
THIS SLIDE
INTENTIONALLY
LEFT BLANK.

# WRITING A TOP-10 CHESS ENGINE IN RUST

CSMR KRAR

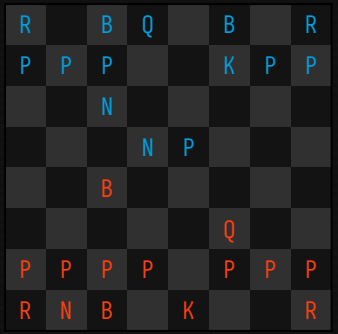
# UNIT DESIGNATIONS

**COSMO** Author of Viridithas (SPCC rank 9)
https://cosmo.tardis.ac

**KORA** Author of Expositor (*defunct*)
https://typ.dev

# PRELIMINARY DEFINITIONS

A CHESS ENGINE is a program that, given the state of a game of chess, attempts to both evaluäte that state and recommend a sequence of moves.




⟨ King to e6, +0.44 in White's favor ⟩

# THE COMPONENTS OF AN ENGINE

**STATE** How is the chessboard represented in the program?

**SEARCH** Constructing and pruning trees of future possibilities*!*

**EVALUATION** How do we estimate the value of a chess position?

# STATE REPRESENTATION

One must be able to

- represent the present state of the game
- query legal moves
- apply transitions caused by moves
- determine when & how a game has ended

This list is highly non-exhaustive.

# TYPES

We use enums to represent colors, pieces, ranks, files...

```
#[derive(...)]
#[repr(u8)]
enum PieceType {
  Pawn   = 0,
  Knight = 1,
  Bishop = 2,
  Rook   = 3,
  Queen  = 4,
  King   = 5,
}
```

# TYPES

We use enums to represent colors, pieces, ranks, files...

By doing this, we benefit manifold:

**RECTITUDE** Precisely specifying the valid values for a type eliminates swathes of bugs.

```
Rank::Third ≠ File::C ≠ PieceType::Knight
```

**CONCISION** Methods on enums are a joy to use.

```
Square::F2.relative_to(Color::Black) == Square::F7
```

**ALACRITY** The more constraints we have on our data, the more we can optimise.

# INDEXING

Suppose we have a type for White *vs* Black like so:

```
enum Color {
  White = 0,
  Black = 1,
}
```

# INDEXING

Suppose we have a type for White *vs* Black like so:

```
enum Color {
  White = 0,
  Black = 1,
}
```

There are many places where we have something like:

```
// occupancy : [u64; 2]
// side : Color
occupancy[side as usize]
```

# INDEXING

Typing out usize every time is annoying, and we'd like to avoid bounds-checks. Fortunately, we can implement the trait std::ops::Index!

```
impl<T> std::ops::Index<Color> for [T; 2] {
  type Output = T;

  fn index(&self, idx : Color) -> &Self::Output {
    unsafe { self.get_unchecked(idx as usize) }
  }
}
```

With this, we can now write "occupancy[side]" without worrying about bounds-checks or usize casts.

*Typically, LLVM can be relied upon to elide bounds-checks.*
*Avoid this pattern unless profiling reveals that elision has failed.*

# BITBOARDS

Bitboards are unsigned 64-bit integers that represent *sets of squares* on the chessboard.

$$\{A1, C1, D1\} \rightleftharpoons \ldots00001101$$

Conveniently, 64-bits is the size of the machine word on modern hardware.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 |
| 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 |
| 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |
| 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |

# BITBOARDS

Typically one stores a chess position using 8 bitboards.

```
struct Position {
  sides  : [Bitboard; 2], // occupancy of each side
  pieces : [Bitboard; 6], // occupancy of each piece-type
}
```

# BITBOARDS

Typically one stores a chess position using 8 bitboards.

```
struct Position {
  sides  : [Bitboard; 2], // occupancy of each side
  pieces : [Bitboard; 6], // occupancy of each piece-type
}
```

Then a query like “how many unblocked pawns are there” is merely:

```
let pawns = sides[Color::White] & pieces[PieceType::Pawn];
let movable_pawns = pawns & !sides[Color::Black].south_one();
return pawns.count_ones();
```

# ITERATION

Given a bitboard (a square-set), how would you efficiently loop over the squares present in the set?

# BIT MANIPULATION

**TZCNT** Count the number of trailing zeroes.

**BLSR** Clear the least-significant set bit.

**POPCOUNT** Count the number of set bits.

```
    tzcnt(10110100) == 2
     blsr(10110100) == 0b10110000
popcount(10110100) == 4
```

`TZCNT` and `BLSR` can be paired to form efficient loops over set bits.

# BIT MANIPULATION

`TZCNT` and `BLSR` can be paired to form efficient loops over set bits.

```
impl Iterator for SquareIter {
  fn next(&mut self) -> Option<Square> {
    if self.value == 0 {
      None
    } else {
      let lsb = self.value.trailing_zeros() as u8;  // TZCNT
      self.value &= self.value - 1;                 // BLSR
      Square::new(lsb)
    }
  }
}
```

# BIT MANIPULATION

Abstraction in hand, we can loop over square-sets with maximum efficiency!

```
let our_knights = board.pieces[Knight] & board.sides[side];
for src in our_knights {
    let moves = knight_attacks(src);
    for tgt in moves & !blockers {
        moves.push(Move::new(src, tgt));
    }
}
```

# EVALUÄTION

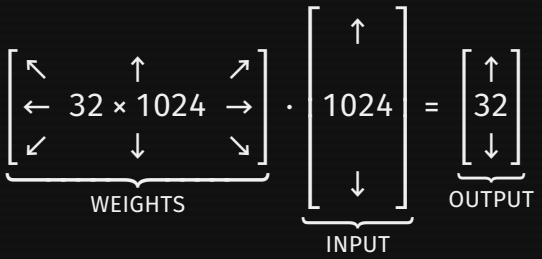
Modern chess engines use Efficiently Updatable Neural Networks (NNUE).

The input features of the network are entirely binary, therefore whenever a piece moves, we simply add or subtract columns of the embedding matrix to generate a new hidden state from an old one.

$$\begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix} \cdot \underbrace{\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}}_{\text{BOARD}} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} \cdots & \cdots & e_{02} & \cdots & \cdots & \cdots \\ \cdots & \cdots & e_{12} & \cdots & \cdots & \cdots \\ \cdots & \cdots & e_{22} & \cdots & \cdots & \cdots \\ \cdots & \cdots & e_{32} & \cdots & \cdots & \cdots \end{bmatrix} \cdot \underbrace{\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}}_{\text{BOARD}} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} + \begin{bmatrix} e_{02} \\ e_{12} \\ e_{22} \\ e_{32} \end{bmatrix}$$

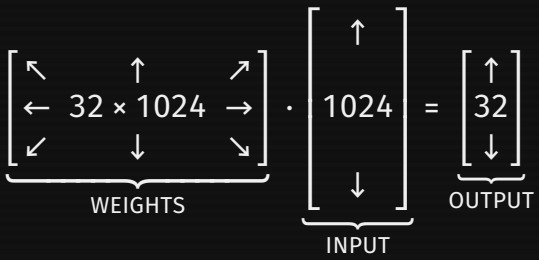
# OPTIMIZATION OF INFERENCE

The most expensive part of the engine is a 1024 → 32 fully-connected layer.



# OPTIMIZATION OF INFERENCE

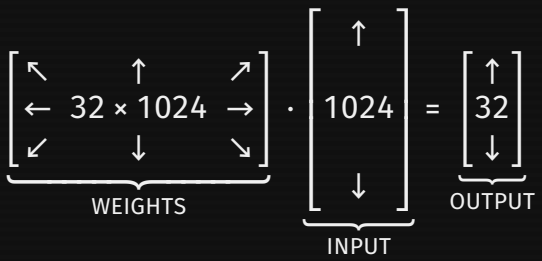
The most expensive part of the engine is a 1024 → 32 fully-connected layer.




**QUANTIZATION** We compress the weights and inputs to i8.

# OPTIMIZATION OF INFERENCE

The most expensive part of the engine is a 1024 → 32 fully-connected layer.

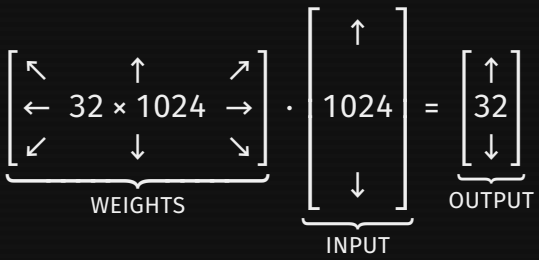



**QUANTIZATION** We compress the weights and inputs to i8.

**SPARSITY** We train the input activations to be mostly zero.

# OPTIMIZATION OF INFERENCE

The most expensive part of the engine is a 1024 → 32 fully-connected layer.




**QUANTIZATION** We compress the weights and inputs to i8.

**SPARSITY** We train the input activations to be mostly zero.

**SIMD** We find and index the non-zero activations in parallel.

## DEDUPLICATION

Many copies of an engine may live and play simultaneously inside the same machine. Reading so many copies of the network causes cache pressure, so we map the same copy of the weights into all processes.

# DEDUPLICATION

Many copies of an engine may live and play simultaneously inside the same machine. Reading so many copies of the network causes cache pressure, so we map the same copy of the weights into all processes.

```
fn weights() -> &'static Network {
  static MAP : OnceLock<Mmap> = OnceLock::new();
  MAP.get_or_init(|| {
    if !network_path.exists() {
      decompress_network_into(&network_path)
    }
    unsafe { Mmap::map(network_path) }
  })  // transmutation and inter-process coördination elided
}
```

# THE SAFE ELIMINATION OF BOUNDS CHECKS

**TASK**

Sort a list of moves by quality, best-to-worst.

```
moves.sort_by_key(|m| {
  (exchange_winning(m), history(m)) // (bool, i32)
});
```

# THE SAFE ELIMINATION OF BOUNDS CHECKS

**TASK**

Sort a list of moves by quality, best-to-worst.

```
moves.sort_by_key(|m| {
  (exchange_winning(m), history(m)) // (bool, i32)
});
```

**REFINEMENT A** We usually only need the first 1–2 elements of the sorted list.

# THE SAFE ELIMINATION OF BOUNDS CHECKS

**TASK**

Sort a list of moves by quality, best-to-worst.

```
moves.sort_by_key(|m| {
  (exchange_winning(m), history(m)) // (bool, i32)
});
```

**REFINEMENT A** We usually only need the first 1–2 elements of the sorted list.

**REFINEMENT B** The sorting key is dominated by exchange_winning.

# THE SAFE ELIMINATION OF BOUNDS CHECKS

**TASK**

Sort a list of moves by quality, best-to-worst.

```
moves.sort_by_key(|m| {
  (exchange_winning(m), history(m)) // (bool, i32)
});
```

**REFINEMENT A** We usually only need the first 1–2 elements of the sorted list.

**REFINEMENT B** The sorting key is dominated by exchange_winning.

**REFINEMENT C** Evaluating exchange_winning is very costly.

# THE SAFE ELIMINATION OF BOUNDS CHECKS

```
// select the maximal element each time around the loop
while let Some((idx, max)) = moves.iter()
    .enumerate()
    .max_by_key(|(_, m)| history(m)) {
  // max is only the max-history move prior
  if !exchange_winning(max) { set_bad(max); continue; }
  emit(max);
  moves.swap(0, idx); //    :(
  moves = &mut moves[1..];
}
```

# THE SAFE ELIMINATION OF BOUNDS CHECKS

```
// select the maximal element each time around the loop
while let Some((idx, max)) = moves.iter()
    .enumerate()
    .max_by_key(|(_, m)| history(m)) {
  // max is only the max-history move prior
  if !exchange_winning(max) { set_bad(max); continue; }
  emit(max);
  moves.swap(0, idx); //    :(
  moves = &mut moves[1..];
}
```

Frustratingly, moves.swap(0, idx) emits a bounds-check for idx.

# THE SAFE ELIMINATION OF BOUNDS CHECKS

Ideally, we'd just write through our max reference — we could just mem::swap it with element zero.

```
while let Some(max) = moves.iter_mut().max_by_key(history) {
  if !exchange_winning(max) { set_bad(max); continue; }
  emit(max);
  if done() { break; }
  std::mem::swap(&mut moves[0], max);
  //             ^^^^^^^^^^^^^ fails to borrow-check
  moves = &mut moves[1..];
}
```

# THE SAFE ELIMINATION OF BOUNDS CHECKS

Ideally, we'd just write through our max reference — we could just mem::swap it with element zero.

```
while let Some(max) = moves.iter_mut().max_by_key(history) {
  if !exchange_winning(max) { set_bad(max); continue; }
  emit(max);
  if done() { break; }
  std::mem::swap(&mut moves[0], max);
  //             ^^^^^^^^^^^^^ fails to borrow-check
  moves = &mut moves[1..];
}
```

This doesn't work — max is uniquely borrowing moves, so "&mut moves[0]" fails to borrow-check.

# CELLS

*A mutable memory location.*

```
#[repr(transparent)]
pub struct Cell<T : ?Sized> {
  value : UnsafeCell<T>,
}

impl<T> Cell<T> {
  pub fn new(value : T) -> Cell<T>;
  pub fn get(&self) -> T where T : Copy;
  pub fn set(&self, value : T);
  //         ^^^^^ a shared borrow!
}
```

# THE SLICE-CELL TRICK

```
//  &mut [T] → &Cell<[T]> → &[Cell<T>]
let mut moves = Cell::from_mut(&mut moves).as_slice_of_cells();

while let Some(max) = moves.iter().max_by_key(|m| history(m.get())) {
  if !exchange_winning(max) { set_bad(max); continue; }
  emit(max);
  if done() { break; }
  Cell::swap(&moves[0], max);
  moves = &moves[1..];
}
```

Because we can make multiple *shared* borrows of moves, all is well in the world, and this routine emits no bounds-checks.

# THE TRACK MACRO

A lot goes on inside a chess engine!

```
// what's this value on average?
if cheap_heuristic(...) > 42 {
  // how often does this fire?
  if expensive_heuristic(...) {
    return;
  }
}
```

# THE TRACK MACRO

With macros, one can trace and aggregate statistics in an *ad-hoc* manner.

# THE TRACK MACRO

With macros, one can trace and aggregate statistics in an *ad-hoc* manner.

```
macro_rules! track {
  ($name:expr; $v:expr) => {{
    #[distributed_slice(TRACKED_VALUES)]
    static ENTRY : TrackedValue = TrackedValue::new(const { $name });
    let value = $v;
    ENTRY.record(value as i64);
    value
  }};
  ($v:expr) => {{ track!(stringify!($v); $v) }};
}
```

# DISTRIBUTED SLICE

The linkme crate provides the #[distributed_slice] macro, which is *incredible*.

```
#[distributed_slice]
static CONSTANTS : [i32];

#[distributed_slice(CONSTANTS)] // foo.rs
static A : i32 = 45;
#[distributed_slice(CONSTANTS)] // bar.rs
static B : i32 = 20;
#[distributed_slice(CONSTANTS)] // baz.rs
static C : i32 = 12;

// main.rs
println!("{:?}", CONSTANTS); // → [20, 45, 12]
```

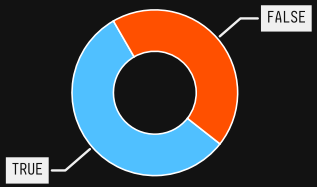
# THE TRACK MACRO

The track! macro behaves just like the identity function!

```
if track!(cheap_heuristic(...)) > 42 {
  if track!(expensive_heuristic()) {
    return;
  }
}
```

## RFP

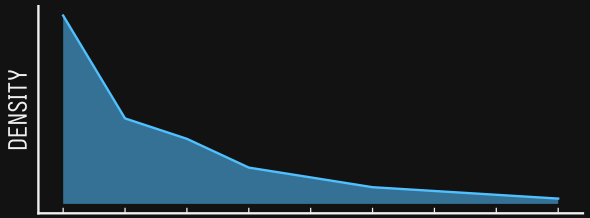
N = 18 964 497  μ = 0.6




## DEPTH

N = 18 837 802  μ = 2.8  |μ| = 2.8

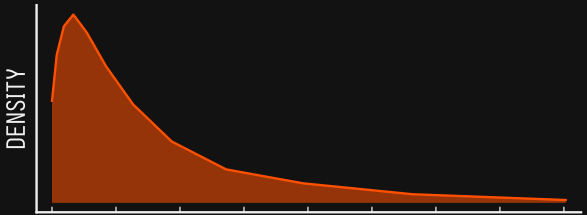
σ = 2.3  LO = 1  HI = 22




## (EVAL - BETA) / 3

N = 10 635 066  μ = 201.2  |μ| = 201.2

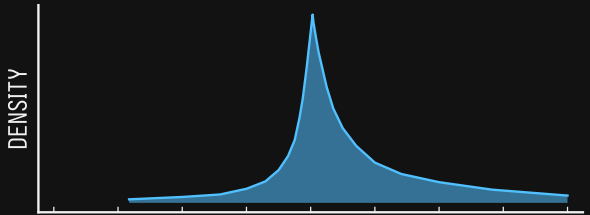
σ = 199.5  LO = 0  HI = 5658




## EVAL

N = 18 194 503  μ = 437.1  |μ| = 709.3

σ = 1548.5  LO = -5916  HI = 32398



# SHARED CACHE

*Or the "transposition table", in traditional parlance.*
*(The size of the transposition table is also known as the "hash size", despite the fact that the data structure is not a hash map but a cache.)*

The cache is the only means by which threads communicate during search.

```rust
#[repr(C)]
struct CacheEntry {
  tag   : u16,
  move  : Option<Move>,
  eval  : i16,
  score : i16,
  depth : u8,
  meta  : PackedMetadata,
}
```

```rust
#[repr(C, align(32))]
struct CacheSet {
  entries : [CacheEntry; 3],
  padding : [u8; 2],  // to avoid UB
}

struct Cache {
  sets : Vec<CacheSet>
}
```

# SHARED CACHE

```rust
#[repr(C)]
struct CacheEntry {
  tag   : u16,
  move  : Option<Move>,
  eval  : i16,
  score : i16,
  depth : u8,
  meta  : PackedMetadata,
}
```

```rust
#[repr(C, align(32))]
struct CacheSet {
  entries : [CacheEntry; 3],
  padding : [u8; 2],  // to avoid UB
}

struct Cache {
  sets : Vec<CacheSet>
}
```

```rust
fn derive_index_tag(num_sets : u64, key : u64) -> (usize, u16)
```

# ACCESSING THE CACHE

We've two options:

- Guard access with locks.
  *The compiler can optimize our lookup into a single 256-bit load if that's better, but the performance penalty from obtaining a lock is unacceptable (or at least unnecessary, however slight it may be — hence unacceptable).*

# ACCESSING THE CACHE

We've two options:

- Guard access with locks.
  *The compiler can optimize our lookup into a single 256-bit load if that's better, but the performance penalty from obtaining a lock is unacceptable (or at least unnecessary, however slight it may be — hence unacceptable).*
- Use atomics.
  *Four 64-bit loads, which appears to be as fast as anything else in practice.*

# ACCESSING THE CACHE ATOMICALLY

```
pub fn lookup(&self, key : u64) -> CacheEntry
{
  let (index, tag) = derive_index_tag(self.num_sets, key);
  let b0 = self.sets[index].block[0].load(Relaxed);
  let b1 = self.sets[index].block[1].load(Relaxed);
  let b2 = self.sets[index].block[2].load(Relaxed);
  let b3 = self.sets[index].block[3].load(Relaxed);
  let cache_set = unsafe {
    transmute::<[u64; 4], CacheSet>([b0, b1, b2, b3])
  };
  for entry in cache_set.entries {
    if entry.tag == tag { return entry; }
  }
  return CacheEntry::ZERO;
}
```

# THE DUALITY OF MEM

**FEARLESS CONCURRENCY**

Rust is a great systems language because of its powerful safe primitives (like Arc, Condvar, Mutex, OnceLock, and so on).

# THE DUALITY OF MEM

**FEARLESS CONCURRENCY**

Rust is a great systems language because of its powerful safe primitives (like Arc, Condvar, Mutex, OnceLock, and so on).

**FEARFUL CONCURRENCY**

Rust is *also* a great systems language because it takes you seriously as a user and lets you create your own primitives*!* (After all, the standard library is largely implemented in ordinary Rust.)

[ACHTUNG]

*When you use unsafe, the compiler will not check that soundness is upheld; you are responsible for doing so.*

DU HAST VERSPROCHEN

REMEMBER YOUR PROMISE

INJUNCTION

READ MARA'S BOOK.

# TORN READS

The widest atomic type is AtomicU64 (eight bytes), but an entry is ten bytes and a set is thirty(-two) bytes, so neither can be accessed atomically.

Reading an entry requires multiple loads, and in theory, if another thread is issuing stores at the same time, part of the data we read might be from the ongoing write and part of the data might be from a previous write.

**NOTE**

Even if torn reads weren't possible, since we store too few tag bits to recover the key, we have to handle entries that appear valid but are in fact invalid. (There's also the very small chance of key collisions.)

# TORN READS

We don’t have to mitigate this, amazingly enough.

- It’s safe because all bit patterns are valid.
- The performance* impact is small (but enough that it’s worth countering).

*Why might this be so? We already expect the engine to be mistaken in its evaluätion of positions, and reading an incorrect entry is simply another misevaluätion, which merely degrades the quality of analysis but does not threaten algorithmic correctness.*

We can reject the vast majority of incorrect lookups by checking the legality of the stored move. *This guards against torn reads, tag aliasing, and key collisions, but imperfectly — a move might be legal in two colliding positions.*

* Game-playing strength measured in Elo rating points.

# ATOMIC PER BYTE

RFC **3301** proposes an “atomic memcpy” that permits tearing.

*It might also provide the means to read uninitialized memory, allowing more performant implementations of data structures like sparse sets.*

# RUST THROUGH THE AGES

We both started using Rust in January 2021. What's changed since then?

- 1.51 const generics · {integer}::unsigned_abs
- 1.58 identifiers in format strings
- 1.59 inline assembly · destructuring assignments
- 1.60 {integer}::abs_diff
- 1.65 let-else statements · break in labelled blocks
- 1.79 inline const expressions
- 1.82 raw pointer syntax
- 1.87 {integer}::is_multiple_of · {integer}::midpoint
- 1.88 let chains
- 1.92 Box::new_zeroed
- 1.93 [T]::as_array

△

{ 终 }

THIS SLIDE
INTENTIONALLY
LEFT BLANK.