A Runtime SPIR-V patcher for code specialization of graphics and compute kernel

Tendsin Mende
Technische Universität Dresden
tendsin.mende@mailbox.tu-dresden.de

Abstract

Vulkan, SysCL as well as OpenCL, can be programmed on the GPU using the SPIR-V format. SPIR-V acts as IR between the (high level) programming language (e.g. GLSL, SysCL, OpenCL C / C++) and the graphics driver. The SPIR-V programs must be completely defined before they are passed to the graphics driver. That is, no driver-side linking of program parts can be assumed.

The project tries to extend the concept of specialization constants to specialization code. This allows shader code to be runtime transformed by user generated content, or procedurally generated content.

Motivation

Vulkan as well as OpenCL, two modern, open graphics and GPGPU APIs, can be programmed on the GPU using the SPIR-V format. SPIR-V acts as IR between the (high level) programming language (e.g. GLSL, SysCL, OpenCL C / C++) and the graphics driver [1]. The SPIR-V programs must be completely defined before they are passed to the graphics driver. That is, no driver-side linking of program parts can be assumed.

SPIR-V's standard includes linking capabilities, but these are not implemented in the high-level graphics frontends (both GLSL and HLSL) [2]. Furthermore, the planned system could not only link functions, but change whole parts of the program.

For example, in the Godot project it is necessary to redefine code that uses specialisation constants to make it DXIL compatible [3]. This cannot be done by linking.

Currently, shader code-specialisation (or optimisation) is done by compiling every permutation of a shader into a separate file which is loaded at runtime. Source [2] goes into depth on how those systems work in practice. Apart from their complexity, such systems have the disadvantage, that every possible state must be known at compile time, which is why integrating user-generated content, or procedurally generated content is difficult.

The project tries to extend the concept of specialisation constants [4]/[5] to *specialisation code*. This is to be realised conceptually via a SPIR-V $\rightarrow$ SPIR-V patch mechanism.

IR-Analysis

A seen by Khronos documentation, SPIR-V is intended as a *communication format* between compiler infrastructure (at compile time) and driver infrastructure at runtime.

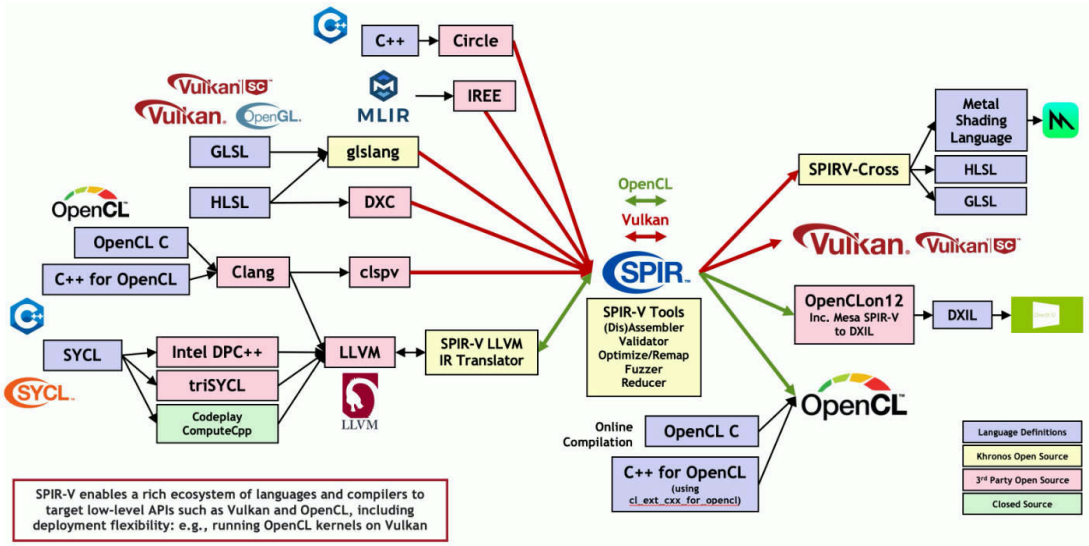


Figure 1: SPIR-V Language Ecosystem
https://www.khronos.org/spir/

None of its stated goals (as seen in section *1.1 Goals* of the specification [5]) contain strictly compiler related transformation goals. Instead it focuses on stability, easy parseability and easy translation from and into other IR formats.

As a result most compilers and drivers use another internal IR to do either compilation to SPIR-V, or from SPIR-V to GPU specific code.

As Figure 1 shows, multiple languages as well as compiler infrastructures like LLVM and MLIR have the capability to compile to SPIR-V. On the other site compute and graphics APIs like Vulkan or OpenCL consume SPIR-V directly, or translate it into other intermediate formats like DXIL before supplying it to the API. Internally at least Linux's MESA driver uses another custom IR, called NIR [6], to translate SPIR-V to the actual GPU code.

Another interesting opensource shader-compiler is the *AMD compiler* (ACO) within mesa as well [7]. It is a backend to the former mentioned NIR specifically for AMD-Hardware.

Conceptually we can split Shader related IRs based on their position in relation to SPIR-V. On one hand we have compilation focused IRs like LLVM, MLIR or, the more shader oriented IRs like SPIR-T. On the other hand we have runtime GPU-Code generation focused IRs like NIR.

Compiler related IRs

On the compiler site we have roughly two approaches to translating a highlevel language to SPIR-V. First we have common LLVM based compiler stacks like SYSCL's. Secondly we have more monolithic approaches like GLSL's and HLSL's stack. An observation is, that GPGPU related languages seem to favour the LLVM (or MLIR) stacks, while graphics related languages favour a custom monolithic stack.

While I couldn't make out a single common reason for this, two main factors play a roll. The first one being controll over the compiler stack, including (simple) distribution and design decisions (See *In defense of NIR*[1] for better explanation). The second being simplicity. Graphics shader are often focused on a certain kind of work (like fragment shading, vertex transformation etc.). Therefore, more informed transformation's can be implemented directly, compared to general-purpose GPU programs.

[1] https://www.gfxstrand.net/faith/blog/2022/01/in-defense-of-nir/

Another reason for which I couldn't find a citeable source is the history of Shader compilers. Only the latest graphics and GPGPU APIs target some kind of byte-format as input. APIs before that where either semi-non-programmable (DirectX up to version 8, and OpenGL until version 2.0), or took actual programme code as input(DirectX until version 12 which introduces DXIL, OpenGL until version 4.6 which introduces SPIR-V capabilities similar to Vulkan). The compiler would therefore recite within the driver stack. This has two implications.

1. The compiler must be shipped with the driver
2. The compiler must be fast enough to compile the code to executable GPU-Code at runtime.

Shader related IRs

For Shader focused IRs we have specialised IRs for programming languages like MLIR dialects (IREE or the SPIR-V dialect) as well as custom solutions like SPIR-T which is used internally in Rust-Gpu. They focus mostly on specialising code for GPU usage. Interestingly we can see that OpenSource driver stack have their own internal IRs that compile SPIR-V (or some other communication IR format) down to the actual ISA instructions. Two notable toolchains are Mesa's NIR, and a special Shader compiler for AMD's ISA called ACO (*AMD_Compiler*[1])

Decision

I decide to use SPIR-V directly for the most part. If I need to do more complex Shader code transformation's I'll try to use SPIR-T, which is not that different to Mesa's NIR. The lifting and lowering are proven to be fast (I spoke to the main developer *eddyb*).

I decided against lifting to MLIR because the patching mechanism is in its nature a rather technical procedure. I anticipate that I wouldn't gain the right type of flexibility I'd need. This only means the *patching* part though. MLIR would probably be the right choice if I want to compile some DSL down to SPIR-V, which then gets patched into a template SPIR-V program.

Patches

NonUniform decoration

Problem description

Currently parts of the program are analysed by the driver (see ACO description) for diverging execution. Others have to be explicitly tagged by the programmer. Mostly when descriptors are indexed non-uniformly. In GLSL this is done via `nonuniformEXT(int i)`. For instance like this:

```
layout(location = 0) flat in int i;
layout(set = 0, binding = 0) uniform sampler2D tex[2];
/*void main(){...*/
vec4 color = texture(tex[nonuniformEXT(i)], ...);
/*...}*/
```

This effectively marks the index `i` as *possibly different per invocation group*. However in practice this has several problems:

1. When this is needed is not always easy to see
2. When forgotten, bugs are subtil
3. Some drivers seem to handle it well if forgotten.

Intuitive solution

The first observation is, that only descriptor indexing related instructions need to be marked `NonUniform`. Therefore, the pass does not have to explore all indexing, but just the ones indexing into descriptors bindings.

[1] https://gitlab.freedesktop.org/mesa/mesa/-/blob/main/src/amd/compiler/README.md

A second observation is, that *per-invocation non-uniform indexing* has a finite count of sources. One is non-uniform control flow, the other is non-uniform input variables. The latter is found by tracing the index calculation for known non-uniform input variables like `invocation-index` or `vertex-index` etc.

Finding non-uniform control-flow is not as easy. The ACO compiler actually does most of its work in that are. Therefore, we reverse the problem and decorate *every* descriptor_indexing as *NonUniform* by default, and just remove the decoration, if we are absolutely sure that it isn't needed.

Implementation

The implementation has four stages:

1. find and cache variables with an potentially non-uniform value (`seed_variables`)
2. find all `OpAccessChain` OPs that access a RuntimeArray, and trace the indices. (`trace_indices`)
3. decorate `OpAccessChain` on RuntimeArrays with non-uniform index (`decorate`)
4. fix module extensions and capabilities by adding `SPV_EXT_descriptor_indexing` and all non-uniform capabilities (`fix_capabilities_and_extensions`)

In practice some of the added capabilities might not be needed. However, they do not influence the resulting ISA-code, since capabilities only signal the possibility of non-uniform access. The pass might decorate *too many* access as `NonUniform`.

Function injection

Handling functions in shader code

Shaders/Kernels are often small, GPU specialised programs. A property of this GPU-Specialisation is that the programmer, and compiler try to eliminate invocation-group wide controll-flow divergence. This combination often results in highly inline programs with few function calls.

SPIR-V contains a `DontInline` hint, but this cannot be used in all front ends. Namely GLSLang (the GLSL-compiler) which is often used for Shader programming does not contain a way to annotate functions in any way.

We therefore have a problem when it comes to identifying a certain callsite in SPIR-V modules. Because those might already be inlined.

The solution is to come up with two versions of the function-patching mechanism.

1. Linking-like replacement of non-inlined functions
2. Injecting custom function call as variable assignment

Linking and replacing

Both following passes rely on a common function-identification. We allow two types. Identification by type-signature, and identification by debug-name-string. The latter is similar to the way a linker might use a identification string for dynamic linking of two functions. The functionality can be found in `crates/patch-function/src/function_finder.rs`

For injection we have to distinguish two versions:

1. Merging known code into an known template. This is similar to standart linking. We'll call this *Constant Replacement*
2. Loading a template, and, (possibly based on its content) injecting new code. We'll call this *Dynamic Replacement*

Constant Replacement

The described code can be found in `crates/patch-function/src/constant_replace.rs`

Since constant replacement is similar to linking, we use the `spirv-link` program provided by Khronos for the most part. Since our patcher can not rely on the correct linking annotations in the template code, as well as the *to be injected* code, we start by modifying both with the correct linking annotations.

Afterwards we assemble both into byte-code and let the linker work. The resulting module is read back into the patcher.

Right now this relies on three files being written (both input files and the output file), which leads to subpar performance compared to the in-memory patches.

Dynamic Replacement

The described code can be found in `crates/patch-function/src/dynamic_replace.rs`

After identifying our *to be replaced* function, we copy the function's definition and start a new basic-block. At this point we hand over control flow to a user specified function (or *closure* in Rust terms), that is free to append any custom code. The function is supplied with knowledge about the provided parameters, as well as the expected return type.

In a last pass all call sites of the former function are replaced with a call to the new function. Since all parameters, return IDs etc. do match by definition, this is safe to do.

There are two advantages to the *Constant Replacement* approach. The first is our runtime knowledge over the whole SPIR-V-Module. We don't need to merge different modules (meaning type-ids, capabilities, extensions etc.), instead we can naturally use and extend the template module.

The second advantage is the in-memory nature of the patch. The template is already loaded and can be mutated largely in-place. In theory this is also possible for the *Constant Replacement* patch, but for that we'd have to re-implement the `spirv-link` application.

Callsite injection

Sadly I didn't have time to implement the call-site injection methode. While the injection itself wouldn't be too difficult, identifying *where* to place the call-site is more complex. We can't rely on any signature matching, since any assigned variable type can potentially be used multiple times. The only relyable way I found is, by relying on debug information (specifically the `OpLine` instruction of SPIR-V) to identify the correct source-code line, and then searching for the correct assignment operation.

Testing

Code-Validation

Before running the generated code, we can validate it against the SPIR-V specification. For that a CLI programme is provided by Khronos called `spirv-val`. It simply takes a shader module on stdin (or as a file) and outputs an error if any invalid code was found. In theory any module passing that check should be valid SPIR-V code and therefore be executable. In practice `spirv-val` does not catch all errors, which is the reason for enabling Vulkan's runtime validation as well.

Runtime Validation

The second validation happens at runtime. We have two ways of checking the validity of the shader module. One is Vulkan's validation layers. They are mainly used to check that a Vulkan application conforms to the Vulkan specification. However, it also detects runtime invalid shader execution, or malformed code. We check that by turning on the validation layer and routing error output into our test.

Secondly, we check that the test returns the expected output. For that, we download the test run's result and check it against our *blessed* results. Those *blessed* results are a list of prior saved results from which we know that they are correct. This lets us ultimately find regression for code that is *valid* in itself, but produces an incorrect (or unexpected) output.

Benchmarking

There are two *domains* we can test. One is the CPU side, where we mostly measure how long any of the patches takes. The other side is GPU-runtime of the resulting code after patching. The `spv-benchmark` application reflects that. When running the app, only the GPU-side benchmarks are executed. For the CPU-side we use Criterion.rs. In can be invoked via `cargo bench`, which will run the benchmark and generate a HTML report at `target/criterion/report/index.html`.

All patching benchmarks take a *simple* code template that calculates the distance of a pixel to an image's center, and replace that with a simple mandelbrot set calculation.

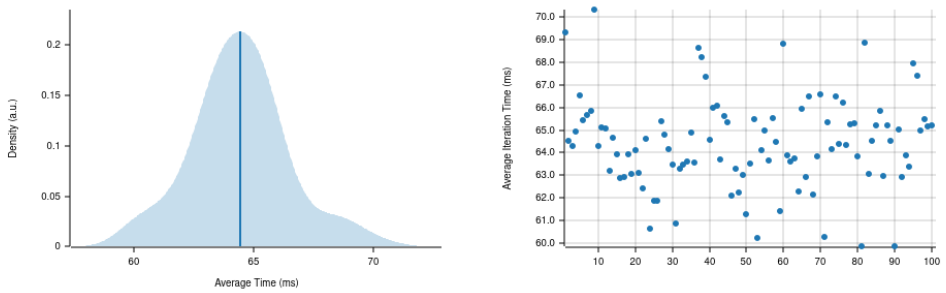
Hardware note

All numbers measured in the following benchmark are taken on a AMD Ryzen 5900X (12 core/24 threads @ 3.7GHz up to 4.8GHz). The GPU is a AMD 6800XT, the GPU benchmark uses Vulkan and the latest open-source Mesa/RADV driver.

CPU

GLSL Compilation time

For reference to *patching* we also measure a full compilation of the mandelbrot-set shader via `glslangValidator`. The shader that is compiled can be found in `crates/spv-benchmark/resources/mandelbrot.comp`.



Additional Statistics:

	Lower bound	Estimate	Upper bound
R^2	0.0016507	0.0017127	0.0016472
Mean	64.015 ms	64.401 ms	64.798 ms
Std. Dev.	1.6653 ms	2.0013 ms	2.3032 ms
Median	63.894 ms	64.359 ms	64.905 ms
MAD	1.2733 ms	1.5827 ms	2.0045 ms

Additional Plots:

- Typical
- Mean
- Std. Dev.
- Median
- MAD

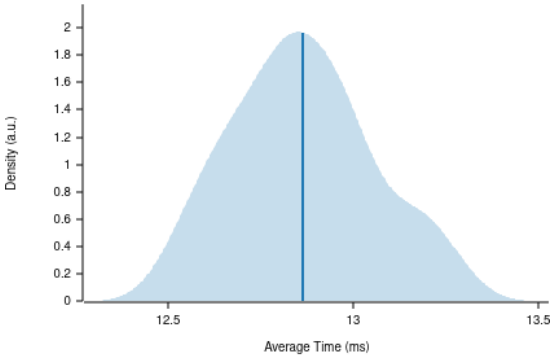
Figure 2: GlslangValidator compile time for the Mandelbrot compute shader.

The mean compiletime is 64.4ms with a standard deviation of 2ms. Therefore, for the patcher to make sense compile-time wise, we need to be faster than 64ms. Otherwise a standard recompilation would make more sense than patching.

Constant Replace

Using the *Constant Replace* patch we measure two timinings. One is only the time the patch needs, the other one is *timing including the assembling into the SPIR-V bytecode*. This allows us to

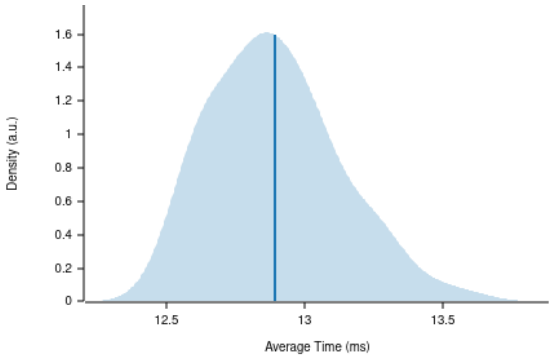
distinguish between the time introduced just by patching, and the actual time needed at runtime to get the final shader code out of the system.



Additional Statistics:

	Lower bound	Estimate	Upper bound
R^2	0.0001716	0.0001781	0.0001713
Mean	12.828 ms	12.864 ms	12.901 ms
Std. Dev.	163.62 µs	187.62 µs	208.35 µs
Median	12.814 ms	12.844 ms	12.902 ms
MAD	143.36 µs	198.35 µs	236.94 µs

Figure 3: Constant patching without assembling.



Additional Statistics:

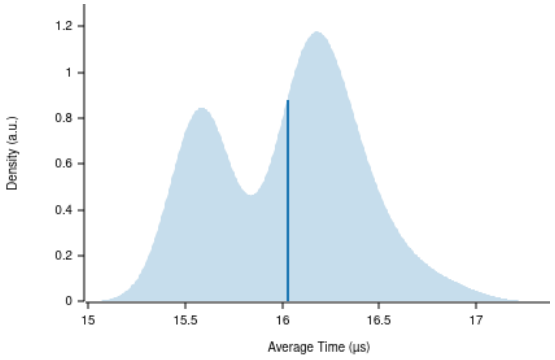
	Lower bound	Estimate	Upper bound
R^2	0.0101258	0.0105023	0.0101001
Mean	12.849 ms	12.893 ms	12.939 ms
Std. Dev.	196.74 µs	229.97 µs	259.52 µs
Median	12.817 ms	12.874 ms	12.917 ms
MAD	176.06 µs	243.48 µs	289.44 µs

Figure 4: Constant patching including assembling into shader bytecode.

The measured timings show, that the assembling time does not play a notable role in the overall runtime. However, the mere linking is faster compared to the full recompilation with a mean timing of 12.8ms.

Dynamic Replace

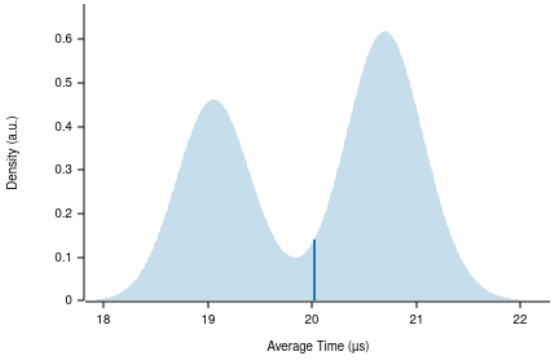
For constant replacement we measure the same times, once *only patching* and once including the assembly into a usable SPIR-V bytecode module.



Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	15.759 µs	15.839 µs	15.928 µs
R^2	0.9501268	0.9529228	0.9494593
Mean	15.956 µs	16.027 µs	16.097 µs
Std. Dev.	320.33 ns	360.09 ns	396.27 ns
Median	16.084 µs	16.104 µs	16.158 µs
MAD	235.53 ns	345.64 ns	600.83 ns

Figure 5: Dynamic replacement only



Additional Statistics:

	Lower bound	Estimate	Upper bound
Slope	20.519 µs	20.603 µs	20.674 µs
R^2	0.9340341	0.9359712	0.9346036
Mean	19.856 µs	20.018 µs	20.178 µs
Std. Dev.	773.98 ns	820.95 ns	850.50 ns
Median	19.433 µs	20.599 µs	20.650 µs
MAD	163.67 ns	397.84 ns	1.2133 µs

Figure 6: Dynamic replacement including assembling into shader bytecode.

The overall timing of the replacement patch is much shorter with a mean timing of 20µs for a usable SPIR-V bytecode module. The assembly step takes around 4-5µs. The reason for that rather fast assembly time lies in the implementation of the `rspirv` library which keeps the data representation in a easily assembled way in memory. Since the patch is applied directly on that *data representation*, no lifting or lowering pass is needed. The patching comes down to simple memory operations that append the new byte-instructions to the shader module, or mutate existing ones (specifically when rewriting the function call-site).

GPU

The GPU benchmark compares the timing of both, dynamic and constant replacement of the *simple* shader to the compiled *mandelbrot* shader, to the patched *mandebrot* shader. For compilation, we use Rust-Gpu, since we have to base our shader template on code generated by that compiler. GLSL inlines the mandebrot calculation with no way of preventing it, which makes the resulting template code unsable for our patching methods that currently rely on un-inlined functions.

Constant Replace

For constant replacement we have the following runtimes:

	Dedicated Graphics			Integrated Graphics		
Name	avg	min	max	avg	min	max
Unmodified	0.05ms	0.02ms	0.07ms	0.35ms	0.37ms	0.4ms
Rust-GPU compiled	2.31ms	2.3ms	2.35ms	11.41ms	11.41ms	11.42ms
Patched Runtime	1.55ms	1.50ms	1.58ms	13.6ms	13.65ms	13.7ms

As expected, the runtimes are pretty uniform on a per-test basis. The difference in the compiled and patched runtime can be explained with the actual resulting code. Rust-GPU seems to lose some knowledge about possible special instructions available to SPIR-V/GPU architectures. For instance, a call to `OpLength` (which calculate the length of an n-dimensional vector) is broken down into individual square operations and a following square-root of the sums of all components. The handwritten code however retains that knowledge, which in turn, is responsible for considerable shorter runtimes on the dedicated hardware. Interestingly the integrated Ryzen chip's graphics card does not profit from that.

Dynamic Replace

	Dedicated Graphics			Integrated Graphics		
Name	avg	min	max	avg	min	max
Unmodified	0.05ms	0.02ms	0.07ms	0.35ms	0.37ms	0.4ms
Rust-GPU compiled	2.0ms	2.05ms	2.2ms	11.45ms	11.45ms	11.45ms
Patched Runtime	1.55ms	1.50ms	1.58ms	13.6ms	13.65ms	13.7ms

For dynamic patching the runtime does not differ that much to constant replacement. This is to be expected, since ideally the code shouldn't differ at all. Combined with the CPU side testing we show, that it is possible to replace performant GPU code in a timely manor via runtime patching. The actual patching in this specific case has a negelectable overhead of 15µm-20µs without any runtime penalty compared to fully compiled code. The latter part however depends on the system that supplies the patched code.

Conclusion

We showed that it is feasible to replace SPIR-V byte code at runtime before supplying it to the graphics driver API. Appending new code to a known module (called *DynamicReplace* in this documentation) provides a opportunity for such operations. The reason for that is the rather good abstraction layer that SPIR-V provides in that case. Its not too low (still high-level SSA code), but also low enough that no costly graph operations are necessary to facilitate the replacement.

The linking like replacement (called *ConstantReplace* in this documentation) is currently not as performant. While reasonable fast, it could probably be implemented faster if kept in memory like the dynamic alternative. The main obstacle here is, that both modules need to be combined into a single SPIR-V context. This means that we have to analyse common data-types, header compatibility etc. before we can effectively merge both modules. A possible solution would be to lift both modules into a common, more easily mergeable IR. Alternatively one could try to replay the *to be merged* module's instruction in the context of the template module using the already existing *DynamicReplace* patch.

Finally we showed that the SPIR-V level IR is also suitable of post-compiler fix-passes. An implemented scenario patches the module to fullfill the `non-uniform` decoration requirement that is often overlooked in practice by programmers. While it can be argued that this is a shortcoming of the source programming language, this kind of patch can be helpful in realworld toolchains.

Bibliography

[1] Khronos-Group, "Spir," May 2023. [Online]. Available: https://www.khronos.org/spir/

[2] M. Pettineo, "The shader permutation problem - part 2: how do we fix it?," Oct. 2021. [Online]. Available: https://therealmjp.github.io/posts/shader-permutations-part2/#offline-linking

[3] P. J. Estébanez, "Direct3d 12: adventures in shaderland," Apr. 2023. [Online]. Available: https://godotengine.org/article/d3d12-adventures-in-shaderland/

[4] Intel, "Specialization constants," Jan. 2023. [Online]. Available: https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2023-0/specialization-constants.html

[5] Khronos-Group, "Spir-v specification," Apr. 2023. [Online]. Available: https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html#_specialization_2

[6] The-Mesa-3D-Graphics-Library, "Nir intermediate representation (nir)," Jan. 2023. [Online]. Available: https://docs.mesa3d.org/nir/index.html#nir-intermediate-representation-nir

[7] The-Mesa-3D-Graphics-Library, "Amd compiler," Apr. 2023. [Online]. Available: https://docs.mesa3d.org/drivers/radv.html?highlight=aco#aco