



ECE 555 Group Presentation

Igor Semyonov Robert Laverne Griffin

George Mason University, Department of Electrical and Computer Engineering

April 29, 2025

Why not Just Use C?

- No memory safety
- Fragmented support across platforms
- Fragmented, or simply missing, tooling
- Skill issues, leading to lack of safety
- Because writing Rust makes me happy

Why Rust

Type System

Listing 1: Cats

```
1 struct FakeCat {  
2     alive: bool,  
3     hungry: bool,  
4 }  
5  
6 enum RealCat {  
7     Alive {  
8         hungry: bool,  
9     },  
10    Dead,  
11 }
```

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

Listing 2: Unphysical (zombie) Cat

```
1 let fake_cat = FakeCat {  
2     alive: false,  
3     hungry: true,  
4 };
```

Why Rust Type System

```
1 # main.rs ▶ cats.rs main
5
4 let real_cat1 = RealCat::Dead;
3 let real_cat2 = RealCat::Alive {
2     hungry: true,
1 };
23 let real_cat3 = RealCat::Dead{hungry: true};
```

rust-analyzer: no such field [E0559]
rustc: variant `RealCat::Dead` has no field named `hungry`
all struct fields are already assigned [E0559]

NOR cats.rs 5 1 utf-8 rust 88% 23:1

Figure 1: Error when declaring an unrealistic cat

Why Rust Safety

- Result: Errors are values, i.e., no hidden 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

Why Rust

Ergonomics

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

```
1 a.chunks_mut(split_length)
2   .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

```
1  a.chunks_mut(split_length)
2      .par_bridge()
3      .for_each(bsort);
```

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

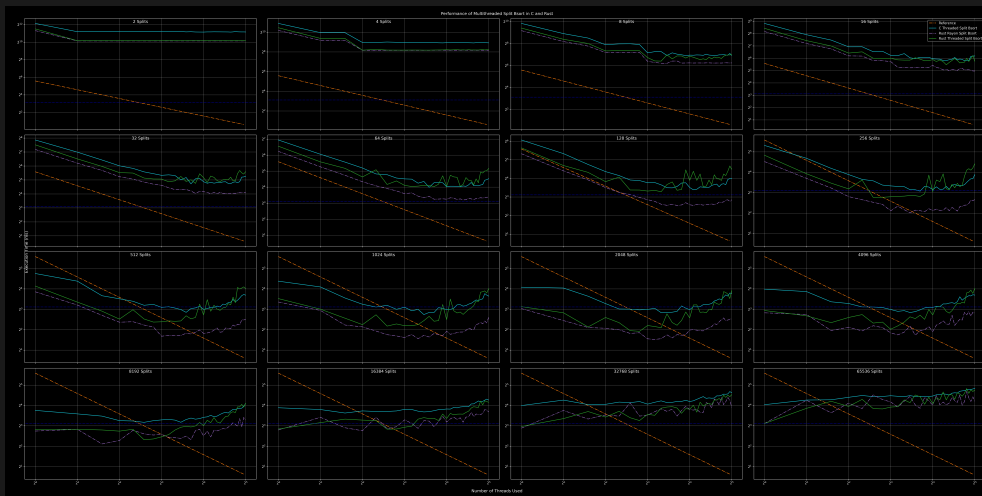


Figure 2: Preformacne results from bsort project. Note that the Rust reference sort is not plotted as it was under 1ms, compared to 8ms for the C qsort.

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 Rules¹
 - 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 Rules²
 - 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.

¹Klabnik et al. 2024, Ch04-01

²Klabnik 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
- Vulkan
- 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 → tokens (identifiers, literals, operators)
- Syntax & semantic analysis: tokens → AST, type checking
- Emit language independent IR

Optimization

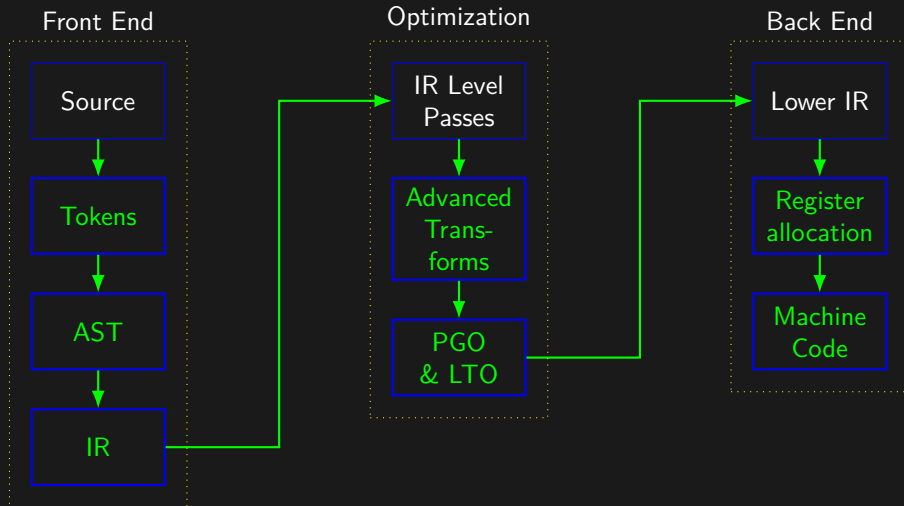
- IR level passes: constant folding, dead code elimination, algebraic simplification
- Advanced transforms: function inlining, loop unrolling, vectorization
- Profile or link time optimizations for extra performance

Back End

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

What is a compiler?

Translation pipeline



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

¹Bychkov 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

Fractals

Mandelbrot and Burning Ship

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

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

The Burning Ship fractal is defined similarly but the sequence is

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

Mandelbrot Kernel

```
1 pub unsafe fn mandelbrot(  
2     n_re: usize,  
3     n_im: usize,  
4     re_min: f32,  
5     re_max: f32,  
6     re_range: f32,  
7     im_min: f32,  
8     im_max: f32,  
9     im_range: f32,  
10    zn_limit: u32,  
11    out: *mut u8,  
12 ) {  
13     let idx_linear = thread::index() as usize;  
14     let idx = thread::index_2d();  
15     let idx_re = idx[1];  
16     let idx_im = idx[0];  
17  
18     let c_re = re_range  
19         * (idx_re as f32 / (n_re - 1) as f32)  
20         + re_min;  
21     let c_im = im_range  
22         * (idx_im as f32 / (n_im - 1) as f32)  
23         + im_min;
```

```
24     let mut z_re = c_re;  
25     let mut z_im = c_im;  
26     for _ in 0..zn_limit / ZN_SKIP {  
27         for _ in 0..ZN_SKIP {  
28             (  
29                 z_re, z_im,  
30             ) = (  
31                 z_re * z_re - z_im * z_im +  
32                     c_re,  
33                 2.0 * z_re * z_im + c_im,  
34             );  
35             if z_re * z_re + z_im * z_im > 4.0 {  
36                 let elem = &mut *out.add(  
37                     idx_linear);  
38                 *elem = 255;  
39                 break;  
40             }  
41         }  
}
```

Launching the Kernel

```
1      unsafe {  
2          launch!(  
3              module.mandelbrot<<<<grid_size, block_size,  
4                  N_RE,  
5                  N_IM,  
6                  re_min,  
7                  re_max,  
8                  re_range,  
9                  im_min,  
10                 im_max,  
11                 im_range,  
12                 zn_limit,  
13                 out_gpu.as_device_ptr(),  
14             )  
15         )?;  
16     }
```

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

Output Image

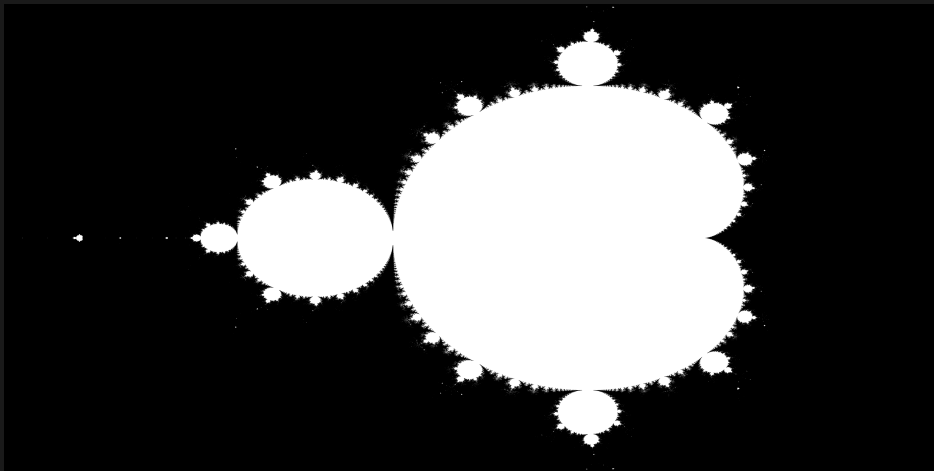


Figure 3: Mandelbrot set image of size 8192x4096

Fractals

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

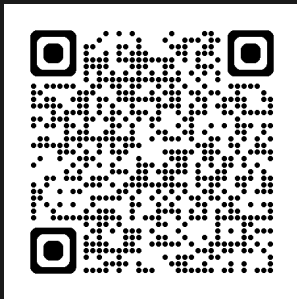
Method	Time (ms)	
	Machine ¹	Machine ²
CPU create point grid	175.357	62.11
GPU using CPU points	0.552	5.23
CPU using rayon	143.647	499.39
GPU f32	0.453	3.48
GPU f64	12.272	101.42

¹12C/24T 3900X // RTX 4090

²4C/8T 7700K // RTX 2070 Super

Live Demo

Repository



<https://github.com/igor-semyonov/ece555-presentation>

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.