VECTORWARE

DISPATCHES

Rust's standard library on the GPU

January 20, 2026 · 11 min read

Pedantic mode: Off

GPU code can now use Rust's standard library. We share the implementation approach and what this unlocks for GPU programming.

At VectorWare, we are building the first GPU-native software company. Today, we are excited to announce that we can successfully use Rust's standard library from GPUs. This milestone marks a significant step towards our vision of enabling developers to write complex, high-performance applications that leverage the full power of GPU hardware using familiar Rust abstractions.

This post is a preview of what we've built. We're preparing our work for potential upstreaming and will share deeper technical details in future posts.

Rust's `std` library

Rust's standard library is organized as a set of layered abstractions:

1. `core` defines the language's foundation and assumes neither a heap nor an operating system.
2. `alloc` builds on `core` by adding heap allocation.
3. `std` sits at the top of `core` and `alloc`. It adds APIs for operating system concerns such as files, networking, threads, and processes.

A defining feature of Rust is that layers 2 and 3 are *optional*. Code can opt out of `std` via the `#![no_std]` annotation, relying only on `core` and, when needed, `alloc`. This

makes Rust usable in domains such as embedded, firmware, and drivers, which lack a traditional operating system.

Why `std` isn't supported on GPUs today

We are the maintainers of `rust-cuda` and `rust-gpu`, open source projects that enable Rust code to run on the GPU. When targeting the GPU with these projects, Rust code is compiled with `#![no_std]` as GPUs do not have operating systems and therefore cannot support `std`.

Because `#![no_std]` is part of the Rust language, `#![no_std]` libraries on crates.io written for other purposes can generally run on the GPU without modification. This ability to reuse existing open source libraries is much better than what exists in other (non-Rust) GPU ecosystems.

Still, there is a cost to opting out of `std`. Many of Rust's most useful and ergonomic abstractions live in the standard library and the majority of open source libraries assume `std`. Enabling meaningful `std` support on GPUs unlocks a much larger class of applications and enables even more code reuse.

Why `std` might be feasible on GPUs soon

Modern GPU workloads like machine learning and AI require fast access to storage and networking from the GPU. Technologies such as NVIDIA's GPUDirect Storage, GPUDirect RDMA, and ConnectX make it possible for GPUs to interact with disks and networks more directly in the datacenter. With systems like NVIDIA's DGX Spark and Apple's M-series devices, similar capabilities are starting to appear in consumer hardware. APIs and features traditionally provided by an operating system are becoming available to GPU code and we only see that trend continuing.

We believe CPUs and GPUs are converging architecturally, as evidenced by designs such as AMD's APUs as well as NPUs and TPUs being integrated into various CPUs. This convergence makes it both more practical and more compelling to

share abstractions across devices rather than treating GPUs and other cores as isolated accelerators.

`std` as a GPU abstraction layer

The GPU ecosystem is younger than the CPU ecosystem. GPU hardware capabilities are changing rapidly and software standards are still emerging. Rust's `std` can provide a stable surface while allowing the underlying implementation to change until the ecosystem matures. For example, `std` may use GPUDirect on one system, fall back to host mediation on another, or adapt between unified memory and explicit transfers depending on hardware support. The key is that user code does not need to change due to the underlying feature support.

At VectorWare, we are building extremely complex GPU-native applications where higher-level abstractions are critical. We are excited to pioneer this space and look forward to sharing more of what we have done in the future.

A world first: Rust's `std` on the GPU

The following is a simple GPU kernel that uses Rust's `std` library to perform various tasks such as printing to stdout, reading user input, getting the current time, and writing to a file. **This code runs on the GPU**:

```rust
use std::io::Write;
use std::time::{SystemTime, UNIX_EPOCH};

#[unsafe(no_mangle)]
pub extern "gpu-kernel" fn kernel_main() {
    println!("Hello from VectorWare and the GPU!");

    print!("Enter your name: ");
    let _ = std::io::stdout().flush();
    let mut name = String::new();
    std::io::stdin().read_line(&mut name).unwrap();

    let name = name.trim_end();
    println!("Hello, {}! Nice to meet you from the GPU!", name);
```

```
    let now = SystemTime::now();
    let duration_since_epoch = now.duration_since(UNIX_EPOCH).unwrap();
    println!(
        "Current time (seconds since epoch): {}",
        duration_since_epoch.as_secs()
    );

    let msg = format!(
        "This file was created *from a GPU*, using the Rust standard library 🦀\n\
         We are {:?}\n\
         Current time (seconds since epoch): {}\n\
         User name: {}",
        "VectorWare".to_owned(),
        duration_since_epoch.as_secs(),
        name
    );
    std::fs::File::create("rust_from_gpu.txt")
        .unwrap()
        .write_all(msg.as_bytes())
        .unwrap();

    println!("File written successfully!");
}
```

The only difference from regular Rust code is the `#[unsafe(no_mangle)]` and `extern "gpu-kernel"` annotations, which indicate that this function is a GPU kernel entry point. We will get rid of those with some compiler transformations in the future.

Hostcalls

This works because of our custom `hostcall` framework. A hostcall is analogous to a syscall. A hostcall is a structured request from GPU code to the host CPU to perform something it cannot execute itself. You can think of it like a remote procedure call from the GPU to the host to achieve syscall-like functionality.

To end users, Rust's `std` APIs appear unchanged and act as one would expect. Under the hood, however, certain `std` calls are reimplemented via hostcalls. While we can implement any API with hostcalls, in this instance we implement a `libc` facade to minimize changes to Rust's `std` (which uses `libc` to communicate with Unix-like operating systems). For example, `std::fs::File::open` is reimplemented to issue an

`open` hostcall to the host, which performs the actual file open operation using the host's filesystem APIs.

With `std` available from the GPU, developer ergonomics improve significantly and a much larger portion of the Rust ecosystem can now be used on the GPU.

Device/host split

"Hostcall" is something of a misnomer. In reality, hostcall requests can be fulfilled on either the host or the GPU. This enables progressive enhancement as GPU hardware and software matures. For example, `std::time::Instant` is implemented on the GPU using a device timer on platforms with APIs such as CUDA's `%globaltimer`, while `std::time::SystemTime` is implemented on the host due to missing wall-clock support on the GPU.

Being able to choose where a hostcall is fulfilled enables advanced features such as device-side caches (where the GPU can cache hostcall results locally to avoid repeated round-trips) and filesystem virtualization (where the GPU can maintain its own view of the filesystem and sync with the host as needed).

We can even innovate and add new semantics without breaking compatibility. For example, one could write a file to `/gpu/tmp` to indicate it should stay on the GPU, or one could make a network call to `localdevice:42` to communicate with warp/workgroup 42. We will share some of our work in this space in the future.

Implementation details

While the idea is fairly simple, the implementation has many details to ensure correctness and performance. We'll deep-dive into the implementation in a later blog post. For now, we will cover some high-level details.

We use standard GPU programming techniques such as double-buffering and atomic operations and take care to avoid data tearing and ensure memory consistency. The protocol is deliberately minimal to keep GPU-side logic simple and we support

features like packing results to avoid GPU heap allocations where appropriate. We leverage APIs like CUDA streams to avoid blocking the GPU while the host processes requests. The host-side hostcall handlers are implemented using `std`, not as a 1:1 `libc` trampoline, so that they are portable and testable.

As Rust programmers, we very much care about safety and correctness. We've run a modified version of the hostcall runtime / kernel under miri, with CPU threads emulating the GPU, to check if our code is minimally sound. We'll share more about our testing and verification efforts in a future post.

Our current implementation targets Linux hosts with NVIDIA GPUs via CUDA. There is nothing fundamental preventing support for other host operating systems or GPU vendors—the hostcall protocol is vendor-agnostic, and we could target AMD GPUs via HIP or use Vulkan with `rust-gpu`.

Prior work

There is substantial prior art in exposing system-like APIs to GPU code, primarily from the CUDA and C++ ecosystems.

NVIDIA ships a GPU-side C and C++ runtime as part of CUDA, including `libcu++`, `libdevice`, and the CUDA device runtime (see the CUDA Programming Guide). These components provide implementations of large portions of the C and C++ standard libraries for device code. However, they are tightly coupled to CUDA, are C/C++-specific, and are not designed to present a general-purpose operating system abstraction. Facilities such as filesystems, networking, and wall-clock time remain host-only concerns, exposed through CUDA-specific APIs rather than a unified standard library interface.

There have also been experiments such as libc for GPUs and related projects that attempt to provide a C or POSIX-like layer for GPUs by forwarding calls to the host. While we only became aware of these efforts after we started our own work, we are excited to see we arrived at similar ideas and implementations independently. We are

also unaware of any prior work that integrates such a layer with a higher-level language standard library like Rust's `std`.

Other GPU programming models, including OpenCL, HIP, and SYCL, provide their own standard libraries and runtime abstractions. These libraries are intentionally minimal and domain-specific, focusing on portability and performance rather than source compatibility with existing CPU codebases or reuse of a general-purpose standard library ecosystem.

Our approach differs in two key ways. First, we target Rust's `std` directly rather than introducing a new GPU-specific API surface. This preserves source compatibility with existing Rust code and libraries. Second, we treat host mediation as an implementation detail behind `std`, not as a visible programming model.

In that sense, this work is less about inventing a new GPU runtime and more about extending Rust's existing abstraction boundary to span heterogeneous systems. At VectorWare, we are bringing the GPU to Rust instead of just bringing Rust to the GPU.

Open sourcing and upstreaming

We are cleaning up our changes and preparing to open source them. As the VectorWare team includes multiple members of the Rust compiler team, we are keen to work upstream as much as possible. That said, changes to Rust's standard library require careful review and take time to upstream.

One open question is where the correct abstraction boundary should live. While this implementation uses a `libc`-style facade to minimize changes to Rust's existing `std` library, it is not yet clear that `libc` is the right long-term layer. The hostcall mechanism is quite versatile and we could alternatively make `std` GPU-aware via Rust-native APIs. This requires more work on the `std` side but could be more safe and efficient than mimicking `libc`.

Is VectorWare only focused on Rust?

As a company, we understand that not everyone uses Rust. Our future products will support multiple programming languages and runtimes. However, we believe Rust is uniquely well suited to building high-performance, reliable GPU-native applications and that is what we are most excited about.

Follow along

Follow us on X, Bluesky, LinkedIn, or subscribe to our blog to stay updated on our progress. We will be sharing more about our work in the coming months, including deeper technical dives into our hostcall implementation and other compiler work, testing and verification efforts, and our thoughts on the future of GPU programming with Rust. You can also reach us at hello@vectorware.com.