Introduction

Welcome to the Rust-GPU dev guide! This documentation is meant for documenting how to use and develop on Rust-GPU.

If you're looking to get started with writing your own shaders in Rust, check out the *"Writing Shader Crates"* section for more information on how to get started.

Alternatively if you're looking to contribute to the `rust-gpu` project, have a look at *"Building Rust-GPU"* section.

Building Rust-GPU

Getting started

1. Clone the repository.

```
git clone --recurse-submodules https://github.com/rust-gpu/rust-gpu
```

2. **optional** Install SPIRV-Tools and add it to your `PATH`. You can skip this step if you just want to run examples with the defaults. See Using installed SPIRV-Tools if you decide to go with this option.

3. Next, look at the examples folder. There are two kinds of targets here: [runners] and [shaders]. The projects inside `shaders` are "GPU crates", i.e. ones that will be compiled to one or more SPIR-V modules. The `runner` projects are normal, CPU crates that use some graphics backend (currently, we have a `wgpu` runner, a Vulkan runner through `ash`, and a barebones pure software CPU runner) to actually run one of the "GPU crate" shaders.

   Run the example:

   ```
   cargo run --bin example-runner-wgpu
   ```

   This will build `rustc_codegen_spirv`, the compiler, then use that compiler to build `sky-shader` into a SPIR-V module, then finally, build a `wgpu` sample app (modified from `wgpu`'s examples) using the built SPIR-V module to display the shader in a window.

Prerequisite linux packages recommended to install before building Rust-GPU

You may need the development version (i.e. headers/etc. included) of some packages in some distributions to be able to build the examples - specifically, x11 and libxkbcommon, as well as gcc/clang with c++ support. These packages may be called (fedora) `libX11-devel`, `libxkbcommon-x11-devel`, and `gcc-c++`, or (ubuntu) `libxkbcommon-x11-dev`, `libx11-dev`, and `gcc`.

Using installed SPIRV-Tools

By default, all of the crates and examples in this repo will compile the `spirv-tools-sys` crate, including a lot of C++ code from SPIRV-Tools. If you don't want to build the C++ code because you already have SPIRV-Tools installed, or just don't want to spend more time compiling, you can build/run the crate with the `use-installed-tools` feature.

```
cargo run \
    --manifest-path examples/example-runner/Cargo.toml \
    --features use-installed-tools \
    --no-default-features
```

You should see `warning: use-installed-tools feature on, skipping compilation of C++ code` during the compilation, but otherwise the build will function just the same as if you compiled the C++ code, with the exception that it will fail if you don't have SPIRV-Tools installed correctly.

Testing Rust-GPU

Rust-GPU has a couple of different kinds of tests, most can be ran through `cargo test`, however Rust-GPU also has end-to-end tests for compiling Rust and validating its SPIR-V output, which can ran by running `cargo compiletest`.

```
cargo test && cargo compiletest
```

Adding Tests

Rust-GPU's end-to-end test's use an external version of the `compiletest` tool as a testing framework. Be sure to check out the repository and the rustc Dev-Guide for more information about how it works, how to configure it, and add new tests.

Blessing Tests

You will occasionally need to "bless" the output from UI tests to update the normalised output, you can do this by passing a `--bless` flag to `cargo compiletest`.

```
cargo compiletest --bless
```

Filtering Tests

When working on tests, you may need to run `cargo compiletest` a few times, while changing only a small number of tests. You can avoid having to run all the other (unrelated) tests, by passing substrings of their paths, to `cargo compiletest`, for example:

```
cargo compiletest arch/u image
```

The above command will only test `ui/arch/u_*.rs` and `ui/image/*.rs`, and skip everything else. You can also add `--bless` to update expected outputs, as well.

Testing Different Environments

You can test against multiple different SPIR-V environments with the `--target-env` flag. By default it is set to `unknown`.

```
cargo compiletest --target-env=vulkan1.1
# You can also provide multiple values to test multiple environments
cargo compiletest --target-env=vulkan1.1,spv.1.3
```

Debugging

Tracing

`rust-gpu` has a lot of debug! (or trace!) calls, which print out logging information at many points. These are very useful to at least narrow down the location of a bug if not to find it entirely, or just to orient yourself as to why the compiler backend is doing a particular thing.

To see the logs, you need to set the `RUSTGPU_LOG` environment variable to your log filter (note the "GPU" in the name). The full syntax of the log filters can be found in the rustdoc of tracing-subscriber.

Use `RUSTGPU_LOG_FORMAT` to control log output format (`"tree"` , `"flat"` , or `"json"`) and `RUSTGPU_LOG_COLOR` to manage color output (`"always"` , `"never"` , or `"auto"`).

To trace non- `rust-gpu` parts of the compiler, set the standard `RUSTC_LOG` environment variable.

Replacements for old codegen arguments

Before `rust-gpu` supported tracing, there were special codegen arguments to aid observability. As of PR#196 the they have been removed and replaced with the following:

- `--specializer-debug` → `RUSTGPU_LOG=rustc_codegen_spirv::specializer=debug`
- `--print-zombie` → `RUSTGPU_LOG=print_zombie=debug`
- `--print-all-zombie` → `RUSTGPU_LOG=print_all_zombie=debug`

Minimizing bugs in SPIR-V

When debugging problems with SPIR-V generated by rust-gpu, you occasionally need to reduce the SPIR-V in order to make it easily shareable with others. We've created a short guide on how to do that.

Prerequisites

In order to build and validate SPIR-V you're going to install SPIR-V tools.

SPIR-V Template

SPIR-V has some amount of required boilerplate in order to be considered valid, we've created a small template to help to get started. This file creates a single empty vertex entry-point with a single floating-point constant.

```
; bug.spvasm
OpCapability Shader
OpCapability VulkanMemoryModel
OpMemoryModel Logical Vulkan

; == Entry-points ==
OpEntryPoint Vertex %vert_fn "vert"

; == Types ==
%void = OpTypeVoid
%f32 = OpTypeFloat 32

; Function Types
%void_fn = OpTypeFunction %void

; == Constants ==
%f32_1 = OpConstant %f32 1

; == Functions ==
%vert_fn = OpFunction %void None %void_fn
  %block = OpLabel
    OpReturn
OpFunctionEnd
```

Steps

1. Assemble your spirv with `spirv-as ./bug.spvasm`, this will produce a `out.spv` file containing the assembled code.

2. The assembled code also needs to be validated with `spirv-val out.spv`

3. Once the code has been validated as having no issues, you can use `spirv-cross` to compile the code to various outputs.

    - **GLSL** `spirv-cross out.spv`
    - **HLSL** `spirv-cross --hlsl out.spv`
    - **MSL** `spirv-cross --msl out.spv`
    - **Vulkan GLSL** `spirv-cross -V out.spv`

"Codegen args" (flags/options) supported by the Rust-GPU codegen backend

Please keep in mind that many of these flags/options are for internal development, and may break output unexpectedly and generally muck things up. Please only use these if you know what you're doing.

Help is also appreciated keeping this document up to date, "codegen args" flags/options may be added/removed on an ad-hoc basis without much thought, as they're internal development tools, not a public API - this documentation is only here because these flags/options may be helpful diagnosing problems for others.

It's recommended that "codegen args" options that take paths to files or directories are set to full paths, as the working directory of the compiler might be something wonky and unexpected, and it's easier to set the full path.

How to set "codegen args" flags/options

The most convenient method is relying on `spirv-builder` reading the `RUSTGPU_CODEGEN_ARGS` environment variable, e.g.:

```
$ RUSTGPU_CODEGEN_ARGS="--no-spirv-val --dump-post-link=$PWD/postlink" cargo
run -p example-runner-wgpu
...
    Finished release [optimized] target(s) in 15.15s

$ file postlink/*
postlink/module: Khronos SPIR-V binary, little-endian, version 0x010300,
generator 0x1b0000
```

Notably, `RUSTGPU_CODEGEN_ARGS="--help"` can be used to see a "usage" message (which lists all the flags/options, including ones not listed in this document), via e.g. running a Cargo build that relies on `spirv-builder`.

However, it's only a convenient alias for for `RUSTGPU_RUSTFLAGS=-Cllvm-args="..."` (without having to expose the fact that LLVM's name is still attached to `rustc`'s interface for this functionality), and if in direct control of `rustc`, you can still pass such "codegen args" flags/options wrapped in `-C llvm-args="..."`.

Historical note about past "environment variables"

Many of these flags/options were at one point, individual environment variable (e.g. the `--dump-pre-link` option used to be the environment variable `DUMP_PRE_LINK`).

However, that approach is prone to various failure modes, because the environment variables would not get registered as a "dependency" (without extra work that never happened), and the idea of "codegen args" fits better with existing practices (e.g. `rustc -C llvm-args="..."` for the LLVM codegen backend of `rustc`).

For more context see also PR #959, which made the transition to this system.

Where are all the rest of the flags/options documented?

If you do run a build with `RUSTGPU_CODEGEN_ARGS="--help"` (or `-C llvm-args="--help"`), you will notice more flags/options than are listed in this documented.

This is a historical artifact: as mentioned above, these used to be environment variables, and this document only described those, without talking about the older "codegen args" at all.

While most of those flags are usually only exposed through higher-level `spirv-builder` APIs, it would be nice to have all of them documented in one place (eventually?).

Debugging "codegen args" flags/options

As mentioned above, these form the bulk of "codegen args", but keep in mind the list is not exhaustive and you will want to check the full list with e.g. `RUSTGPU_CODEGEN_ARGS="--help"`.

`--dump-mir DIR`

Dumps the MIR of every function rust-gpu encounters, to files in `DIR`. Yes, rustc does have options to do this by default, but I always forget the syntax, and plumbing through the option to `spirv-builder` is annoying, so this is handy to just hack an output.

FIXME(@eddyb) *this may be irrelevant now given* `RUSTGPU_RUSTFLAGS`

`--dump-module-on-panic FILE`

If codegen panics, then write the (partially) emitted module to `FILE`. Note that this only exists for codegen, if the linker panics, this option does nothing, sadly.

`--dump-pre-link DIR`

Dumps all input modules to the linker, to files in `DIR`, before the linker touches them at all.

`--dump-post-merge DIR`

Dumps the merged module, to a file in `DIR`, immediately after merging, but before the linker has done anything else (including, well, linking the methods - `LinkageAttributes` will still exist, etc.). This is very similar to `--dump-pre-link`, except it outputs only a single file, which might make grepping through for stuff easier.

`--dump-pre-inline DIR`

Dumps the module, to a file in `DIR`, immediately before the inliner pass runs.

`--dump-post-inline DIR`

Dumps the module, to a file in `DIR`, immediately after the inliner pass runs.

`--dump-post-split DIR`

Dumps the modules, to files in `DIR`, immediately after multimodule splitting, but before final cleanup passes (e.g. DCE to remove the other entry points).

`--dump-post-link DIR`

Takes: path to directory

Dumps all output modules from the linker, to files in `DIR`. This may be multiple files due to the multimodule/module splitting option, hence it takes a directory instead of a file path.

This is the final output binary before `spirv-opt` is executed, so it may be useful to output this to check if an issue is in Rust-GPU, or in `spirv-opt`.

`--specializer-dump-instances FILE`

Dumps to `FILE` all instances inferred by the specializer.

`--no-spirv-val`

Disables running `spirv-val` on the final output. Spooky scary option, can cause invalid modules!

`--no-spirv-opt`

Forcibly disables running `spirv-opt` on the final output, even if optimizations are enabled.

`--no-dce`

Disables running dead code elimination. Can and probably will generate invalid modules or crash the linker, hasn't been tested for a while.

`--no-compact-ids`

Disables compaction of SPIR-V IDs at the end of linking. Causes absolutely ginormous IDs to be emitted. Useful if you're println debugging IDs in the linker (although spirv-opt will compact them anyway, be careful).

`--no-early-report-zombies`

Delays reporting zombies (aka "deferred errors") even further, to allow more legalization. Currently this also replaces the zombie reporting with a SPIR-T-based version (which may become the default in the future).

`--no-infer-storage-classes`

Disables the old SPIR-V "Storage Class" (i.e. address space) inference pass, to allow testing alternatives to it (such as SPIR-T `qptr` passes).

Note that this will produce illegal SPIR-V by default, and you need e.g. `--spirt-passes=qptr` in order to regain legal "Storage Class" assignments
(see SPIR-T `qptr` PR for more information on `qptr` in general)

`--no-structurize`

Disables CFG structurization. Probably results in invalid modules.

`--spirt` *(until 0.6.0)*

~~Note: as of `rust-gpu 0.6.0`, `SPIR-🇹` is enabled by default. Use `--no-spirt` to disable.~~
Note: as of `rust-gpu 0.8.0`, `SPIR-🇹` is always being used and cannot be disabled (to reduce the cost of maintenance, testing and further feature development).

Enables using the experimental `SPIR-🇹` shader IR framework in the linker - more specifically, this:

- adds a `SPIR-V -> SPIR-🇹 -> SPIR-V` roundtrip
(future `SPIR-🇹` passes would go in the middle of this, and eventually codegen might not produce `SPIR-V` at all)
- replaces the existing structurizer with `SPIR-🇹` structurization (which is more robust and can e.g. handle `OpPhi` s)
- runs some existing `SPIR-V` legalization/optimization passes (`mem2reg`) *before* inlining, instead of *only after* (as the `OpPhi` s they would produce are no longer an issue for structurization)

For more information, also see the `SPIR-🇹` repository.

`--no-spirt` *(0.6.0 and 0.7.0)*

Note: as of `rust-gpu 0.8.0`, `SPIR-🇹` is always being used and cannot be disabled (to reduce the cost of maintenance, testing and further feature development).

Note: if you were using `--no-spirt` to work around `naga` issue #1977
(valid loops causing `The 'break' is used outside of a 'loop' or 'switch' context`),

you may be able to `cargo update -p naga` to update to a fixed `naga` version (`0.11.1` for `wgpu 0.15` , `0.12.1` for `wgpu 0.16` , and any later versions).

Disables the `SPIR-🇹` shader IR framework in the linker.

`--spirt-passes PASSES`

Enable additional `SPIR-🇹` passes, as listed in `PASSES` (comma-separated). Their implementation can be found in `rustc_codegen_spirv::linker::spirt_passes` .

Note: passes that are not already enabled by default are considered experimental and likely not ready for production use, this flag exists primarily for testing.

`--dump-spirt-passes DIR`

Dump the `SPIR-🇹` module across passes (i.e. all of the versions before/after each pass), as a combined report, to a pair of files (`.spirt` and `.spirt.html`) in `DIR` .

(the `.spirt.html` version of the report is the recommended form for viewing, as it uses tabling for versions, syntax-highlighting-like styling, and use->def linking)

`--spirt-strip-custom-debuginfo-from-dumps`

When dumping (pretty-printed) `SPIR-🇹` (e.g. with `--dump-spirt-passes`), strip all the custom (Rust-GPU-specific) debuginfo instructions, by converting them to the standard SPIR-V debuginfo (which `SPIR-🇹` understands more directly).

The default (keeping the custom instructions) is more verbose, but also lossless, if you want to see all instructions exactly as e.g. `--spirt-passes` see them.

`--spirt-keep-debug-sources-in-dumps`

When dumping (pretty-printed) `SPIR-🇹` (e.g. with `--dump-spirt-passes`), preserve all the "file contents debuginfo" (i.e. from SPIR-V `OpSource` instructions), which will end up being included, in full, at the start of the dump.

The default (of hiding the file contents) is less verbose, but (arguably) lossier.

`--spirt-keep-unstructured-cfg-in-dumps`

When dumping (pretty-printed) `SPIR-🇹` (e.g. with `--dump-spirt-passes`), include the initial unstructured state, as well (i.e. just after lowering from SPIR-V).

The default (of only dumping structured SPIR-T) can have far less noisy dataflow, but unstructured SPIR-T may be needed for e.g. debugging the structurizer itself.

Publishing rust-gpu on crates.io

This is a task list for the maintainers of rust-gpu to remember to do when publishing a new version of rust-gpu (probably not useful for contributors without access to our crates.io account 😋)

The published crates and their relative locations are:

1. `spirv-std-types` (`crates/spirv-std/shared`)
2. `spirv-std-macros` (`crates/spirv-std/macros`)
3. `spirv-std` (`crates/spirv-std`)
4. `rustc_codegen_spirv-types` (`crates/rustc_codegen_spirv-types`)
5. `rustc_codegen_spirv` (`crates/rustc_codegen_spirv`)
6. `spirv-builder` (`crates/spirv-builder`)

Publishing the crates in above order prevents dependency issues. These are the steps:

1. Bump all the versions to the next one in the workspace's `Cargo.toml` . This project uses workspace inheritance, so this is the only place you'll find these actual versions. Make sure to pin the rust-gpu dependencies to their *exact* versions using the `=` -notation, such as: `=0.4.0` . All crates are built and published in tandem so you're not expected to be able to mix and match between versions.
2. Add this new version to the table in `crates/spirv-builder/README.md` and make sure the correct nightly version is listed there as well.
3. Create a PR with that change. Wait for CI and a review, and merge it.
4. Pull the merged `main` branch.
5. Tag `main` with the version: `git tag v0.4.0`
6. Push the tag: `git push origin v0.4.0`
7. Publish the crates: `cd [crate] && cargo publish` in the order of the list above. The crates.io index might take some seconds to update causing an error if the crates are published in quick succession. Wait a couple of seconds and try again 🙃.

Platform Support

The `rust-gpu` project currently supports a limited number of platforms and graphics APIs. Right now we're not distributing build artifacts and we're primarily focused on the development of the project, so this is based on the current `main` branch. There are a lot of different configurations and hardware out there to support, this document is intended to document what is currently supported, what we intend to support, and what isn't supported. Over time as the project stabilises and our CI improves, more platforms and APIs will be supported and tested. Currently support for each topic is divided into the following three categories.

- **Primary —** Built and tested on CI.
- **Secondary —** Built but *not fully tested* on CI.
- **Tertiary —** Present in the codebase but not built or tested.

Operating System

Operating System	Version	Support	Notes
Windows	10+	Primary	
Linux	Ubuntu 18.04+	Primary	
macOS	Catalina (10.15)+	Secondary	Using MoltenVK, requires v1.1.2+
Android	Tested 10-11	Secondary	

Graphics APIs

Name	Version	Support	Notes
SPIR-V	1.3+	Primary	
Vulkan	1.1+	Primary	
WGPU	0.6	Primary	Uses a translation layer to Metal/DX12
OpenGL	???	Tertiary	

SPIR-V Targets

- `spirv-unknown-spv1.0`
- `spirv-unknown-spv1.1`
- `spirv-unknown-spv1.2`
- `spirv-unknown-spv1.3`
- `spirv-unknown-spv1.4`
- `spirv-unknown-spv1.5`

Vulkan Targets

- `spirv-unknown-vulkan1.0`
- `spirv-unknown-vulkan1.1`
- `spirv-unknown-vulkan1.1spv1.4`
- `spirv-unknown-vulkan1.2`

WebGPU Targets

- `spirv-unknown-webgpu0`

OpenGL Targets

- `spirv-unknown-opengl4.0`
- `spirv-unknown-opengl4.1`
- `spirv-unknown-opengl4.2`
- `spirv-unknown-opengl4.3`
- `spirv-unknown-opengl4.5`

OpenCL Targets

- `spirv-unknown-opencl1.2`
- `spirv-unknown-opencl1.2embedded`
- `spirv-unknown-opencl2.0`
- `spirv-unknown-opencl2.0embedded`
- `spirv-unknown-opencl2.1`
- `spirv-unknown-opencl2.1embedded`

- `spirv-unknown-opencl2.2`
- `spirv-unknown-opencl2.2embedded`

GPU

Currently we don't have specific generations of GPUs for support, as long they support Vulkan 1.1+ with the latest officially installed drivers it should be able build and run the examples. You can check your Vulkan version using the `vulkaninfo` command from the `vulkan-sdk`.

Drivers

- **AMD**
- **Intel:** Linux, macOS, Windows
- **Nvidia**

Writing Shader Crates

This is section is going to walk you through writing a shader in Rust and setting up your shader crate.

Be aware that this project is in a very early phase, please file an issue if there's something not working or unclear.

Online

You can now test out and try building shaders with rust-gpu from the browser!

- SHADERed A shader IDE which has a lite version, which allows you to build and run shaders on the web.
- Shader Playground A playground for building and checking the output of shader code similar to godbolt or play.rust-lang.org.

Local Setup

There are two main ways to setup your shader project locally.

1. Using the `spirv-builder` crate. The `spirv-builder` is a crate designed to automate the process of building and linking the `rust-gpu` to be able to compile SPIR-V shaders into your main Rust crate.
2. Using `.cargo/config`. Alternatively if you're willing to do the setup yourself you can manually set flags in your cargo configuration to enable you to run `cargo build` in your shader crate.

Using `spirv-builder`

If you're writing a bigger application and you want to integrate SPIR-V shader crates to display, it's recommended to use `spirv-builder` in a build script.

1. Copy the `rust-toolchain.toml` file to your project. (You must use the same version of Rust as `rust-gpu`. Ultimately, the build will fail with a nice error message when you don't use the exact same version)

2. Reference `spirv-builder` in your Cargo.toml:

   ```
   [build-dependencies]
   spirv-builder = "0.9"
   ```

   All dependent crates are published on crates.io.
3. Create a `build.rs` in your project root.

`build.rs`

Paste the following into `build.rs`

```
use spirv_builder::{MetadataPrintout, SpirvBuilder};

fn main() -> Result<(), Box<dyn std::error::Error>> {
    SpirvBuilder::new(shader_crate, target)
        .print_metadata(MetadataPrintout::Full)
        .build()?;
    Ok(())
}
```

Substituting `shader_crate` with a relative path to your shader crate. The values available for the `target` parameter are available here. For example, if building for vulkan 1.1, use `"spirv-unknown-vulkan1.1"`.

The `SpirvBuilder` struct has numerous configuration options available, see documentation.

`main.rs`

The following will directly include the shader module binary into your application.

```
const SHADER: &[u8] = include_bytes!(env!("<shader_crate>.spv"));
```

> **Note** If your shader name contains hyphens, the name of environment variable will be the name with hyphens changed to underscores.

Keep in mind that by default, build-dependencies are built in debug mode. This means that the rust-gpu compiler (`rustc_codegen_spirv`) will be built in debug mode, and will be *incredibly* slow. You can solve this by placing this bit of configuration in your workspace

`Cargo.toml` :

```
# Compile build-dependencies in release mode with
# the same settings as regular dependencies.
[profile.release.build-override]
opt-level = 3
codegen-units = 16
[profile.dev.build-override]
opt-level = 3
```

Keep in mind this will optimize *all* build script dependencies as release, which may slow down full rebuilds a bit. Please read this issue for more information, there's a few important caveats to know about this.

Using `.cargo/config.toml`

> **Note** This method will require manually rebuilding `rust-gpu` each time there has been changes to the repository.

If you just want to build a shader crate, and don't need to automatically compile the SPIR-V binary at build time, you can use `.cargo/config.toml` to set the necessary flags. Before you can do that however you need to do a couple of steps first to build the compiler backend.

1. Clone the `rust-gpu` repository
2. `cargo build --release` in `rust-gpu` .

Now you should have a `librustc_codegen_spirv` dynamic library available in `target/release` . You'll need to keep this somewhere stable that you can reference from your shader project.

Copy the `rust-gpu/rust-toolchain.toml` file to your project. You must use the same version of Rust as `rust-gpu` so that dynamic codegen library can be loaded by `rustc` .

Now we need to add our `.cargo/config.toml` file that can be used to teach `cargo` how to build SPIR-V. Here are a few things we need to mention there.

- Path to a spec of a target you're compiling for (see platform support). These specs reside in a directory inside the `spirv-builder` crate and an example relative path could look like `../rust-gpu/crates/spirv-builder/target-specs/spirv-unknown-spv1.3.json` .
- Absolute path to the `rustc_codegen_spirv` dynamic library that we built above.

- Some additional options.

```toml
[build]
target = "<path_to_target_spec>"
rustflags = [
    "-Zcodegen-backend=<absolute_path_to_librustc_codegen_spirv>",
    "-Zbinary-dep-depinfo",
    "-Csymbol-mangling-version=v0",
    "-Zcrate-attr=feature(register_tool)",
    "-Zcrate-attr=register_tool(rust_gpu)"
]

[unstable]
build-std=["core"]
build-std-features=["compiler-builtins-mem"]
```

Now we can build our crate with cargo as normal.

```
cargo build
```

Now you should have `<project_name>.spv` SPIR-V file in `target/debug` that you can give to a renderer.

Writing your first shader

Configure your shader crate as a `"dylib"` type crate, and add `spirv-std` to its dependencies:

```toml
[lib]
crate-type = ["dylib"]

[dependencies]
spirv-std = { version = "0.9" }
```

Make sure your shader code uses the `no_std` attribute and makes the `spirv` attribute visible in the global scope. Then, you're ready to write your first shader. Here's a very simple fragment shader called `main_fs` as an example that outputs the color red:

```rust
#![no_std]

use spirv_std::spirv;
use spirv_std::glam::{vec4, Vec4};

#[spirv(fragment)]
pub fn main_fs(output: &mut Vec4) {
    *output = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Attribute syntax

rust-gpu introduces a number of SPIR-V related attributes to express behavior specific to SPIR-V not exposed in the base rust language.

Before you'll able to use these attributes, make sure you import the attribute from the `spirv-std` crate:

```
use spirv_std::spirv;
```

There are a few different categories of attributes:

Entry points

When declaring an entry point to your shader, SPIR-V needs to know what type of function it is. For example, it could be a fragment shader, or vertex shader. Specifying this attribute is also the way rust-gpu knows that you would like to export a function as an entry point, no other functions are exported.

Example:

```
#[spirv(fragment)]
fn main() { }
```

Common values are `#[spirv(fragment)]` and `#[spirv(vertex)]` . A list of all supported names can be found in spirv_headers - convert the enum name to snake_case for the rust-gpu attribute name.

Compute shader dimensions

The dimensions (`local_size_*` in openGL, `numthreads` in DX) of a compute shader must be specified (eg. `#[spirv(compute(threads(32, 16, 97)))]`). Trailing ones may be elided.

Example:

```
// the x dimension is required
// same as threads(32, 1, 1)
#[spirv(compute(threads(32)))]
pub fn compute_1() {}

// same as threads(32, 57, 1)
#[spirv(compute(threads(32, 57)))]
pub fn compute_2() {}
```

Override entry point name

You can override the default `OpEntryPoint` name for any entry point with the `entry_point_name` sub-attribute on any of the execution model attributes. (e.g. `#[spirv(vertex(entry_point_name="foo"))]`)

Builtins

When declaring inputs and outputs, sometimes you want to declare it as a "builtin". This means many things, but one example is `gl_Position` from glsl - the GPU assigns inherent meaning to the variable and uses it for placing the vertex in clip space. The equivalent in rust-gpu is called `position` .

Example:

```
#[spirv(vertex)]
fn main(
    #[spirv(position)] out_pos: &mut Vec4,
) { }
```

Common values are `#[spirv(position)]` , `#[spirv(vertex_id)]` , and many more. A list of all supported names can be found in spirv_headers - convert the enum name to snake_case for the rust-gpu attribute name.

Descriptor set and binding

A SPIR-V shader must declare where uniform variables are located with explicit indices that match up with CPU-side code. This can be done with the `descriptor_set` and `binding` attributes. Note that `descriptor_set = 0` is reserved for future use, and cannot be used.

Example:

```
#[spirv(fragment)]
fn main(
    #[spirv(uniform, descriptor_set = 2, binding = 5)] var: &mut Vec4,
) { }
```

Both descriptor_set and binding take an integer argument that specifies the uniform's index.

Flat

The flat attribute corresponds to the flat keyword in glsl - in other words, the data is not interpolated across the triangle when invoking the fragment shader.

Example:

```
#[spirv(fragment)]
fn main(#[spirv(flat)] obj: u32) { }
```

Invariant

The invariant attribute corresponds to the invariant keyword in glsl. It can only be applied to output variables.

Example:

```
#[spirv(vertex)]
fn main(#[spirv(invariant)] var: &mut f32) { }
```

Workgroup shared memory

The `workgroup` attribute defines shared memory, which can be accessed by all invocations within the same workgroup. This corresponds to `groupshared` memory in hlsl or `shared` memory in glsl.

Example:

```
#[spirv(compute(threads(32)))]
fn main(#[spirv(workgroup)] var: &mut [Vec4; 4]) { }
```

Generic storage classes

The SPIR-V storage class of types is inferred for function signatures. The inference logic can be guided by attributes on the interface specification in the entry points. This also means it needs to be clear from the documentation if an API requires a certain storage class (e.g `workgroup`) for a variable. Storage class attributes are only permitted on entry points.

Specialization constants

Entry point inputs also allow access to SPIR-V "specialization constants", which are each associated with an user-specified numeric "ID" (SPIR-V `SpecId`), used to override them later ("specializing" the shader):

- in Vulkan: during pipeline creation, via `VkSpecializationInfo`
- in WebGPU: during pipeline creation, via `GPUProgrammableStage` *#constants*
  - note: WebGPU calls them "pipeline-overridable constants"
- in OpenCL: via `clSetProgramSpecializationConstant()` calls, before `clBuildProgram()`

If a "specialization constant" is not overridden, it falls back to its *default* value, which is either user-specified (via `default = ...`), or `0` otherwise.

While only "specialization constants" of type `u32` are currently supported, it's always possible to *manually* create values of other types, from one or more `u32` s.

Example:

```rust
#[spirv(vertex)]
fn main(
    // Default is implicitly `0`, if not specified.
    #[spirv(spec_constant(id = 1))] no_default: u32,

    // IDs don't need to be sequential or obey any order.
    #[spirv(spec_constant(id = 9000, default = 123))] default_123: u32,

    // Assembling a larger value out of multiple `u32` is also possible.
    #[spirv(spec_constant(id = 100))] x_u64_lo: u32,
    #[spirv(spec_constant(id = 101))] x_u64_hi: u32,
) {
    let x_u64 = ((x_u64_hi as u64) << 32) | (x_u64_lo as u64);
}
```

Note: despite the name "constants", they are *runtime values* from the perspective of compiled Rust code (or at most similar to "link-time constants"), and as such have no connection to *Rust constants*, especially not Rust type-level constants and `const` generics - while specializing some e.g. `fn foo<const N: u32>` by `N` long after it was compiled to SPIR-V, or using "specialization constants" as Rust array lengths, Rust would sadly require *dependent types* to type-check such code (as it would for e.g. expressing C `T[n]` types with runtime `n`), and the main benefit over truly dynamic inputs is a (potential) performance boost.

Inline Assembly

Rust-GPU has support for inline SPIR-V assembly. In addition the backend provides several conveniences for writing inline assembly that are documented below. For more information on specific instruction behaviour and syntax, please refer to the SPIR-V specification.

Basic syntax & usage.

You can write inline assembly using the new `asm!` macro available with the `asm` feature on nightly. Refer to the Rust unstable book for more information on how to use the macro.

Non-ID arguments are written as-is, e.g.

```
asm! {
    "OpCapability DerivativeControl"
}
```

ID based arguments are prefixed with `%` and their name. `Result<id>` s accessed with a `=` and a ID on the left hand side of the expression. E.g.

```
let vector = spirv_std::glam::Vec2::new(1.0, 0.0);
let mut result = f32::default();

asm! {
    "%vector = OpLoad _ {vector}",
    "%element = OpVectorExtractDynamic _ %vector {index}",
    "OpStore {element} %element",
    vector = in(reg) &vector,
    index = in(reg) index,
    element = in(reg) &mut result
}
```

`asm!` only accepts integers, floats, SIMD vectors, pointers and function pointers as input variables. However you can have the pointer point to a generic variable, so you can write generic assembly code like so.

```rust
use spirv_std::{scalar::Scalar, vector::Vector};

// This fn is available as `spirv_std::arch::vector_extract_dynamic`
pub unsafe fn vector_extract_dynamic<T: Scalar, V: Vector<T>>(vector: V, index:
usize) -> T {
    let mut result = T::default();

    asm! {
        "%vector = OpLoad _ {vector}",
        "%element = OpVectorExtractDynamic _ %vector {index}",
        "OpStore {element} %element",
        vector = in(reg) &vector,
        index = in(reg) index,
        element = in(reg) &mut result
    }

    result
}
```

Additional syntax

Syntax	Description
`%<name>`	Used to refer to an abstract ID, every unique `<name>` use generates a new ID.
`typeof{<variable>}`	Returns the type of `variable`
`_` (underscore)	Equivalent to `typeof{<variable>}`, but uses inference to determine the variable

Image type syntax

There are a huge number of combinations of image types in SPIR-V. They are represented by a const generic type called `spirv_std::image::Image`, however, specifying the generic parameters of this type is incredibly tedious, so a wrapper macro, `spirv_std::Image!` can be used to write the type instead.

The specific syntax and meaning of the arguments to the `Image!` macro can be found in rustdoc.

Some type aliases for common image formats can be found in the `spirv_std::image` module. For example, `Image2d` is a very commonly used type, corresponding to `texture2D` in GLSL, and is likely what you want if you want a regular old sampled texture.

```
type Image2d = Image!(2D, type=f32, sampled);
```