CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

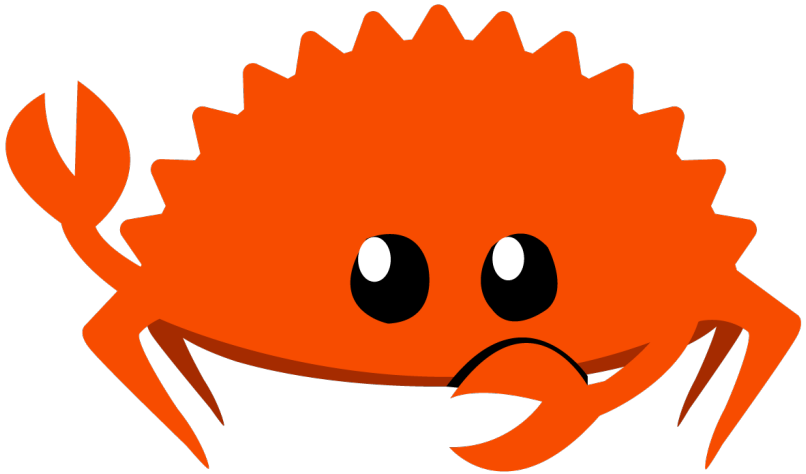
ETH zürich

An Overview of Rust

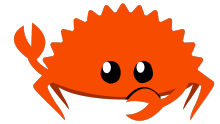
Webinar
Michal Sudwoj
26.06.2020

An Overview of Rust
Webinar
Michal Sudwoj
26.06.2020

Questions



Questions



1. Ferris the Crab
2. Use Zoom Q&A feature, or use the ”Raise pincer” button

What is Rust?

- 2006: personal project of Graydon Hoare
- 2010: announced by Mozilla Research
- 2015: v1.0
- 2016–2020: ”most loved programming language”
- 6 week release cycle

What is Rust?
- 2006: personal project of Graydon Hoare
- 2010: announced by Mozilla Research
- 2015: v1.0
- 2016–2020: ”most loved programming language”
- 6 week release cycle

1. Used in Firefox (Servo engine), Redox OS
2. Sponsored by Microsoft, Mozilla, Amazon, Google
3. Ferrous Systems GmBH: Sealed Rust - Rust in Safety Critical Domain

Installation

Using `spack`:

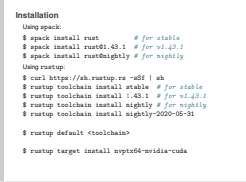
```
$ spack install rust          # for stable
$ spack install rust@1.43.1   # for v1.43.1
$ spack install rust@nightly  # for nightly
```

Using `rustup`:

```
$ curl https://sh.rustup.rs -sSf | sh
$ rustup toolchain install stable  # for stable
$ rustup toolchain install 1.43.1  # for v1.43.1
$ rustup toolchain install nightly # for nightly
$ rustup toolchain install nightly-2020-05-31

$ rustup default <toolchain>

$ rustup target install nvptx64-nvidia-cuda
```

1. `cd <project-dir>; rustup override set <toolchain>`
2. `rustup target install nvptx64-nvidia-cuda`

Hello, World!

```
$ cargo new --bin hello_world
$ cd hello_world
$ cat src/main.rs
fn main() {
    println!("Hello, World!");
}
$ cargo run
Hello, World!
```


Hello, World!
$ cargo new --bin hello_world
$ cd hello_world
$ cat src/main.rs
fn main() {
    println!("Hello, World!");
}
$ cargo run
Hello, World!

1. `cargo new --bin` for example binary
2. `cargo new --lib` for example library
3. `cargo +nightly` for version change

Features

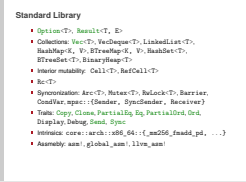
- C++-like syntax
- non-mutable and private by default
- `struct`s (incl. tuple structs)
- `enum`s (tagged unions)
- `trait`s
- `mod`ules
- `Iterator`s (think C++ ranges / Python for-loops)
- UTF-8 support
- generics
- **hygenic** macros
- attributes incl. *`#[derive(...)]`*
- `Result<T, E>` instead of exceptions
- `rustc` LLVM backend

Features
- C++-like syntax
- non-mutable and private by default
- structs (incl. tuple structs)
- enums (tagged unions)
- traits
- modules
- Iterators (think C++ ranges / Python for-loops)
- UTF-8 support
- generics
- hygenic macros
- attributes incl. #[derive(...)]
- Result<T, E> instead of exceptions
- rustc LLVM backend

1. `cfg` attributes for architectures/os/...
2. `repr` for different `struct` and `enum` representations

Standard Library

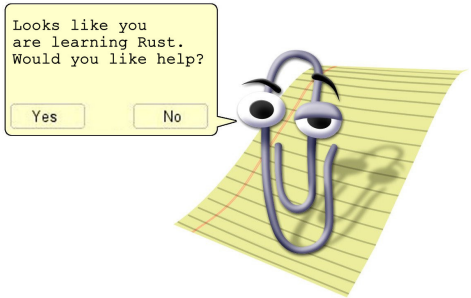
- `Option<T>`, `Result<T, E>`
- Collections: `Vec<T>`, `VecDeque<T>`, `LinkedList<T>`, `HashMap<K, V>`, `BTreeMap<K, V>`, `HashSet<T>`, `BTreeSet<T>`, `BinaryHeap<T>`
- Interior mutability: `Cell<T>`, `RefCell<T>`
- `Rc<T>`
- Syncronization: `Arc<T>`, `Mutex<T>`, `RwLock<T>`, `Barrier`, `CondVar`, `mpsc::{Sender, SyncSender, Receiver}`
- Traits: `Copy`, `Clone`, `PartialEq`, `Eq`, `PartialOrd`, `Ord`, `Display`, `Debug`, `Send`, `Sync`
- Intrinsics: `core::arch::x86_64::{_mm256_fmadd_pd, ...}`
- Assmebly: `asm!`, `global_asm!`, `llvm_asm!`

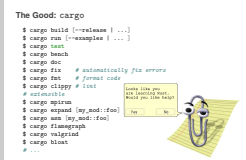


1. eg. `Option` optimized for references (null pointer = `None`)
2. Interior Mutatbility: mutable via `&T`, runtime borrow checking, `!Sync`
3. `Send`: type not sendable to another thread
4. `Sync`: not shareable between thread (`&T: !Send`)

The Good: `cargo`

```
$ cargo build [--release | ...]
$ cargo run [--examples | ... ]
$ cargo test
$ cargo bench
$ cargo doc
$ cargo fix     # automatically fix errors
$ cargo fmt     # format code
$ cargo clippy # lint
# extensible
$ cargo mpirun
$ cargo expand [my_mod::foo]
$ cargo asm [my_mod::foo]
$ cargo flamegraph
$ cargo valgrind
$ cargo bloat
# ...
```





1. critcmp `https://github.com/BurntSushi/critcmp`
2. cargo-benchcmp
   `https://github.com/BurntSushi/cargo-benchcmp`

The Good: `build.rs`

- custom build script (written in Rust!)
- crates to call `cc`, `cmake`, generate C bindings (`bindgen`)
- compile-time embedding and environment access

Demo: `https://git.cscs.ch/msudwoj/rust-in-hpc/-/tree/master/interop/rust2cpp`

The Good: `build.rs`
- custom build script (written in Rust!)
- crates to call `cc`, `cmake`, generate C bindings (`bindgen`)
- compile-time embedding and environment access

Demo: `https://git.cscs.ch/msudwoj/rust-in-hpc/-/tree/master/interop/rust2cpp`

The Good: Ecosystem

- `crates.io`
  - package registry
  - custom registries
- `docs.rs`

The Good: Ecosystem
crates.io
package registry
custom registries
docs.rs

1. Properly semversion'ed
2. commit `Cargo.lock` to freeze dependencies
3. high quality packages
4. some used in `rustc`
5. Supports multiple crate versions in the same program/library
6. Cargo.toml
7. patches, overrides, …
8. build profiles, eg. build dependencies always in release mode

The Good: Ecosystem

- `ndarray`
- `blas`, `cblas`, `lapack`, `lapacke`, `netlib-src`, `openblas-src`, `intel-mkl-src`
- `num`
- `lazy_static`
- `serde`
- `rayon`, `crossbeam`, `parking_lot`
- `rand`
- `libc`
- `bitflags`
- `proc-macro2`, `syn`, `quote`
- `hdf5`
- `mpi`
- `smallvec`
- `tokio`, `async-std`
- `reqwest`
- `criterion`, `bencher`
- `thiserror`, `anyhow`

The Good: Ecosystem
ndarray
blas, cblas, lapack, lapacke, netlib-src, openblas-src, intel-mkl-src
num
lazy_static
serde
rayon, crossbeam, parking_lot
rand
libc
bitflags
proc-macro2, syn, quote
hdf5
mpi
smallvec
tokio, async-std
reqwest
criterion, bencher
thiserror, anyhow

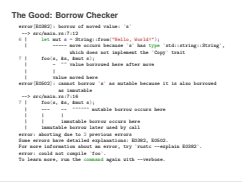
The Good: Borrow Checker

```
$ cat src/main.rs
fn foo<T>(x: T, y: &T, z: &mut T) {}
fn main() {
  let mut s = String::from("Hello, World!");
  foo(s, &s, &mut s);
}
$ cargo build
```


The Good: Borrow Checker
$ cat src/main.rs
fn foo<T>(x: T, y: &T, z: &mut T) {}
fn main() {
let mut s = String::from("Hello, World!");
foo(s, &s, &mut s);
}
$ cargo build

The Good: Borrow Checker

```
error[E0382]: borrow of moved value: `s`
 --> src/main.rs:7:12
6 |     let mut s = String::from("Hello, World!");
  |         ----- move occurs because `s` has type `std::string::String`,
  |               which does not implement the `Copy` trait
7 |     foo(s, &s, &mut s);
  |         -  ^^ value borrowed here after move
  |         |
  |         value moved here
error[E0502]: cannot borrow `s` as mutable because it is also borrowed
              as immutable
 --> src/main.rs:7:16
7 |     foo(s, &s, &mut s);
  |     ---    --  ^^^^^^ mutable borrow occurs here
  |     |      |
  |     |      immutable borrow occurs here
  |     immutable borrow later used by call
error: aborting due to 2 previous errors
Some errors have detailed explanations: E0382, E0502.
For more information about an error, try `rustc --explain E0382`.
error: could not compile `foo`.
To learn more, run the command again with --verbose.
```

1. Spans
2. `rustc --explain EXXXX`

The Good: String printing and formatting

```
$ cat src/main.rs
```

```rust
fn main() {
  let e = std::env::VarError::NotPresent;
  println!("Display: {}", e);
  println!("Debug:   {:?}", e);
}
```

```
$ cargo run
Display: environment variable not found
Debug:   NotPresent
```

On your own types:

```rust
#[derive(Debug)]
struct Foo { /* ... */ }
impl std::fmt::Display for Foo { /* ... */ }
```


The Good: String printing and formatting
$ cat src/main.rs
fn main() {
let e = std::env::VarError::NotPresent;
println!("Display: {}", e);
println!("Debug: {:?}", e);
}
$ cargo run
Display: environment variable not found
Debug: NotPresent
On your own types:
#[derive(Debug)]
struct Foo { /* ... */ }
impl std::fmt::Display for Foo { /* ... */ }

The Good: Pattern matching

```rust
use std::num::NonZeroU8;
use rand::random;
let x = NonZeroU8::new(random::<u8>());
if let Some(x) = x {
  match x {
    1       => print!("x = 1"),
    2..10   => print!("2 <= x < 10"),
    10..=20 => print!("10 <= x <= 20"),
    _       => print!("20 < x"),
  }
}
```


The Good: Pattern matching


```rust
use std::num::NonZeroU8;
use rand::random;
let x = NonZeroU8::new(random::<u8>());
if let Some(x) = x {
  match x {
    1       => print!("x = 1"),
    2..10   => print!("2 <= x < 10"),
    10..=20 => print!("10 <= x <= 20"),
    _       => print!("20 < x"),
  }
}
```

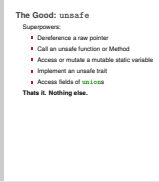

1. Also: guards, bindings (@)

The Good: `unsafe`

Superpowers:

- Dereference a raw pointer
- Call an unsafe function or Method
- Access or mutate a mutable static variable
- Implement an unsafe trait
- Access fields of `union`s

Thats it. Nothing else.



In particular, some things are still UB:

- data races
- dereferencing dangling or unaligned pointers
- uninitialized memory (use `std::mem::MaybeUninit<T>`)
- …

see `https://doc.rust-lang.org/reference/behavior-considered-undefined.html`

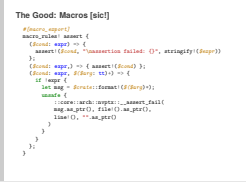
Inversely, things that are still safe, but generally undesireable:

- deadlocks
- memory leaks
- exiting without calling destructors
- integer overflow

see `https://doc.rust-lang.org/reference/behavior-not-consi`

The Good: Macros [sic!]

```
#[macro_export]
macro_rules! assert {
  ($cond: expr) => {
    assert!($cond, "\nassertion failed: {}", stringify!($expr))
  };
  ($cond: expr,) => { assert!($cond) };
  ($cond: expr, $($arg: tt)+) => {
    if !expr {
      let msg = $crate::format!($($arg)*);
      unsafe {
          ::core::arch::nvptx::__assert_fail(
          msg.as_ptr(), file!().as_ptr(),
          line!(), "".as_ptr()
        )
      }
    }
  };
}
```

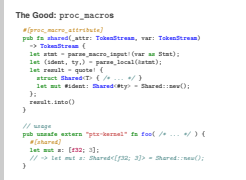



1. ”macros-by-example”
2. hygenic
3. on syntactic level (token trees)

The Good: proc_macros

```rust
#[proc_macro_attribute]
pub fn shared(_attr: TokenStream, var: TokenStream)
  -> TokenStream {
  let stmt = parse_macro_input!(var as Stmt);
  let (ident, ty,) = parse_local(&stmt);
  let result = quote! {
    struct Shared<T> { /* ... */ }
    let mut #ident: Shared<#ty> = Shared::new();
  };
  result.into()
}

// usage
pub unsafe extern "ptx-kernel" fn foo( /* ... */ ) {
  #[shared]
  let mut s: [f32; 3];
  // -> let mut s: Shared<[f32; 3]> = Shared::new();
}
```

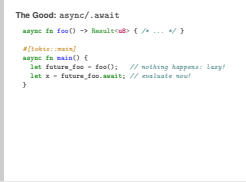



1. Rust function that transforms the `TokenStream` at compile time
2. DSLs! Especially using eg. `combine_proc_macro`

The Good: `async/.await`

```rust
async fn foo() -> Result<u8> { /* ... */ }

#[tokio::main]
async fn main() {
  let future_foo = foo();   // nothing happens: lazy!
  let x = future_foo.await; // evaluate now!
}
```

1. `Futures` in rust are lazy!
2. They get evaluated only upon being `.await`ed
3. Zero-cost
4. Requires executor

The Bad(?): Function overloading

```
impl String {
  pub fn new() -> Self { /* ... */ }
  pub fn with_capacity(capacity: usize) -> Self { /* ... */ }
  pub fn from_raw_parts(
    buf: *mut u8, length: usize, capacity: usize
  ) -> Self { /* ... */ }
  pub fn from_utf8(vec: Vec<u8>)
    -> Result<Self, FromUtf8Error> { /* ... */ }
  pub fn from_utf8_unchecked(bytes: Vec<u8>)
    -> Self { /* ... */ }
  pub fn from_utf8_lossy(v: &[u8]) -> Cow<str> { /* ... */ }
  pub fn from_utf16(v: &[u16])
    -> Result<Self, FromUtf16Error> { /* ... */ }
  pub fn from_utf16_lossy(v: &[u16]) -> Self { /* ... */ }
}
impl From<&'_ String>   for String { /* ... */ }
impl From<&'_ str>      for String { /* ... */ }
impl From<Box<str>>     for String { /* ... */ }
impl From<Cow<'a, str>> for String { /* ... */ }
// ...
```


1. By design choice
2. https://internals.rust-lang.org/t/justification-for-rust-not-supporting-function-overloading-directly/7012
3. Can be alleviated by traits, but is considered an anti-pattern (other than eg. From)

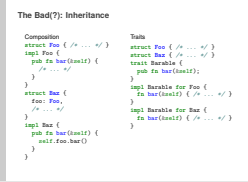
The Bad(?): Inheritance

Composition

```
struct Foo { /* ... */ }
impl Foo {
  pub fn bar(&self) {
    /* ... */
  }
}
struct Baz {
  foo: Foo,
  /* ... */
}
impl Baz {
  pub fn bar(&self) {
    self.foo.bar()
  }
}
```

Traits

```
struct Foo { /* ... */ }
struct Baz { /* ... */ }
trait Barable {
  pub fn bar(&self);
}
impl Barable for Foo {
  fn bar(&self) { /* ... */ }
}
impl Barable for Baz {
  fn bar(&self) { /* ... */ }
}
```

1. Can also do blanket implementation in Barable

The Bad(?): Unstable features

- `#![feature(const_generics)]`
- `#![feature(specialization)]`
- `#![feature(const_fn)]`
- `#![feature(fn_traits, unboxed_closures)]`

The Bad(?): Unstable features
- `#![feature(const_generics)]`
- `#![feature(specialization)]`
- `#![feature(const_fn)]`
- `#![feature(fn_traits, unboxed_closures)]`

The Bad(?): Tier 3 CUDA support

As of May 15th 2020: Tier 2!

Works, but …

- (only on nightly)
- lots of boilerplate
- `unsafe` / unsound
- can still call (C++) CUDA functions using FFI

The Bad(?): Tier 3 CUDA support
As of May 15th 2020: Tier 2!
Works, but …
(only on nightly)
lots of boilerplate
unsafe / unsound
can still call (C++) CUDA functions using FFI

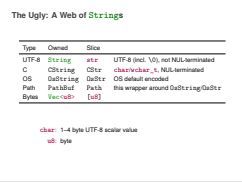
1. ”Hacky”: requires `ptx-linker`, which fixes link issues
2. rustc is unaware of CUDA threading model -> cannot use slices/borrowck in CUDA device code
3. No support for `__shared__` or `__const__`
4. Tier 1: guaranteed to work (automated testing): currently i686 and x68_64
5. Tier 2: guaranteed to build: a lot
6. Tier 3: preliminary support

The Ugly: A Web of `Strings`

Type	Owned	Slice	
UTF-8	`String`	`str`	UTF-8 (incl. \0), not NUL-terminated
C	`CString`	`CStr`	`char`/`wchar_t`, NUL-terminated
OS	`OsString`	`OsStr`	OS default encoded
Path	`PathBuf`	`Path`	this wrapper around `OsString`/`OsStr`
Bytes	`Vec<u8>`	`[u8]`	

`char`: 1–4 byte UTF-8 scalar value

`u8`: byte



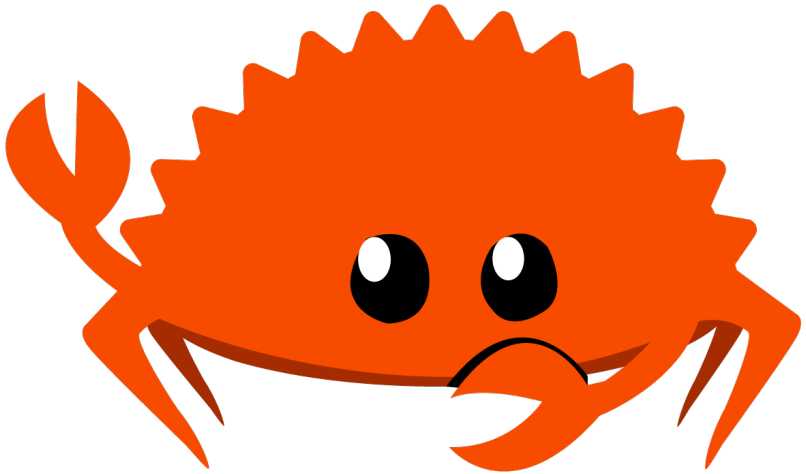
Conclusion

Rust…

- … takes getting used to
- … has great tooling
- … in still missing some crucial features (on stable)
  - const generics
  - specialization
- … has zero-cost abstractions
- … has good interop with C (and other languages)
- … helps you avoid memory errors
- … makes shared memory parallelism hard

Conclusion
Rust…
… takes getting used to
… has great tooling
… in still missing some crucial features (on stable)
const generics
specialization
… has zero-cost abstractions
… has good interop with C (and other languages)
… helps you avoid memory errors
… makes shared memory parallelism hard

Questions?



Questions?

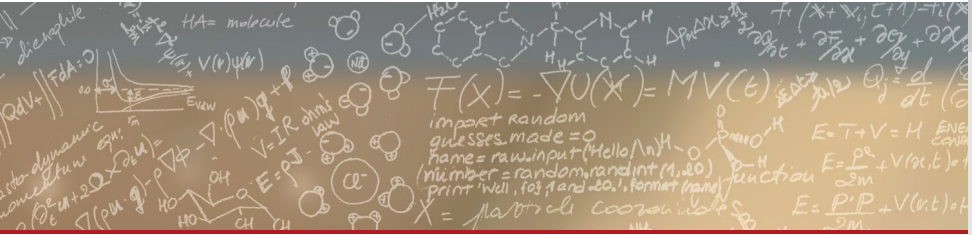
Hit me up!

- https://git.cscs.ch/msudwoj/rust-in-hpc
- Slack: @Michal Sudwoj
- Email: msudwoj@student.ethz.ch

Hit me up!
- https://git.cscs.ch/msudwoj/rust-in-hpc
- Slack: @Michal Sudwoj
- Email: msudwoj@student.ethz.ch

CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Thank you for your attention.

Thank you for your attention.

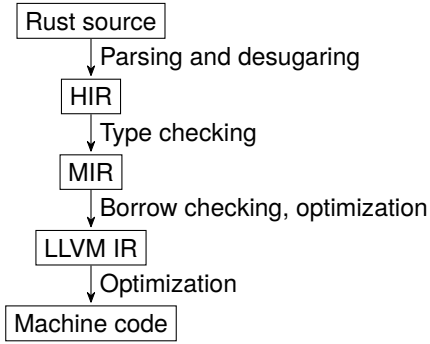
References

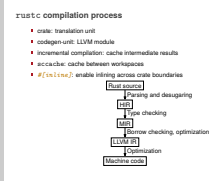
- The Rust Programming Language: `https://doc.rust-lang.org/book/title-page.html`
- The Rustonomicon: `https://doc.rust-lang.org/nomicon/index.html`
- The Cargo Book: `https://doc.rust-lang.org/cargo/index.html`
- Rust Forge: `https://forge.rust-lang.org/index.html`

References
- The Rust Programming Language: https://doc.rust-lang.org/book/title-page.html
- The Rustonomicon: https://doc.rust-lang.org/nomicon/index.html
- The Cargo Book: https://doc.rust-lang.org/cargo/index.html
- Rust Forge: https://forge.rust-lang.org/index.html

`rustc` compilation process

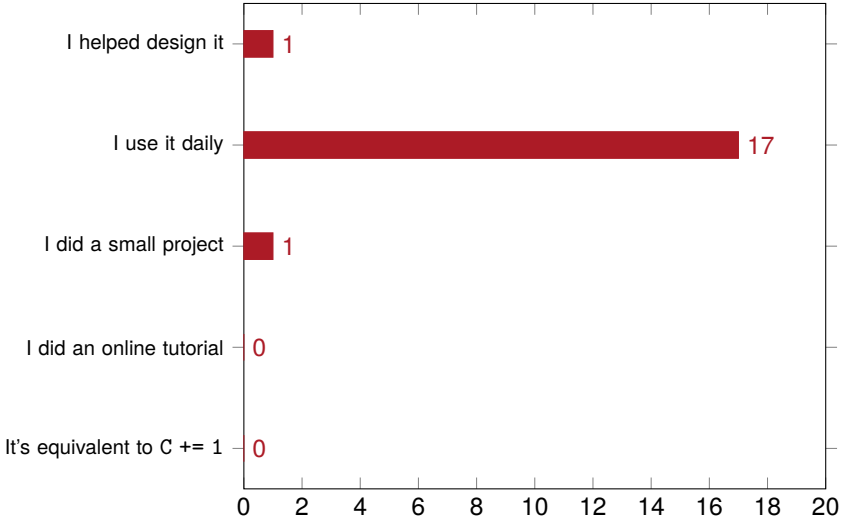
- crate: translation unit
- codegen-unit: LLVM module
- incremental compilation: cache intermediate results
- `sccache`: cache between workspaces
- *`#[inline]`*: enable inlining across crate boundaries

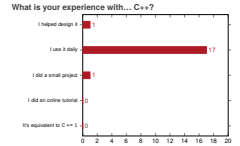




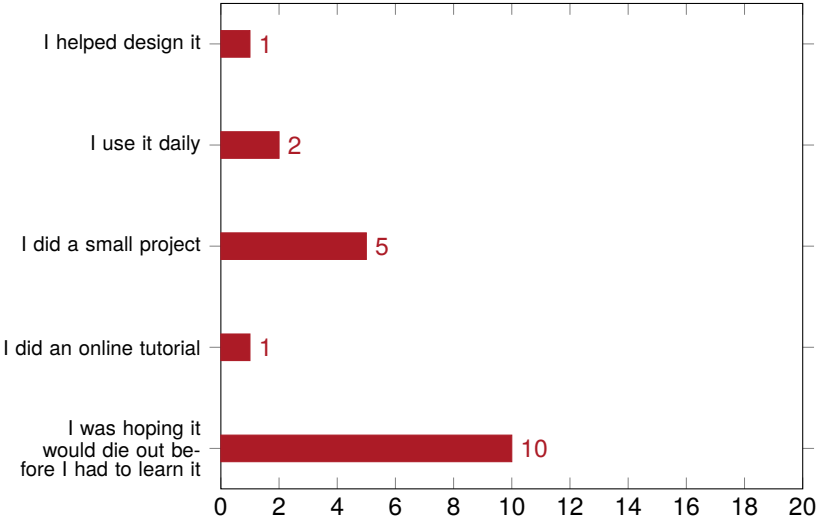
1. codegen-unit allows parallel compilation
2. incremental compilation caches intermediate results and only recompiles if necessary

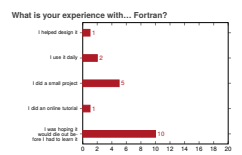
What is your experience with… C++?



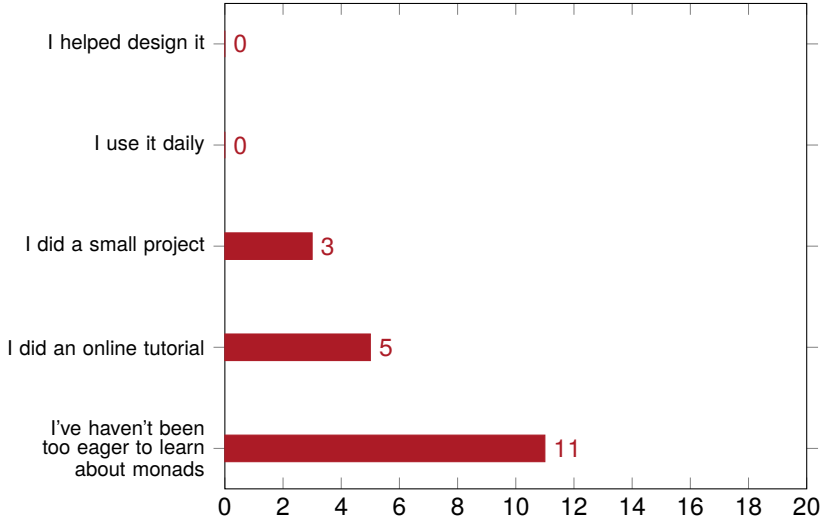


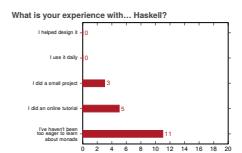
What is your experience with… Fortran?



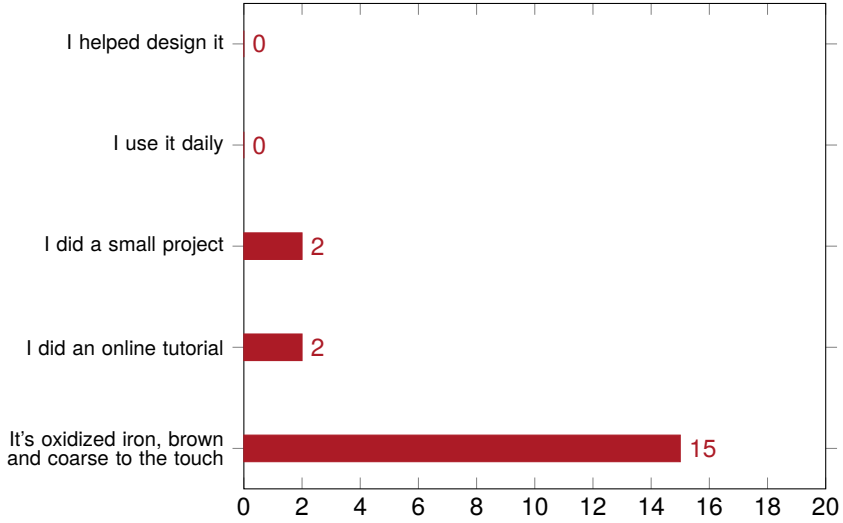


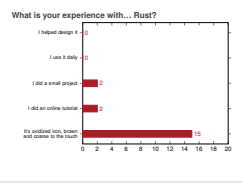
What is your experience with… Haskell?



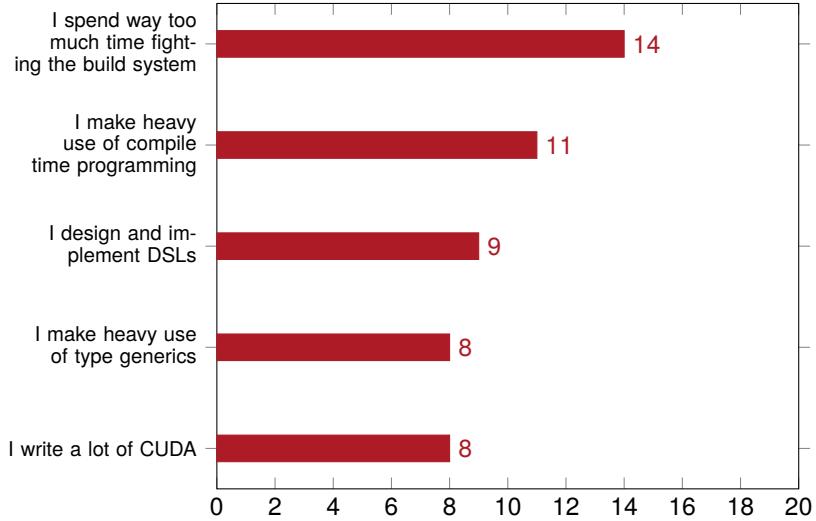


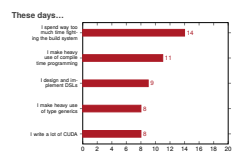
What is your experience with… Rust?



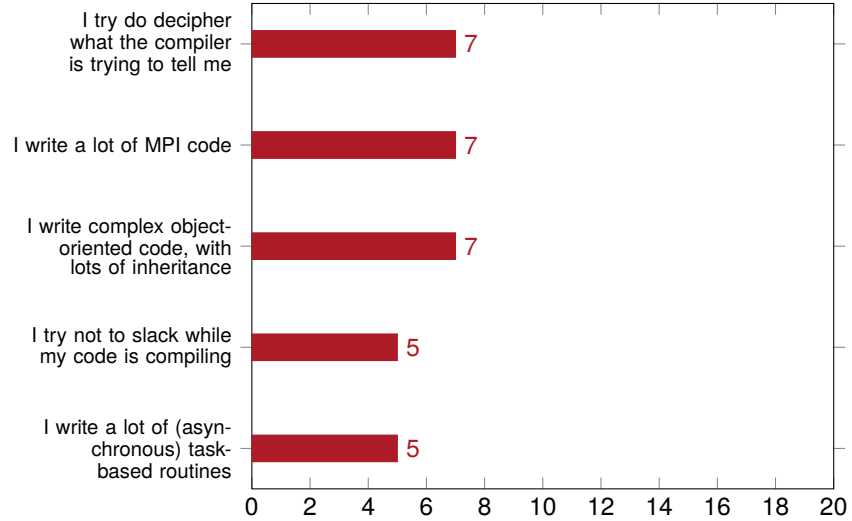


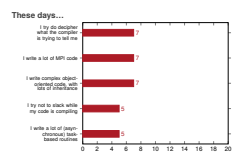
These days…


I spend way too much time fighting the build system
14
I make heavy use of compile time programming
11
I design and implement DSLs
9
I make heavy use of type generics
8
I write a lot of CUDA
8
0 2 4 6 8 10 12 14 16 18 20

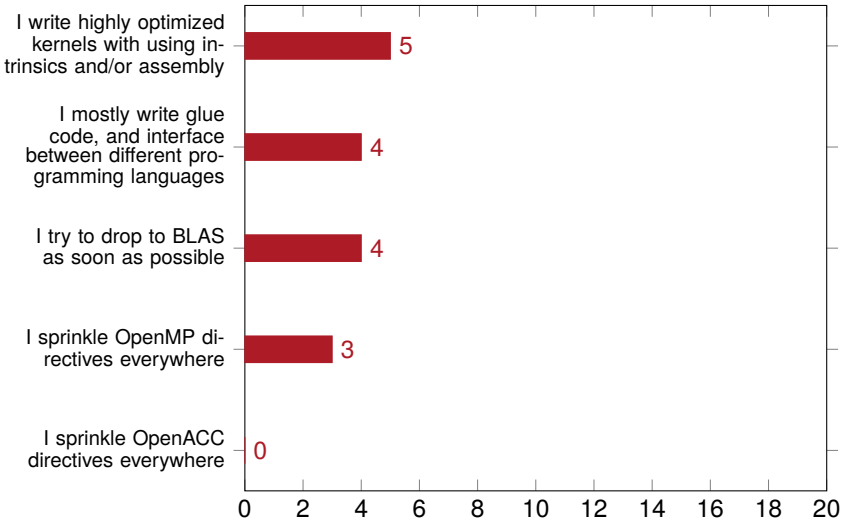


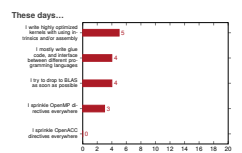
These days…


I try do decipher what the compiler is trying to tell me
7
I write a lot of MPI code
7
I write complex object-oriented code, with lots of inheritance
7
I try not to slack while my code is compiling
5
I write a lot of (asyn-chronous) task-based routines
5
0 2 4 6 8 10 12 14 16 18 20

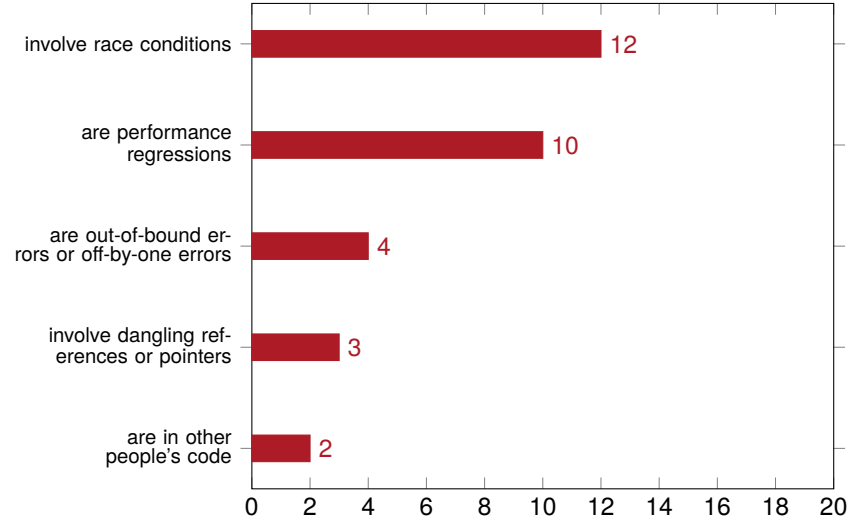

These days…
I try do decipher what the compiler is trying to tell me
7
I write a lot of MPI code
7
I write complex object-oriented code, with lots of inheritance
7
I try not to slack while my code is compiling
5
I write a lot of (asyn-chronous) task-based routines
5
0 2 4 6 8 10 12 14 16 18 20

These days…

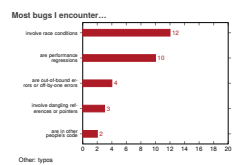

I write highly optimized kernels with using intrinsics and/or assembly
5
I mostly write glue code, and interface between different programming languages
4
I try to drop to BLAS as soon as possible
4
I sprinkle OpenMP directives everywhere
3
I sprinkle OpenACC directives everywhere
0
0 2 4 6 8 10 12 14 16 18 20



Most bugs I encounter…



Other: typos



In the webinar, I would like to see…