Bridging CPU and GPU in Rust

Niek Aukes[1], Cristian-Andrei Begu[1] and Georgiana Caltais[1]

[1]University of Twente, The Netherlands

Abstract

As heterogeneous computing becomes widespread, fragmented workflows for managing separate CPU and GPU codebases create significant challenges for software evolution. This paper introduces a unified compilation model for Rust that addresses this fragmentation by treating GPU kernels as native functions. Using a prototype compiler, we demonstrate the benefits of treating GPU kernels as native code with a case study: migrating an existing multi-threaded CPU program to use GPU acceleration. We introduce a unified compilation model that contributes to reducing maintenance overhead and technical debt in migrating and evolving heterogeneous systems.

Keywords

Rust, compiler, hybrid compilation, GPU acceleration, CUDA, heterogeneous programming

1. Introduction

Heterogeneous computing, in which a CPU orchestrates GPUs executing highly parallel kernels, has become widespread across various disciplines, including scientific simulation, real-time rendering, and emerging artificial intelligence workloads. This is reflected in real-world adoption: the June 2025 TOP500 list [1] indicates that over one-third of high-performance systems use GPU accelerators. Yet, despite this hardware revolution, the software infrastructure remains fragmented: host code is traditionally written in C, C++ or Python, while performance-critical kernels are implemented in specialized languages like CUDA C [2] or OpenCL C [3]. This divide may result in developers maintaining parallel code bases, which may lead to increased maintenance costs. GPU-specific languages further suffer from a narrower ecosystem: they lack the rich standard libraries and package-management facilities of mainstream CPU languages. Consequently, even light refactoring requires edits in two languages, validation across two toolchains, and custom build steps. These fractured workflows may not only hinder development but also incur technical debt, undermining the maintainability and long-term evolution of heterogeneous applications.

Several tools have attempted to reduce this complexity. In C and C++, OpenMP [4] provides directives that allow loops to be parallelized on GPU hardware via annotations. This lowers the entry cost compared to writing explicit CUDA or OpenCL, but the model is applicable to loop-based parallel patterns and offers less control over execution and optimization than explicit GPU programming. High-level languages have also partially addressed the divide: Numba [5], for example, compiles annotated Python functions into CUDA kernels, offering in-place acceleration within a single Python syntax and integration with the broader Python stack. While performance limitations from Python's dynamic typing, interpreted nature, and the global interpreter lock primarily affect non-accelerated code, Numba introduces its own challenges. Its programming model is often non-idiomatic for Python developers, and its error messages can be difficult to understand. Compiled languages have fared no better: GPU programs typically resort to OpenCL APIs, while NVIDIA's CUDA C++ compiler still requires a two-stage build that extracts and compiles kernels separately [6]. These approaches leave developers with split toolchains and additional build complexity.

Despite these challenges, Rust's combination of zero-cost abstractions, a compile-time ownership model and borrow checker (the compiler enforcing safe memory sharing rules), and a comprehensive standard library [7] positions it as an ideal base for unified heterogeneous programming. These checks,

BENEVOL25: The 24th Belgium-Netherlands Software Evolution Workshop, Autumn 2025, Enschede, The Netherlands
n.aukes@student.utwente.nl (N. Aukes); c.begu@student.utwente.nl (C. Begu); g.g.c.caltais@utwente.nl (G. Caltais)


together with a strong static type system, catch many concurrency and memory safety errors at compile time, while Cargo and `crates.io` provide a mature package ecosystem with over 190,000 packages [8]. Prior work by Holk et al. [9] already demonstrated the possibility of extending Rust with GPU support. Their prototype showed that Rust could be used as a base for hybrid compilation, by using a pre-processing step to separate host and device code.

More recently, projects such as Rust-GPU [10] and Rust-CUDA [11] have advanced the state of GPU programming in Rust by showing that safe, idiomatic Rust can directly target GPU execution environments. Their progress highlights the potential for a convergent toolchain where unified compilation and GPU-first abstractions reinforce each other. Alongside this, Faé and Griebler developed a macro-based GPU accelerator [12] demonstrating an alternative path, trading language flexibility for a simplified experience. These efforts point toward a growing ecosystem where our compiler can interoperate and extend the reach of Rust on heterogeneous platforms.

Our contribution is to showcase a unified compilation approach for Rust that treats GPU kernels as native Rust functions, building on prior compiler efforts. This reduces the need for separate kernel languages and build steps, making it easier for developers to write, migrate, refactor, and test heterogeneous applications within a familiar Rust workflow.

The remainder of this paper is organized as follows. Section 2 presents the system design, detailing the approach that enables unified CPU–GPU compilation. Section 3 demonstrates the proposal through a case study: migrating, testing, and evolving an existing Rust program using the hybrid compiler. Section 4 discusses the implications of our compiler, its limitations, and potential directions for the future.

2. The Unified Compiler

In this section, we describe our unified compiler and the changes made to the Rust compilation pipeline to treat GPU kernels as native Rust functions, without disrupting the original host workflow. A high-level overview of the unified compiler is shown in Figure 1. For an in-depth description of its design and implementation, we refer readers to our earlier work [13, 14]. The source code, along with installation instructions, is available at https://github.com/NiekAukes/rust-gpu-hybrid-compiler

In standard Rust, the compiler parses source code into an Abstract Syntax Tree, lowers it to the High-level Intermediate Representation for type checking, then to the Mid-level Intermediate Representation (MIR) for borrow checking and optimizations, and finally emits LLVM IR for machine code generation [15]. The compiler executes these stages via "queries", on-demand computations that retrieve information such as a variable's type or the body of a function when required.

The design of Holk et al. [9] introduces a pre-processing step that identifies functions marked with the `#[kernel]` attribute and generates two separate source code artifacts: one for the host and one for the device. The host code is compiled with the standard Rust compiler, while the device code is handled

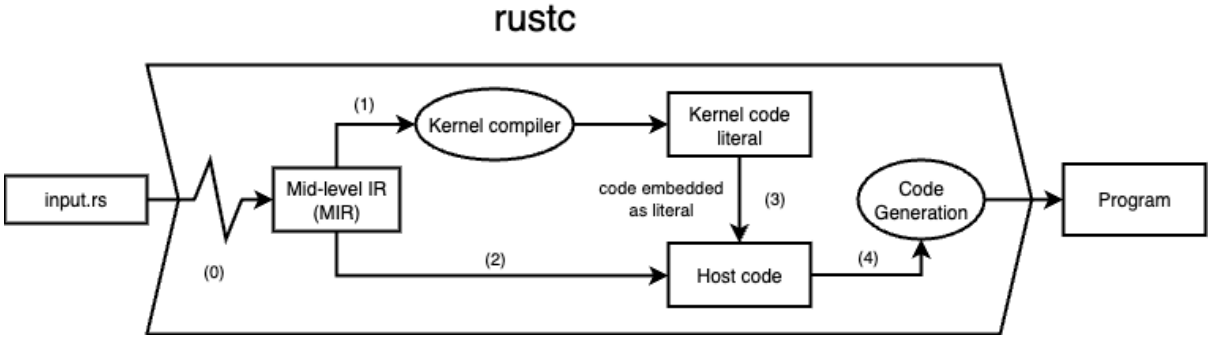

Figure 1: Compilation of annotated functions: MIR splits into GPU (1-3) and CPU (2) paths, converging in LLVM backend (4); parsing, type, and borrow checks are shared (0)

by a modified Rust compiler that emits GPU bytecode. These two outputs are then re-integrated at runtime via OpenCL bindings.

In our approach, functions marked with a `#[kernel]` attribute trigger the compiler to fork the MIR into two paths. As illustrated in Figure 1, the GPU path (1) lowers the MIR into GPU bytecode, which is then embedded back into the CPU MIR as a static object (3). The CPU path (2) continues with both the host logic and the embedded GPU code. From there, the combined MIR is passed into the standard LLVM backend for code generation (4), producing a single executable that contains both CPU and GPU components. This design allows all parsing, type checking, and borrow checking (0) to remain shared, while the final binary is produced by a single toolchain from a single source.

Our modified compiler preserves the full set of Rust's core language features[1], including type checking, borrow checking, lifetimes, generics, and traits. Beyond these, it also supports several advanced capabilities, such as defining custom panic (exception) handlers and performing dynamic memory allocation directly on the GPU. All supported features should behave consistently with their CPU counterparts, which provides several advantages: developers can reuse the same source code across CPU and GPU, and rely on identical execution semantics. Importantly, our compiler retains full compatibility with existing Rust projects and tooling.

Although our model has many benefits, there are also limitations with the current implementation of the programming model. Most importantly, only CUDA-enabled GPUs are currently supported by our compiler. This is because the architectural design of Nvidia GPUs allows many CPU execution constructs that are not available on other platforms. Examples include dynamic dispatch of functions (object-oriented programming) and function recursion. CUDAs open-specification counterpart, Vulkan [16], simply do not support these constructs that the Rust Compiler relies on [17, ch.2.16].

Beyond traditional models, it is also useful to contrast our work with ongoing efforts in the Rust ecosystem. Rust-GPU targets shader and kernel development and focuses on compiling Rust directly into GPU targets [11, 10]. While it enables a safe, expressive programming model for GPU code, it still requires complex build steps and careful separation between CPU, GPU, and shared logic [18]. By contrast, our approach emphasizes single-source development and unifies CPU and GPU code within one compiler pipeline, which simplifies maintenance and testing across heterogeneous backends.

Another approach is that of the macro-based accelerator[12], which provides a higher-level abstraction by transforming constrained Rust code into GPU-executable code through procedural macros. This model reduces developer burden, but it achieves this simplicity by enforcing restrictions on how GPU code is written. Our compiler takes the opposite stance: it supports the full Rust core language and many advanced features within GPU kernels, while preserving execution consistency between CPU and GPU paths.

3. A Mandelbrot Example

To showcase the potential of our hybrid compilation approach, we adapt an existing open-source Mandelbrot set generator, originally a traditional multi-threaded CPU program in Rust, to our modified compiler. The Mandelbrot set (Figure 2) is a well-known example in parallel computing: a fractal rendered by applying the same iterative computation independently to each pixel of the image, making it an ideal workload for both CPU parallelism and GPU acceleration.

The original project is available at https://github.com/JohnTWilkinson/Gendelbrot and our migrated version is available at https://github.com/NiekAukes/Gendelbrot. By working with an existing codebase, we provide a supporting example that illustrates how our approach can introduce GPU

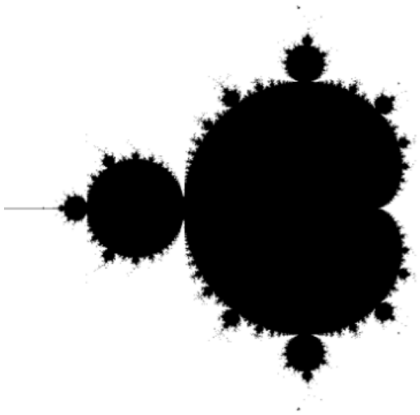

Figure 2: A Mandelbrot fractal.

[1]Core features include all language constructs and the core library, but exclude the standard library

acceleration into software not originally designed for heterogeneous execution.

The first stage in the migration moves the computation from CPU threads to GPU execution. In a conventional CUDA or OpenCL port, the main computation function would be rewritten in a GPU-specific language, moved into a separate module, compiled with a separate toolchain, and manually integrated with the host code. In contrast, our migration, shown in Figure 3, keeps the computation entirely in Rust. The existing threaded loop (lines 1–18, left) is replaced by a GPU kernel (lines 1–17, right). Instead of spawning threads explicitly (line 2, left), the kernel is launched once from the host (lines 19–21, right). The nested loops over pixel coordinates (lines 4–5, left) are replaced by a computation based on the thread index (lines 5–7, right), which directly maps each GPU thread to a pixel. The coordinate calculation (lines 6–8, left vs. lines 9–11, right) remains the same, and the `Complex` type used in both versions refers to the same type definition (line 10, left vs. line 13, right), reused directly inside the GPU kernel. The image update (lines 11–13, left vs. lines 13–15, right) also maps directly between the two versions. The rest of the codebase, including its use of Rust libraries and the Cargo build system, remain identical.

Compared with traditional migration approaches, our model sits at a middle ground between OpenMP and OpenCL. Like OpenMP, the changes required are extensions within the same source language, which avoids splitting the program into separate codebases. At the same time, similar to OpenCL, developers must still understand the parallel execution model: loops do not transform automatically but are re-expressed through thread identifiers. In addition, knowledge of GPU primitives such as synchronization and thread grouping, remains important when adapting more complex computations.

```
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 +
7                      (j * real_step);
8              let y = i_start - (i * i_step);
9
10             let p = Complex::new(x, y);
11             if p.is_stable(iterations) {
12                 slice[i * image_width + j] = 0;
13             }
14         }
15         }
16         tx.send((t, slice)).unwrap();
17     });
18 }
```

```
1  #[kernel]
2  fn mandelbrot(
3      mut image: Buffer<u8>, offset: usize, ...
4  ) {
5      let pos = offset + gpu::global_tid_x();
6      let i = pos / image_width;
7      let j = pos % image_width;
8
9      let x = real_start +
10             (j * real_step);
11     let y = i_start - (i * i_step);
12
13     let p = Complex::new(x, y);
14     if p.is_stable(iterations) {
15         image.set(i * image_width + j, 0);
16     }
17 }
18
19 mandelbrot.launch(
20     threads, blocks, image, 0, ...
21 ).unwrap();
```

Figure 3: Migration from CPU threads (left) to GPU kernel (right).

An additional benefit during migration is that testing requires no changes to existing workflows. Since the hybrid compiler allows GPU kernels to call the same logic as the CPU path, existing unit tests, such as those checking the behavior of the `Complex` type, run without modification within Cargo's testing framework. Beyond these unchanged unit tests, the unified workflow makes it trivial to add regression tests that compare the outputs of the CPU and GPU implementations. Such comparisons are more complex in a split-language environment, where separate toolchains make it impractical to compare host and device code in a single test suite.

After migrating Gendelbrot, we validated the new implementation using Rust's built-in testing framework [7, ch.11]. Before migration, Gendelbrot already included a suite of unit tests testing the CPU implementation of the algorithm. During migration, we preserved all of these existing tests without modification, since the hybrid compiler allows GPU kernels to invoke the same logic as the

```rust
#[test]
fn test_mandelbrot_gpu_simple_default() {
    let options = MandelbrotCpu::default();
    let image = build_mandelbrot_gpu_simple(&options);
    assert_eq!(image.len(), options.image_width * options.image_height);
    let expected_image = build_mandelbrot_cpu_simple(&options);

    if image != expected_image {
        println!("GPU image does not match expected output.");
        export_image(&image, options.image_width, options.image_height, "gpu_output.png");
        export_image(&expected_image, options.image_width, options.image_height,
        ↪  "expected_output.png");
        assert!(false, "GPU image does not match expected output.");
    }
}
```

Figure 4: Example of testing the GPU implementation against the verified CPU implementation

CPU path. In addition, we extended the suite with regression tests that directly compare the output of the GPU execution against the verified CPU baseline (Figure 4). These tests also provide a systematic way to detect potential divergences with further optimization or refactoring.

Looking forward, the single-source model also has implications for long-term evolution. In a traditional CUDA or OpenCL program, every structural or behavioral change requires coordination and mirroring between host and device definitions, often across different languages and toolchains. This duplication can add extra work during refactoring and may increase the likelihood of inconsistencies, such as mismatched data layouts, incomplete updates to kernel logic, or divergent type definitions. By ensuring that both CPU and GPU code operate on the same shared structures and are validated in the same compiler pass, our approach should reduce this type of user errors. Our approach should also reduce technical debt and enable heterogeneous programs to evolve similarly to their CPU-only counterparts.

4. Conclusion

We presented a single-source compilation model that facilitates the evolution of existing Rust programs into heterogeneous CPU/GPU applications with minimal code changes. This model operates within the standard Cargo ecosystem, enabling developers to leverage familiar tooling and a large package registry. Our approach also improves long-term maintainability by allowing host and device code to share data structures and logic, which simplifies future refactoring efforts. However, this approach introduces its own set of challenges. Currenty, our work only supports CUDA-enabled GPUs, which makes the current work impractical for cross-platform programming. Supporting cross-platform tools like Vulkan remains challenging, but possible. Additionally, the reliance on a modified compiler requires dedicated maintenance to keep pace with the official Rust toolchain's evolution.

Several paths for future work could improve the model's viability. A performance analysis against established toolchains could guide the implementation of GPU-specific optimization passes. Additional developer support could be provided through GPU-specific lints that detect anti-patterns, which could be integrated into Cargo's workflow. Moving beyond the current NVIDIA-only backend toward cross-platform APIs like Vulkan would greatly increase the compiler's relevance. Collaborating with initiatives such as Rust-GPU [10] and Rust-CUDA [11] could play a pivotal role in achieving this. Refactoring tools could also be developed to help automate the migration of existing CPU code. Finally, exploring ways to reduce the maintenance of the compiler fork, such as through a more modular architecture, would help with its maintainability.

Acknowledgements. This work is partially supported by the CYCLIC project (file no. OCENW.XL.23.089) of the research programme Open Competition Domain Science XL, by the Dutch Research Council

(NWO) under the grant https://doi.org/10.61686/FHYZO53064.

References

[1] June 2025 TOP500, 2025. URL: https://www.top500.org/lists/top500/2025/06/, Accessed: August 2025.

[2] NVIDIA Corporation, CUDA C Programming Guide, NVIDIA, 2024. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/, Accessed: August 2025.

[3] Khronos Group, The OpenCL C Specification, Khronos Group, 2025. URL: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_C.html, Accessed: August 2025.

[4] OpenMP Architecture Review Board, OpenMP Application Programming Interface, OpenMP ARB, 2024. URL: https://www.openmp.org/specifications/, Accessed: August 2025.

[5] S. K. Lam, A. Pitrou, S. Seibert, Numba: a LLVM-based Python JIT compiler, in: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, LLVM '15, Association for Computing Machinery, New York, NY, USA, 2015. URL: https://doi.org/10.1145/2833157.2833162. doi:10.1145/2833157.2833162.

[6] NVIDIA CUDA Compiler Driver 13.0 documentation, 2025. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/, Accessed: August 2025.

[7] S. Klabnik, C. Nichols, The Rust Programming Language, 2nd Edition, No Starch Press, 2023. URL: https://doc.rust-lang.org/book/.

[8] crates.io: Rust package registry, 2025. URL: https://crates.io/, Accessed: August 2025.

[9] E. Holk, M. Pathirage, A. Chauhan, A. Lumsdaine, N. D. Matsakis, GPU programming in Rust: Implementing high-level abstractions in a systems-level language, in: 2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, 2013, pp. 315–324. doi:10.1109/IPDPSW.2013.173.

[10] Rust-GPU, Embark Studios, rust-gpu: Rust as a first-class language and ecosystem for GPU graphics & compute shaders, https://github.com/Rust-GPU/rust-gpu, 2020-2025. Accessed: September 2025.

[11] Rust-CUDA: Ecosystem of libraries and tools for writing and executing fast GPU code fully in Rust, https://github.com/Rust-GPU/Rust-CUDA, 2021-2025. Accessed: September 2025.

[12] L. Faé, D. Griebler, Towards gpu parallelism abstractions in rust: A case study with linear pipelines, in: Anais do XXIX Simpósio Brasileiro de Linguagens de Programação, SBC, Porto Alegre, RS, Brasil, 2025, pp. 75–83. URL: https://sol.sbc.org.br/index.php/sblp/article/view/36951. doi:10.5753/sblp.2025.13152.

[13] N. Aukes, Hybrid compilation between GPGPU and CPU targets for Rust, Thesis, University of Twente, Enschede, 2024. URL: https://purl.utwente.nl/essays/100981.

[14] C.-A. Begu, Enabling Idiomatic Rust for Hybrid CPU/GPU Programming, Thesis, University of Twente, Enschede, 2025. URL: https://purl.utwente.nl/essays/107371.

[15] The Rust Project Developers, The rustc Development Guide, https://rustc-dev-guide.rust-lang.org/, 2025. Accessed: August 2025.

[16] Vulkan® 1.4.326 - a specification, 2025. URL: https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html, Accessed: September 2025.

[17] J. Kessenich, B. Ouriel, R. Krisch, Spir-v specification, Khronos Group 3 (2018) 17.

[18] C. Legnitto, Rust running on every gpu, 2025. URL: https://rust-gpu.github.io/blog/2025/07/25/rust-on-every-gpu.