Pragmatic ways of using Rust in your data project

Christopher Prohm
PyCon.DE / PyData Berlin 2023

@chmp@hachyderm.io

Code + Slides: https://github.com/chmp/PyConDE23

Motivation

(+) Interactive exploration:

- data analysis
- model building
- writing tests
- ...

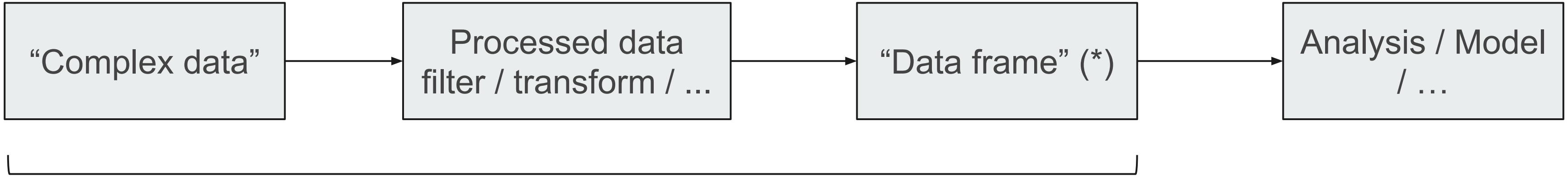
(-) Only fast if data fits NumPy, Pandas, ...
& it's really easy to fall off the performance cliff

Strategy: Do not replace Python

- Sprinkle Rust in for performance
- Rust & Python have complementary strengths

Logo: python.org

How to turn raw data to “Python-compatible” data? (fast)



(*) the Arrow format expands the range of what can be stored in data frame
[arrow.apache.org]

Examples:

1. Parsing bank statement PDFs
2. Converting the “Spotify Million Playlist Dataset” into a data frame
   [aicrowd.com/challenges/spotify-million-playlist-dataset-challenge]

Using Rust in your data project

WARNING: Rust Code ahead

What I like about Rust

Performance & memory efficiency

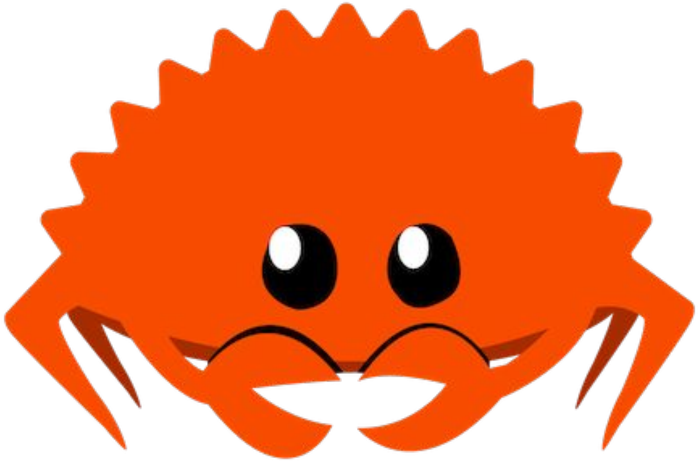
Easy to integrate into other runtimes

- No garbage collector, no runtime
- Built to interface with C code

Well designed features fit together into high-level interfaces

- Macros (“Code generation”)
- Traits (“Interfaces”)
- Type inference
- Sum-types (“Unions”)
- …

Great tooling & documentation!



[rustacean.net]

Python + Rust: PyO3 - the gold standard

PyO3 [pyo3.rs] allows to build Pythonic interfaces in Rust (*)

Some Python libraries are fully written in Rust

Leverages Rust features to great effect:

- Macros for code generation
- Traits ("Interfaces")
- Type inference

```rust
use pyo3::prelude::*;

#[pyfunction]
fn double(x: i32) -> i32 {
    x * 2
}

#[pymodule]
fn my_extension(py: Python<'_>, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(double, m)?)?;
    Ok(())
}
```

Code: pyo3.rs

The hacky / easy alternative: building custom command line tools

(*) See also Rusty Python: A Case Study by Robin Raymond

Basic strategy (1): build custom CLI tools

CLI tool with JSON in, JSON out:

```
> echo '{"value": 21}' | io-patterns-double.exe
{"value":42}
```

Using from Python

```
import subprocess

res = subprocess.run(
    ["./io-patterns-double.exe"],
    encoding="utf-8",
    capture_output=True, check=True,
    input=inp,
)
out = res.stdout
```

Strings in Rust are UTF-8 encoded

JSON: lingua franca of data exchange

Potential alternative: binary encoding
(e.g., bincode, Python code can be generated with serde-generate)

How to encode the input / decode the output?

Logo: python.org

Basic strategy (2): types + codegen in Python

```python
from dataclasses import dataclass

@dataclass
class Input:
    value: int

@dataclass
class Output:
    value: int
```

cattrs[catt.rs]: simple & configurable way to convert between Python types and JSON compatible types

Simple cases require no custom code

No standardized encoding for unions

```python
inp = Input(value=21)
inp = cattrs.unstructure(inp)
inp = json.dumps(inp)
```

unstructure( )

Python object

JSON compatible objects

structure( )

```python
out = json.loads(out)
out = cattrs.structure(out, Output)
out
```

Basic strategy (3): types + codegen in Rust

```rust
use serde::{Serialize, Deserialize};

#[derive(Deserialize)]
struct Input {
    value: i64,
}

#[derive(Serialize)]
struct Output {
    value: i64,
}
```

```rust
let input: Input = serde_json::from_reader(std::io::stdin())?;

let output = Output {
    value: 2 * input.value,
};

serde_json::to_writer(std::io::stdout(), &output)?;
```

Serde [serde.rs] : de-facto Rust standard for (de)serialization

Rely on code generation via macros

PyO3 vs. CLI Tools

PyO3 [pyo3.rs]

- \+ High level wrapper around the Python C-API
  - rich conversions between Python & Rust
- \+ Maximum efficiency
  - shared objects
  - minimal call overhead
- \- Requires build & installation step
- \- No reloading (Python limitation)

Custom CLI tool

- \+ Easy to get started:
  - No Rust / PyO3 specific concepts to learn
  - Easy to debug
- \+ Easy to build & distribute
- \- Not super efficient
- \- No shared objects

Code can still be refactored into PyO3

Case study: Parsing PDFs

Performance of PDF parsing

Girokonto		
Buchungstag **Valuata**	**Auftraggeber / Empfänger** **IBAN/BIC**	Ausgang Eingang
Alter Saldo		+XXXX,00
08.02.2021 08.02.2021	Restaurant XYZ	-40,00
10.03.2021 10.03.2021	Supermarkt XYZ	-30,00
...		
Neuer Saldo		+XXXX,00

PDF is a sequence of commands

```
Td 1.0 2.0
Tj "Alter Saldo"

Td 1.0 3.0
Tj "08.02.2021"
```

Needs to be interpreted

```
def do_Td(self, tx: PDFStackT, ty: PDFStackT):
    """Move text position"""

def do_Tj(self, s: PDFStackT):
    """Show text"""
...
```

Code from pdfminer.six

Extracting structured information

Girokonto		
Buchungstag Valuata	Auftraggeber / Empfänger IBAN/BIC	Ausgang Eingang
Alter Saldo		+XXXX,00
08.02.2021 08.02.2021	Restaurant XYZ	-40,00
10.03.2021 10.03.2021	Supermarkt XYZ	-30,00
…		
Neuer Saldo		+XXXX,00

```
(1.0, 2.0, "Alter Saldo")
(1.0, 3.0, "08.02.2021")
(1.0, 4.0, "08.02.2021")
(12.0, 3.0, "-40,00")
(5.0, 3.0, "Restaurant XYZ")
…
```

```
[
  date(2021, 2, 8), -40.0),
  date(2021, 3, 10), -30.0),
  ...,
]
```

Step 1:
Extract position
+ text from PDF

Step 2:
1. Find header / footer of transactions
2. Group blocks
3. Parse dates, amounts

Implementation

17 lines Python
using pdfminer.six

106 lines Python

122 lines Rust CLI Tool
5 lines Python
using pdf-extract

106 lines Python

```
> pdf-parser.exe statement_2023-03-01.pdf
{"number":1,
 "words": [
   {"x": 1.0, "y": 2.0, "text": "..."},
   ...
 ]
}
```

For one year of statements:

25.1s

11 x faster end to end

2.3s

pdf-extract less mature than pdfminer.six

- e.g., content of the first page is not parsed

speed up depends on PDF: more complex -> bigger effect

Logos: foundation.rust-lang.org, python.org

Case study: processing JSON files

The Arrow revolution (*) [arrow.apache.org]

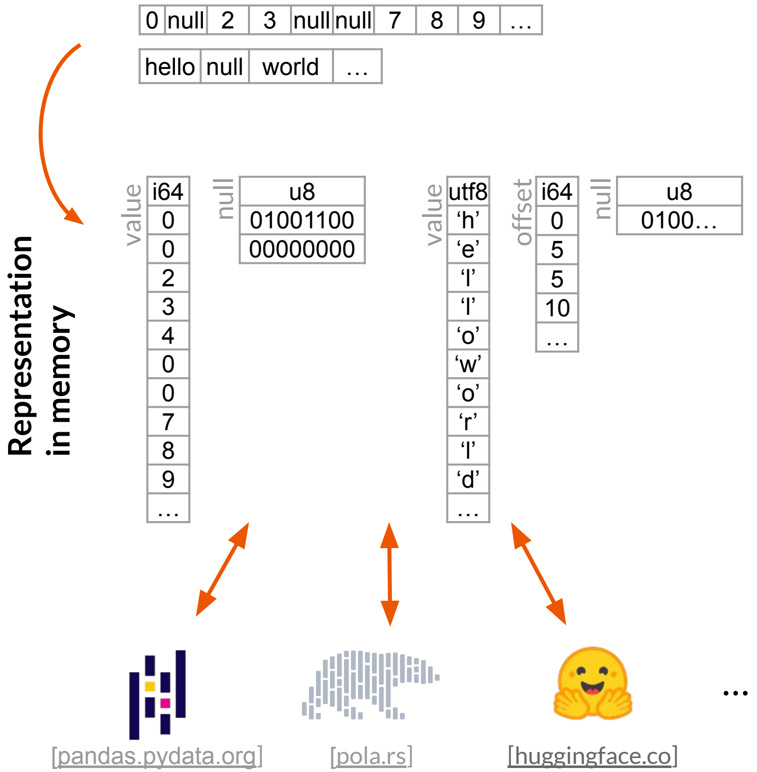
Specification how to arrange data frames in memory

Allows to exchange data without copies

Official support for C, C++, C#, Go, Java, JavaScript, Julia, MATLAB, **Python**, R, Ruby, **Rust**

Supports complex array types:

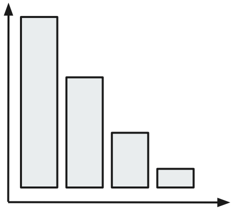
- Primitives (int64, float32, utf8, …)
- Structs
- Lists
- …



(*) See also Apache Arrow: connecting and accelerating dataframe libraries across the PyData ecosystem by Joris Van den Bossche

Processing JSON documents

[arrow.apache.org]



Spotify Million Playlist Dataset
[aicrowd.com/challenges/
spotify-million-playlist-dataset-challenge]

2.3 GB as Arrow IPC file

JSON documents with Spotify playlists

5.4 GB compressed, 30+ GB uncompressed

Python Implementation

```python
import pyarrow as pa

schema = {
    "name": pa.string(),
    'collaborative': pa.string(),
    'pid': pa.int64(),
    # ...
    'tracks': pa.list_(
        pa.struct({
            "pos": pa.int16(),
            'artist_name': pa.string(),
            'track_uri': pa.string(),
            # ...
        }),
    ),
}
```

```python
with zipfile.ZipFile(root / "[...].zip", "r") as z:
    for i in range(1000):
        with z.open("...", "r") as fobj:
            d = json.load(fobj)

        for pl in d["playlists"]:
            pl["modified_at"] = 1000 * pl["modified_at"]

        table = pa.table({
            name: pa.array(
                [pl[name] for pl in d["playlists"]],
                type=ty,
            )
            for name, ty in schema.items()
        })
```

Rust Implementation

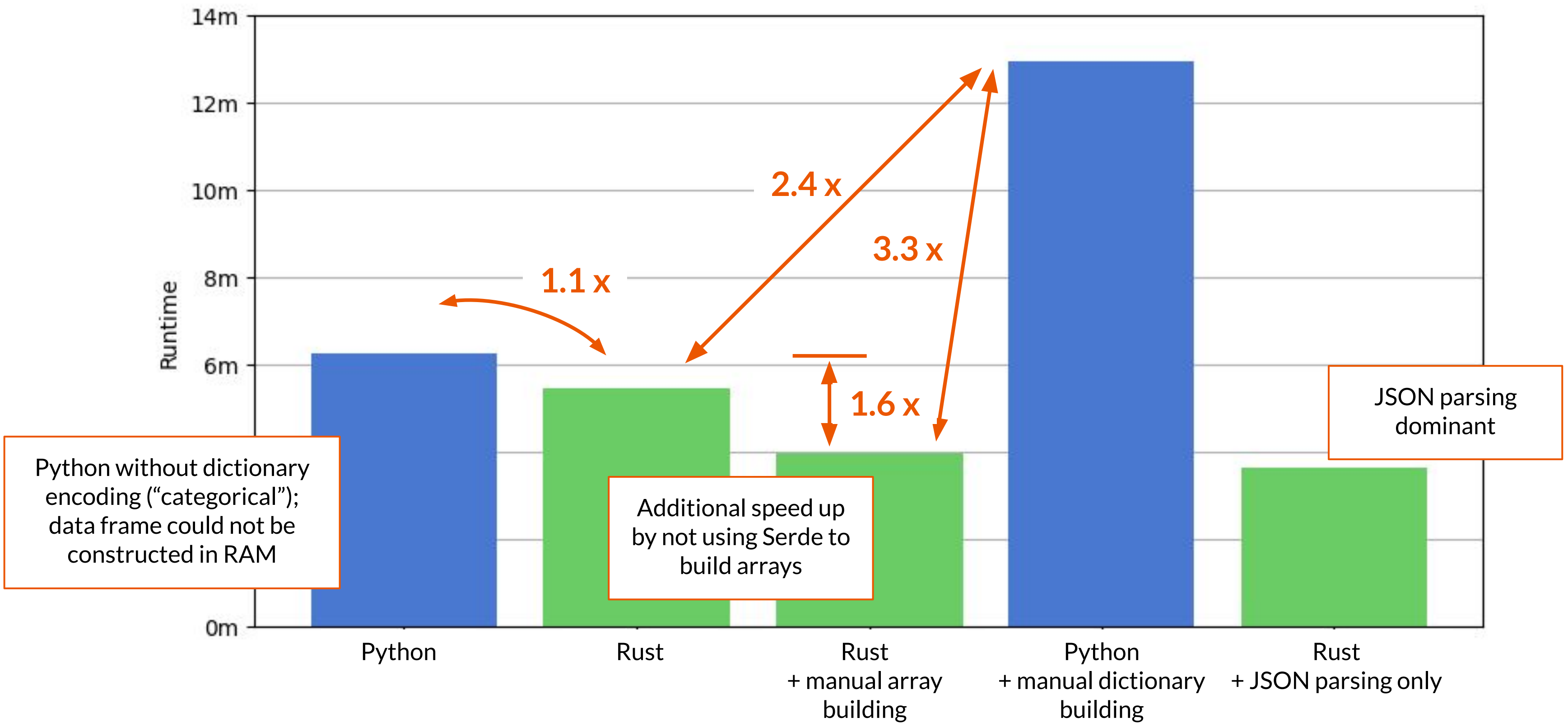
```rust
#[derive(Deserialize, Serialize)]
struct Playlist {
    name: String,
    collaborative: String,
    pid: i64,
    modified_at: i64,
    num_tracks: u16,
    num_albums: u16,
    num_followers: i64,
    tracks: Vec<Track>,
}
```

```rust
let mut builder = ArraysBuilder::new(&fields)?;
for i in 0..n {
    let mut content = Vec::new();
    zip.by_name("...",)?.read_to_end(&mut content)?;
    let mut data: Container = serde_json::from_slice(&content)?;

    for item in data.playlists.iter_mut() {
        item.modified_at = 1000 * item.modified_at;
    }

    builder.extend(&data.playlists)?;
}
```

Rust -> Arrow conversion: serde_arrow[github.com/chmp/serde_arrow]

- Serde used in JSON and Arrow conversion
- Easy to write, but performance overhead
- Arrow 37.0.0 has similar feature built in

Comparison


14m
12m
10m
8m
6m
0m
Runtime
2.4 x
3.3 x
1.1 x
1.6 x
JSON parsing dominant
Python without dictionary encoding (“categorical”); data frame could not be constructed in RAM
Additional speed up by not using Serde to build arrays
Python
Rust
Rust + manual array building
Python + manual dictionary building
Rust + JSON parsing only

Loading the data frame in Python using Polars [https://pola.rs]

```python
import polars as pl
df = pl.read_ipc("data/spotify_million_playlist_dataset.ipc", memory_map=True)
```

```python
(
    df.lazy().select(
        pl.col("tracks").arr.explode()
        .struct.field("artist_name")
        .value_counts().alias("counts")
    )
    .unnest("counts")
    .sort("counts").tail(10)
    .collect()
)
```

artist_name	counts
cat	u32
"J. Cole"	241560
"Justin Bieber"	243119
"Future"	250734
"Ed Sheeran"	272116
"Eminem"	294667
"The Weeknd"	316603
"Rihanna"	339570
"Kendrick Lamar...	353624
"Kanye West"	413297
"Drake"	847160

Extending Pandas / Polars / …

Built In PyO3 support in arrow-rs

```
#[pyfunction]
fn transform(array: &PyAny, py: Python) -> PyResult<PyObject> {
    let array = make_array(ArrayData::from_pyarrow(array)?);

    let result = todo!();

    result.to_data().to_pyarrow(py)
}
```

Converts only metadata
Array data is shared

See also

github.com/apache/arrow-rs/tree/master/arrow-pyarrow-integration-testing
docs.rs/arrow/latest/arrow/pyarrow/index.html

Conclusion

Conclusion

Rust for data processing

- Rust is fast and memory efficient out of the box (no performance cliffs)
- Caveat: Rust libraries for data processing not yet as high quality as in Python

Strategies

- Build your own CLI tools
- Use types & code generation to define interfaces
- Leverage Arrow to exchange structured data in a unified format
- Build extension modules with PyO3

When to incorporate Rust?

- Performance benefit not always clear cut
- Bring data into a Python compatible format
- Complex processing steps (in particular string processing)
- Data not in data frame format

Code + Slides:
github.com/chmp/PyConDE23

References

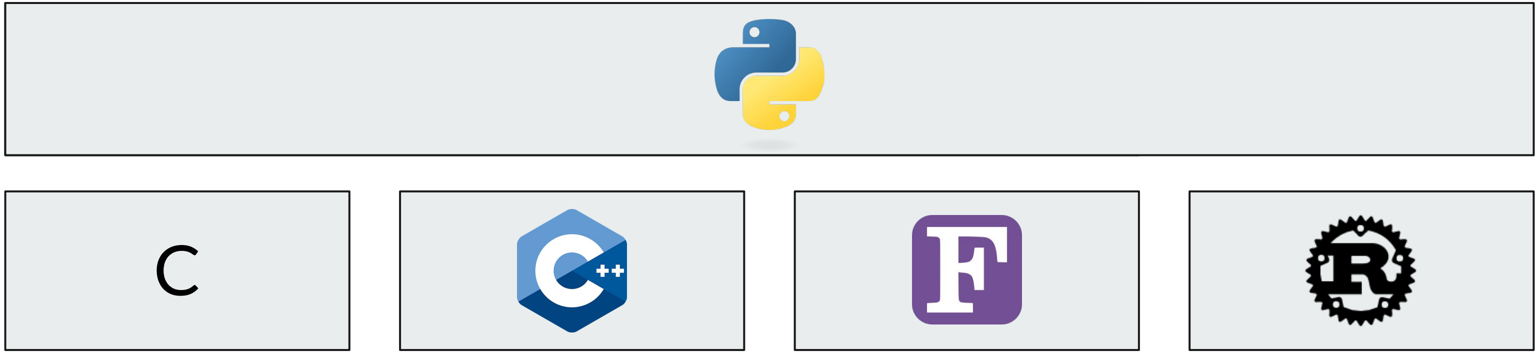
Code to talk: https://github.com/chmp/PyConDE23

Useful Rust libraries:

- serde - serialization & deserialization
- serde_json - serialize into JSON
- arrow & arrow2 - create arrow compatible data & write parquet
- polars - dataframes in Rust & Python
- PyO3 - Python modules written in Rust
- anyhow - simplified error handling
- rayon - simple parallelization

Backup

Why Rust & Python?



Python builds on C, C++, Fortran

Rust is modern language, designed to fit into this group

Logos: foundation.rust-lang.org, python.org, isocpp.org, fortran-lang.org

Streaming inputs / outputs (1)

```
> io-patterns-echo.exe
 "foo"
 "Echo: foo"
 "bar"
 "Echo: bar"
 "baz"
"Echo: baz"
```

Often the output can be generated in parts

Strategy: use JSON lines / one line per message

Simple parallelization: Rust & Python can work in parallel

Streaming inputs / outputs

```python
with subprocess.Popen(
    [path], encoding="utf-8",
    stdin=subprocess.PIPE, stdout=subprocess.PIPE,
) as proc:
    # ...
    proc.stdin.write(inp)
    proc.stdin.write("\n")
    proc.stdin.flush()

    out = proc.stdout.readline()
    # ...
    proc.stdin.close()

assert proc.returncode == 0
```

Deadlocks without flushing

Signal end

```rust
std::io::stdout().write_all(&out)?;
std::io::stdout().write_all(b"\n")?;
std::io::stdout().flush()?;
```

Preventing deadlocks requires correct flushing & handling of EOF (end of file)

Logos: foundation.rust-lang.org, python.org

Borrow checker: xor mutability

either-or:

- A **single** mutable reference
- Multiple immutable references

```rust
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```

won't compile

Code: doc.rust-lang.org

Tips:

- Consider cloning data if it simplifies the program
- Localize mutable access
- Split structs into smaller pieces

github.com/luser/keep-calm-and-call-clone

Rust tip: keep it simple (1) (or don't to write Python in Rust)

```rust
struct S {
    a: i32,
    b: i32
}

impl S {
    pub fn get_a(&mut self) -> &mut i32 {
        &mut self.a
    }
    /* ... */
}

let mut s = S { a: 0, b: 0 };
```

```rust
fn update(a: &mut i32, b: &mut i32) {
    *a = 1;
    *b = 2;
}
```

```rust
// compiles
update(&mut s.a, &mut s.b);
```

```rust
// does not compile
update(&mut s.get_a(), &mut s.get_b());
```

See also: steveklabnik.com/writing/rusts-golden-rule

Rust tip: keep it simple (2) (or don’t to write Python in Rust)

Python makes stream processing simple

```python
def process_items(items: Iterable[T]) -> Iterable[U]:
    for item in items:
        ...
        yield result
```

In Rust use out arguments

```rust
fn process_items(items: &[T], result: &mut Vec<U>) {
    for item in items {
        // ...
        result.push(result)
    }
}
```

`&[T]` read-only view of a “list” of T

`&mut Vec<U>` write access to a “list” of U

Iterators can be written in Rust, but Lifetimes can make it complicated

Rust tip: parallelize in Python

- Rust parallelism requirements are encoded in the type system (Send / Sync)
- Often: independent units of works (e.g., files)

⇒ parallelize with Python threads

```python
from concurrent.futures import ThreadPoolExecutor
from multiprocessing import cpu_count

with ThreadPoolExecutor(
    max_workers=cpu_count(),
) as executor:
    results = executor.map(process_files_in_rust, files)
```