

# Parallel Programming in Rust:

## Techniques for Blazing Speed

# Background

- Evgenii Seliverstov
- Senior Software Engineer in Bloomberg
  - *Opinions expressed in this talk are solely my own and do not express the views or opinions of my employer*
- Write mostly C++, Rust, a bit less Python and Erlang
- High-performance systems, infrastructure tools and moving JSONs around
- PhD and academic research on parallel computing systems, GPUs and compilers
- Curious about making things better and blazingly fast

# Topics

- Ways to improve performance
- Concurrent vs. parallel vs. asynchronous computing
- High-level parallelism
- Internals of well-known parallel primitives
- Leveraging low-level hardware capabilities
- How to measure performance
- Concurrency challenges
- Future for GPU

# Case study

Crates.io dump <https://crates.io/data-access#database-dumps>

```
uv init && uv add polars ruff
```

```
import polars as pl
import sys

df_crates = pl.read_csv(sys.argv[1])
df_bz = df_crates.with_columns(
    pl.col("readme").str.extract(r"(?im)(blazing|blazingly|\d+)").alias("fast")
).filter(pl.col("fast").is_not_null())
print("{:.2}%".format(100 * df_bz.shape[0] / df_crates.shape[0]))
```

# Case study

- Run it

```
uv run main.py $(fd crates.csv)
```

- Guesses?
- 0.79%. Not bad! Humble and worthy.

# Case study

Turns out, a lot of tools you use every day are rewritten in Rust

- uv
- polars
- ruff
- fd
- cat (batcat)
- I guess you're running it in Warp and editing in Zed? 😊
- Parts of the kernel



# Case study

## Why the RIIR?

- ~~Add "blazingly fast 🚀" announcement to your Readme~~
- Make tools faster
- Make code safer
- Achieve more work for the same wall time
- Introduce parallelism on different levels

# Source of parallelism

## 1. Instruction-level

- Instruction pipelines (ILP)
- Superscalar
- Vectorized SIMD (single instruction, multiple data)

## 2. Task-level

- Processes
- Threads
- Green threads

## 3. System-level

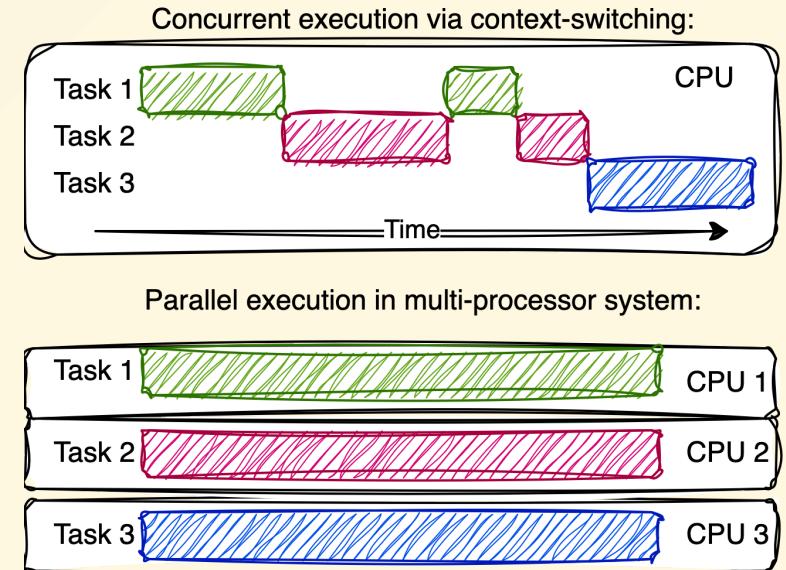
- Accelerators (GPU, TPU, \*PU)

## 4. Network-level

- Distributed systems
- Message-passing task parallelism

# Parallelism, concurrency and asynchronicity

- Concurrency paradigm
  - Running multiple tasks using the same processor
  - Context-switched
  - Multi-threading
- Parallelism paradigm
  - Running multiple tasks using multiple processors
  - On multiple systems / processors / cores
- Synchronous model
  - execute tasks in a sequence
- Asynchronous model
  - execute tasks simultaneously
  - tokio utilizes multi-threading concurrency



# Abstraction level

- Libraries - high-level data parallelism
- Thread pools - high-level concurrency
- Mutexes, queues, lock-free - concurrency primitives
- Atomic operations, cache - low-level optimisation
- SIMD, GPU - hardware-supported data parallelism
- Well-isolated
- Composable

cargo build --release	&str	
Arc<Mutex<T>>		
channel()	rayon	std::mpsc
sync_channel()		parking_lot
into_par_iter()	ThreadPool	
num_cpus	thread affinity	
crossbeam::queue	RwLock	
perf	_unsafe	
AtomicU32	criterion	
flamegraph	libc::	
spinlock	inlining	
smallvec	weak vs strong	
hazard pointers	lock-free	ABA
Ordering::Relaxed		
std::intrinsics	portable_simd	
	cargo show-asm	
#[target_feature]	Amdahl Law	
L1d	false sharing	AVX512
CachePadded	asm!	NEON
	_mm256_add_ps	godbolt

# Rayon: task-based data parallelism

```
use rayon::prelude::*;

let res: i64 = (0..5).into_par_iter()
    .map(|x| x * 2).filter(|x| x % 2 == 0)
    .sum();
```

- Implementations:
  - sequential
  - thread pool
- A library, no built-in functionality in std
- Similar approaches in other languages:
  - C++ 17 - parallel `<algorithm>` + `<execution>` libraries



```
std::for_each(std::execution::par, vec.begin(), vec.end(), [](auto x) { return x * 2; });
```

- Java - `Collection::parallelStream`.
- Python - only `concurrent.futures.ThreadPoolExecutor.map`

# Rayon iterator trait

- Mirror traits for std library (`Iterator` -> `ParallelIterator`)

```
impl<T: Send> IntoParallelIterator for VecDeque<T> { }
```

- Plenty of methods (`map`, `for_each`, `filter`, `sum`)
- Random access - via different trait `IndexedParallelIterator`.

```
(0..5).into_par_iter().take(3)
```

- Possible to wrap std iterators with

```
(0..5).iter().par_bridge()
```

# Rayon chunk and window struct

## Specialized functions

- Chunks - almost the same struct as `std::slice::Chunks`

```
fn par_chunks(&self, size: usize) -> Chunks
```

```
let chunks: Vec<_> = [1, 2, 3, 4, 5].par_chunks(2).collect();
assert_eq!(chunks, vec![&[1, 2][..], &[3, 4], &[5]]);
```

- Windows

```
fn par_windows(&self, size: usize) -> Windows
```

- Split

```
fn par_split(&self, sep: T) -> Split
```

# Rayon for strings

The same idea for other major types

```
impl ParallelString for str {  
    fn as_parallel_string(&self) -> &str {}  
}
```

```
let char_count: usize = "Hello\nRustNation\nUK".as_parallel_string().par_lines().map(|x| x.len()).sum();
```

- Sorting - parallel merge sort (timsort-based)

```
fn par_sort(&mut self)
```

- Processing plan is not known in advance
- Divide and conquer

# Rayon internals: threads

- Functions (tasks) should be thread-safe
  - ideally pure
  - synchronized (with `std::sync::Arc` or `std::sync::Atomic`)
- A fork-join model over a thread pool
  - But better
- Dynamic balanced scheduler
  - Work-stealing
  - Data is `Send`
  - Better performance
  - Data moves to threads

# Rayon internals: data

- Data-driven
  - push / pull
  - Multiple steps are coordinated with callbacks
  - A bunch of producers for different iterators
  - Data is fed to threads queues
- Dynamic data split during `join`
  - A building primitive

```
rayon::join(|| op1(), || op2());
```

- Hierarchical split
- Specials: `by_exponential_blocks()` or `by_uniform_blocks(block_size: usize)`
- Send

# Rayon: thread pool

- Predetermined size
- Thread creation is expensive - a global thread pool is kept
- Understands CPU count of the system
- Customize global pool:
  - Explicitly

```
rayon::ThreadPoolBuilder::new().num_threads(32).build_global().unwrap();
```

- Or by `RAYON_NUM_THREADS`
- Multiple pools for different purposes

```
let pool = rayon::ThreadPoolBuilder::new().build().unwrap();
pool.install(|| { /* do work */ })
```

# Checking threads

In Unix, a thread appears as a process and has a PID

Linux:

```
ps h -T -o tid -p $(pgrep app)
```

macOS or BSD:

```
ps -M -p $(pgrep app)
```

In Windows, thread is a separate concept

# How parallel am I?

```
std::thread::available_parallelism()
```

- Logical CPUs
- Applies cgroup quotas
- Includes hyper-threading cores

# Deep dive: logical processors

Create [num\\_cpus](#) crate

`num_cpus::get()` - logical CPUs

The number of logical CPUs can change.

To get alive processors:

- Linux sysconf API `_SC_NPROCESSORS_ONLN`
- vs. old `_SC_NPROCESSORS_CONF` (static)

`num_cpus` now returns offline cores too via `num_cpus::get()`

# Deep dive: physical processors

OS-aware - num\_cpus crate

```
num_cpus::get_physical()
```

Physical cores:

- Linux: `/proc/cpuinfo`
- Windows: `GetLogicalProcessorInformation` / `GetSystemInfo`
- Macos: `hw.physicalcpu`

NUMA

# CPU affinity

- Threads migrate between CPU cores
- Good:
  - Devs do not care
  - Hardware scheduler could be fair
- Bad:
  - Cache misses and contention
  - Migration overhead

Threads could be pinned to specific CPU core

```
unsafe fn pin_to_cpu(cpu: usize) {  
    let mut set: libc::cpu_set_t = std::mem::zeroed();  
    libc::CPU_ZERO(&mut set);  
    libc::CPU_SET(cpu, &mut set);  
    let ret = libc::sched_setaffinity(0, std::mem::size_of::<libc::cpu_set_t>(), &mut set);  
}
```

# To unsafe or not to unsafe?

- Performant, safe or both?
- `unsafe` does not mean program is not memory-safe
  - Developer is responsible for ensuring invariants
- Interfacing with system libraries, libc and kernel
- All FFIs are unsafe
- Hardware interfaces (atomics, SIMD) are unsafe
- Core data structures (queues, channels, ring buffers) are built on unsafe
- Isolate libc/kernel calls and algorithms in wrappers and crates

# Platform-specific features

```
error: cannot find type `cpu_set_t` in crate `libc`
```

- Handling differences between operating systems
- Conditional compilation with `cfg`
- Don't confuse with crate features
- `target_os` (linux, macos, windows) or `target_arch`
- Like `rustc --cfg` but implicitly set

# Platform-specific features

- Tailored implementations within one source file

```
#[cfg(target_os = "linux")]
fn pin_to_cpu(cpu: usize) { /* linux impl */ }

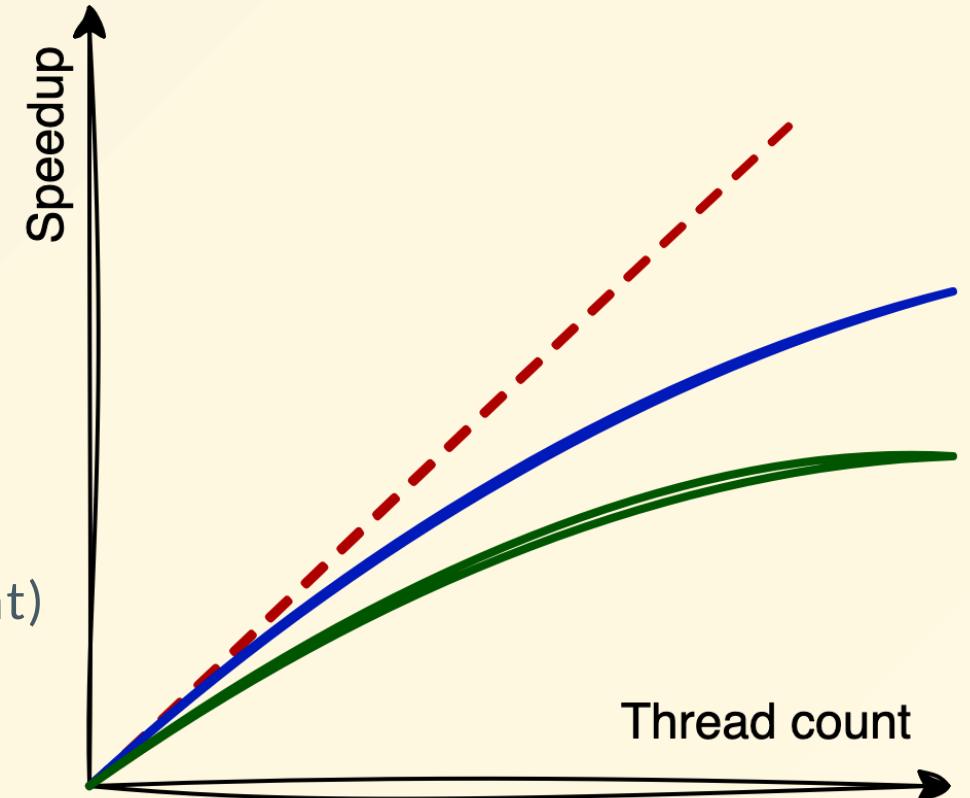
#[cfg(all(target_os = "macos", target_arch="x86_64"))]
fn pin_to_cpu(cpu: usize) { /* intel mac impl */ }
```

- There is no "else". Non-exhaustive match
- Powerful `target_feature` feature

```
#[target_feature(enable = "sha")]
unsafe fn sha_hardware_accelerate() {}
```

# Scalability

- Inherently limited by sequential code
- Amdahl law
  - 10% sequential code -> 10x max speedup
- Scalability blockers
  - Critical sections
  - Synchronisation overhead
  - Message passing overhead (raises with thread count)
  - Lock contention
  - Cache contention



# Performance: sequential metrics

- perf - Linux observability tool (performance counters / events)
  - `perf record`
  - `perf report --stdio` for simple scenario
- Flamegraph - visualisation of execution time
  - Brendan Gregg's stackcollapse Perl scripts
  - `perf report flamegraph` (OS packages are broken)
  - Easier: `inferno` crate
  - `cargo flamegraph && open flamegraph.svg`
  - Spot performance blockers
- No difference with threads
  - Still can collect per-process, per-thread, per-cpu events



# Performance: parallel metrics

- Naive approach - `time`

```
% time ./target/release/bench 50000000 4
./target/release/bench 50000000 4  0.00s user 0.00s system 80% cpu 0.005 total
```

- Scientific approach - `hyperfine`

```
% hyperfine --warmup 3 "./target/release/bench 50000000 4"
Benchmark 1: ./target/release/bench 50000000 4
  Time (mean ± σ):     1.252 s ±  0.009 s    [User: 4.950 s, System: 0.003 s]
  Range (min ... max): 1.244 s ... 1.269 s    10 runs
```

- Statistical benchmarking - multi-run, mean and deviation
- Warmup
- Commercial approach - Intel VTune
- Pragmatic approach - `criterion-rs`

# Communication patterns

Three different approaches

- Shared state - thread-safe `Arc` protected by mutex
- Shared state - with atomics
- Message passing - `std::sync::mpsc`

Mix and match

# Shared state

- Multiple threads sharing the same object in shared memory
- `Arc<T>` - thread-safe access to immutable object
- `Arc<Mutex<T>>` - thread-safe mutable object
- `T: Send + Sync`

# Shared state: example

```
let counter = Arc::new(Mutex::new(0u64));
let mut handles = vec![];
for _ in 0..5 {
    let thread_counter = counter.clone();
    let handle = std::thread::spawn(move|| -> () {
        let mut data = thread_counter.lock().expect("locked");
        *data += 1;
    });
    handles.push(handle);
}
for handle in handles {
    handle.join().unwrap();
}
println!("Sum is {}, should be 5", counter.lock().unwrap());
```

- Cloning `Arc` just changes the reference count
- Each thread owns its own `Arc` object
- The underlying object is not cloned (no need to be `Clone` or `Copy`-able)

# Shared state: downsides

- Mutex lock contention
- Heavy - OS/library primitive
  - uncontended: 0.1 - 10 microseconds
  - contended: 10 - 1000 microseconds
- Usually calls kernel
- Or stays in user-space with futex syscall
- Readers block each other
  - Solution - `std::sync::RwLock` for multiple readers
- Advanced mutex implementation - `parking_lot` crate

# Message passing

- Threads do not share state
- Communicate by sending messages
- Well-known in scientific computing (MPI)
- CSP (communicating sequential processes) - Erlang, Go, Crystal
- Distributed computing
- Foundation for the actor model

# Example

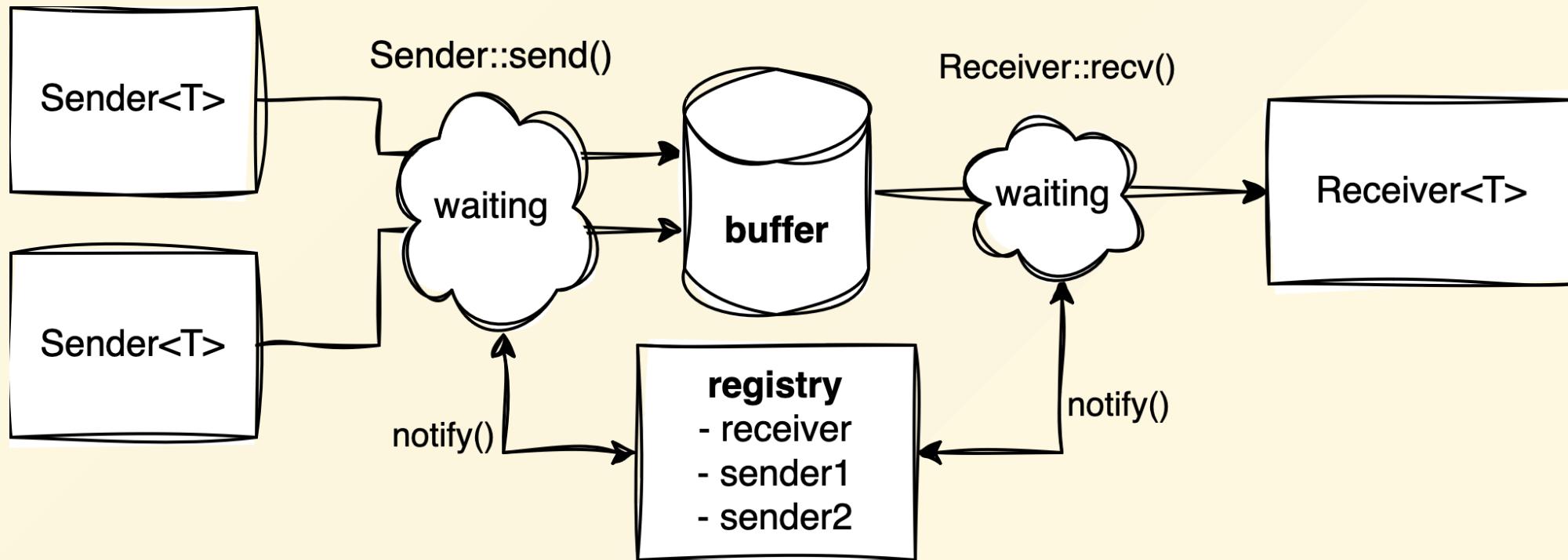
- MPSC: Multi-producer, single-consumer FIFO queue
- One-directional

```
let (sender, receiver) = mpsc::channel::<u64>();  
  
let sender2 = sender.clone();  
std::thread::spawn(move || {  
    sender.send(10).unwrap();  
    sender.send(11).unwrap();  
});  
std::thread::spawn(move || {  
    sender2.send(20).unwrap();  
    sender2.send(21).unwrap();  
});  
for _ in 0..4 {  
    println!("{}: {:?}", _, receiver.recv().unwrap());  
}  
receiver.recv().unwrap_err(); // RecvError
```

# Channel flavours

- `Sender<T>` is cloneable to support multiple producers
  - `Receiver<T>` is not
  - Type `T` implements `Send` trait
  - Simple unbounded FIFO queue
  - Bounded FIFO queue
- ```
channel() -> (Sender, Receiver)
```
- ```
fn sync_channel(bound: usize) -> (SyncSender, Receiver)
```
- `Sender::send` does not block
    - `SyncSender::send` can block
    - Backpressure
    - Still can use `Sender::send` and `SyncSender::try_send`
  - Could use `Receiver::recv_timeout` to avoid blocking

# Channel internals



# Crate crossbeam

A handy toolbox of parallel utilities. Nightly for nightly

- `crossbeam::channel` - channel for message passing
- A source of `std::sync::{mpsc, mpmc}` since Rust 1.67
  - `std::sync::mpmc` is in nightly std
  - MPMC - multiple consumers aka `receiver.clone()`

# Crate crossbeam

Why use it?

- Bleeding edge
- Better performance
- Orchestrating multiple channels with `select!`
- Extra channels - e.g. `crossbeam_channel::tick`

```
let ticker: Receiver<_> = crossbeam_channel::tick(Duration::from_millis(100));
ticker.recv().expect("tick 1");
ticker.recv().expect("tick 2");
```

# Crate crossbeam

- `crossbeam::deque` - work-stealing deque for Rayon
- `crossbeam::queue::ArrayQueue` - thread-safe MPMC-queue
  - Bounded
  - Thread-safe: share between threads via `Arc<ArrayQueue>`
  - Blocking interface (can check size)

# Channel and queues internals

- Not your mother's mutex
  - `Arc<Mutex<List<T>>>`
- Lock-free data structure
  - Buffer - linked list (unbounded) or array (bounded)
  - Head and tail atomic indices
  - Updates via atomic operation compare-and-swap
  - Waiting is spinning in the loop

# Atomics

- Atomic variables
  - Low-level hardware primitive
  - Guaranteed execution to a given state
  - Without interruption and locking
- A limited set of basic types (1-8 bytes)
- A limited set of foundational operations
  - Highly platform dependent
  - Load / store
  - Atomic add / sub
  - Compare and exchange: "if current value equal to desired value, set to target value"

# Rust API

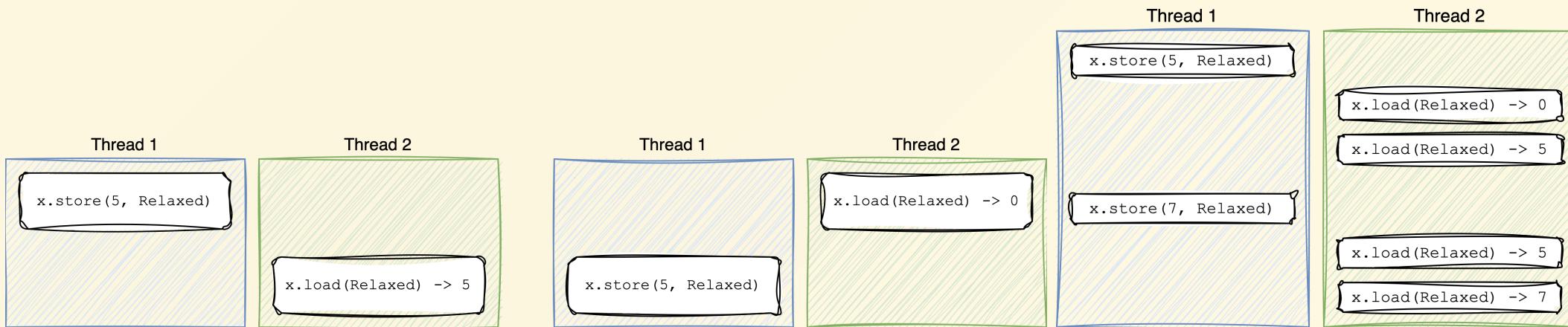
- Three major scenarios:
  - Atomic counters
  - Boolean flags
  - Complex data types based on indices
- Standard types and operations

```
use std::sync::atomic::{AtomicU32, Ordering};
let atomic = AtomicU32::new(0);
// 1a. Atomic load
atomic.store(10, Ordering::Relaxed);
// 1b. Atomic increment
atomic.fetch_add(32, Ordering::Relaxed); // 42
// 2. Boolean swap
let atomicBool = AtomicBool::new(false);
atomicBool.swap(true); // false
// 3a. try to set to 10 if current is 42
atomic.compare_exchange(42, 10, Ordering::Acquire, Ordering::Acquire); // Ok(10) - current value
// 3b. try to set to 20 if current is 99
atomic.compare_exchange(99, 20, Ordering::Acquire, Ordering::Acquire); // Err(10) - current value
```

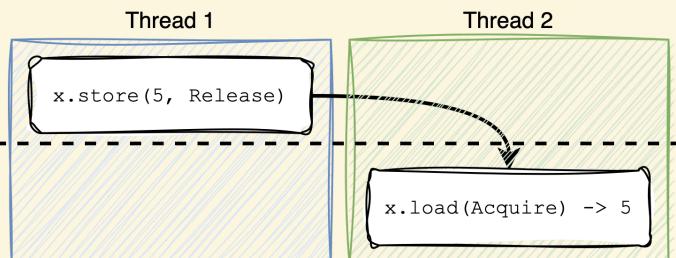
# Atomic ordering

Ordering - access semantics for one atomic variable

- Relaxed (total modification order)



- Release / Acquire (happens-before relationship)



- Sequential (total order relationship)

# Atomic operations internals

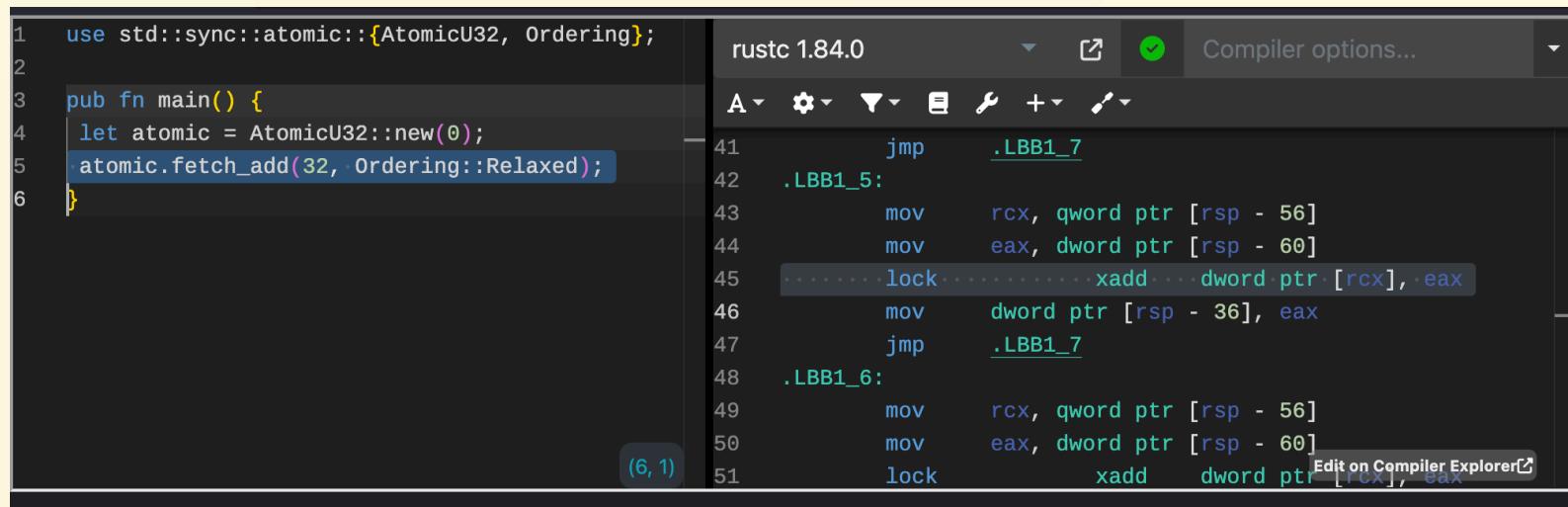
- Implementation lies in standard library

- Rust: `std::intrinsics::atomic_xadd_acquire`
  - C builtin: `__atomic_add_fetch`

- Translates directly to CPU machine code

- `cargo show-asm` or godbolt

- Processor instructions for x86-64



The screenshot shows the Compiler Explorer interface with the Rustc 1.84.0 compiler. On the left, the Rust source code for a main function is displayed:

```
1 use std::sync::atomic::{AtomicU32, Ordering};
2
3 pub fn main() {
4     let atomic = AtomicU32::new(0);
5     atomic.fetch_add(32, Ordering::Relaxed);
6 }
```

On the right, the generated assembly code is shown:

```
rustc 1.84.0
Compiler options...
A  Compiler options...
41     jmp    .LBB1_7
42 .LBB1_5:
43     mov    rcx, qword ptr [rsp - 56]
44     mov    eax, dword ptr [rsp - 60]
45     lock   xadd    dword ptr [rcx], eax
46     mov    dword ptr [rsp - 36], eax
47     jmp    .LBB1_7
48 .LBB1_6:
49     mov    rcx, qword ptr [rsp - 56]
50     mov    eax, dword ptr [rsp - 60]
51     lock   xadd    dword ptr [rcx], eax
```

The instruction at line 45, `lock xadd dword ptr [rcx], eax`, is highlighted with a yellow box. A tooltip for this instruction is visible at the bottom right of the assembly window, reading "Edit on Compiler Explorer".

- Reordering by a compiler or a processor

# Atomic operations internals

- **x86-64** is strongly ordered
  - Relaxed = Release/Acquire
- Processor instructions for **ARMv8** (AARCH64) (weakly-ordered)
  - **LDR** and **STR** - normal Load/Store RISC (non-atomic aka **MOV**)
  - **LDAR** and **STLR** - atomic Load/Store (acquire/release)
  - **LDAXR** and **STAXR** - exclusive Load/Store instructions (acquire/release)
  - **LDADDL w8, w8, [x9]** - newer CISC instructions (strong)
  - **CAS** - CISC instruction for compare-and-swap
  - ARM becomes "stronger" :)
  - Acquire/release is expensive, while relaxed is cheap
- Other weak architectures: RISC-V, Power (PPC64)
- Test on the weakest architecture

# Lock-free algorithms over atomics

- Primitives: load / store / compare-and-exchange / compare-and-swap
- Atomic operations on Boolean up to 64-bit data
- How about heavier data structures?

# Lock-free algorithms over atomics

- Primitives: load / store / compare-and-exchange / compare-and-swap
- Atomic operations on Boolean up to 64-bit data
- How about heavier data structures?
- Locking primitive over atomics
  - Try to set atomic variable to a target state
  - If it doesn't work, try again
  - Sleep a little to allow thread rescheduling
  - While locked, spin and burn CPU
  - Compare: kernel mutex lock puts the thread to sleep state

# Very basic spinlock

```
pub struct SpinLock {
    lock: AtomicBool,
    value: *mut u64,
}

impl SpinLock {
    pub fn new() -> Self {
        Self {
            lock: AtomicBool::new(false),
            value: Box::into_raw(Box::new(0)),
        }
    }

    pub fn lock(self: &Self) -> SpinLockGuard {
        while self.lock.swap(true, Ordering::Acquire) == true {
            std::thread::yield_now();
        }
        SpinLockGuard{ lock: self }
    }

    pub fn unlock(&self) {
        self.lock.store(false, Ordering::Release);
    }
}

pub struct SpinLockGuard<'a> {
    lock: &'a SpinLock
}

impl<'a> DerefMut for SpinLockGuard<'a> {
    fn deref_mut(&mut self) -> &u64 {
        unsafe { &mut *self.lock.value }
    }
}
```

# Spinlock usage

- Trust me, I am a safe mutex (tag traits)

```
unsafe impl Send for SpinLock {}  
unsafe impl Sync for SpinLock {}
```

- Usage (same as with other locking primitives like `std::sync::Mutex`)

```
let spinlock = Arc::new(SpinLock::new());  
for _ in 0..2 {  
    let local_spinlock = spinlock.clone(); // ref-counted copy  
    std::thread::spawn(move || -> () {  
        for _ in 0..10 {  
            let guard = &mut local_spinlock.lock();  
            let value: &mut u64 = guard;  
            *value += 1;  
            local_spinlock.unlock();  
        }  
    });  
}
```

# Atomics: more use-cases

- Spinlock only uses Boolean flag
- Lock-free queues - use atomic indices (head / tail)
- ABA problem (missing writes anomaly)
- Reference counting (`Arc`) on atomics
- Rarely acquire/release only
- Please, don't implement lock-free data structures by yourself!

# Performance considerations

- Use lock-free structures with significant contention (many threads, frequent locking)
- Use lock-based structures for long waits (blocks the thread instead spinning the lock)
- Extra size for MPSC/MPMC channels:
  - extra pointer size to each item
  - ~100 bytes per channel
  - under 1 KiB shared between senders and receivers
  - cache padding
  - mutex is smaller
- Implementation is inherently unsafe
  - MaybeUninit
  - Atomics (indirectly)
  - Buffer operations
- **Always measure performance!**

# Performance: kernel counters

- perf stat
- Aggregated metrics

```
perf stat \  
  --event task-clock,migrations,faults,  
  context-switches,cycles,instructions,  
  branches,branch-misses,cache-references,  
  cache-misses \  
  ./target/release/bench
```

- Per-thread metrics

```
perf stat \  
  --per-thread -t 3033001,3033002,3033003 \  
  --event branch-misses,cache-misses
```

- Analyze fine-grained thread behaviour
- Cache access, core migrations

```
% perf stat \  
  --event task-clock,migrations,faults,context-switches,cycles,instructions,branches,branch-misses,cache-references,cache-misses \  
  ./target/release/bench 500000000 3  
  
Performance counter stats for './target/release/bench 500000000 3':  
  
      115,638.82 msec task-clock          #    2.597 CPUs utilized  
              5   migrations            #    0.043 /sec  
             95   faults              #    0.822 /sec  
           1,792 context-switches        # 15.497 /sec  
 278,968,011,471 cycles            #    2.412 GHz          (33.34%)  
 208,935,082,575 instructions       #    0.75  insn per cycle     (33.35%)  
 22,000,730,970 branches           # 190.254 M/sec        (33.34%)  
 2,053,586,366 branch-misses       #    9.33% of all branches  (33.33%)  
    24,262,120 cache-references     # 209.809 K/sec        (33.32%)  
      549,902 cache-misses          # 2.267 % of all cache refs (33.33%)  
  
 44.529077311 seconds time elapsed  
  
 115.627403000 seconds user  
  0.015999000 seconds sys
```

# Performance: eBPF

- Turing-complete program event-driven program running in kernel 🤖
- Example of `bpftrace`: a user-defined program to track thread pool lifecycle

```
% bpftrace -e '
    tracepoint:sched:sched_process_fork { printf("> thread fork %s %ld\n", args->child_comm, args->child_pid)},
    tracepoint:sched:sched_process_exit { printf("< thread exit %ld\n", args->pid)}
    ' -c "./target/release/bench 500000 2"

Attaching 2 probes...
> thread fork bench 3043015
> thread fork bench 3043016
< thread exit 3043015
< thread exit 3043016
> thread fork bench 3043017
> thread fork bench 3043018
< thread exit 3043017
< thread exit 3043018
< thread exit 3043014
```

# Performance: eBPF aggregates

- Flavours of eBPF programs:
    - BCC (complex)
    - bpftrace (one-liners)
    - bpftrace (programs)

```
tracepoint:syscalls:sys_enter_futex
/comm == "rnrayon"/
{
    @start[tid] = nsecs;
}

tracepoint:syscalls:sys_exit_futex
/comm == "rnrayon"/
{
    @end[tid] = nsecs;
    @duration = hist(nsecs - @start[tid]);
    @totalspent[tid] = sum(nsecs - @start[tid]);
    delete(@start[tid]);
}
```

- Track futex syscall

```
bpftrace -c rnrayon futex.epbf
```

- Statistics for lock duration
  - Statistics for total lock time

# Performance: lock contention

- Performance counters for locks

```
perf lock record ./target/release/bench
```

- `perf lock contention -a` (5 threads, 100k locks each)

contended	total wait	max wait	avg wait	type	caller
8146	41.48 ms	30.96 us	5.09 us	spinlock	futex_wake+0xc4
2513	10.58 ms	14.24 us	4.21 us	spinlock	futex_q_lock+0x26
4	1.25 ms	1.15 ms	313.23 us	rwsem:W	__vm_munmap+0x92
6	185.29 us	51.42 us	30.88 us	rwsem:R	lock_mm_and_find_vma+0x9b

- For 3 threads

contended	total wait	max wait	avg wait	type	caller
1511	6.38 ms	16.94 us	4.22 us	spinlock	futex_wake+0xc4
136	526.25 us	9.26 us	3.87 us	spinlock	futex_q_lock+0x26

# Hard choice

- High-level iterator abstraction - Rayon
- One-directional communication - channels
  - One receiver - `std::mpsc`
  - Multiple receivers - `crossbeam_channel` or nightly
  - Bounded or unbounded channel - `channel_sync` or `channel`
- Omni-directional access - lock-free queue
  - bounded `crossbeam::ArrayQueue` and unbounded `crossbeam::SegQueue`
- Async channels? Sorry, check `flume` or `tokio`
- Locking data structures - `Arc<Mutex<Container>>`
- Two levels of data parallelism (mix and match):
  - High: threads or tasks
  - Low: hardware SIMD extensions

# Low-level data parallelism: SIMD

- Hardware-accelerated vectorised operations
- Enabled data parallelism in truly parallel manner
- Operates on primitive scalar types (8 to 64 bits)
- Basic scalar operations - add, sub, mul, mask, sum, reduce
- And much more
- Highly processor-dependent

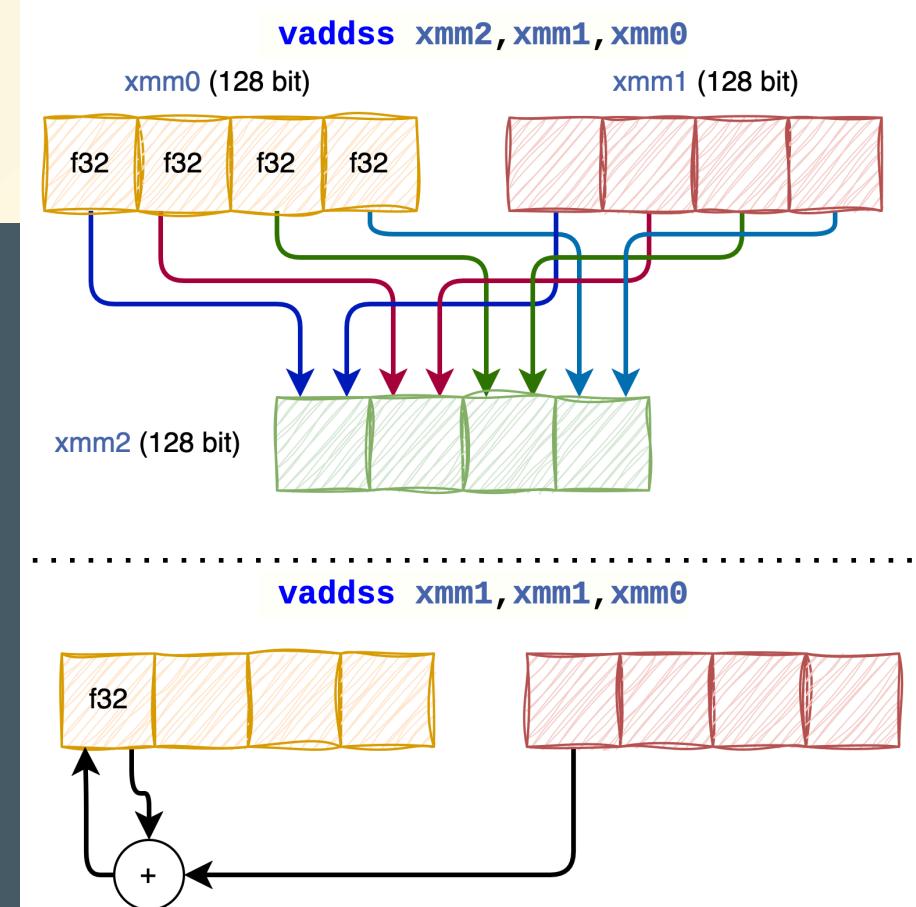
# Auto-vectorisation

- Automatic SIMD generation
- LLVM-based
- Release (`-c opt-level=3`)
- `cargo show-asm` is your friend
- Help compiler:
  - Simpler is better
  - Isolated functions
  - `#[inline(never)]`
  - `for` loops
  - Asserts on ranges
  - Slices, not containers like `Vec`
  - Ranges are pre-defined
  - Restrict reuse of variables

# Low-level SIMD

- Stable API `core::arch`
- intrinsics - exact wrappers for CPU instructions

```
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
#[target_feature(enable = "avx")]
unsafe fn add_f32_avx(a: &[f32], b: &[f32], c: &mut [f32]) {
    assert!((a.len() % 4) == 0);
    let chunk_len = a.len() / 4;
    for idx in 0..chunk_len {
        let chunk = idx as isize;
        let achunk: *const f32 = a.as_ptr().offset(chunk);
        let bchunk: *const f32 = b.as_ptr().offset(chunk);
        let avec: __m256 = _mm256_loadu_ps(achunk); // VMOVUPS
        let bvec: __m256 = _mm256_loadu_ps(bchunk); // VMOVUPS
        let cvec: __m256 = _mm256_add_ps(avec, bvec); // VADDPS
        let cchunk: *mut f32 = c.as_mut_ptr().offset(chunk);
        _mm256_storeu_ps(cchunk, cvec); // VMOVUPS
    }
}
```



- Register ordering is still done by compiler

# The state of SIMD support

- As for beginning of 2025:
  - stable x86-64 SSE3, SSE4, AVX, AVX2
  - nightly x86-64 AVX512
  - stable NEON support for AARCH64
  - nightly NEON support for other ARMs (32-bit ARM etc)

# portable-simd

- High-level SIMD library
- Ergonomic
- Nightly only
- Works best with nightly slice features (`[array_chunks]`)
- `std::simd` (nightly feature `portable_simd`) or crate `portable-simd`

# portable-simd: primitive operations

- Foundation type

```
#[repr(simd)]
pub struct Simd<T, const N: usize> /* */ where LaneCount<N>: SupportedLaneCount, T: SimdElement;
```

- Concrete types

```
pub type f32x8 = Simd<f32, 8>;
```

- Bridges between registers and Rust types:

- `Simd::from_array`, `Simd::from_slice`
- `Simd::to_array`
- `Simd::splat(default_value: T)`
- `Simd::gather(slice: &[T], idx: &[usize]) -> Self`
- `Simd::scatter(slice: &mut [T], idx: &[usize]) -> Self`

# portable-simd: dot-product with AVX

```
#[inline(never)]
fn dot_product1(a: &[f32; 8], b: &[f32; 8]) -> f32 {
    // 32bit * 8 elements = 256 bit (one YMM)
    let avec = f32x8::from_array(*a);
    let bvec = f32x8::from_array(*b);
    let one_sum: f32x8 = avec * bvec;

    let one_scalar: f32 = one_sum.reduce_sum();
    one_scalar
}
```

- Build for Haswell with AVX (YMM - 256-bit registers, XMM - 128-bit registers)
- Check the assembly `cargo asm --target-cpu=haswell --rust dot_product1`

```
// simplified! run yours
vmovups ymm0, ymmword ptr [rdi]           // load register ymm0 (256 bit) from memory
    unsafe { core::intrinsics::simd::$simd_call($lhs, $rhs) }
vmulps ymm0, ymm0, ymmword ptr [rsi]         // multiply ymm0 with memory location (CISC-y)
    unsafe { core::intrinsics::simd::simd_reduce_add_ordered(self, 0.) }
vaddss xmm1, xmm0, xmm1                      // sum all eight 32-bit elements
```

# portable-simd: dot-product with SSE4

- Now build for Nehalem (pre-historical 2008 CPU with SSE4.2) - only XMM registers
- How could it work? `fx32x8` is "256-bit" SIMD register!
- Check the assembly `cargo asm --target-cpu=nehalem --rust dot_product1`

```
movups xmm0, xmmword ptr [rdi]           // load register xmm0 (256 bit) from memory
movups xmm1, xmmword ptr [rdi+16]         // load register xmm1 (256 bit) from memory
    unsafe { core::intrinsics::simd::$simd_call($lhs, $rhs) }
mulps xmm2, xmm0                         // multiply xmm2 with xmm0 (first part)
movups xmm0, xmmword ptr [rdx + 16]
    unsafe { core::intrinsics::simd::$simd_call($lhs, $rhs) }
mulps xmm0, xmm1                         // multiply xmm0 with xmm1 (second part)
addss xmm1, xmm2                         // sum four 32-bit elements (partially)
```

- Magic!
- portable-simd internally splitted operation to supported instruction set (SSE4 with XMM)
- `std::core::Simd` lane is virtual
- Leeway to choose `N` part in `Simd<T, N>`

# portable-simd: dot-product of arbitrary length

- Input is not limited
- Iterator interface (compare with Rayon)

```
#[inline(never)]
fn dot_product2(a: &[f32], b: &[f32]) -> f32 {
    letavec = a.array_chunks::<8>().map(|&x| f32x8::from_array(x));
    letbvec = b.array_chunks::<8>().map(|&x| f32x8::from_array(x));
    avec.zip(bvec).map(|(x,y)| (x * y).reduce_sum()).sum();
}
```

- Choose lane count with performance benchmark
  - Not less than hardware register width (8 for AVX)
- No need to guard with `cfg`
- Worth checking assembly

# Compiler settings

- `RUSTFLAGS` env var
- More usable - build settings for Cargo in `.cargo/config.toml`

```
[target.'cfg(all(target_arch = "x86_64"))']
rustflags = ["-C", "target-feature=+avx", "-C", "target-cpu=haswell"]

[target.'cfg(all(target_arch = "aarch64"))']
rustflags = ["-C", "target-feature=sve", "-C", "target-cpu=cortex-a53"]
```

- Note: `cfg target_arch` does not match your target triple `aarch64-unknown-linux-gnu`
- Allows the compiler to generate instructions for the whole binary
- Can fail if executed on CPU without that feature
- `target_cpu=native` is harmful

# Code generation

- How to emit different code in the same binary?
- Spot a difference?

```
#[cfg(target_feature="avx")]
fn fun1() { }

#[target_feature(enable="avx")]
fn fun2() { }
```

- Function `fun1` can be built with a global `avx` feature only
- Function `fun2` enables feature locally
- Run on Nehalem

```
#[target_feature(enable="avx")]
fn fun2() {
    unsafe { _mm256_setzero_ps(); }
}
```

- Crashes

# Dynamic feature detection

- Fine-grained architecture detection
- Lookup `/proc/cpuinfo` or `cpuid` asm call and route calls
- Rust `std` library helps
- Helpers
  - `std::arch::is_x86_feature_detected`
  - `std::arch::is_aarch64_feature_detected`
  - the rest is nightly
- Target architectures and features, not CPU families

# Dynamic feature detection example

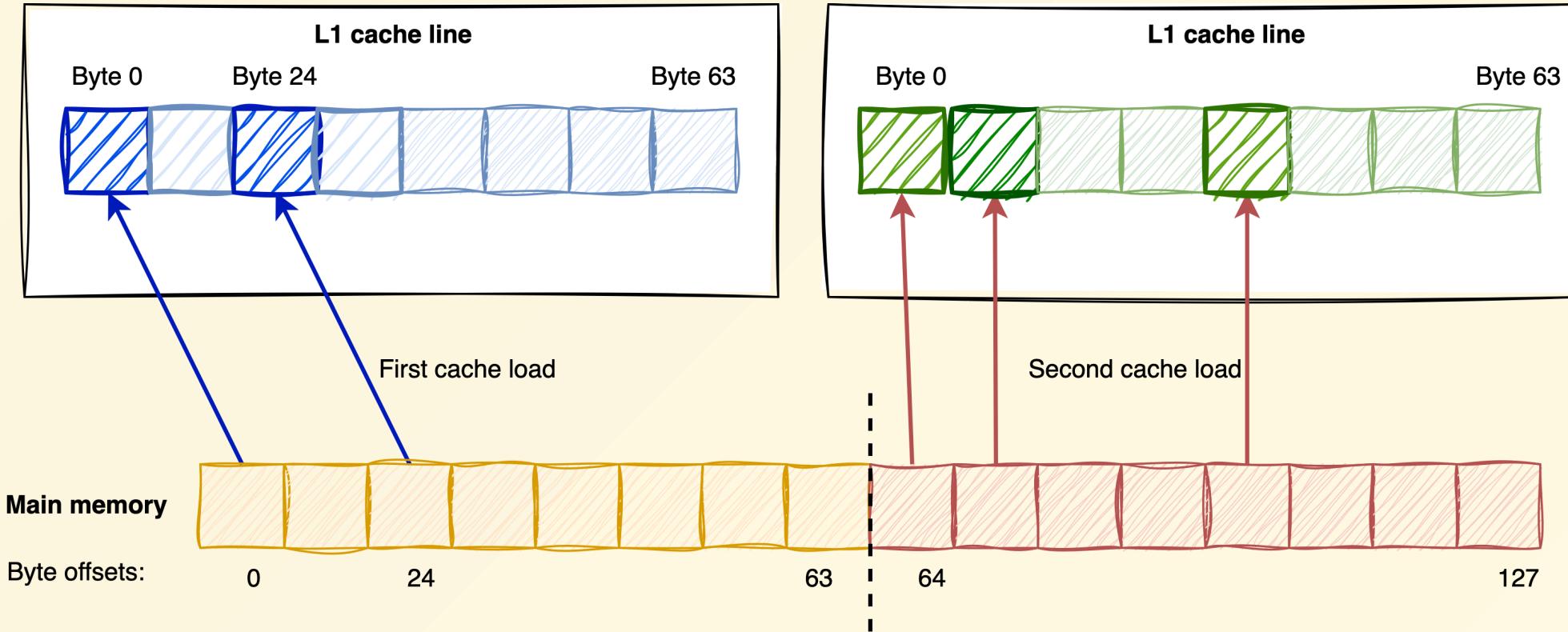
- Safe multiplexer

```
fn dot_product(a: &[f32], b: &[f32]) -> f32 {
    #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
    {
        if is_x86_feature_detected!("avx512f") { unsafe { dot_product_avx512(a, b) } }
        else if is_x86_feature_detected!("avx") { unsafe { dot_product_avx(a, b) } }
        else { dot_product_native(a, b) }
    }
    #[cfg(any(target_arch = "aarch64"))]
    {
        if is_x86_feature_detected!("neon") { unsafe { dot_product_neon(a, b) } }
        else { dot_product_native(a, b) }
    }
    unimplemented!("unsupported architecture");
}

#[cfg(any(target_arch = "aarch64"))]
#[target_feature(enable = "neon")]
unsafe fn dotprod_neon(a: &[f32], b: &[f32]) -> f32 {
    use std::arch::aarch64::*;
    42.0
}
```

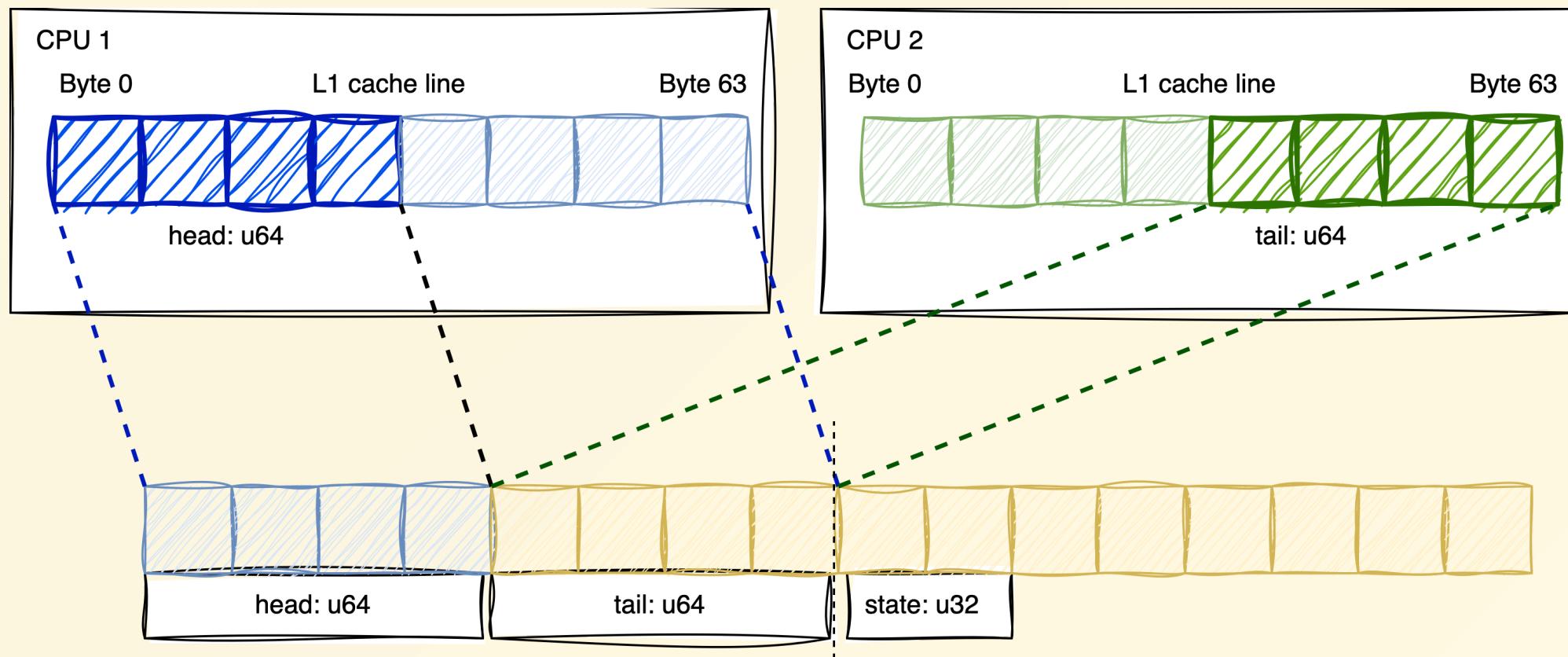
# Still not blazingly fast?

- Doesn't scale well even with modern algorithms & data structures?
- Per-core L1 cache
  - Fully transparent for sequential access

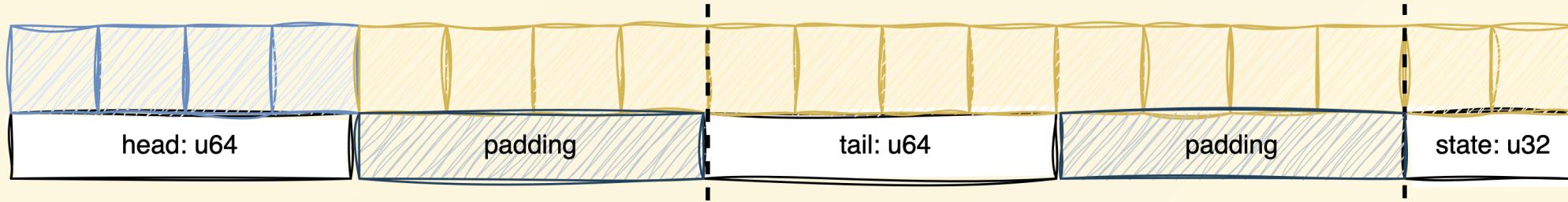


# Concurrent cache access

- It's harmful sometimes
- Write requires exclusive cache access (E state in MESI protocol)
- Invalidating cache line



# Cache padding



```
use crossbeam::utils::CachePadded;
struct List {
    head: CachePadded<u64>,
    tail: CachePadded<u64>,
    state: CachePadded<u32>,
}
```

- Platform-dependent (thanks to `cfg` and `target_arch`)
- Crossbeam use 128-byte cache size for modern x86-64 / ARM64 processors
- Used to be 64 bytes for x86, older ARM processors
- Align beginning of data to 64 bytes - helps SIMD

# Extra techniques

- Consider using `lto` (`-c lto`)
  - Off by default in release profile
- Slower build, faster binary with `-c codegen-units=1`
- Inline
  - Per function `#[inline(always)]`
- Choose your arithmetic
  - `checker_`, `saturating_`, `overflowing_`
  - checks are on in non-release profile
- High-level abstractions like Apache Arrow

# GPU: state of the ecosystem

- GPU is a coprocessor
- Nvidia CUDA
- Massively parallel (thousands of arithmetic units)
- Multi-level hierarchy of memory
- SIMD-like execution - SIMT (single instruction, multiple threads)
- Specialized code kernels running on-chip

# GPU: binary formats

- CUDA kernels - C++ dialect with language extension (like OpenMP or OpenACC)
- LLVM compiler to translate to IR (intermediate representation)
- Nvidia-specific NVVM IR
- IR is translated to PTX binary code
- PTX is shipped to the GPU
- Host runtime coordinates asynchronous kernel execution

# GPU: state of the ecosystem

- Rust project [Rust-GPU](#)
  - graphics (Vulcan)
  - emit SPIR-V binary code
  - write shaders in Rust
  - alpha
- Rust core project [nvptx](#)
  - Compiler backend to directly generate PTX
  - Based on LLVM capabilities
  - Tier 2 / experimental
- Rust project [Rust-CUDA](#) - rebooting
  - emits LVVM IR
  - write kernels in Rust

## Rebooting the Rust CUDA project

January 27, 2025 · 4 min read



**Christian Legnitto**

Rust GPU and Rust CUDA maintainer



We're excited to announce the reboot of the [Rust CUDA project](#). Rust CUDA enables you to write and run [CUDA](#) kernels in Rust, executing directly on NVIDIA GPUs using [NVVM IR](#).

Our immediate focus is on modernizing the project and integrating with other GPU-related efforts in the Rust ecosystem.

# GPU: Rust-CUDA goals

- Native `rustc` backend to emit LLVM IR
  - Like cranelift
  - Wraps LLVM API
- Host toolkit
  - Usually C++ Runtime API - the CUDA SDK
  - `rust-cuda` wraps a low-level Driver API
  - All host-gpu communications
- GPU part
  - Supports a small subset of kernel features
  - A lot of work to reach feature parity with LLVM

# Conclusion

- Start with high-level parallel abstractions
  - Rayon
  - channels
  - queues
  - portable-simd
- Then, delve into low-level optimisation
  - Rust is a system language
  - All hardware primitives is available
- Learn build tooling to target modern architectures
- Always measure with benchmarks, user- and kernel profilers
- Tailor implementations for different platforms with the powerful `cfg`

# Read more

- [Nomicon](#)
- [What Every Programmer Should Know About Memory](#)
- [Rust Atomics and Locks](#)
- [Hands-on Concurrency with Rust](#)
- [C++ Concurrency in Action: Practical Multithreading](#)
- [Software optimization resources \(<https://agner.org/optimize>\)](#)
- [The Rust Performance Book](#)
- [Rayon internals](#)
- [rust-cuda host](#) and [GPU](#) code
- Intel 64 and IA-32 Architectures Software Developer's Manual



# Thank you for your attention!

- Questions?
- Reach out after the talk
- X [@theirix](https://twitter.com/theirix)
- Code <https://github.com/theirix>
- Blog <https://omniverse.ru>

