

the
matter lab

Why would you use Rust for scientific computing?

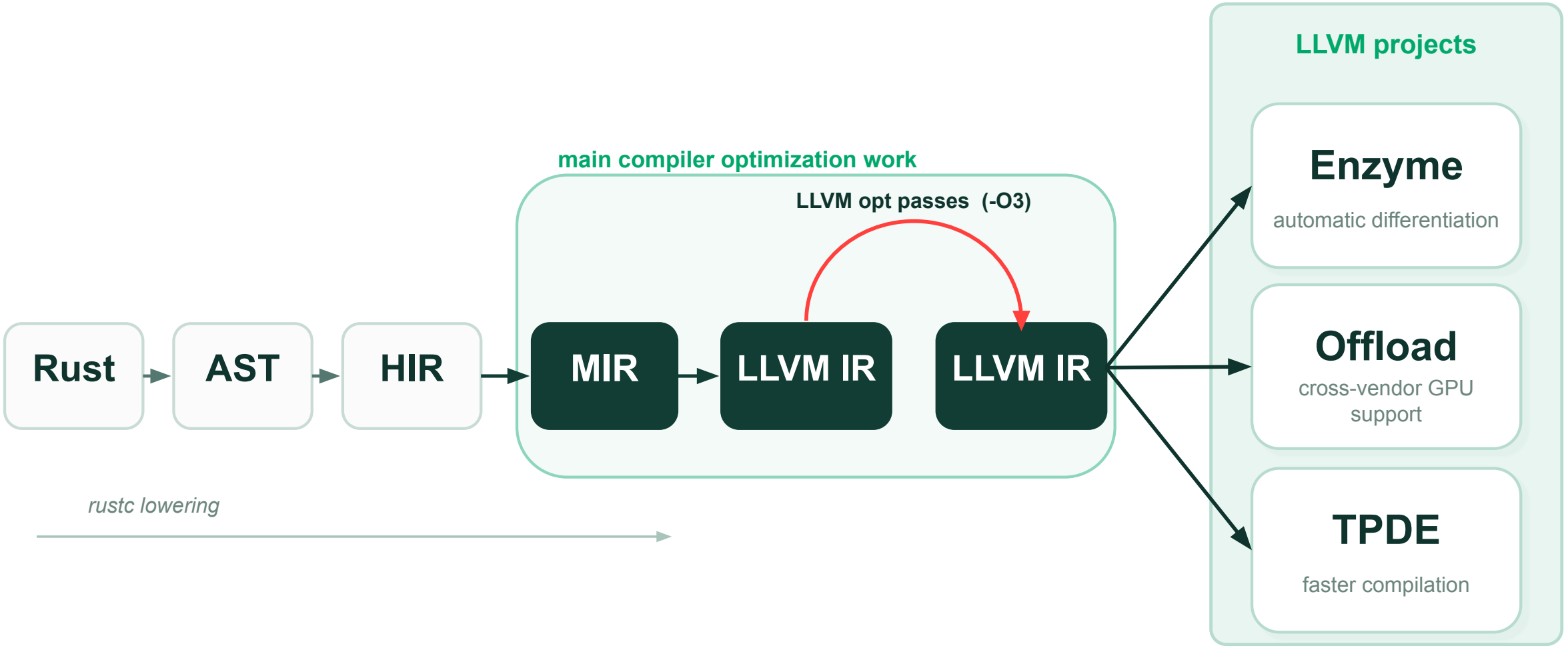
Manuel S. Drehwald

SFU's Research Computing Group Meeting

About me

- PhD student at the University of Toronto, in the Matter Lab
- Rust compiler team member (GPU support & automatic differentiation)
- Previous intern and collaborator at Lawrence Livermore National Lab
- Previous intern at the Julia Lab
- Just starting at NVIDIA's Rust GPU Compiler team

Where am I working?

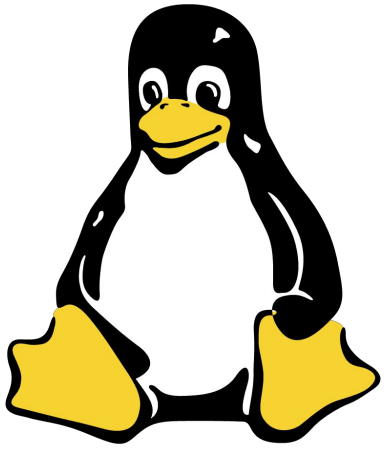


About Rust

- Originally developed at Mozilla: A safe & fast language for Browsers
- A safe system programming language







- But why care for HPC? Performance and Features

Part I: Performance

Performance

- C++ vs. Fortran vs. Rust: All can achieve the same peak performance
  - But at what cost
- Defaults matter
- Compilers have limited “Fuel” they can spent on optimizations
  - Otherwise we would compile forever

Alias Information defaults

Do **src** and **dst** point to the same memory?

C++: *void copy(double *dst, const double *src, long n)* - **Maybe**

Fortran: *subroutine copy(dst, src, n)* - **No** (Hopefully)

Rust: *fn copy(dst: &mut [f64], src: &[f64])* - **No** (Guaranteed)

Alias Information alternatives

Do **src** and **dst** point to the same memory?

C++: *void copy(double *__restrict dst, const double *__restrict src, long n)*

- **No** (Hopefully)

unsafe Rust: unsafe fn copy(dst: *mut f64, src: *const f64, n: usize)

- **Maybe**

Alias Information impacts



Can your language vectorize this function?

```
function (pointer src, pointer dst, int n)
    for i in 0..n
        dst[i] = src[i]
```



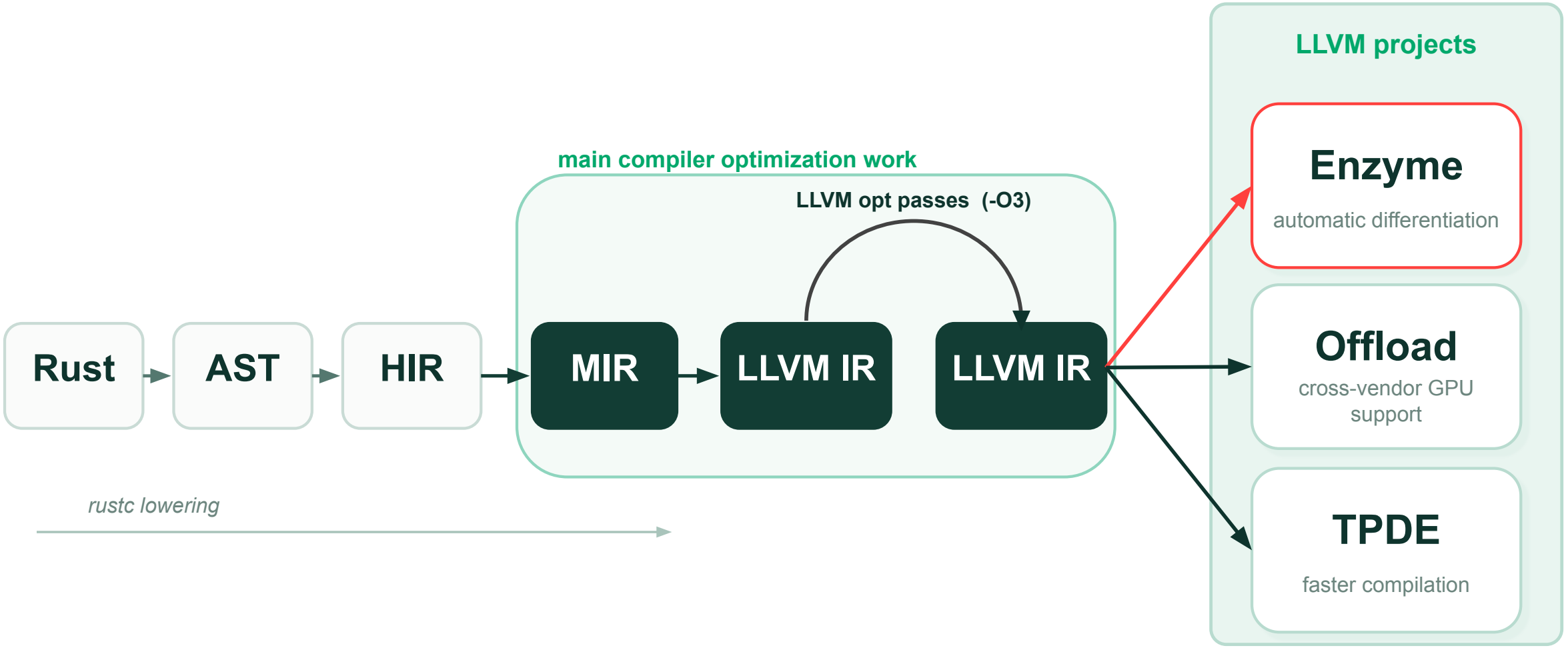
Alias Information impacts

Can your language vectorize this function?

```
function (pointer src, pointer dst, int n)
    for i in 0..n
        dst[i] = src[i]
```

Fortran: gfortran Yes, lfortran No, Flang Partially
C++: Partially: 100 lines
C++ (restrict): Yes: 6 lines
Rust: Yes: 7 lines
Rust (unsafe): Partially: 70 lines

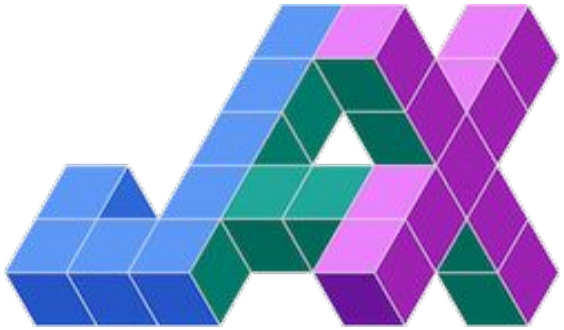
What are we optimizing?



Optimizing automatic differentiation

fn f(x) = x*x

fn f’ = ?

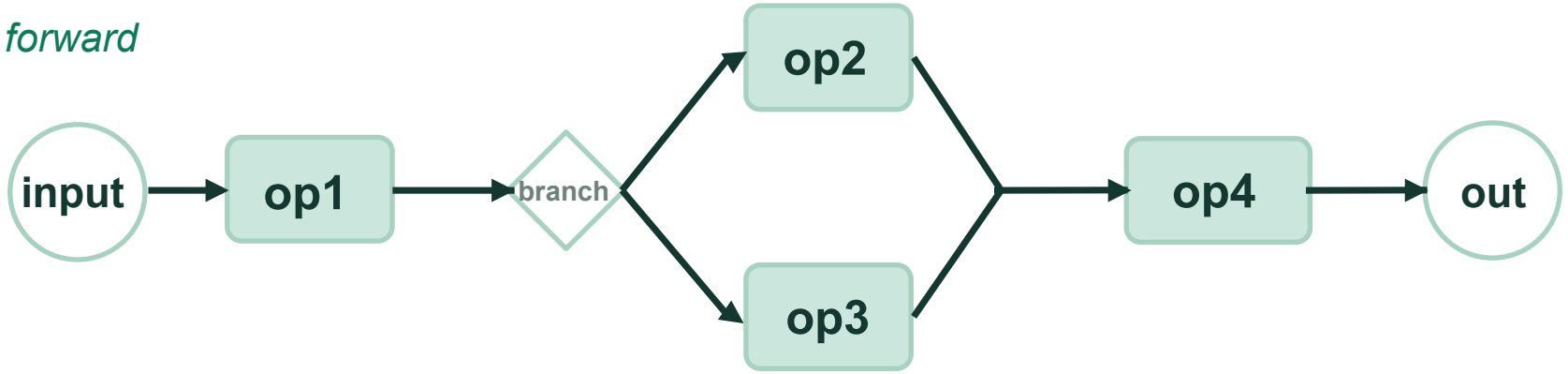


jacrev()



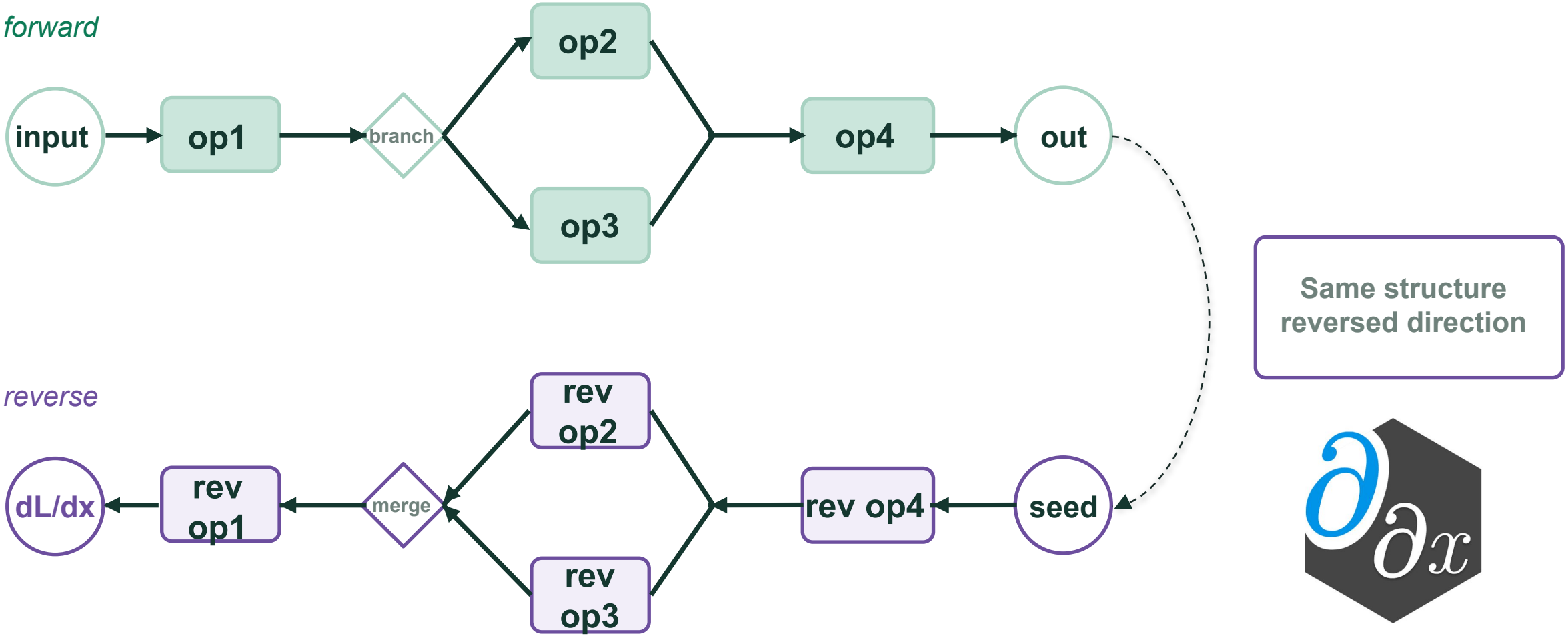
torch.autograd.backward()

std::autodiff

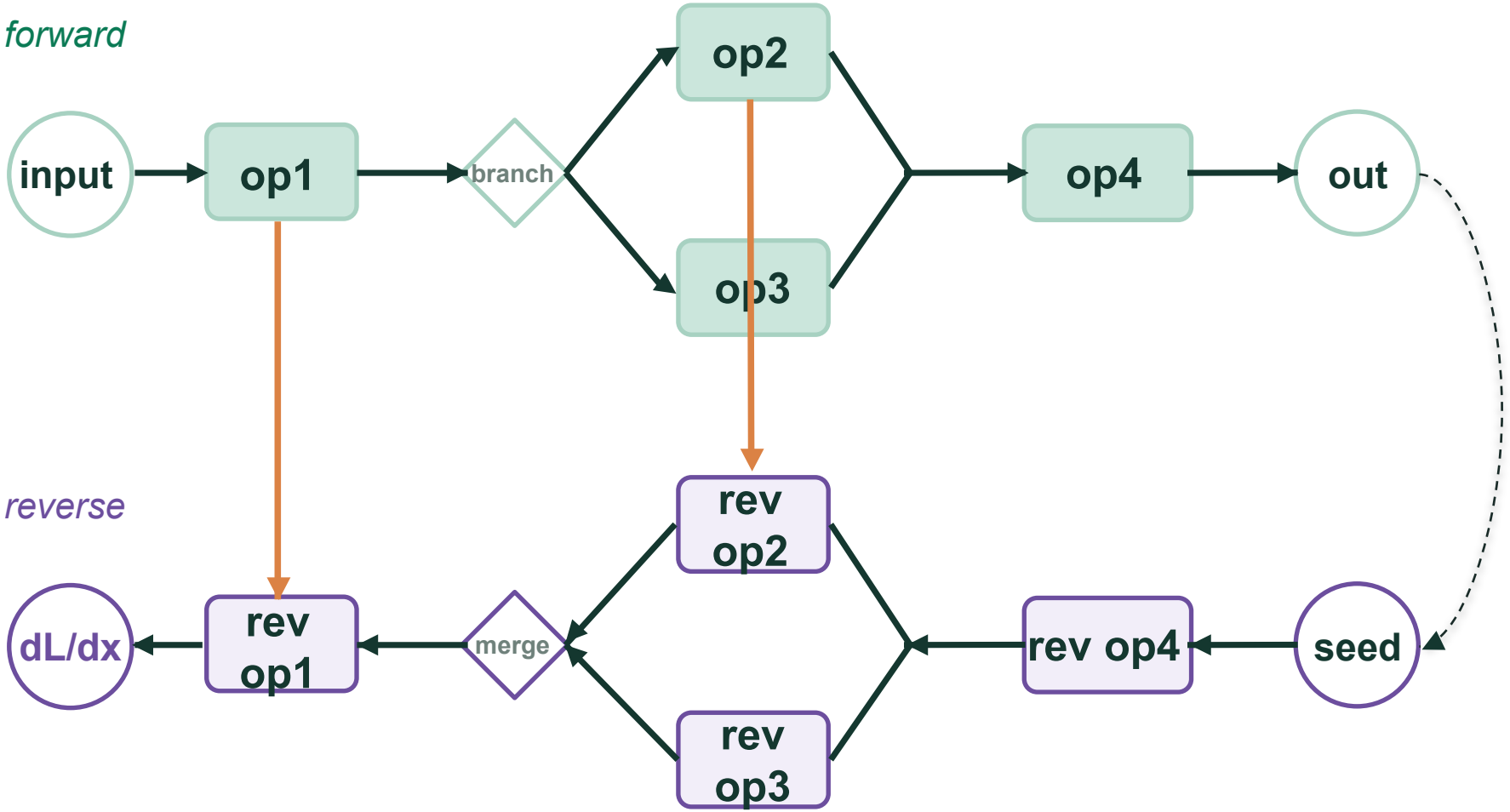


Original “primal” function

std::autodiff (Enzyme)



std::autodiff



Some primal values will be used later

→ cache or recompute

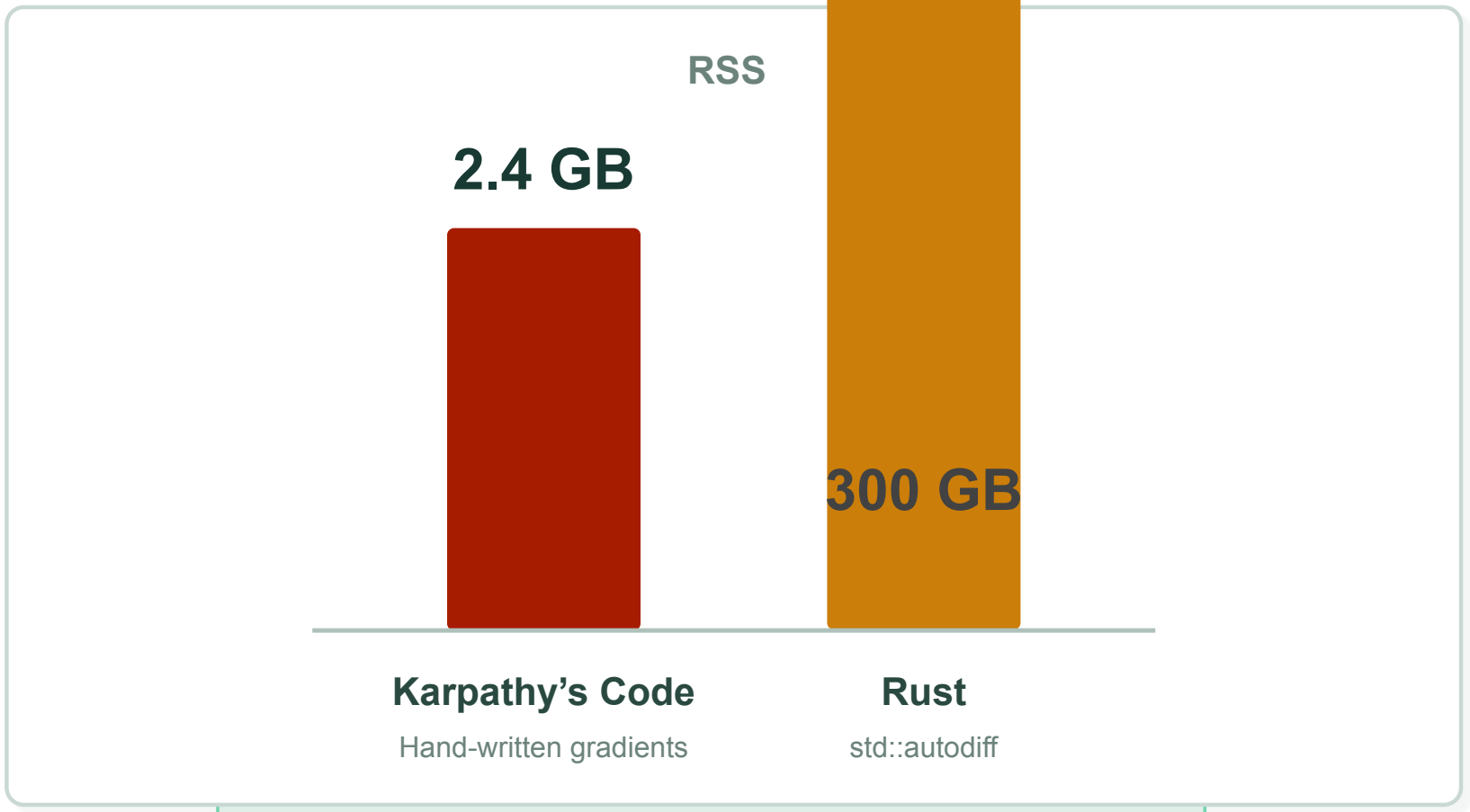
Caching challenges - GPT2

We ported Karpathy's GPT2.c to Rust

- C2Rust + rewrite into idiomatic, safe Rust
- Both C and safe Rust had identical performance

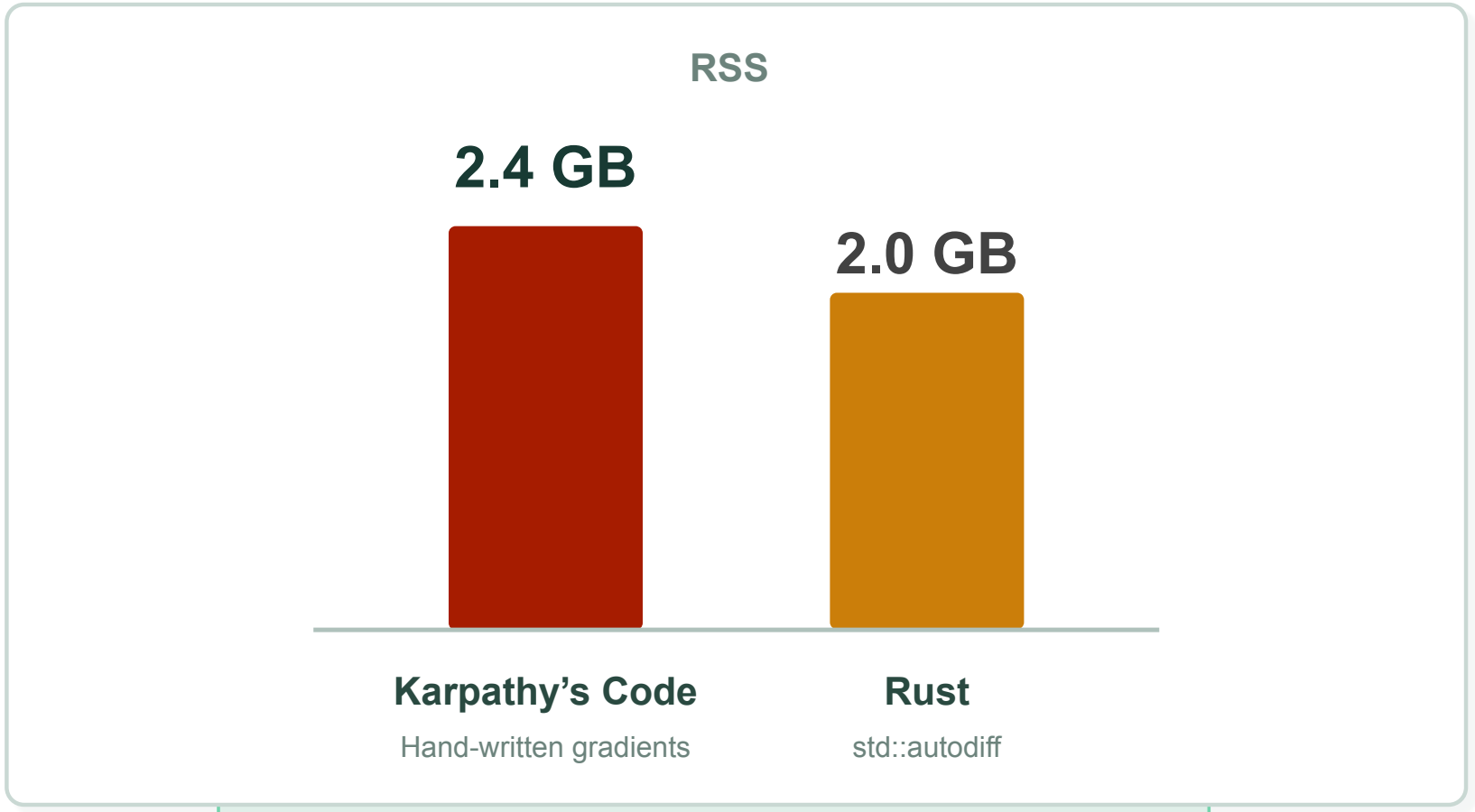
- Can we replace handwritten backpropagation with *std::autodiff*?
  - Of course we can! We replaced all manual <x>_backward() functions:

Caching everything - GPT2



Alias Info is crucial for autodiff

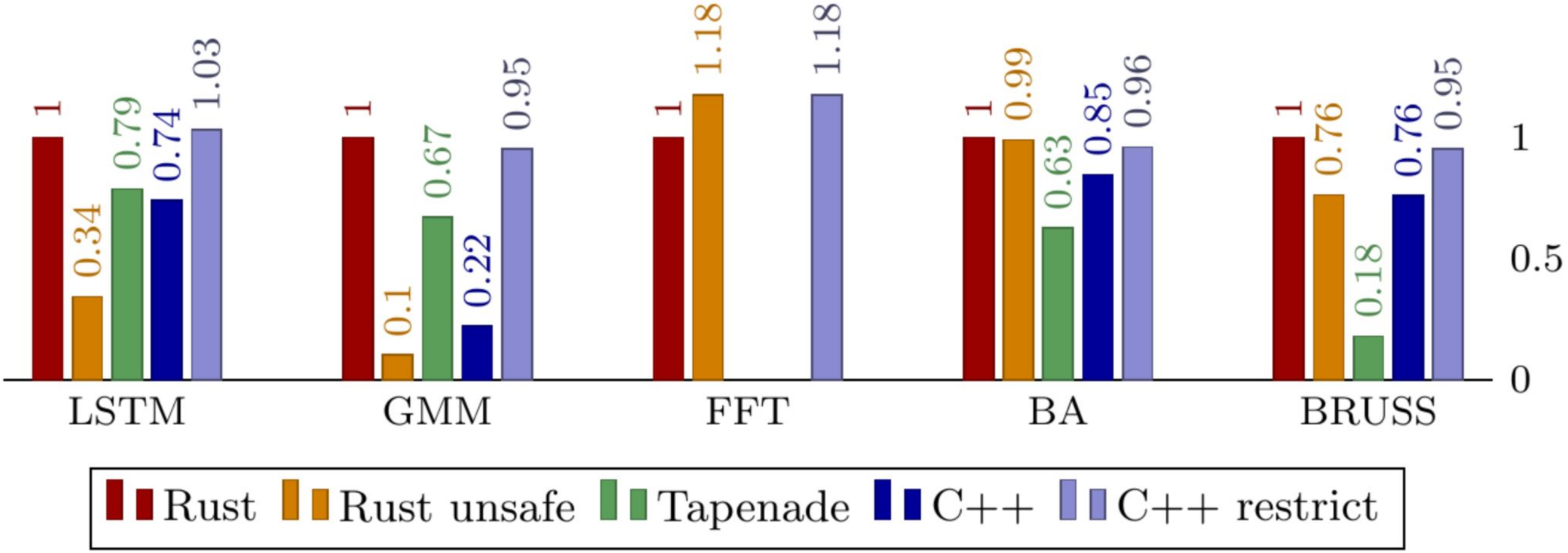
Caching fixed - GPT2



(Rust runtime still 4x slower)

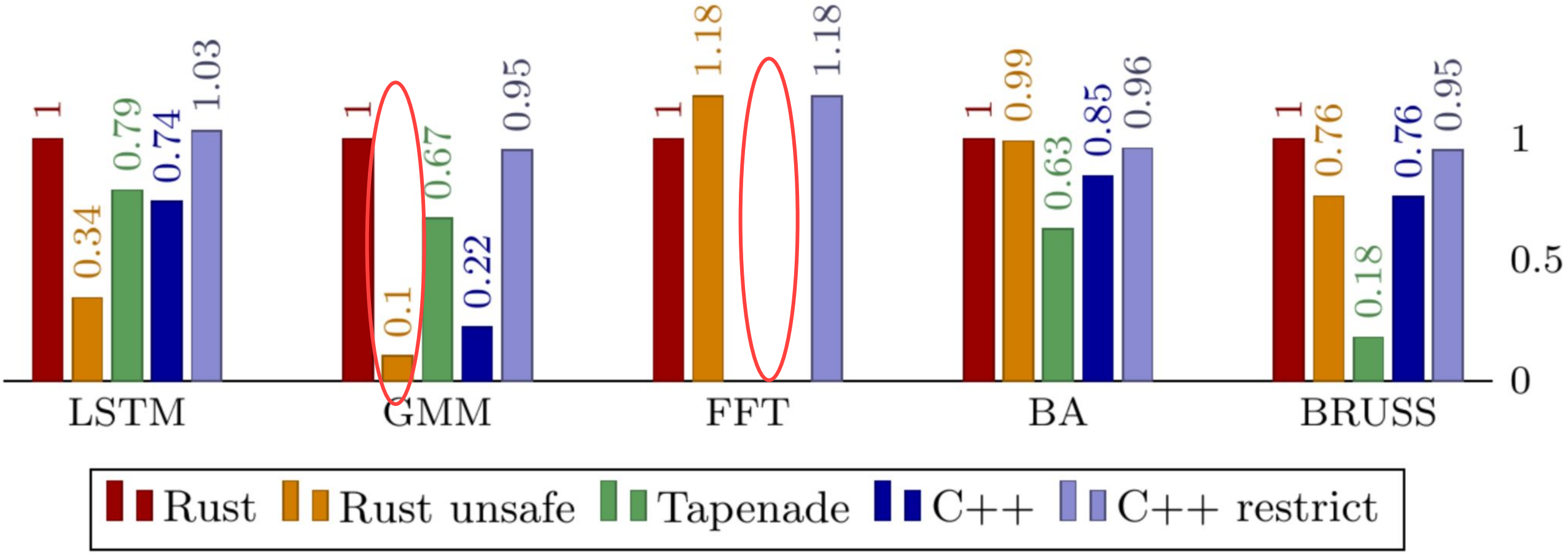
ADBench - autodiff benchmark suite

- Speedup over **safe Rust** code, which is normalized to 1
- Unsafe Rust is using **raw pointers** instead of slices and references.

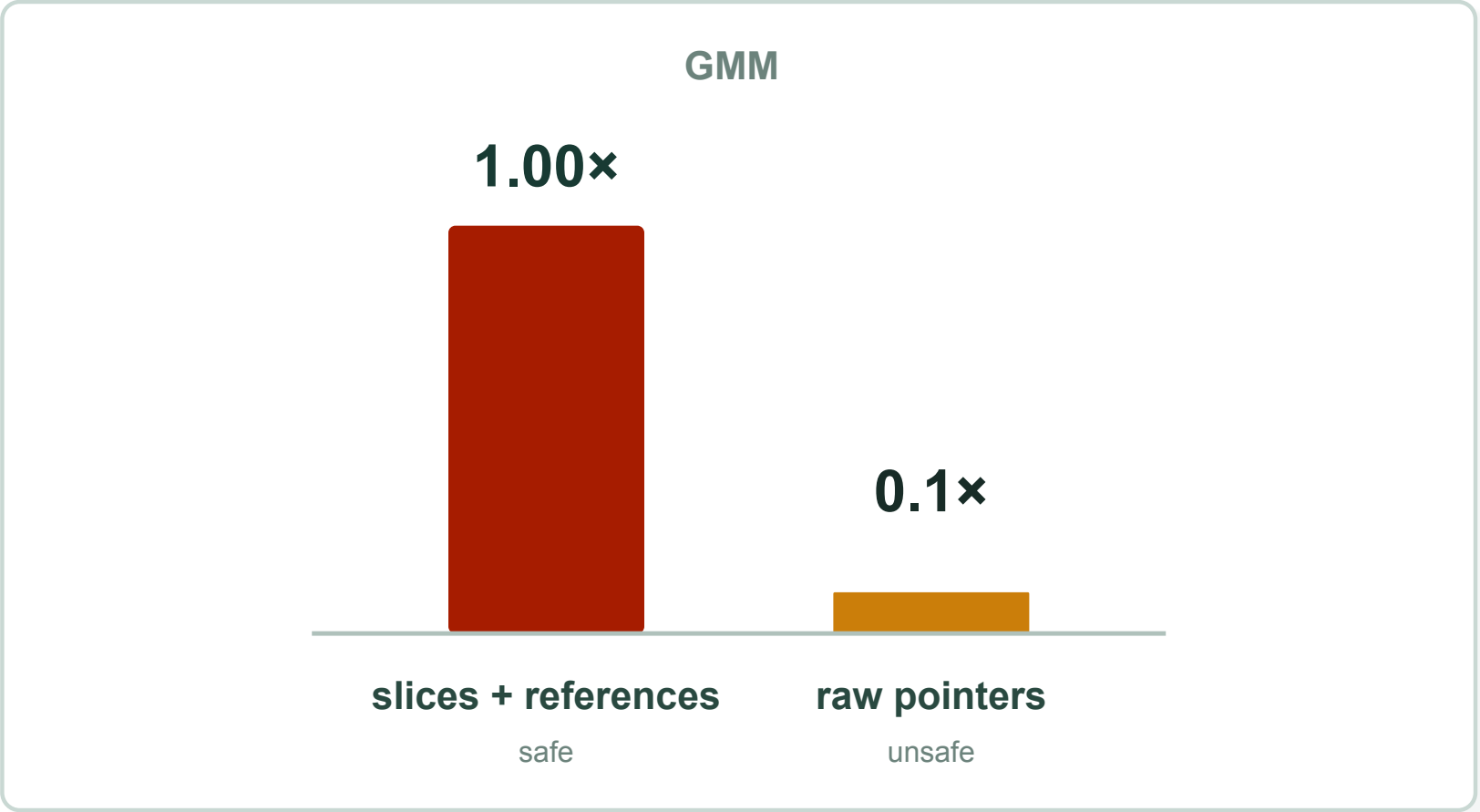


ADBench - autodiff benchmark suite

- Speedup over **safe Rust** code, which is normalized to 1
- Unsafe Rust is using **raw pointers** instead of slices and references.



Performance pitfalls - GMM



Why is the gap so extreme?

Performance pitfalls - GMM

```
// unsafe
unsafe fn qtimesx(d: usize, q_diag: *const f64, ltri: *const f64, x: *const f64, out: *mut f64) {
    for i in 0..d {
        let mut lparamsidx = i*(2*d-i-1)/2;
        for j in i + 1..d {
            *out.add(j) = *out.add(j) + *ltri.add(lparamsidx) * *x.add(i); // <= bad
            lparamsidx += 1;
        }
    }
}

// safe
fn qtimesx(d: usize, q_diag: &[f64], ltri: &[f64], x: &[f64], out: &mut [f64]) {
    for i in 0..d {
        let mut lparamsidx = i*(2*d-i-1)/2;
        for j in i + 1..d {
            out[j] = out[j] + ltri[lparamsidx] * x[i];
            lparamsidx += 1;
        }
    }
}
```

Performance pitfalls - GMM

- **out.add(j)* could overwrite *ltri* and *x*, so these must be cached.
- One cache allocation per outer loop iteration

```
for i in 0..d {
    let mut lparamsidx = i*(2*d-i-1)/2;
    for j in i + 1..d {
        *out.add(j) = *out.add(j) + *ltri.add(lparamsidx) * *x.add(i);
        lparamsidx += 1;
    }
}
```

- **malloc calls**: 11 in safe Rust, > 1 million for unsafe Rust.
- 37 MB vs 843 MB vs cached

Molpipx

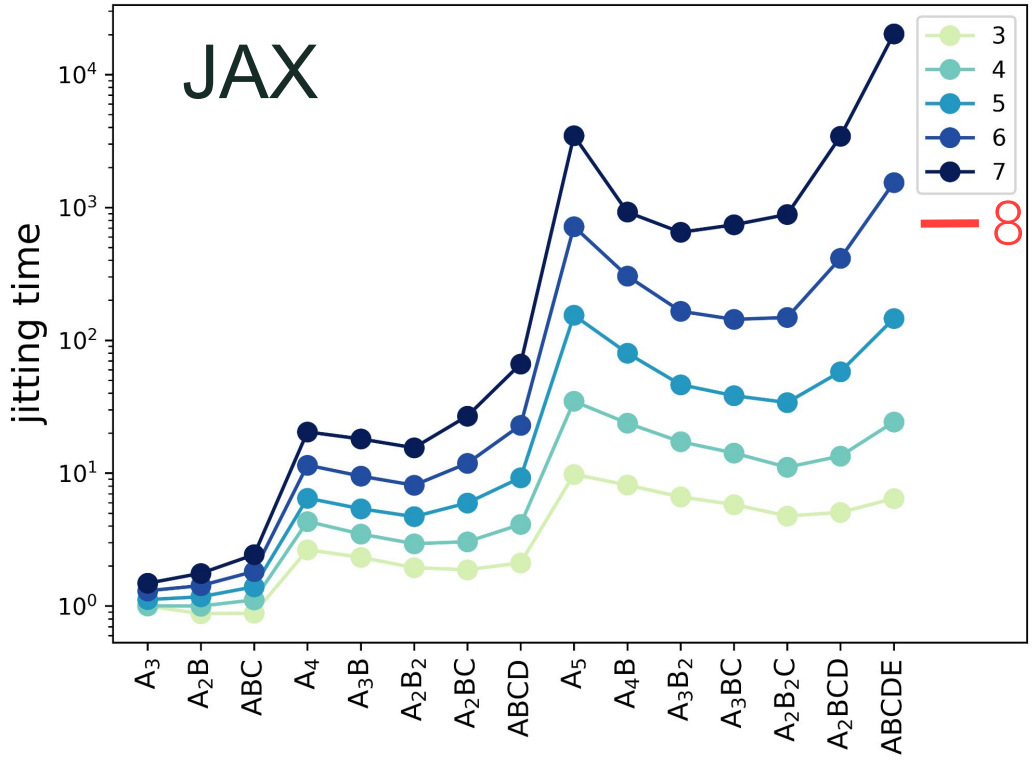
In this work, we present MOLPIPx, a versatile library designed to seamlessly integrate permutationally invariant polynomials with modern machine learning frameworks.These methodologies are widely employed for parameterizing potential energy surfaces across diverse molecular systems.

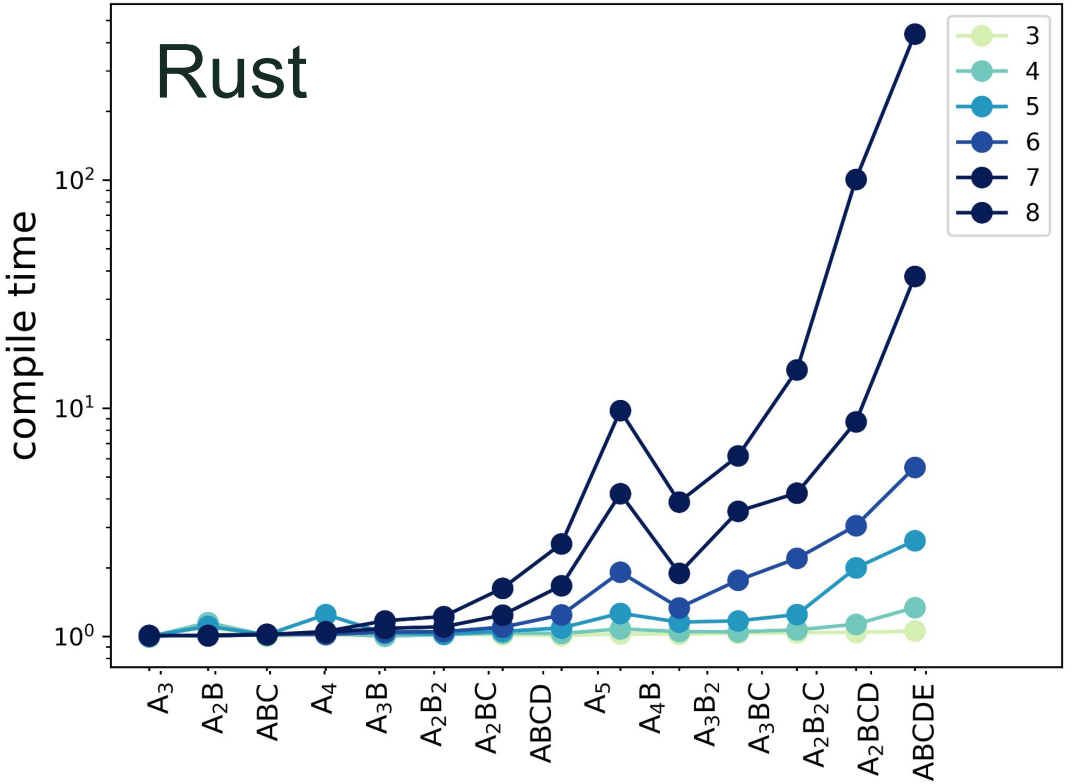
MOLPIPx leverages two powerful automatic differentiation engines —JAX in Python and std::autodiff in Rust —to facilitate the efficient computation of energy gradients and higher-order derivatives, which are essential for tasks such as force field development and dynamic simulations.

JIT/Compile times - JAX vs Rust

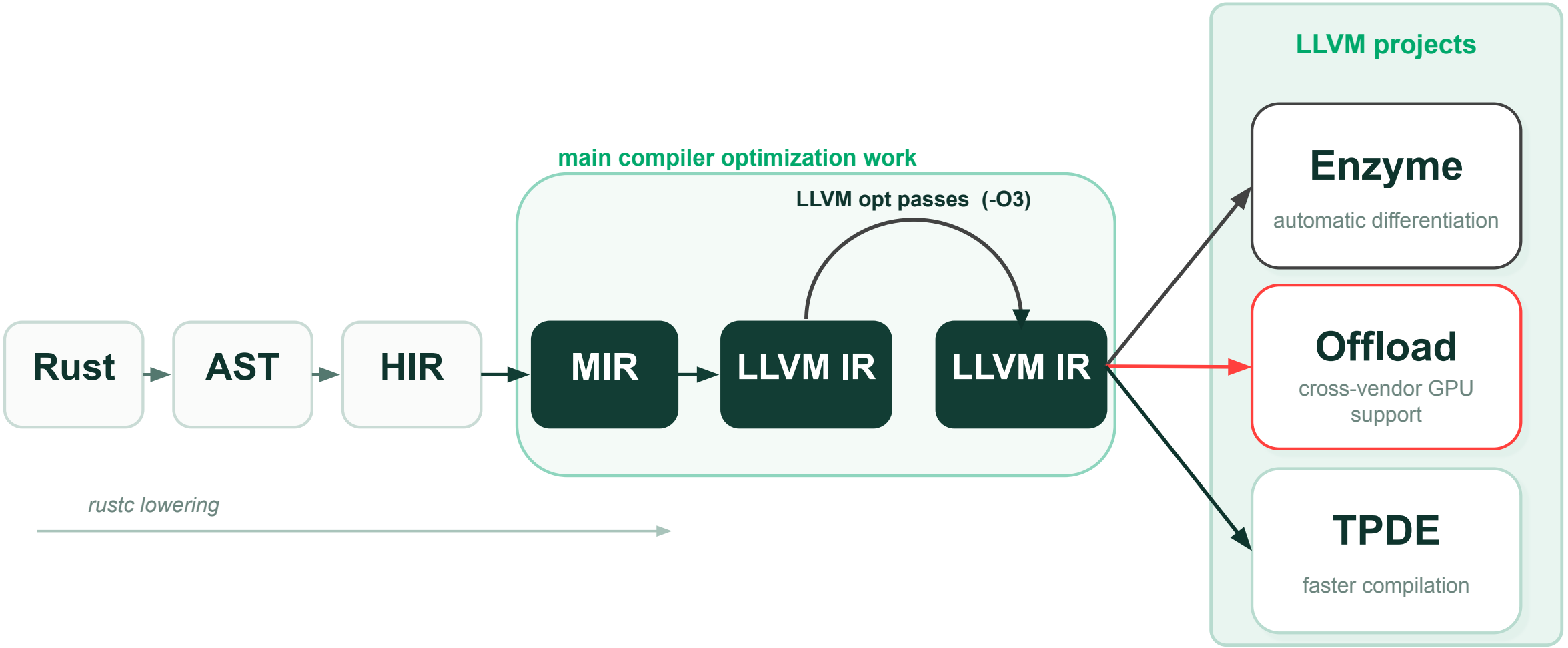
Rust compilation is ~2 magnitudes faster than JAX jit times.

JAX ran out of memory while jitting degree 8 (using > 200GB)





What are we optimizing?



GPU Offload in Rust

Rust supports safely writing and launching GPU kernels

```
#[offload_kernel]
fn fir(
    mut m_out: Region<Real, Linear1D>,
    m_in: &[Real; INLEN],
    coeff: &[Real; COEFFLEN],
    coefflen: usize,
    iend: usize,
) {
    let i = Linear1D::index();
```

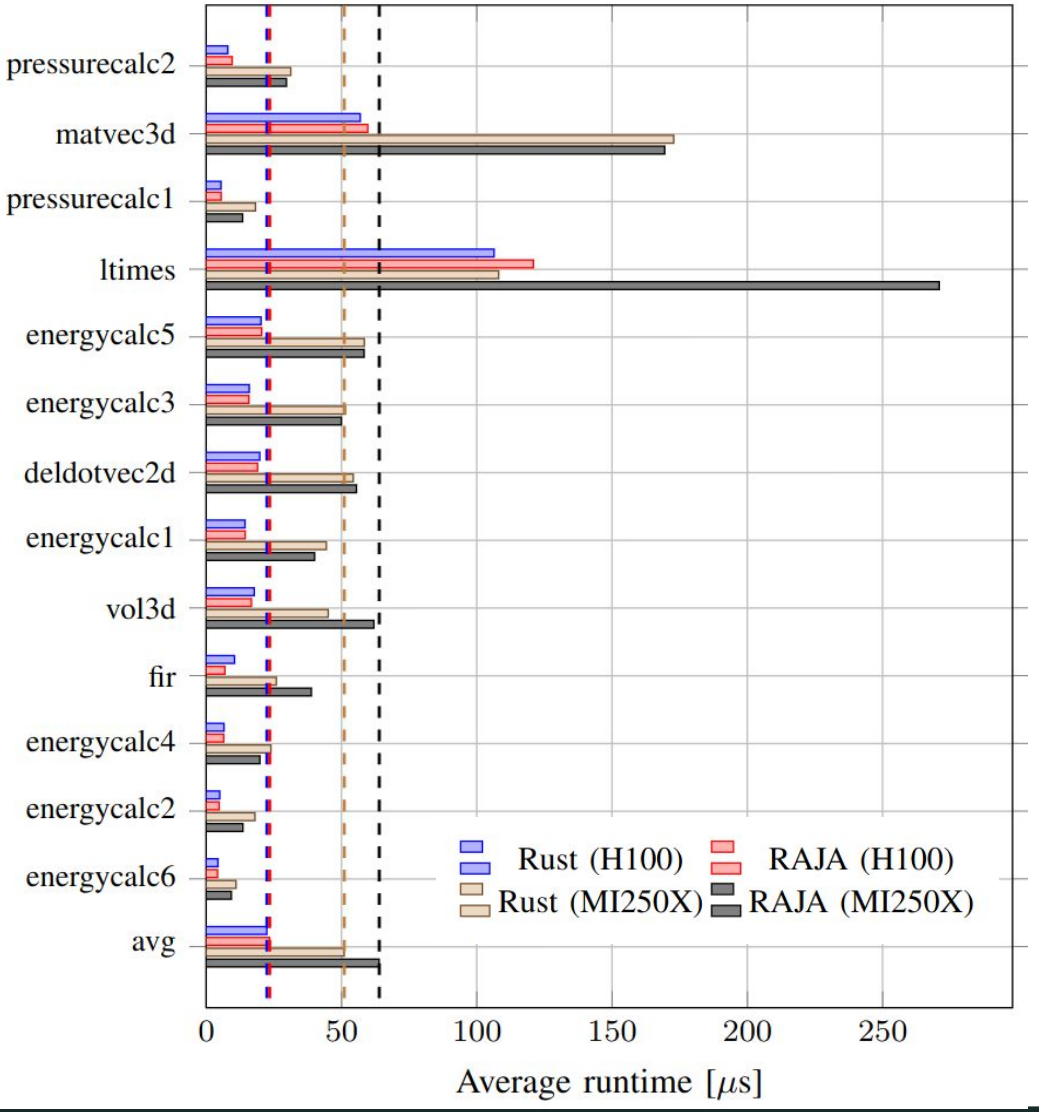
```
core::offload::offload! {
    kernel = energycalc4,
    grid_dim = [BLOCKS, 1, 1],
    block_dim = [THREADS_PER_BLOCK, 1, 1],
    args = (e_new_reg, self.e_cut, self.emin),
};
```

We implemented LLNL's Benchmark suite, RajaPerf, in safe Rust

GPU Offload performance in Rust

Rust matches CUDA performance and outperforms HIP on 3 kernels.

- C++ ltimes lacks alias information:
  - 64 stores per C++ thread, 1 store for Rust
- fir & vol3d are small kernels
  - unlucky loop unroll decisions under HIP



Memory transfers

on both AMD and NVIDIA GPUs. On H100 and across all benchmarks, RAJA performs slightly more and larger host-to-device transfers than Rust: 55 vs. 53 transfers, totalling 468 MB vs. 423 MB. After the kernels, both perform 9 device-to-host transfers, but RAJA transfers 99 MB compared to Rust's 69 MB. Rust moves less data overall, but this does

- Letting the compiler generate memory transfers paid off

Inline assembly

- C++ vs. Fortran vs. Rust: All can achieve the same peak performance
  - In the worst case using Assembly?
- Great to test features not (yet) part of your language

```
fn add(mut a: u64, b: u64) -> u64 {
    unsafe {
        asm!("add {a}, {b}",
             a = inout(reg) a, b = in(reg) b,
             options(pure, nomem, nostack));
    }
    a
}
```

Inline assembly



Not so great if you are on a GPU

```
fn smem_asm() -> *mut f32 {
    let p: *mut f32;
    unsafe {
        asm!("{{ .shared .align 4 .f32 buf[256]; cvta.shared.u64 {p}, buf; }}",
             p = out(reg64) p, options(pure, nomem, nostack));
    }
    p
}
```

Inline assembly was 10% slower due to blocking LLVM optimizations

```
fn smem_rustc() -> *mut f32 {
    let p = gpu_launch_sized_workgroup_mem::<f32>();
    p
}
```

Part II: Features

Rust has official tooling

// Starting a new project

- cargo init

// Installing a Rust binary

- cargo install

// Adding a dependency

- cargo add

// Publishing and deleting a project

- cargo publish
- cargo yank

- cargo bench
- cargo build
- cargo check
- cargo clean
- cargo clippy
- cargo doc
- cargo fetch
- cargo fix
- cargo fmt
- cargo miri
- cargo run
- cargo rustc
- cargo rustdoc
- cargo test

Rust -> Python: Molpipx example

Exposing Rust packages to Python is trivial.

```rust
#[pymodule]
fn piprx(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(py_f_energy, m)?)?;
    m.add_function(wrap_pyfunction!(py_df_energy, m)?)?;
    m.add_function(wrap_pyfunction!(get_N_XYZ, m)?)?;
    m.add_function(wrap_pyfunction!(get_N_DISTANCES, m)?)?;
    m.add_function(wrap_pyfunction!(py_read_geometry_energy, m)?)?;
    Ok(())
}

#[pyfunction]
pub fn py_read_geometry_energy(filename: std::path::PathBuf)
    -> (Vec<f64>, Vec<f64>, Vec<f64>, Vec<String>) {
    read_geometry_energy(filename)
}
```

C -> Rust

C2Rust[1] is a very reliable translate from C to unsafe Rust.

Correctly transpiled all C codebases we looked at:

- > 100k LoC
- Function pointers
- Aliasing code
- Type punning
- Bitmasks
- ...

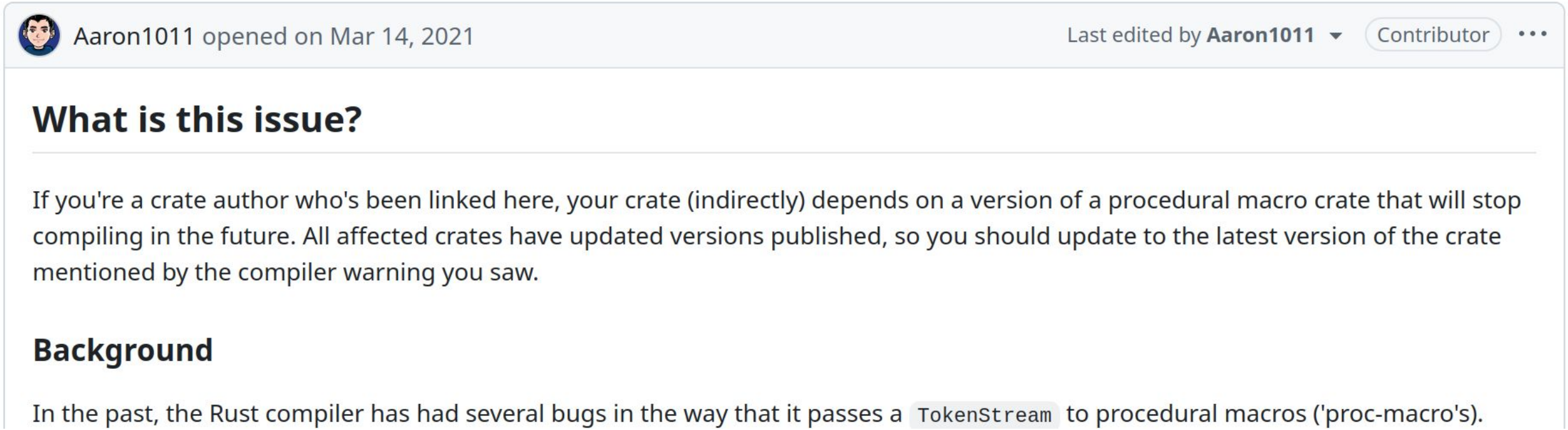
Also DARPA - TRACTOR[2]: to TRanslate All C To (safe) Rust

[1] https://github.com/immunant/c2rust [2] https://www.darpa.mil/research/programs/translating-all-c-to-rust

Stability

Rust has been stable since 2015, with few exceptions for e.g. bugfixes.

If your code is public (github or crates.io) and affected by a breaking change, maintainers will try to reach out and offer a fix.



Miri - The only sanitizer you'll (hopefully) need



An interpreter to verify your unsafe code

```
→  ~  rustup +nightly component add miri
info: downloading component miri
        miri installed                          4.67 MiB
```

And then run it via

```
cargo +nightly miri test
```

Compile time improvement efforts

Faster compile times are on our roadmap[1]

We already vendor our own linker (LLD)

More work in progress

How we get there

Goal	Timespan	What and why
Add an experimental TPDE compilation option	2026	Significant speedup of debug builds
Compiler performance optimizations	2026-2027	Make `rustc` faster through sustained, incremental, profile-driven optimization.
Improve `rustc_codegen_cranelift` performance	2026	This goal aims to improve the rust development experience through faster incremental code generation with `rustc_codegen_cranelift`.
Incremental Systems Rethought	2026	Design, propose, and implement a redesign of the incremental system to allow for shared common bases between different Rustc invocations with different arguments (i.e.
Polymorphic code generation experiment	2026	Implement experimental support in the Rust compiler for polymorphic code generation.
Promoting Parallel Front End	2026	Continue with stabilization and performance improvements to the parallel front-end, continuing from the 2025h2 goal.
Relink Don't Rebuild - Land in nightly, with limitations	2026	Massive speedup of rebuilds in large multi-crate projects
Relink Don't Rebuild - Prepare for stabilization	2027	Massive speedup of rebuilds in large multi-crate projects
Target directory size reduction	2026	Reduce the size of temporary compilation artifacts so developers need less disk space and the compiler spends less time on disk I/O
Incremental, efficient linking with Wild	2026 and beyond	Integrate support for Wild, an innovative, Rust-based linker focused on performance and incremental link times

[1] https://goals.rust-lang.org/2026/roadmap-fast-builds.html

“Fearless Concurrency”

Rayon is Rust’s closest equivalent of OpenMP

```
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter() // <-- just change that!
         .map(|&i| i * i)
         .sum()
}
```

“Fearless Concurrency”

With some extra safeguards

```
error[E0596]: cannot borrow `sum` as mutable, as it is a captured variable in a `Fn` closure
 --> src/main.rs:6:31
  |
5 |     let mut sum = 0u64;
  |         ------- `sum` declared here, outside the closure
6 |     v.par_iter().for_each(|x| sum += x); // error[E0594]
  |                   -------------^^^------
  |                   |          |   |
  |                   |          |   cannot borrow as mutable
  |                   |          in this closure
  |                   expects `Fn` instead of `FnMut`

For more information about this error, try `rustc --explain E0596`.
```

Fearless Concurrency - GPU edition

- We found it frustrating to tell the compiler information it already has

- ***“Dear compiler, just move this object to the GPU, thank you”***

GPU Interface A

- Interface A: Memory movements before and after each kernel launch, we launch the Kernel for you

```
core::offload::offload! {
    kernel = energycalc4,
    grid_dim = [BLOCKS, 1, 1],
    block_dim = [THREADS_PER_BLOCK, 1, 1],
    args = (e_new_reg, self.e_cut, self.emin),
};
```

GPU Interface B

- Interface B: Automatic memory movements, but the called function manages kernel launches (e.g. cuBLAS, rocBLAS)

```
core::intrinsics::offload_args::<_, _, ()>(
    rocblas_sgemv_wrapper,
    (&A, &x, &mut y),
);
```

GPU Interface C

- Interface C: Separate, explicit data-transfers. Otherwise like A)

```
let p_vol = preload(vol_ref);
// other code
offload! {
    kernel = vol3d,
    grid_dim = [count.div_ceil(256) as u32, 1, 1],
    block_dim = [256, 1, 1],
    args = (p_vol, self.vnormq, jp, kz),
};
```

GPU Interface C

- Lifetimes allow us to safely reload data with no risk for data-races

```rust
pub struct Preload<'a, T: ?Sized> {
    cpu_ptr: *const T,
    _marker: PhantomData<&'a T>,
}
pub struct PreloadMut<'a, T: ?Sized> {
    cpu_ptr: *mut T,
    _marker: PhantomData<&'a mut T>,
}
```

Jupytr notebooks for Rust

Evcxr[1] supports interactive programming in Rust

```
[1]
✓ 16s
    !curl -fsSL -o setup.sh https://raw.githubusercontent.com/xavierforge/colab-rust/main/setup.sh
    !bash setup.sh
    %load_ext colab_rust

▶ colab-rust setup (ref: main)
▶ Installing Rust (stable, minimal profile)...
✅ Rust 1.98.0
▶ Attempting prebuilt evcxr_jupyter download...
✅ Installed prebuilt evcxr_jupyter
✅ evcxr_jupyter at /root/.cargo/bin/evcxr_jupyter
✅ Rust kernel registered with Jupyter
▶ Installing colab_rust.py magic module...
✅ colab_rust.py ready at /content/colab_rust.py

🎉 Setup complete.

Next steps in your notebook:

    %load_ext colab_rust

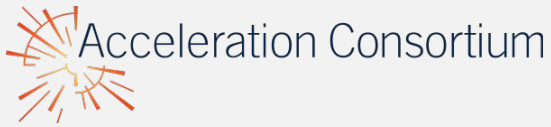
    %%rust
    println!("Hello from Rust on Colab!");
```

[1] https://github.com/evcxr/evcxr

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-2023077)

The views and opinions expressed in this account are those of my own and do not represent those of my employers









Collaborators



Alán Aspuru-Guzik
(University of Toronto)



Marcelo Domínguez
(GSoC)



Shota Sugano
(GSoC)



Kevin Sala
(LLNL)



Johannes Doerfert
(LLNL)