Enabling Idiomatic Rust for Hybrid CPU/GPU Programming

CRISTIAN-ANDREI BEGU, University of Twente, The Netherlands

Rust's strong safety guarantees and performance make it a compelling candidate for General-Purpose GPU (GPGPU) programming. Building upon prior work that demonstrated a hybrid Rust compiler capable of targeting both CPU and GPU from a single source, we address a key limitation: the inability to use many standard Rust features within GPU kernel code. This limitation arises because compiler 'language items' ('lang-items'), which implement core features, often rely on CPU-specific assumptions that are invalid in the GPU context. To overcome this, we extend the existing hybrid compiler with a context-dependent lang-item system. This system enables the compiler to select specialized, GPU-safe implementations for lang-items when compiling kernel code, while retaining standard implementations for host code. As a result, a significantly broader range of idiomatic Rust features becomes available within GPU kernels, substantially improving the practicality and expressiveness of integrated hybrid compilation in Rust.

1 INTRODUCTION

The Rust programming language continues to gain traction for systems development, offering compelling advantages in safety, concurrency, and performance [7]. Simultaneously, the practice of using Graphics Processing Units (GPUs), originally designed for rendering complex visuals, for general-purpose computational tasks—known as GPGPU computing—provides mass parallelism computing essential for accelerating demanding workloads across various scientific and industrial domains [9]. Combining Rust's technologies with the parallel processing power of GPUs holds significant promise, potentially leading to safer, more maintainable high-performance applications. However, integrating GPU acceleration smoothly into compiled languages like Rust remains challenging compared to the relative ease seen in dynamic languages like Python, where libraries such as Numba allow defining GPU kernels alongside host code [8].

Recent work by Aukes [1] made significant progress by developing a prototype Rust compiler capable of integrated hybrid compilation. This prototype allows developers to define GPU kernels directly within their Rust source code alongside host logic, processing both through a modified compiler pipeline. While this demonstrated the feasibility of a unified workflow, practical use revealed a fundamental obstacle: many standard Rust features and idioms fail within GPU kernels.

We identify the root cause of this obstacle as the incompatibility of standard Rust 'language items' ('lang-items') [14] with the constrained GPU execution environment. These compiler primitives, implementing core semantics like error handling or dynamic memory allocation, typically assume CPU characteristics (e.g., existence of OS services, specific memory models) [6] that differ fundamentally on GPUs. For example, the standard error handler in Rust expects to print an error message to the console, an operation that is not possible on a GPU, which lacks standard I/O. The prototype, while successfully separating code paths, did not address any functionality related to these lang-items.

In this paper, we design and implement a context-dependent lang-item system within the existing hybrid compiler fork in order to tackle this incompatibility. This system enables the compiler to automatically select appropriate lang item implementations based on the target context (CPU host vs. GPU device). By providing specialized, GPU-aware implementations, we significantly expand the range of idiomatic Rust features usable within GPU kernels, making the integrated hybrid compilation approach substantially more practical.

Author's address: Cristian-Andrei Begu, c.begu@student.utwente.nl, University of Twente, Enschede, The Netherlands, 7500AE.

With the problem in mind, this paper aims to answer the following research question:

How can compiler support for context-dependent lang-items enable idiomatic Rust across diverse execution environments?

We break this down into the following sub-questions:

(1) What compiler architecture modifications should be made for target-aware integration of lang-item implementations?
(2) How should we design lang-item implementations for constrained execution environments (e.g., GPUs)?
(3) What range of core Rust language features, previously unusable in the target constrained environment (GPU), would be supported with the proposed architectural changes?

We provide background on Rust language items, the compiler architecture, and the existing hybrid compilation approach in Section 3. In Section 4, we describe the modifications made to the compiler and support libraries to enable context-dependent lang-items. Section 5 discusses the design and implementation of GPU-specific lang-item replacements. In Section 6, we present the new Rust features that become available in GPU kernels as a result of our approach. Section 8 offers a discussion of the results and current limitations. Finally, Section 9 outlines possible directions for future work.

2 RELATED WORK

The immediate foundation for our project is the hybrid Rust compiler prototype developed by Aukes [1]. This work introduced a novel approach by modifying the Rust compiler itself to handle both CPU host code and GPU device code. The prototype demonstrated separating the compilation paths at the mid-level representation stage and embedding the resulting GPU bytecode back into the host application as a static string.

Beyond the integrated prototype that we extend, other methods exist for leveraging GPUs in Rust. One common strategy involves using API wrappers like 'rust-cuda' [11] or 'ocl' [17], which provide Rust bindings to native CUDA and OpenCL APIs. These allow us to load and launch kernels but typically require writing the kernel code itself in C++/CUDA C or OpenCL C, or managing pre-compiled PTX/SPIR-V modules. This approach requires handling separate kernel codebases and interacting with complex, often unsafe, APIs. Another significant effort is the 'rust-gpu' project [5], which focuses on compiling Rust code, primarily targeting graphics shaders, into the SPIR-V intermediate representation. This operates as a distinct compiler backend, generating SPIR-V modules for later use. While advancing Rust for GPU programming, its focus on shaders and separate compilation model differs from the goal of enabling general-purpose kernels compiled via an integrated Rust compiler.

Moreover, the concept of providing alternative implementations for lang-items is well-established within the Rust ecosystem, particularly for environments such as microcontrollers or embedded systems [13]. When targeting platforms lacking an operating system or standard library, developers must often supply their own implementations for critical lang items to bridge the gap. For instance, embedded applications must define a custom panic handler (`panic_impl`), perhaps by looping indefinitely or triggering a hardware reset, as standard unwinding mechanisms are unavailable. Similarly, if heap allocation is required in such a context, the developer must provide a suitable static allocator implementation. This existing practice demonstrates Rust's flexibility in accommodating target-specific overrides for core primitives.

3 BACKGROUND

To understand the proposed system, it is essential to understand the role of lang-items in Rust, the constraints of GPU execution environments, and the architecture of both the standard Rust compiler and the extensions provided by Aukes in [1].

3.1 The GPU Execution Model

Graphics Processing Units (GPUs) are specialized hardware designed for highly parallel computation. Unlike CPUs, which are optimized for sequential processing and complex control flow, GPUs excel at running thousands of lightweight threads in parallel, each performing similar operations on different data. This parallelism comes with important constraints, such as:

- **No Operating System Services:** GPU code (called *kernels*) cannot perform system calls, file I/O, or interact with the operating system in the way CPU code can.
- **Distinct Memory Model:** GPUs have their own memory hierarchy, including global, shared, and local memory. Standard heap allocation is often unavailable or must be managed differently.
- **Simplified Error Handling:** Mechanisms like stack unwinding (cleaning up unused memory objects) or process termination are not generally available. In a GPU context, if a kernel encounters an error, it may simply halt or signal an error flag.

These differences mean that code written for CPUs cannot always be reused directly on GPUs, especially if it relies on features like dynamic memory allocation or standard error handling.

3.2 Rust Language Items (Lang-Items)

One of Rust's unique features is its use of *language items* (commonly called *lang-items*), which are special functions or types that the compiler relies on to implement core language features. These lang-items are not part of the language syntax itself, but are instead marked in library code with the `#[lang = "..."]` attribute.

Lang-items serve as the bridge between the language and its standard library. For example, when a program encounters a panic (an unrecoverable error), the compiler expects to find a function marked as the panic handler lang-item, which it then calls. Other lang-items define how dynamic memory allocation works, how destructors are called, and how certain traits (such as copying or iteration) are implemented. In typical Rust programs, these lang-items are provided by the standard library, which assumes a conventional operating system and CPU environment.

However, when targeting environments that lack standard operating system services—such as embedded systems or GPU—developers must provide alternative implementations for these lang-items. This flexibility allows Rust to be used in a wide range of contexts.

3.3 The Rust Compiler Architecture

The Rust compiler, `rustc`, works in several steps to turn source code into an executable program. This section describes the main stages of its pipeline relevant to this paper, drawing from the official Rust Compiler Development Guide [16]. Understanding these components helps explain how Rust features work and how the compiler can be adapted for different targets.

3.3.1 Intermediate Representations

After parsing the source code, the compiler processes it through several intermediate representations:

- **High-level Intermediate Representation (HIR):** This is a structured, desugared version of the source code, closely reflecting the original syntax but normalized for further analysis. The HIR is used for tasks like type checking.
- **Mid-level Intermediate Representation (MIR):** The MIR is a simplified, control-flow-oriented representation of the program. It abstracts away many syntactic details and is designed to make optimizations and analyses easier. The MIR is the main input for later stages such as monomorphization, borrow checking, and code generation.

3.3.2 The Query System

In a traditional compiler, the compilation process is organized as a fixed sequence of stages or "passes." For example, the compiler might first parse the source code, then perform type checking, then optimize the code, and finally generate machine code. Each stage processes the entire program before passing its results to the next stage. This approach is straightforward, but it can be inefficient: even a small change in the source code may require re-running many stages for the whole program, and adding new analyses or transformations often means modifying the pipeline itself.

The Rust compiler takes a different approach with its *query system.* Instead of fixed passes, the compiler organizes its work as a collection of small, focused queries. Each query is responsible for computing a specific piece of information, such as the type of an expression, the borrow-checked version of a function, or the MIR for a particular item.

The query system makes the compiler more modular and extensible. New analyses or transformations can be added as new queries, and the system automatically manages their dependencies and caching. In the context of this work, the query system is leveraged to distinguish between host and kernel code and to apply target-specific transformations, such as swapping lang-items, only where necessary in the compilation pipeline, without affecting unrelated parts of the program.

3.3.3 Monomorphization and Code Generation

After the compiler has generated and optimized the MIR, it performs *monomorphization*. In Rust, generics allow functions and types to be written abstractly, so they can operate on different data types without code duplication. During monomorphization, the compiler creates concrete versions of each generic function or type for every set of type parameters actually used in the program.

Once monomorphization is complete, the compiler lowers the MIR to target-specific machine code, such as x86, ARM, or GPU bytecode (e.g., PTX for NVIDIA GPUs). This is typically done using a backend like LLVM, which handles the final optimization and code generation steps, allowing Rust to target a wide range of hardware platforms.

3.4 Hybrid Compiler Modifications

The hybrid Rust compiler developed by Aukes [2] extends the standard Rust compiler to support integrated compilation of both CPU and GPU code from a single source file. The key modifications relevant to this work are:

- **Kernel Annotation and Separation:** The compiler recognizes special annotations (`#[kernel]`) to identify functions intended for GPU execution. During compilation, it separates these kernel functions from the host code by issuing different queries for the relevant context.
- **Dual Compilation Paths:** At the MIR level, the compiler processes host and kernel code along separate paths. Host code is compiled as usual for the CPU, while kernel code is compiled to GPU-compatible bytecode (in the current iteration, PTX for NVIDIA GPUs).
- **Embedding GPU Code:** The resulting GPU bytecode is embedded into the host binary as a static string, allowing the host code to launch GPU kernels at runtime.
- **Shared Intermediate Representations:** Both host and kernel code share the same initial parsing and HIR stages, but diverge at the MIR stage, where target-specific transformations and code generation occur.

In addition to modifying the compiler itself, Aukes also developed a CUDA support library, which provides safe Rust bindings to the CUDA API. This support library will also be extended to interact with the new lang-item system, supplying CUDA-specific implementations for language features as needed.

4 COMPILER MODIFICATIONS

The goal is to allow the same Rust source code to use familiar language features in both host and kernel code, while ensuring that the underlying implementations are correct for each target. This requires the compiler to:

- Distinguish between host and kernel code during compilation.
- Substitute standard lang-items with target-specific versions only in kernel code.
- Allow external libraries to provide their own target-specific lang-item implementations.

4.1 Kernel-Specific Lang-Items in the HIR

To enable this, we introduce new lang-items at the High-level Intermediate Representation (HIR) level, specifically for kernel functions. These kernel lang-items can be implemented in the same way as standard lang-items, which means they can be provided by any library. For example, the CUDA support library written by Aukes [3] is extended to provide these lang-items, mapping them to CUDA specific-operations (such as `cudaTrap` and `cudaMalloc`).

4.2 MIR Query and Lang-Item Swap Pass

As illustrated in Figure 1, the compilation path diverges at the MIR level to handle host and kernel code separately. This separation is managed by an existing *kernel MIR query* that is responsible for producing the MIR for functions marked as GPU kernels. The core modification of our work is the introduction of a dedicated *kernel lang-item swap pass* within this query. This pass maintains a mapping from standard lang items (such as `panic_impl`, `exchange_malloc`) to their kernel-specific counterparts. As the pass traverses the MIR of a kernel function, it identifies calls to standard lang-items and rewrites the MIR to call the kernel version instead.

The choice to perform this transformation on the MIR, rather than the earlier HIR, is deliberate. First, the hybrid compiler's architecture only identifies and separates kernel functions from host code after the initial HIR analysis is complete. More fundamentally, the HIR is designed to be a high-level representation that faithfully reflects the source code's structure and semantics, making it suitable for type-checking and trait resolution [15]. The MIR, by contrast, is the appropriate stage for such changes as it is a lower-level representation intended for target-specific transformations.

Further, the transformation is performed *before* monomorphization. By applying the swap at this stage, we ensure that all generic functions and types are instantiated with the correct, kernel-specific lang-items. This is crucial, as it prevents the original, CPU-specific lang-items from ever reaching the code generation backend for the kernel. Consequently, all downstream code generation for the kernel is guaranteed to use implementations that are safe and appropriate for the target environment, avoiding compilation errors or undefined behavior.

Since this transformation is performed only for requests of the kernel MIR, any changes made to it will only be reflected in the generated kernel bytecode. This ensures that host code continues to use the standard lang-items and is compiled as usual.

5 DESIGN OF GPU-SPECIFIC LANG-ITEMS

With the compiler infrastructure in place to support target-aware lang-items, the next step is to design implementations of these lang-items that are appropriate for the GPU environment. With the replacement lang-items being implemented through the same mechanism as standard lang-items, they can be provided by any support library – in this case, the CUDA support library.

When implementing lang-items for GPU kernels, several key principles should guide the design:

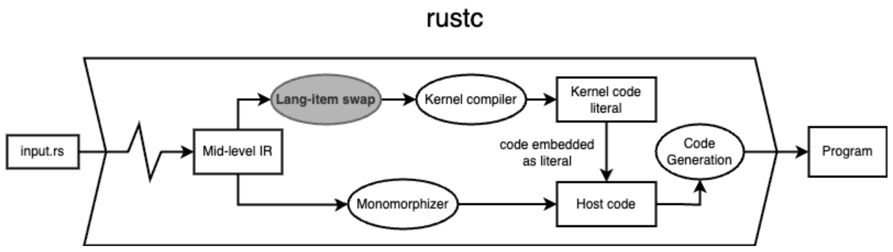

Fig. 1. Compiler pipeline with lang-item swap pass. The figure extends the architecture diagram from Aukes [1], highlighting the lang-item swap pass.

- **Signature Compatibility:** GPU-specific lang-items must have exactly the same function signatures as their CPU counterparts. This ensures seamless substitution during compilation and guarantees that all call sites, including those generated by the compiler or standard library, remain valid.
- **Semantic Alignment:** The GPU implementation should preserve the intent and semantics of the original lang-item as much as possible, so that code behaves consistently across host and device. In practice, this means that the GPU-specific version should match the observable behavior of the CPU version wherever the hardware and execution environment allow. This is typically ensured by designing the GPU implementation to follow the same function signatures, return values, and side effects as the standard version, and by validating behavior through tests. However, not all semantics can be fully preserved due to fundamental differences between CPU and GPU environments. In such cases, the implementation must provide the closest reasonable approximation and clearly document these differences for users.
- **GPU Suitability:** Implementations must avoid dependencies on unavailable features, such as operating system services or host-side I/O, and should be designed with the GPU's memory model, parallelism, and resource limitations in mind.

To demonstrate the approach, we discuss the lang-items currently supported in the CUDA support library: panic handling and dynamic memory allocation. For each, we compare the standard CPU implementation to the current GPU implementation, and outline possible alternative designs.

5.1 Panic Handling

CPU Implementation. On the CPU, the `panic` lang-item is typically implemented to print an error message, unwind the stack[1], and terminate the process or thread. This relies on operating system services and standard I/O, and may provide detailed diagnostics for debugging. Panics can either abort the process immediately or unwind the stack, allowing for resource cleanup and, in some cases, recovery if a suitable handler is present.

GPU Implementation. On the GPU, such facilities are unavailable. There is no standard output, no operating system to report errors to, and stack unwinding is not supported. In the CUDA support library, the `panic` lang-item is implemented using the CUDA *trap* instruction. When a panic occurs in a kernel, this instruction immediately halts the current thread and signals an error to the CUDA runtime. This approach is minimal and robust: it prevents further execution of faulty code and allows the host application to detect that a kernel has failed, typically by checking the CUDA error status after kernel launch. However, it does not provide detailed information about the cause or location of the panic, nor does it allow for resource cleanup or recovery within the kernel.

Other implementations are possible, for example, a panic handler could write an error flag, thread index, or diagnostic information to a pre-allocated region of global memory. After kernel execution, the host could inspect this memory to determine which threads panicked and possibly why. This would allow for more detailed error reporting or debugging, but introduces additional complexity: memory must be reserved and managed for error reporting, and care must be taken to avoid race conditions if multiple threads panic simultaneously.

5.2 Dynamic Memory Allocation

CPU Implementation. On the CPU, Rust's dynamic memory allocation is built on a multi-layered abstraction designed to decouple memory requests from the underlying memory provider. This system revolves around lang-items, compiler-generated code, and a library-provided allocator.

As seen in Figure 2, the process begins when application code uses a type like `Box<T>`[2], which triggers a call to the `exchange_malloc` lang-item. However, this lang-item is merely a placeholder for another function. The compiler translates this high-level request into a call to a 'magic'[3], low-level function symbol: `__rust_allocate`. Similarly, when a heap-allocated value is dropped, the `drop_in_place` lang-item is invoked. This triggers compiler-generated 'drop glue' code specific to the type being dropped, which for heap types like `Box<T>` includes a call to another standard symbol, `__rust_deallocate`.

For our purpose, the important part of this design is how these standard symbols are defined. By default, Rust's standard library provides the implementations, typically by linking to an allocator

[1]Stack unwinding is the process of walking back up the stack after an error, running destructors for all in-scope variables as each stack frame is removed.

[2]`Box<T>` is a standard Rust type that allows a value of type `T` to be stored on the heap rather than on the stack. This enables the creation of data structures whose size is not known at compile time, such as linked lists or trees.

[3]By 'magic', we informally refer to symbols or behaviors that are not explicitly written by the user but are part of a convention understood by the compiler. The compiler automatically generates and links these symbols.

like the operating system's native `malloc`. However, developers can override this default behavior using the `#[global_allocator]` attribute. This attribute is applied to a static item whose type implements the `GlobalAlloc` trait. The compiler then generates the `__rust_allocate` and `__rust_deallocate` functions as thin wrappers[4] that forward calls to the `alloc` and `dealloc` methods of the user-provided allocator. This mechanism allows any part of the program to allocate and free memory without needing to know which specific allocator is being used.

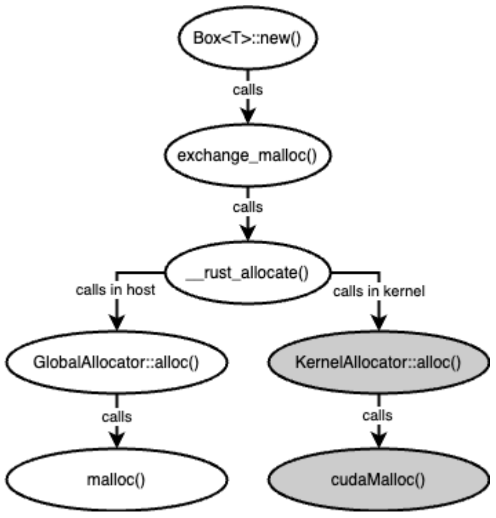

Fig. 2. Flow of dynamic memory allocation for both compilation paths.

Kernel Allocator Attribute. Memory management on the GPU presents additional challenges that a simple lang-item swap cannot solve. While swapping `exchange_malloc` would redirect explicit allocations, it fails to account for deallocations. As on the CPU, deallocation is handled by compiler-generated 'drop glue' that calls the low-level `__rust_deallocate` symbol directly. A swapped lang-item would be bypassed, causing kernel code to incorrectly call the host's deallocator.

To solve this, we introduce a new `#[kernel_allocator]` attribute, which provides a GPU-specific analog to the host's `#[global_allocator]`. This attribute is applied to a custom allocator type designed for the kernel environment. When compiling kernel code, the compiler recognizes this attribute and, for the GPU target only, redefines the `__rust_allocate` and `__rust_deallocate` symbols to call the methods of the designated kernel allocator. As shown in the right branch of Figure 2, this approach ensures that all calls to dynamic memory operations within a kernel, both allocation and deallocation within drop glue, are routed through the GPU-specific implementation.

GPU Implementation. The CUDA support library provides a concrete implementation for this system. It offers an allocator type that is marked with the `#[kernel_allocator]` attribute. This allocator's methods are simple wrappers around CUDA's device memory management functions: its `alloc` method calls `cudaMalloc`, and its `dealloc` method calls `cudaFree`. This setup allows GPU kernels written in traditional Rust to leverage the native memory management mechanisms of the CUDA runtime.

The flexibility of the `#[kernel_allocator]` mechanism, however, is not limited to this single implementation. It is designed, as the lang-item swaps, to be an extensible interface. Alternative allocator strategies could be implemented to suit specific application requirements or backends. For example, a developer could provide a bump allocator [5] in global memory for fast, lock-free allocation, or a memory pool to reduce fragmentation for workloads with frequent, uniformly sized allocations.

6 NEWLY ENABLED FEATURES

The introduction of context-dependent lang-items in the hybrid Rust compiler significantly expands the set of Rust features available within GPU kernels. In this section, we evaluate the impact of these changes by highlighting the newly enabled features and presenting concrete kernel examples that were previously unsupported.

With the new system, GPU kernels can now take advantage of several key Rust features that were previously unavailable:

- **Panic Handling:** Kernels can safely use `panic!` and other error-handling idioms. Instead of causing undefined behavior or compilation failures, panics are now handled in a way that is appropriate for the GPU environment.
- **Dynamic Memory Allocation:** Kernels can create and use heap-allocated data structures, such as `Box<T>` and `Vec<T>`, with memory allocated directly on the device.
- **Idiomatic Rust Patterns:** Many common Rust constructs, such as `unwrap()` and dynamic data structures, can now be used naturally in kernel code, making GPU programming in Rust more expressive and familiar.

6.1 Panic Handling

Previously, GPU kernels could not use the language's built-in panic mechanism, which meant that errors like out-of-bounds access could not be reported in a standard way. With the new system, panics are now supported in device code. This allows kernels to use `panic!` for error reporting, as shown in Listing 1, where a panic is triggered if a thread attempts to access data outside the valid range. Further, common Rust error handling patterns, such as using `unwrap()` on `Option` and `Result` types, are also now available in GPU kernels, as they also rely on the `panic!` macros.

6.2 Dynamic Memory Allocation

With the introduction of GPU-specific lang-items and the `#[kernel_allocator]` system, Rust kernels can now use dynamic memory allocation on the GPU. This enables the use of heap-allocated data structures, such as trees, lists, or buffers, directly within GPU code.

[4]A thin wrapper is a function that performs no computation itself, serving as a call to another function.

[5]A bump allocator is a simple memory allocator that allocates memory by incrementally moving a pointer forward in a pre-allocated region. While, very fast, it does not support the deallocation of individual objects.

```
#[kernel]
unsafe fn add_with_bounds_check(a: &[i32], b: &[i32],
    mut out: Buffer<i32>) {
    let i = gpu::global_tid_x();
    if i >= a.len() {
        panic!("Thread {} out of bounds", i);
    }
    let mut sum = 0;
    for j in 0..a.len() {
        sum += a[j] + b[j];
    }
    out.set(i, sum);
}
```

Listing 1. A kernel that performs bounds-checked parallel addition and panics on out-of-bounds access.

This capability is useful in many practical scenarios. For example, tree structures are commonly used to represent hierarchical or recursive data. In industry, such trees are fundamental for building acceleration structures in computer graphics, such as BVH or kd-trees for ray tracing and collision detection [10], or for spatial indexing in geospatial analysis, including quadtrees and octrees in mapping and GIS [12]. In machine learning, decision trees and random forests are widely used for classification and regression [4], and can benefit from parallel construction and evaluation on the GPU.

Listing 2 shows a kernel where each thread builds a small binary tree and computes the sum of its values. This demonstrates how dynamic memory enables more flexible and idiomatic Rust code on the GPU.

7 TOOLING USAGE

All development for this work was carried out using the same repositories as described in the original hybrid compiler paper [1]. As such, the installation and usage instructions remain unchanged.

To use the prototype, users must have a CUDA-enabled GPU, a supported operating system (Windows or Linux), and the CUDA Toolkit installed. The modified Rust compiler can be obtained by cloning the repository at https://github.com/NiekAukes/rust-gpu-hybrid-compiler [2] and following the installation steps in the provided `README.md`.

The CUDA support library and sample projects are available at https://github.com/NiekAukes/rust-kernels [3]. This repository contains the necessary libraries for kernel execution as well as example code that can serve as a template for new projects.

8 DISCUSSION

The introduction of context-dependent lang-items into the hybrid Rust compiler may be a significant step toward making idiomatic Rust programming practical and robust for GPU development. By allowing the compiler to substitute core language features with GPU-appropriate implementations, we have enabled Rust constructs, such as safe panic handling and dynamic memory allocation, that were previously unavailable or unsound in device code.

```
struct TreeNode {
    value: i32,
    left: Option<Box<TreeNode>>,
    right: Option<Box<TreeNode>>,
}

impl TreeNode {
    fn new(value: i32) -> Box<TreeNode> {
        Box::new(TreeNode {
            value,
            left: None,
            right: None,
        })
    }

    fn sum(&self) -> i32 {
        let mut total = self.value;
        if let Some(ref l) = self.left {
            total += l.sum();
        }
        if let Some(ref r) = self.right {
            total += r.sum();
        }
        total
    }
}

#[kernel]
fn tree_kernel(mut results: Buffer<i32>) {
    let i = gpu::global_tid_x();
    let mut root = TreeNode::new(i as i32);
    if i % 2 == 0 {
        root.left = Some(TreeNode::new(i as i32 * 2));
    }
    if i % 3 == 0 {
        root.right = Some(TreeNode::new(i as i32 * 3));
    }
    results.set(i, root.sum());
}
```

Listing 2. A kernel where each thread dynamically allocates and sums a binary tree.

This work shows that much of Rust's safety and expressiveness can be brought to the GPU domain without forcing developers to abandon familiar idioms. For example, the ability to use `panic!` and `unwrap()` in kernels allows for clear error handling, while support for heap-allocated data structures like trees enables more sophisticated algorithms, including those used in graphics, scientific computing, and data analytics.

A key strength of this approach is its extensibility. By defining kernel-specific lang-items at the HIR level and performing substitutions in a dedicated MIR pass, the system allows any library to provide its own implementations. This modularity makes it possible to support additional GPU backends, improve error reporting, and expand the set of language features available in the future.

However, there are trade-offs. The system introduces additional complexity into the compilation process, as maintaining parallel sets of lang-items for each target increases the maintenance burden for both compiler developers and library authors. As the number of supported lang-items grows, keeping these mappings consistent may become more challenging, especially as the Rust language evolves.

Some language features, such as panic handling, remain more limited on the GPU — for example, panics typically result in thread termination rather than stack unwinding. Dynamic memory allocation is now possible, but is constrained by the GPU's memory model and allocator design. Furthermore, while the current implementation targets CUDA, supporting other GPU backends will require further engineering effort.

Another important aspect is the interaction between Rust's safety guarantees and the introduced compiler extensions. The overall safety of kernel code ultimately relies on the correctness of the GPU-specific lang-item implementations. For instance, dynamic memory allocation on the GPU must be carefully managed to prevent leaks or misuse, and panic handling must avoid leaving the device in an inconsistent state, especially since stack unwinding and resource cleanup are typically unavailable. Notably, since these lang-items are frequently implemented using unsafe code and foreign function interfaces (FFI) to communicate with GPU runtimes, the responsibility for maintaining Rust's safety properties shifts from the language and compiler to the implementer of these lang-items.

Overall, context-dependent lang-items offer a practical compromise: they bring much of Rust's safety and expressiveness to GPU programming, while introducing some new complexity that must be managed as the system matures.

9 FUTURE WORK

The current system establishes a viable framework for context-dependent lang-items, opening several paths for future development. Future work should focus on broadening language feature support, enhancing error reporting mechanisms, and adapting the system for different GPU backends.

9.1 Expanding Lang-item Coverage

While panic handling and dynamic allocation are critical, several other lang-items could be investigated to enable more advanced, idiomatic Rust in GPU kernels. An area of exploration is that of the `Iterator` trait family, specifically the `IteratorNext` and `IntoIterIntoIter` lang-items. These are the foundation of Rust's `for item in collection` loops. Enabling them for GPU kernels would allow developers to write safer and more expressive iteration code.

9.2 Advanced Panic Handling

The current `panic_impl` for CUDA relies on the `trap` instruction, which halts a thread but provides no diagnostic information to the host. A major area for future work is to design a communicative panic handler.

This could be achieved by creating a standardized error-reporting protocol between the host and device. Before launching a kernel, the host application would allocate a small buffer in global GPU memory and pass a pointer to it. The GPU-side `panic_impl` would be modified to, instead of immediately trapping, attempt to write details from the `PanicInfo` struct—such as the error message, file, and line number—into this buffer. After the kernel completes (or fails), the host could check this buffer to retrieve and display a detailed error message, transforming panics from silent failures into actionable debug information.

9.3 Alternative Backends

The core architecture for swapping lang-items is backend-agnostic, but the implementations themselves are highly specific to the target hardware and API. Extending this work to support other backends, such as SPIR-V for Vulkan and OpenCL, presents another research direction.

On one hand, the implementation of items such as `panic_impl` would not change significantly. While the CUDA version uses a specific intrinsic (`cudaTrap`), a SPIR-V implementation would likely use the `OpKill` instruction to terminate the invocation.

On the other hand, dynamic memory allocation (`exchange_malloc`) would require a completely different approach. The CUDA implementation can directly call `cudaMalloc`. In a Vulkan environment, there is no direct equivalent. A SPIR-V kernel allocator would need to interface with a memory management system set up by the host using Vulkan APIs, potentially leveraging extensions like buffer device addresses. This highlights that each new backend requires not just a new code generator but a carefully designed library of lang-item implementations that respect the execution model, memory hierarchy, and available intrinsics of that specific environment.

9.4 Approach Streamlining

As the number of supported lang-items increases, ensuring correctness and reducing manual effort will become increasingly important. In that sense, another direction for future work is to improve the reliability of GPU-specific lang-item implementations through systematic testing. This could include developing targeted tests that compare the behavior of host and kernel lang-items in controlled scenarios, or even exploring formal verification techniques to catch mismatches in semantics. Another direction is to automate the process of adding new lang-items for kernels. For example, Rust's existing macro system for registering lang-items could be supplemented with functionality that automatically generates kernel-specific definitions when a new lang-item is introduced, reducing boilerplate and the risk of human error.

10 CONCLUSION

We have presented a context-dependent lang-item system for Rust, enabling the hybrid compiler to select specialized, GPU-safe implementations for core language features when compiling kernel code. This allows developers to write GPU kernels in idiomatic Rust, using constructs such as `panic!`, `unwrap()`, and dynamic memory allocation, features that were previously unavailable or unsafe in device code.

The system's extensibility and modularity provide a strong foundation for future work. As more lang-items are supported and tooling improves, the gap between CPU and GPU Rust programming will continue to narrow. In summary, context-dependent lang-items make it possible to write safer, more maintainable, and more idiomatic Rust code for heterogeneous computing, opening the door to new applications and more robust GPU software.

REFERENCES

[1] Niek Aukes. 2024. Hybrid compilation between GPGPU and CPU targets for Rust. In *Proceedings of the 41st Twente Student Conference on IT (TScIT 41)*. Enschede, The Netherlands.

[2] Niek Aukes. 2024. rust-gpu-hybrid-compiler. https://github.com/NiekAukes/rust-gpu-hybrid-compiler. Accessed: 2025-06-20.

[3] Niek Aukes. 2024. rust-kernels. https://github.com/NiekAukes/rust-kernels. Accessed: 2025-06-20.

[4] Leo Breiman. 2001. Random Forests. *Machine Learning* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324

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

[6] David B. Kirk and Wen-mei W. Hwu. 2010. *Programming Massively Parallel Processors: A Hands-on Approach*.

[7] Steve Klabnik and Carol Nichols. 2023. *The Rust Programming Language* (2nd ed.). No Starch Press, San Francisco, CA.

[8] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (LLVM '15)*. ACM, New York, NY, USA, Article 7. https://doi.org/10.1145/2833157.2833162

[9] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. 2007. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum* 26, 1 (2007), 80–113. https://doi.org/10.1111/j.1467-8659.2007.01100.x

[10] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically Based Rendering: From Theory to Implementation* (3rd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 255–302 pages.

[11] Rust-GPU Project Contributors. 2021-2024. Rust-CUDA: Ecosystem of libraries and tools for writing and executing fast GPU code fully in Rust. https://github.com/Rust-GPU/Rust-CUDA. Accessed: April 2025.

[12] H. Samet. 2006. *Foundations of Multidimensional and Metric Data Structures*. Elsevier Science. 28–90 pages. https://books.google.nl/books?id=vO-NRRKHG84C

[13] The Rust Embedded Working Group. 2018-2025. The Embedded Rust Book.

[14] The Rust Project Developers. 2025. Lang items - rustc-dev-guide. Accessed: June 3 2025.

[15] The Rust Project Developers. 2025. The HIR (High-level IR) - rustc-dev-guide. https://rustc-dev-guide.rust-lang.org/hir.html. Accessed: June 13, 2025.

[16] The Rust Project Developers. 2025. The rustc Development Guide. https://rustc-dev-guide.rust-lang.org/. Accessed: June 10, 2025.

[17] Brendan Zabarauskas, Vincent Pasquier, and cogciprocate contributors. 2016-2025. ocl: OpenCL Bindings and Wrappers for Rust. https://github.com/cogciprocate/ocl. Rust library providing OpenCL bindings. Accessed: April 2025.

A AI STATEMENT

During the writing of this paper, the author used OpenAI's ChatGPT 4.1 for paraphrasing, spell and grammar checking. All content was reviewed and edited after using AI tools, and the author takes full responsibility for the content of the paper.