College of Engineering and Computing
ELECTRICAL AND
COMPUTER ENGINEERING
George Mason University

ECE 555 Group Presentation

Igor Semyonov   Jordan Carnes   Robert Laverne Griffin

George Macon University, Department of Electrical and Computer Engineering

April 23, 2025

Why not Just Use C?

- No safety
- Skill issues, leading to lack of safety

Listing 1: Cats

```
struct FakeCat {
    alive: bool,
    hungry: bool,
}

enum RealCat {
    Alive {
        hungry: bool,
    },
```

Impossible states can be encoded in the type system, which leads to compile time errors, instead of runtime checks.

Listing 2: Unphysical (zombie) Cat

```
let fake_cat = FakeCat {
    alive: false,
    hungry: true,
};
```

Why Rust
Type System

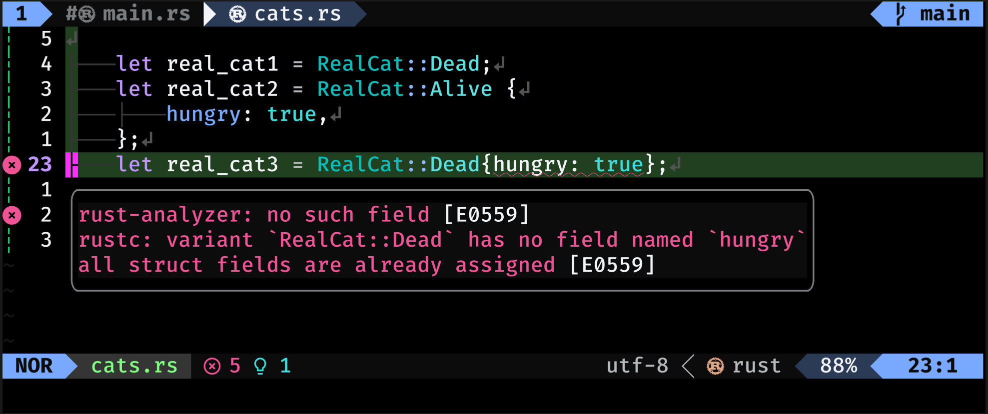


Figure 1: Error when declaring an unrealistic cat

- Result: Errors are values, i.e., no nidden control flow
- Functions that can fail return Result and this must be explicitly handled.

- No manual memory management
- Values are dropped once they go out of scope

High level abstractions that enable multithreading without the error-prone approach of pthreads in C.

Why Rust
Single threaded bsort

Listing 3: Single threaded bsort

```
a.chunks_mut(split_length)
    .for_each(bsort);
```

- a is a vector of numbers
- `chunks_mut` slices the vector into non-overlapping slices of the given length
- `for_each` iterates over the chunks, running the provided function with each chunk as the input.

Why Rust
Concurrency is easy while avoiding race conditions

Listing 4: Example of easy parallelization

```
a.chunks_mut(split_length)
        .par_bridge()
        .for_each(bsort);
```

- Chunks is known to split a vector into distinct slices
- Hence we can safely send each chunk to a different thread

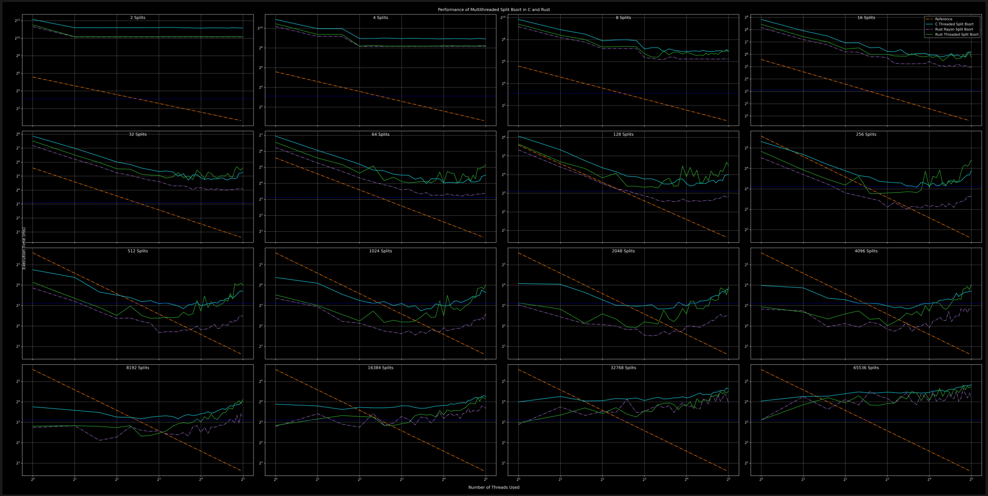


Figure 2: Preformacne results from bsort project. Note that the Rust reference sort is not plotted as it was under 1ms, compared to 8ms for the C qsort.

Why Rust
Zero Cost Abstraction

All these features are provided at compile time and are optimized away when compiling so that they do not affect performance at run time.

Rust Ownership and Borrowing
Problems with shared memory access in Rust

- Ownership Rules[1]
  - Each value has an owner
  - There can only be one owner at a time
  - When the owner goes out of scope, the value will be dropped.
- Borrow Rules[2]
  - At any given time, you can have either one mutable reference or any number of immutable references.
  - References must always be valid.

C, and especially CUDA, style shared mutable memory access violates the ownership and borrow rules.
At this time, we therefore need to use unsafe Rust when writing and calling GPU kernels.

[2] [Ch04-01]rust-book
[2] Klabnik et al. 2024, Ch04-02.

Rust and GPU Programming
Current options

- Rust GPU for vendor agnostic GPU programming
- Rust CUDA for targeting NVIDIA GPUs and their specific libraries
- Vulkano
- wgpu
- Miniquad
- Sierra
- Glium
- Ash
- Erupt

We focused on the Rust CUDA crate, which is currently in active development.

What is a compiler?
Translation pipeline

Front End

- Lexical analysis: source $\rightarrow$ tokens (identifiers, literals, operators)
- Syntax & semantic analysis: tokens $\rightarrow$ AST, type checking
- Emit language independent IR

Optimization

- IR level passes: constant folding, dead code elimination, algebraic simplification
- Advanced transforms: function inlining, loop unrolling, vectorization
- Profile or link time optimizations for extra performance

Back End

- Lower optimized IR to machine instructions (instruction selection & scheduling)
- Register allocation, calling-convention handling
- Emit assembly or object code ready for linking

The problem LLVM solves
front ends & back ends

Fragmented Toolchains

- Historically each language writes its own optimizer & code generator
- Reinventing the same analyses over and over

Reusable IR

- SSA based intermediate form shared by many languages
- Single place to build and maintain optimizations

Multiple back ends

- Add support for x86, ARM, RISC V, NVIDIA GPUs, ... by writing one backend
- All front ends instantly benefit from each new target

Rapid language support

- New languages (Rust, Swift, Julia, etc.) get mature codegen “for free”
- Performance improvements flow to every user automatically

NVVM
NVIDIA's LLVM based GPU IR

What is NVVM IR? [1]

- A dialect of LLVM IR extended for CUDA style GPU programming
- Adds memory space qualifiers (global, shared, constant)

Metadata & intrinsics

- Thread/block IDs and barriers (llvm.nvvm.barrier0)
- Marks kernel entry points and resource usage requirements

[1] *Supercomputing: 8th Russian Supercomputing Days, RuSCDays 2022, Moscow, Russia, September 26–27, 2022, Revised Selected Papers* 2022.

NVVM
NVIDIA's LLVM based GPU IR (continued)

Toolchain flow

1. Front end emits .nvvm.bc bitcode file
2. NVVM compiler (libnvvm) lowers IR to PTX assembly
3. PTX $\rightarrow$ CUBIN or JIT compiled by CUDA driver at runtime

Why use NVVM?

- Reuse LLVM's optimizer on device code
- Keep host and GPU code in one common IR for easier shared analysis

Rust CUDA and NVVM
How the pieces fit

Cargo & rustc target [1]

- Compile kernels with –target=nvptx64-nvidia-cuda
- Separate build profiles for host (x86_64) and device (PTX)

LLVM IR generation

- Annotate GPU functions with `#[kernel]` or `extern "ptx-kernel"`
- rustc emits NVVM compatible IR carrying thread/block metadata

[1] Bychkov and Nikolskiy 2022.

Rust CUDA and NVVM
How the pieces fit (continued)

PTX emission

- Build script (`build.rs`) or `nvptx-link` plugin calls into `libnvvm`
- Produces `.ptx` files you bundle into your binary or load at runtime

Runtime loading & launch

- Use the `rustacuda` crate (or raw CUDA Driver API) to load modules
- Launch kernels inside `unsafe` blocks, mirroring C/CUDA calls

Demo Implementation

$c \in \mathbb{C}$ is in the mandelbrot set if the sequence $\{z_n\}$ converges.

$$z_{n+1} := z_n^2 + c \qquad z_0 = 0$$

The Burning Ship fractal is defined similarly but the sequence is

$$z_{n+1} := (|\mathrm{Re}(z_n)| + |\mathrm{Im}(z_n)|i)^2 + c \qquad z_0 = 0$$

Mandelbrot Kernel

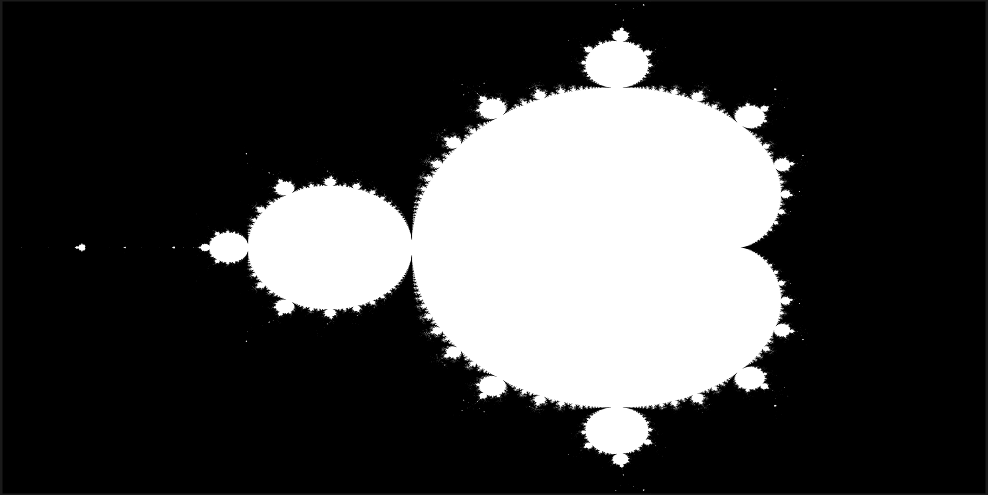
```rust
pub unsafe fn mandelbrot(
    n_re: usize,
    n_im: usize,
    re_min: f32,
    re_max: f32,
    re_range: f32,
    im_min: f32,
    im_max: f32,
    im_range: f32,
    zn_limit: u32,
    out: *mut u8,
) {
    let idx_linear = thread::index() as usize;
    let idx = thread::index_2d();
    let idx_re = idx[1];
    let idx_im = idx[0];

    let c_re = re_range
        * (idx_re as f32 / (n_re - 1) as f32)
        + re_min;
    let c_im = im_range
        * (idx_im as f32 / (n_im - 1) as f32)
        + im_min;
    let mut z_re = c_re;
    let mut z_im = c_im;
    for _ in 0..zn_limit / ZN_SKIP {
        for _ in 0..ZN_SKIP {
            (
                z_re, z_im,
            ) = (
                z_re * z_re - z_im * z_im + c_re,
                2.0 * z_re * z_im + c_im,
            );
        }
        if z_re * z_re + z_im * z_im > 4.0 {
            let elem = &mut *out.add(idx_linear);
            *elem = 255;
            break;
        }
    }
}
```

Launching the Kernel

```
unsafe {
    launch!(
        module.mandelbrot<<<grid_size, block_size,
            0, stream>>>(
            N_RE,
            N_IM,
            re_min,
            re_max,
            re_range,
            im_min,
            im_max,
            im_range,
            zn_limit,
            out_gpu.as_device_ptr(),
        )
    )?;
}
```

- Fiarly similar to launching a kernel in C.
- Must be used inside an `unsafe` block since the kernel itself is an unsafe function.

Output Image



Fractals

Table 1: Timing results for several approaches of producing the fractals

Method	Time (ms)
CPU create point grid	175.357
GPU using CPU points	0.552
CPU using rayon	143.647
GPU f32	0.453
GPU f64	12.272

Live Demo

References

Bychkov, Andrey and Vsevolod Nikolskiy (2022). "Rust Language for GPU Programming". In: *Supercomputing: 8th Russian Supercomputing Days, RuSCDays 2022, Moscow, Russia, September 26–27, 2022, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, pp. 522–532. ISBN: 978-3-031-22940-4. DOI: 10.1007/978-3-031-22941-1_38. URL: https://doi.org/10.1007/978-3-031-22941-1_38.

Klabnik, Steve et al. (2024). *The Rust Programming Language*. No Starch Press. ISBN: 978-1-63190-308-0. URL: https://doc.rust-lang.org/book/.

Supercomputing: 8th Russian Supercomputing Days, RuSCDays 2022, Moscow, Russia, September 26–27, 2022, Revised Selected Papers (2022). Lecture Notes in Computer Science. Moscow, Russia: Springer-Verlag.