
34         shufps  xmm2, xmm2, 0xe0
35         mulps   xmm1, xmm2
36         movlps  QWORD PTR [rdx], xmm1
37         test    sil, 1
38         je      .L1
39         and     esi, -2
40         add     eax, esi
41 .L6:
42         cdqe
43         lea     rax, [rcx+rax*4]
44         mulss   xmm0, DWORD PTR [rax]
45         movss   DWORD PTR [rax], xmm0
46 .L1:
47         ret
48 .L17:
```

Autovectorisation

- Highly dependent on compiler
- Only works with vectors/arrays of int, char, double etc. - no structs
  - Workaround: struct of arrays instead of array of structs
- Traverse data linearly
- for loop, not while loop
- Number of iterations predetermined (ideally, known at compile time)
- No data-dependent break, goto, etc.
- No conditionals
- No data dependencies between array elements
- No aliasing

```
int a[1000], b[1000], c[1000];
// fill with some values...

for (int i = 0; i < 1000; ++i) {
    a[i]   += b[i];
    b[i+1] += c[i];
}
```

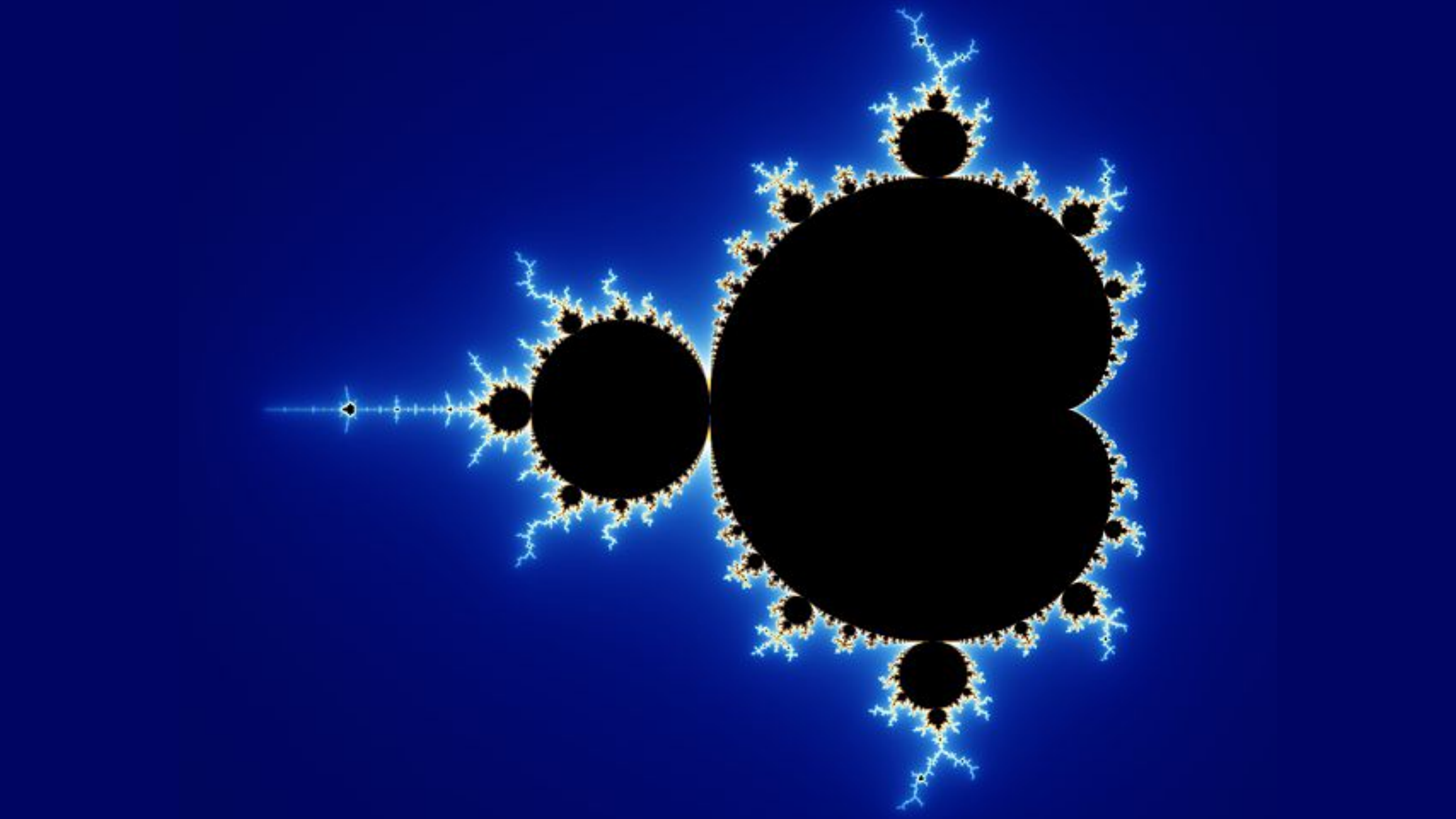
```
int a[1000], b[1000], c[1000];
// fill with some values...

for (int i = 0; i < 1000; ++i) {
    a[i]   += b[i];
    b[i+1] += c[i];
}
```

```
int a[1000], b[1000], c[1000];
// fill with some values...

a[0] +=  b[0];
for (int i = 1; i < 999; ++i) {
    b[i+1] += c[i];
    a[i+1] += b[i+1];
}
b[999] += c[999];
```

```
## non-vectorised -> slow :(
LBB0_1:
  addl  %ecx, 4(%rdi,%rax,4)
  movl  8(%rsi,%rax,4), %ecx
  addl  4(%rdx,%rax,4), %ecx
  movl  %ecx, 8(%rsi,%rax,4)
  addl  %ecx, 8(%rdi,%rax,4)
  movl  12(%rsi,%rax,4), %ecx
  addl  8(%rdx,%rax,4), %ecx
  movl  %ecx, 12(%rsi,%rax,4)
  addq  $2, %rax
  cmpq  $1000, %rax
  jne   LBB0_1
```

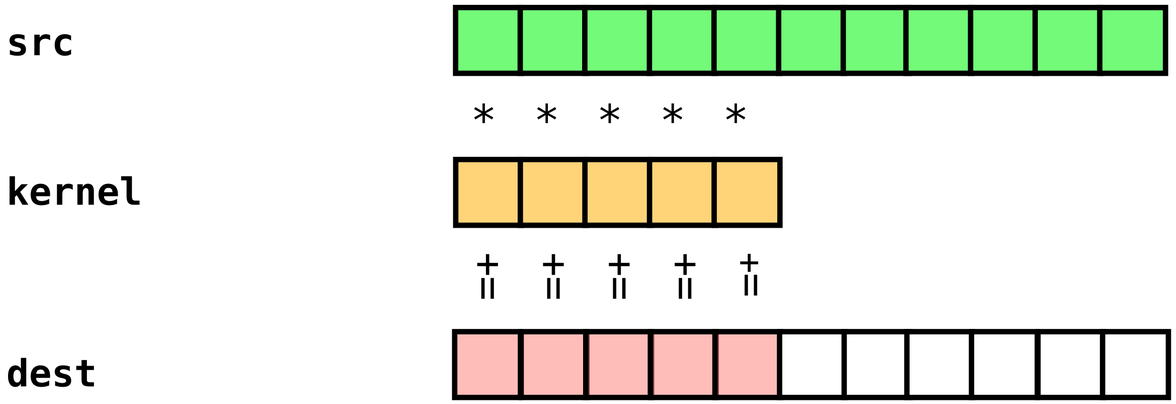
```
## vectorised -> faster :)
LBB0_4:
  movdqu   4(%rdx,%rcx,4), %xmm0
  movdqu   20(%rdx,%rcx,4), %xmm1
  movdqu   8(%rsi,%rcx,4), %xmm2
  movdqu   24(%rsi,%rcx,4), %xmm3
  paddd %xmm0, %xmm2
  paddd %xmm1, %xmm3
  movdqu   %xmm2, 8(%rsi,%rcx,4)
  movdqu   %xmm3, 24(%rsi,%rcx,4)
  movdqu   8(%rdi,%rcx,4), %xmm0
  movdqu   24(%rdi,%rcx,4), %xmm1
  paddd %xmm2, %xmm0
  paddd %xmm3, %xmm1
  movdqu   %xmm0, 8(%rdi,%rcx,4)
  movdqu   %xmm1, 24(%rdi,%rcx,4)
  addq  $8, %rcx
  cmpq  $992, %rcx
  jne   LBB0_4
```

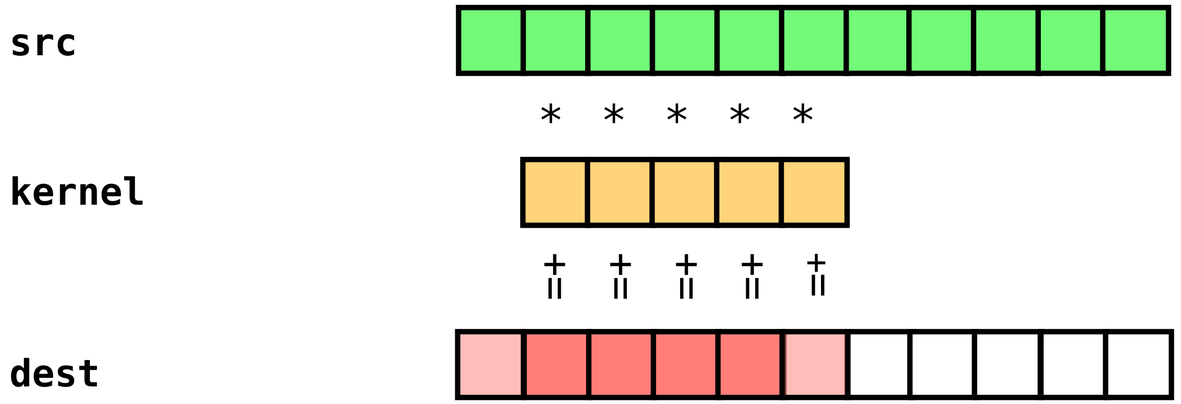
Explicit vectorisation libraries

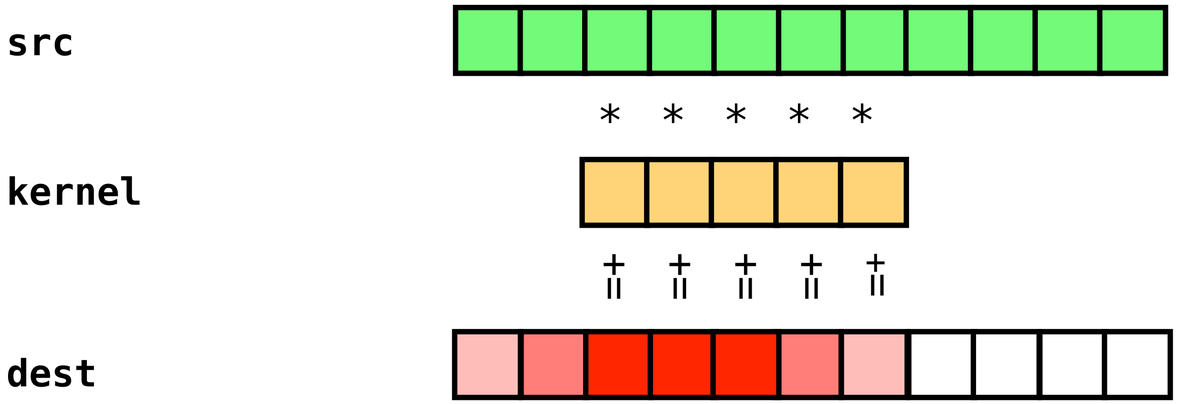


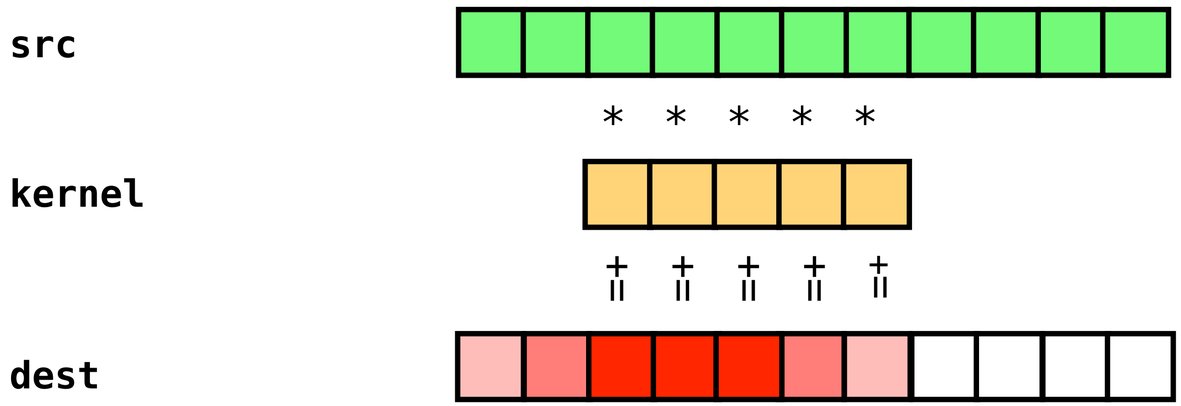
```
int mandelbrot(double real, double imag) {
   int limit = 100;
   double zReal = real;
   double zImag = imag;

   for (int i = 0; i < limit; ++i) {
      double r2 = zReal * zReal;
      double i2 = zImag * zImag;

      if (r2 + i2 > 4.0)
         return i;

      zImag = 2.0 * zReal * zImag + imag;
      zReal = r2 - i2 + real;
   }
   return limit;
}
```

```
int mandelbrot(double real, double imag) {
    int limit = 100;
    double zReal = real;
    double zImag = imag;

    for (int i = 0; i < limit; ++i) {
        double r2 = zReal * zReal;
        double i2 = zImag * zImag;

        if (r2 + i2 > 4.0)
            return i;

        zImag = 2.0 * zReal * zImag + imag;
        zReal = r2 - i2 + real;
    }
    return limit;
}
```

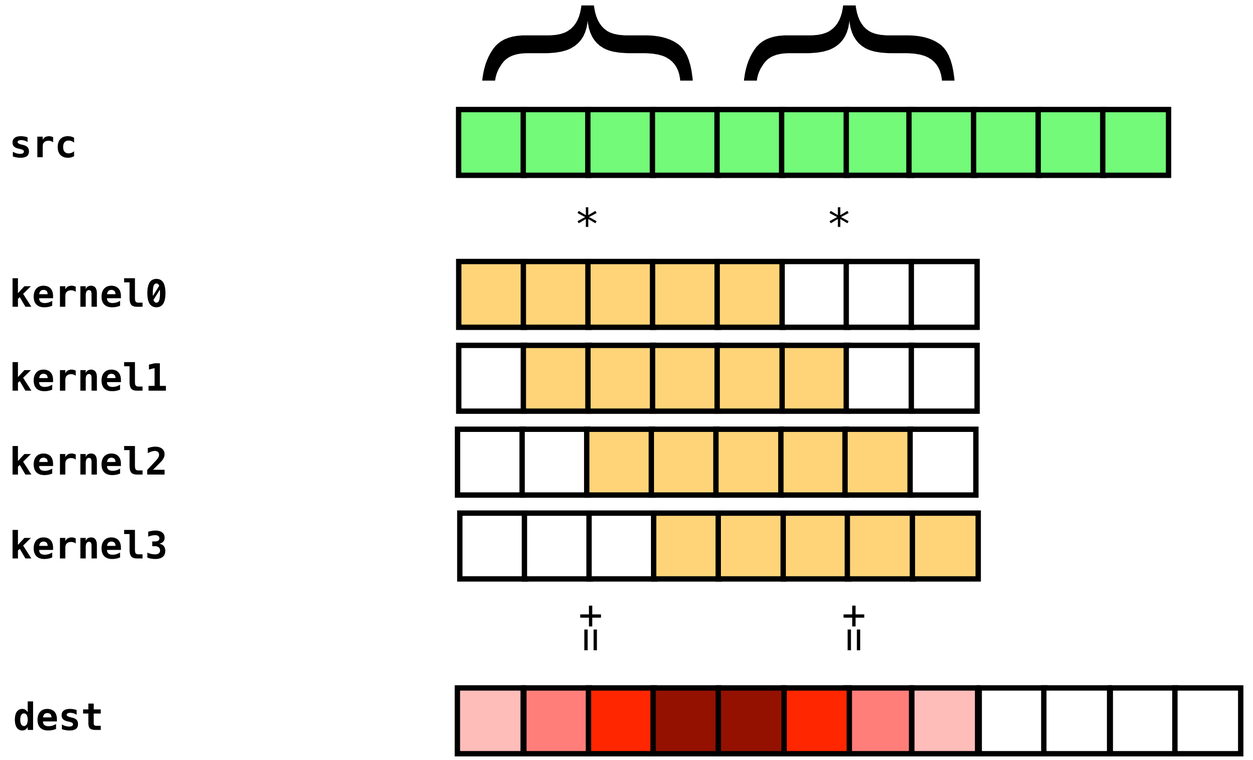

src
kernel
dest

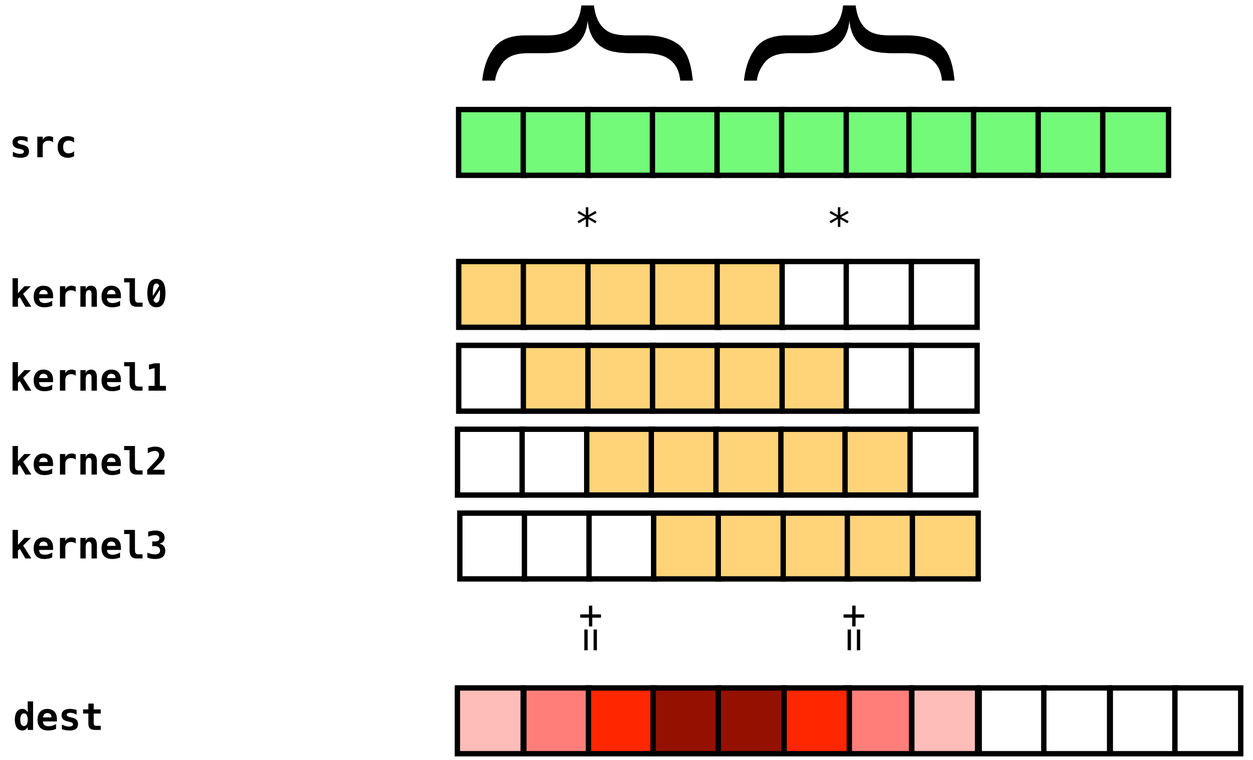

src
kernel
dest


src
kernel
dest




```
for (int i = 0; i < src.size() - kernel.size(); ++i) {
   float sum = 0.0f;
   for (int j = 0; j < kernel.size(); ++j)
      sum += src [i + j] * kernel[j];

   dest[i] = sum;
}
```


src
kernel0
kernel1
kernel2
kernel3
dest



Jeff Garland: *"SIMD Libraries in C++" (CppNow 2023)*

Other SIMD considerations

- If you know the exact target CPU:
  - use arch compiler flags
- If you don't:
  - dynamic dispatch
  - function multiversioning (GCC/Clang only)

```
// Function multiversioning (GCC/Clang only):

__attribute__ ((target ("default"))) int foo () {
  // The default version of foo.
  return 0;
}

__attribute__ ((target ("sse4.2"))) int foo () {
  // foo version for SSE4.2
  return 1;
}

__attribute__ ((target ("arch=atom"))) int foo () {
  // foo version for the Intel ATOM processor
  return 2;
}

__attribute__ ((target ("arch=amdfam10"))) int foo () {
  // foo version for the AMD Family 0x10 processors.
  return 3;
}

int main () {
  int (*p)() = &foo;
  assert((*p)() == foo());
  return 0;
}
```

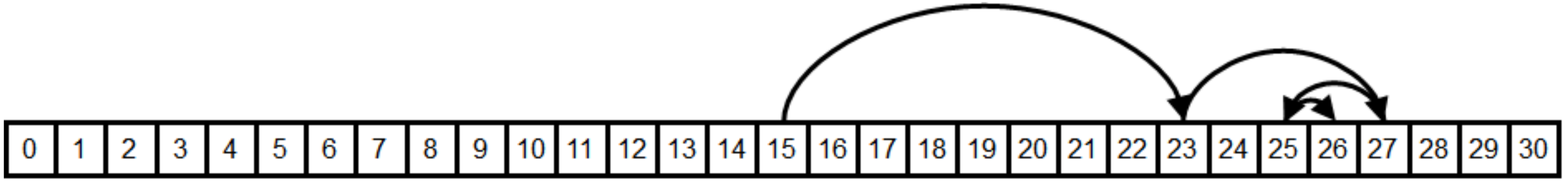
The cache pyramid

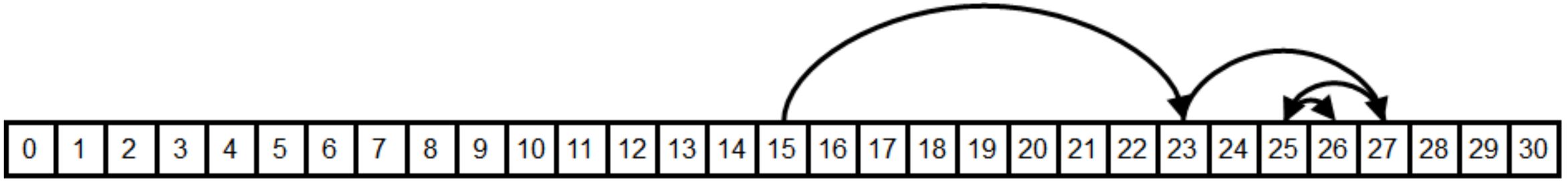
- Registers
- L1 cache (per core)
- L2 cache (shared)
- L3 cache (shared)
- Main memory

Minimise cache misses!

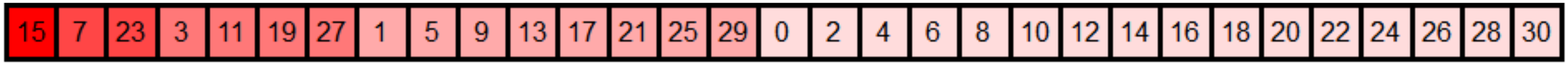
Minimise data cache misses

- Data locality
- Align data on cachelines
- Concurrency aspect (true/false sharing)
- Contiguous data traversal (also good for prefetcher)
- "Almost always vector"
- Cache-friendly associative containers
  - `std::flat_set/std::flat_map` (C++23), Abseil containers
- Cache-friendly algorithms
  - Cache-friendly binary search


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30


15 7 23 3 11 19 27 1 5 9 13 17 21 25 29 0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30

Minimise data cache misses

- Data locality
- Align data on cachelines
- Concurrency aspect (true/false sharing)
- Contiguous data traversal (also good for prefetcher)
- "Almost always vector"
- Cache-friendly associative containers
  - `std::flat_set`/`std::flat_map` (C++23), Abseil containers
- Cache-friendly algorithms
  - Cache-friendly binary search
- *Eduardo Madrid: Rehashing Hash Tables And Associative Containers (C++Now 2022)*

Minimise instruction cache misses

- Consider generated code layout & alignment
- Avoid branches
- Avoid virtual functions
  - std::variant
  - Compile-time polymorphism
    - CRTP, mixins, "deducing `this`" (since C++23)

Keep the cache warm

- Data cache
  - Periodically poke data on a timer
- Instruction cache
  - Periodically run the hot path with dummy input/output

Minimise cache misses

- *Avi Lachmish: "Algorithmic Complexity, Data Locality, Parallelism, and Compiler Optimizations, Seasoned with Some Concurrency – A Deep Dive into Performance" (2022)*
- *Björn Fahller: "What do you mean by cache friendly?" (2020)*
- *Timur Doumler: "Want fast C++? Know your hardware!" (2016)*
- *Scott Meyers: "CPU Caches and why you care" (2014)*

Other fun hardware problems

- **CPU throttling**
  - Due to overheating

    → spread thermal dissipation

Other fun hardware problems

- **CPU throttling**
  - Due to overheating
    → spread thermal dissipation
  - Due to low activity
    → keep CPU busy with pause instructions

C++ techniques for low-latency programming

- Two categories:
  - **Efficient programming**
  - Programming for deterministic execution time

- Most crucial thing: measuring!

C++ techniques for low-latency programming

- Two categories:
  - Efficient programming
  - **Programming for deterministic execution time**

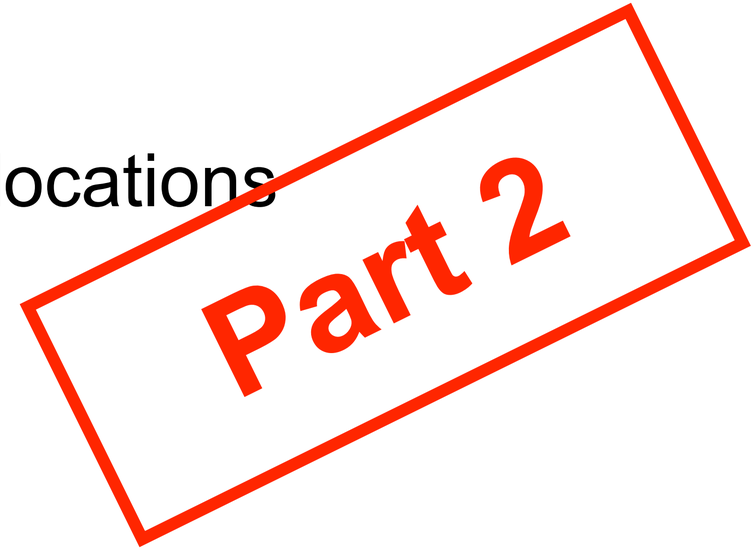
- Most crucial thing: measuring!

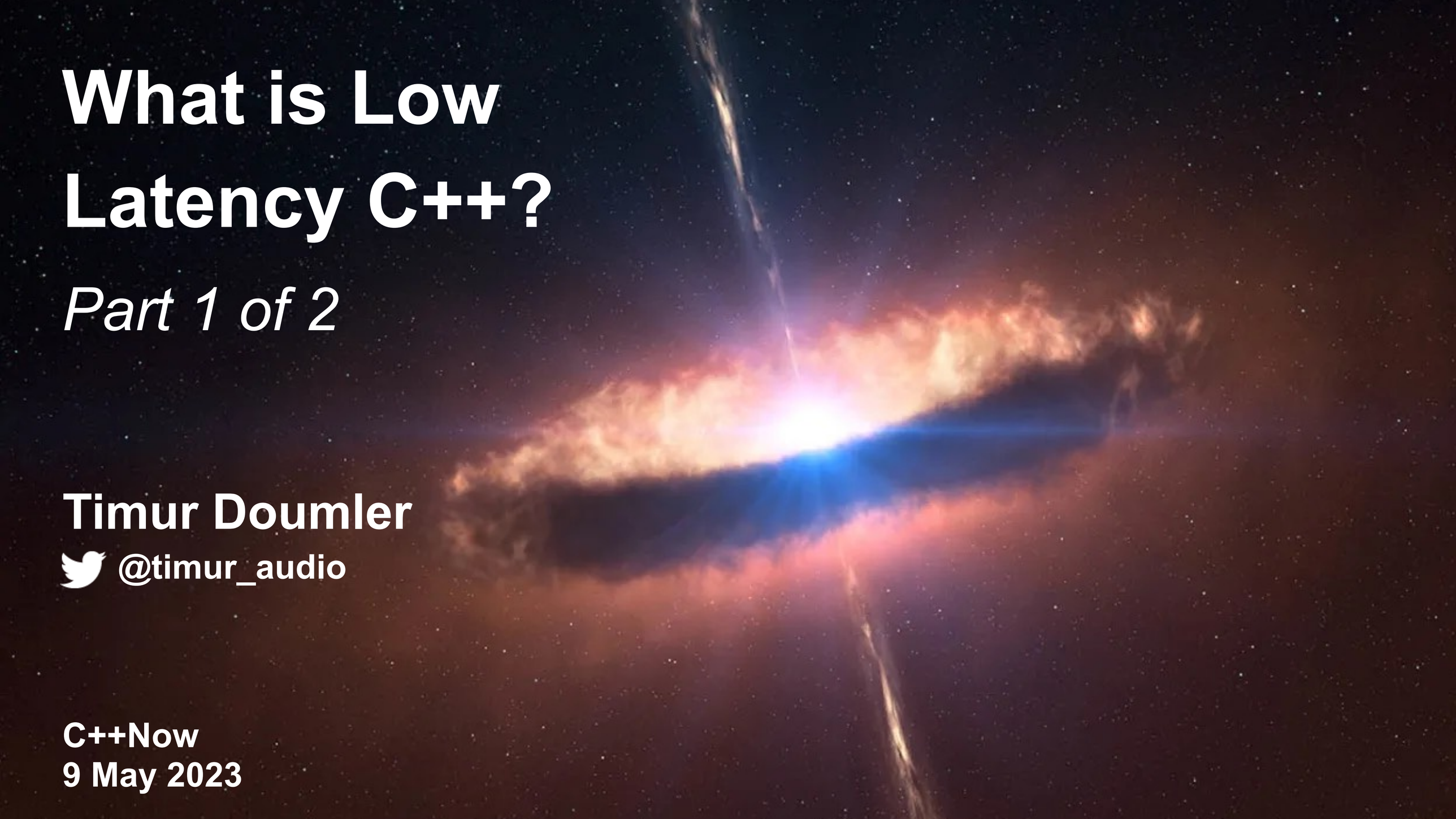
What not to do in the hot path:

- Dynamic memory allocations/deallocations
- blocking the thread
- I/O
- exceptions
- context switches / mode switches (user/kernel space)
- syscalls
- calling into unknown code
- loops without definite bounds
- algorithms > $O(1)$, or with no statically known upper bound on N

What not to do in the hot path:

- Dynamic memory allocations/deallocations
- blocking the thread
- I/O
- exceptions
- context switches / mode switches (user/kernel space)
- syscalls
- calling into unknown code
- loops without definite bounds
- algorithms > *O*(1), or with no statically known upper bound on *N*





What is Low Latency C++?

Part 1 of 2

Timur Doumler
@timur_audio

C++Now
9 May 2023