Bridging CPU and GPU in Rust

Niek Aukes Andrei Begu

University of Twente

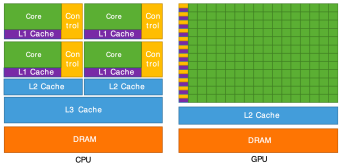
November 17, 2025

Table of Contents

Introduction

GPU Model

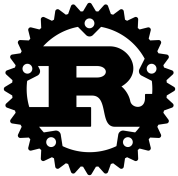
- GPUs execute thousands of lightweight threads in parallel
- One thread runs the same kernel on different data
- Ideal for data-parallel workloads such as image rendering or simulation
- Heterogeneous computing: best of both worlds!



https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Rust

- Modern systems programming language focused on performance and safety
- Enforces memory safety without a garbage collector
- Strong type system and compiler checks
- Built-in tooling: `cargo` for builds and testing, `crates.io` for packages
- Previous work on integration with GPUs



Rust (CPU)

```rust
fn vadd(a: &[f32], b: &[f32]) -> Vec<f32> {
    let mut c = vec![0.0f32; a.len()];
    for i in 0..a.len() {
        c[i] = a[i] + b[i];
    }
    c
}

fn main() {
    let a = vec![1.0, 2.0, 3.0, 4.0, ...];
    let b = vec![10.0, 20.0, 30.0, 40.0, ...];
    let c = vadd(&a, &b);
    println!("{:?}", c);
}
```

OpenCL I

```
1  use ocl::{Buffer, ProQue};
2
3  const KERN: &str = r#"
4  __kernel void vadd(__global const float* A,
5                     __global const float* B,
6                     __global float* C,
7                     const int N) {
8      int i = get_global_id(0);
9      if (i < N) C[i] = A[i] + B[i];
10 }"#;
11
12 fn main() {
13     let a = vec![1.0, 2.0, 3.0, 4.0, ...];
14     let b = vec![10.0, 20.0, 30.0, 40.0, ...];
15     let n = a.len();
16     let mut c = vec![0.0f32; n];
17
18     let pq = ProQue::builder().src(KERN).dims(n).build().unwrap();
```

OpenCL II

```
19      let a_buf: Buffer<f32> = pq.buffer_builder().copy_host_slice(&a).build().unwrap();
20      let b_buf: Buffer<f32> = pq.buffer_builder().copy_host_slice(&b).build().unwrap();
21      let c_buf: Buffer<f32> = pq.create_buffer().unwrap();
22
23      unsafe {
24          pq.kernel_builder("vadd").arg(&a_buf).arg(&b_buf)
25              .arg(&c_buf).arg(n as i32)
26              .build().unwrap()
27              .enq().unwrap();
28      }
29
30      c_buf.read(&mut c).enq().unwrap();
31  }
```

CUDA C I

```
1  __global__ void vadd(const float* A, const float* B, float* C, int N) {
2      int i = blockIdx.x * blockDim.x + threadIdx.x;
3      if (i < N) C[i] = A[i] + B[i];
4  }
5
6  int main() {
7      float hA[] = {1, 2, 3, 4, ...};
8      float hB[] = {10, 20, 30, 40, ...};
9
10     const size_t N = sizeOf(hA)/sizeOf(hA[0]);
11     const size_t bytes = N * sizeof(float);
12     float hC[N];
13
14     float *dA, *dB, *dC;
15     cudaMalloc(&dA, bytes);
16     cudaMalloc(&dB, bytes);
17     cudaMalloc(&dC, bytes);
```

CUDA C II

```
18      cudaMemcpy(dA, hA, bytes, cudaMemcpyHostToDevice);
19      cudaMemcpy(dB, hB, bytes, cudaMemcpyHostToDevice);
20
21      vadd<<<128, 4>>>(dA, dB, dC, N);
22      cudaMemcpy(hC, dC, bytes, cudaMemcpyDeviceToHost);
23
24      cudaFree(dA); cudaFree(dB); cudaFree(dC);
25  }
```

Hybrid Rust Compiler

```rust
#[kernel]
unsafe fn vadd(a: &[f32], b: &[f32], mut c: Buffer<f32>, n: i32) {
    let i = gpu::global_tid_x() as i32;
    if i < n {
        let sum = a[i as usize] + b[i as usize];
        c.set(i as usize, sum);
    }
}
fn main() {
    let a = vec![1.0, 2.0, 3.0, 4.0, ...];
    let b = vec![10.0, 20.0, 30.0, 40.0, ...];
    let c_buf = Buffer::<f32>::alloc(a.len() as usize).unwrap();
    let n = a.len();

    vadd.launch(128, 4, &a, &b, c_buf, n).unwrap();

    let result = c_buf.retrieve().unwrap();
}
```

Migration Case

Migration Case

- Goal: try-out the compiler on a (small, 300 LOC) real-world case
- Target: existing open-source Mandelbrot renderer in Rust for the CPU
- Per-pixel computation; embarrassingly parallel
- Three stages: migration, testing, maintenance

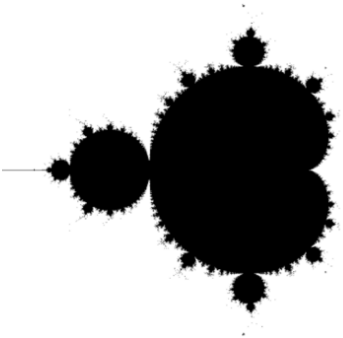


Figure: Mandelbrot fractal render

Migration (1/4): CPU Version

```
1  for t in 0..threads {
2      thread::spawn(move || {
3          let mut slice = vec![u8::MAX; this_height * image_width];
4          for i in 0..this_height {
5              for j in 0..image_width {
6                  let x = real_start + (j * real_step);
7                  let y = i_start - (i * i_step);
8                  let p = Complex::new(x, y);
9                  if p.is_stable(iterations) {
10                     slice[i * image_width + j] = 0;
11                 }
12             }
13         }
14         tx.send((t, slice)).unwrap();
15     });
16 }
```

Migration (2/4): Extract Function + Annotate

```
#[kernel] // annotated as GPU kernel
unsafe fn mandelbrot(mut image: Buffer<u8>, offset: usize, ...) {
    for i in 0..this_height {
        for j in 0..image_width {
            let x = real_start + (j * real_step);
            let y = i_start - (i * i_step);
            let p = Complex::new(x, y);
            if p.is_stable(iterations) {
                image.set(i * image_width + j, 0);
            }
        }
    }
}
```

Migration (3/4): Thread Indexing

```
1  #[kernel]
2  unsafe fn mandelbrot(mut image: Buffer<u8>, offset: usize, ...) {
3      let pos = offset + gpu::global_tid_x();
4      let i = pos / image_width;
5      let j = pos % image_width;
6
7      let x = real_start + (j * real_step);
8      let y = i_start - (i * i_step);
9      let p = Complex::new(x, y);
10     if p.is_stable(iterations) {
11         image.set(i * image_width + j, 0);
12     }
13 }
```

Migration (4/4): Host Launch

```
#[kernel]
unsafe fn mandelbrot(mut image: Buffer<u8>, offset: usize, ...) {
    let pos = offset + gpu::global_tid_x();
    let i = pos / image_width;
    let j = pos % image_width;

    let x = real_start + (j * real_step);
    let y = i_start - (i * i_step);
    let p = Complex::new(x, y);
    if p.is_stable(iterations) {
        image.set(i * image_width + j, 0);
    }
}

mandelbrot.launch(threads, blocks, image, 0, ...).unwrap();
```

Migration: The Rest?

The rest?

The same as before!

Tooling Project structure

Use of packages `Complex` type & logic

Image logic Unit tests

Testing

- Existing CPU unit tests run unchanged
- GPU kernels reuse the same tested logic and data types
- One toolchain → regression tests trivial

```rust
#[test]
fn test_mandelbrot_gpu_simple_default() {
    let options = MandelbrotCpu::default();
    let gpu_image = build_mandelbrot_gpu_simple(&options);
    let cpu_image = build_mandelbrot_cpu_simple(&options);

    assert_eq!(gpu_image, cpu_image);
}
```

Maintenance

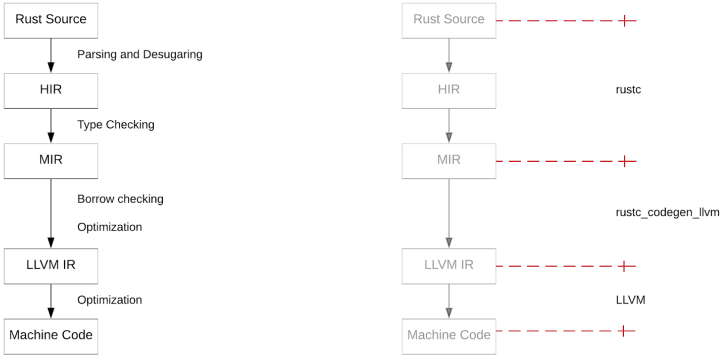
- Single-source model reduces long-term maintenance overhead
- CPU and GPU code share the same data structures, types, and logic
- Refactoring affects both sides simultaneously
- All of Rust's toolchain benefits carry over (borrow checks, code smell hints, refactoring tools)
- Can use third party tools in the Rust ecosystem (linters, IDEs, formatters)

Compiler Insights

From Migration to Compilation

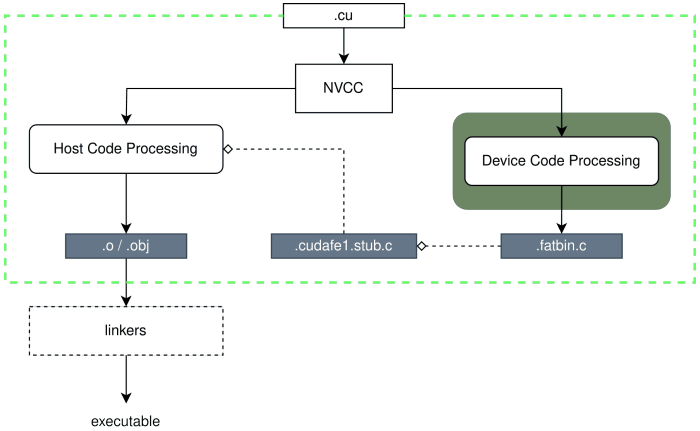
- We've seen how Rust can run on GPUs, how does that *actually* work?
- To understand our design, let's look at existing compiler architectures first.

Compiler Architectures: Rust's Perspective



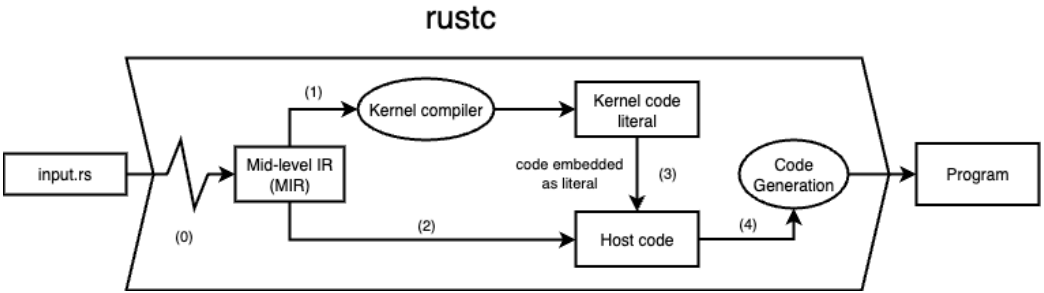
https://jason-williams.co.uk/posts/a-possible-new-backend-for-rust/

NVCC: The CUDA Compiler Architecture



https://hpcgpu.mini.pw.edu.pl/cuda-compilation-toolchain/

Our Hybrid Compiler Architecture



Advantages and Drawbacks

- Advantages

Advantages and Drawbacks

- Advantages
  - Single pass compiler → no build scripts
  - Single source compilation → no fragmented code

Advantages and Drawbacks

- Advantages
  - Single pass compiler → no build scripts
  - Single source compilation → no fragmented code
  - Safety Guarantees from Rust → including boundary

Advantages and Drawbacks

- Advantages
  - Single pass compiler → no build scripts
  - Single source compilation → no fragmented code
  - Safety Guarantees from Rust → including boundary
- Drawbacks
  - Compiler maintenance and synchronization with `rustc`

Conclusion

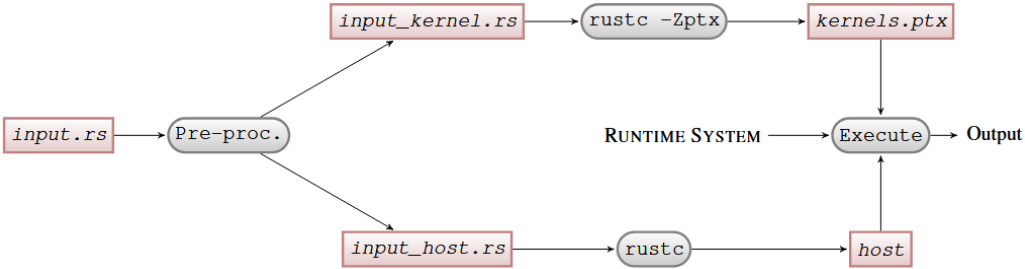
Prototype and Future Work

- Current system is a functional prototype

Prototype and Future Work

- Current system is a functional prototype
- Future directions:
  - Update the compiler to a newer version
  - Integration with `Rust-GPU`
  - Cross-platform support
- Goal: Integrated, cross-platform, safe GPU programming with Rust guarantees

Holk's Rust GPU Compiler (2013)



E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine and N. D. Matsakis

"GPU Programming in Rust: Implementing High-Level Abstractions in a Systems-Level Language"

Rust's Safety Philosophy

- Memory safety without garbage collection
- Ownership model ensures one writer, many readers
- Borrow checker enforces valid lifetimes and no aliasing
- Type system catches data races and invalid access at compile time

Safety: design goals

- Preserve Rust's memory-safety guarantees inside kernel code
- Make illegal kernel invocations a *compile-time* error
- Make it possible to reason about safety in the **CPU-GPU boundary**
- Restrict which types can be moved to device memory
- Keep unsafe surface minimal and easy to audit

Type-Safe Kernel Invocation

```
const vadd: Kernel<u32, (&[f32], &[f32], Buffer<f32>, i32)> = Kernel {
    code: &'static [u8],
    name: &'static str,
    // With Phantom type parameters
}
```

- Compiler encodes argument types in the Kernel type
- Prevents launching kernels with wrong parameters

GPU Memory Management Traits

- **DeepCopy**: asserts a type can be safely `memcpy`'d to the GPU
  - Implemented for plain-old-data types (e.g. `f32`)
  - Guarantees no internal pointers or references

GPU Memory Management Traits

- **DeepCopy**: asserts a type can be safely `memcpy`'d to the GPU
  - Implemented for plain-old-data types (e.g. `f32`)
  - Guarantees no internal pointers or references
- **DSend**: defines how a type is sent to device memory
  - Provides APIs for buffer allocation, transfer, and synchronization
  - Automatically implemented for `DeepCopy` types
  - Can be manually implemented for complex types (e.g. `&[T]` or `Buffer<T>`)

Pointer Aliasing and Mutable Access

- GPU APIs freely expose raw pointers to shared memory
  - Makes aliasing and data races difficult to reason about
  - No compile-time guarantees of exclusivity

Pointer Aliasing and Mutable Access

- GPU APIs freely expose raw pointers to shared memory
  - Makes aliasing and data races difficult to reason about
  - No compile-time guarantees of exclusivity
- Rust forbids mutable aliasing:
  - One mutable reference (`&mut T`) or many shared (`&T`) references
  - Enforced statically by the borrow checker

Pointer Aliasing and Mutable Access

- GPU APIs freely expose raw pointers to shared memory
  - Makes aliasing and data races difficult to reason about
  - No compile-time guarantees of exclusivity
- Rust forbids mutable aliasing:
  - One mutable reference (`&mut T`) or many shared (`&T`) references
  - Enforced statically by the borrow checker
- Our "unsafe" `Buffer<T>` type provides a controlled escape for GPU memory:
  - Signals that aliasing occurs across host/device boundaries
  - Final safety remains with the programmer

Type Checking through a Library

- Our cuda library governs type-checking and kernel launching
- The library defines:
  - Valid kernel argument types
  - (safe) Memory management rules

Type Checking through a Library

- Our `cuda` library governs type-checking and kernel launching
- The library defines:
  - Valid kernel argument types
  - (safe) Memory management rules
- This enables:
  - Eventually platform-dependent safety semantics (CUDA, Vulkan, ...)
  - Compiler simplicity $\rightarrow$ Rust's type system does the enforcement
  - Extendability from the community, due to compiler independence

Type-Safe Kernel Invocation

```rust
#[kernel]
unsafe fn vadd(a: &[f32], b: &[f32], mut c: Buffer<f32>, n: i32) {
    let i = gpu::global_tid_x() as i32;
    if i < n {
        let sum = a.get(i as usize) + b.get(i as usize);
        c.set(i as usize, sum);
    }
}

// Compiles to:
const vadd: Kernel<u32, (&[f32], &[f32], Buffer<f32>, i32)> = Kernel {
    code: &'static [u8],
    name: &'static str,
    // With Phantom type parameters
}
```