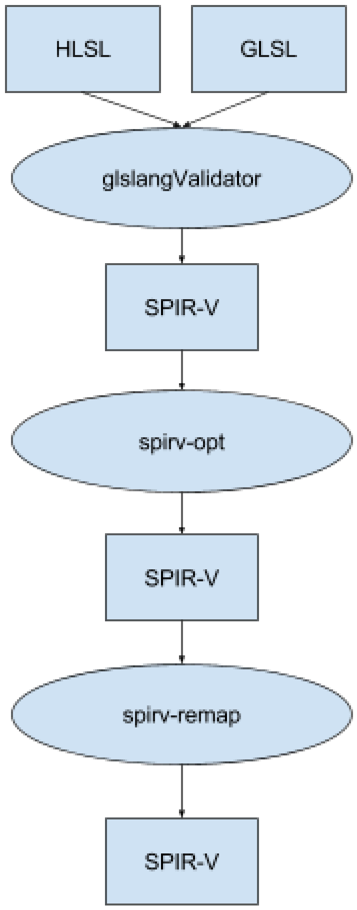
SPIR-V Shader Size Reduction Using spirv-opt

Greg Fischer
LunarG Inc.

Summary

This paper will discuss how code transformations available as passes in spirv-opt (github.com/KhronosGroup/SPIRV-Tools) can be used to reduce the size of SPIR-V (khronos.org/registry/spir-v) shaders for the Vulkan graphics API (khronos.org/vulkan). It will discuss the specific passes that can be used and what they do so that users of spirv-opt can best decide how to reduce the size of their SPIR-V shaders.



Proposed SPIR-V Workflow

Introduction

Since the inception of SPIR-V, there has been interest in optimizing its code, particularly with the goal of reducing its size. Understandably, the raw SPIR-V emitted from the frontend glslangValidator (github.com/KhronosGroup/glslang) is verbose, particularly with regard to function scope variables and their loads and stores. Another significant opportunity for size reduction comes from dead code due to branches with constant conditionals. Another opportunity is replicated accesses to uniform variables, including images and samplers.

We have introduced passes to spirv-opt which use classic code optimization techniques to address these inefficiencies in a SPIR-V module. Combined with spirv-remap (github.com/KhronosGroup/glslang) which we use to remove module-level dead types and functions, these passes can reduce SPIR-V size by over 60%, and bring SPIR-V sizes within %40 of DX Byte Code.

These passes have been initially designed to work with SPIR-V modules for graphics APIs such as Vulkan and OpenGL. Such shaders use logical addressing. Modules with physical addressing, such as those for the OpenCL API will regrettably not derive much benefit at this time. The scope of these passes was restricted to speed implementation and delivery of their benefits to the graphics community. The structured control flow of shaders and the simplified memory accesses of logical addressing simplify these passes. Optimization of kernels is left for future work.

There are several other features which these passes do not support. Please refer to the **Limitations** section. Running these passes on modules with unsupported features will cause the pass to return silently without changing the module.

A Recipe

The following is one suggested recipe of spirv-opt passes to reduce the size of a graphics SPIR-V module*:

```
--inline-entry-points-exhaustive
--convert-local-access-chains
--eliminate-local-single-block
--eliminate-local-single-store
--eliminate-insert-extract
--eliminate-dead-code-aggressive
--eliminate-dead-branches
--merge-blocks
--eliminate-local-single-block
--eliminate-local-single-store
--eliminate-local-multi-store
--eliminate-insert-extract
--eliminate-dead-code-aggressive
--eliminate-common-uniform
```

* (Execution of spirv-opt with this recipe should ideally be followed with execution of “spirv-remap --strip all --dce all” to remove debug instructions and module-level dead types and functions.)

Some passes expose optimization opportunities for other passes, and the recipe orders the passes to take this into account. Performing exhaustive inlining first is an important case in point.

Note that the order of spirv-opt pass options controls the order that the passes are applied and repeated options cause a repeated application of the pass.

This recipe may be somewhat overkill for some shaders, but is meant to show how all the passes might effectively fit together. Some shaders may require fewer passes; some may require more. We will now discuss each pass individually to help users make these adjustments.

Exhaustive Inlining (--inline-entry-points-exhaustive)

Shaders often are broken into separate subroutines to increase modularity. But these subroutine calls create a barrier to dataflow analysis and subsequent optimizations. One method for eliminating these barriers is through exhaustive inlining of function calls in entry point functions. While exhaustive inlining can cause code size increase, in practice the benefits of increased analysis and optimizations have outweighed these costs. This practice is also the reason that all subsequent passes initially only operate on entry point functions.

The inlining of a function in SPIR-V is generally straightforward. One exception is functions with early return. The structured control flow of SPIR-V shaders does not allow for the branch out of a conditional that would be required. For the moment, this is implemented by creating a one-trip loop around the called function, replacing the early return with a branch to the outer loop's merge block, which is permissible in SPIR-V. The only time this doesn't work is when early returns are already inside of a loop. For the moment, the inliner will not inline such functions. The resulting code will be correct, but its optimization will be curtailed by the remaining function call. Improving this situation is left to future work.

Local Variable Access Chain Conversion (--convert-local-access-chains)

Shaders often have code which packs data into and unpacks data from structures, typically across function calls. After inlining, this copying code is a significant opportunity for optimization.

Possibly surprisingly, this pass converts all local variable access chain loads and stores with constant indices into their equivalent load or store combined with an insert or extract. For example, the access chain load:

```
%20 = OpAccessChain %_ptr_Function_v4float %s0 %int_1
%21 = OpLoad %v4float %20
```

is converted to:

```
%24 = OpLoad %S_t %s0
%25 = OpCompositeExtract %v4float %24 1
```

and the access chain store:

```
%19 = OpAccessChain %_ptr_Function_v4float %s0 %int_1
OpStore %19 %18
```

Is converted to:

```
%22 = OpLoad %S_t %s0
%23 = OpCompositeInsert %S_t %18 %22 1
OpStore %s0 %23
```

Since stores are converted to a longer instruction sequence, this conversion is only done for local variables that are only accessed through such loads and stores and are thus guaranteed to ultimately be optimized away. Many of the passes in this discussion restrict their optimizations to such variables.

Among the reasons for this conversion is to allow dataflow analysis to concentrate on one form of composite reference, inserts and extracts, and allow it to ignore interaction with access chains. Another reason is that extracts have the desirable property that a group of loads from a single composite object can share a single load operation. Likewise, a sequence of inserts to the same composite object ultimately can share a single load and a single store. This allows for many loads and stores to be easily eliminated during single block local variable elimination, discussed next.

Local Store/Load Elimination - Single Block (--eliminate-local-single-block)

Elimination of local store and load instructions is a significant opportunity for SPIR-V size reduction, but elimination of loads and stores is also a de facto method for value propagation, so elimination of stores and loads aids analysis and optimizations that are dependent on values, such as dead branch elimination.

General elimination of local variables and their loads and stores across an entire function requires a complex and expensive algorithm. It is therefore often beneficial in compile time to eliminate some load and stores (and possibly their variable) with simpler and cheaper algorithms first. Eliminating stores and loads within a single block allows control flow analysis to be ignored.

This pass eliminates store/load and load/load pairs to the same local variable in the same block. It optimizes only direct loads and stores of variables. If a store is not live at the end of the block, it will be deleted.

For example, the store-load sequence in:

```
OpStore %v %14
%15 = OpLoad %v4float %v
OpStore %gl_FragColor %15
```

could be optimized to:

```
OpStore %gl_FragColor %14
```

And the load-load sequence in:

```
%31 = OpLoad %v4float %v
OpStore %32 %31
%33 = OpLoad %v4float %v
OpStore %34 %33
```

would be optimized to:

```
%31 = OpLoad %v4float %v
OpStore %32 %31
OpStore %34 %31
```

Access chain loads and stores are not optimized and may actually inhibit optimization. This optimization is therefore heavily dependent on the access chain conversion described above. Function calls can also inhibit dataflow analysis and optimization, so inlining beforehand is also highly recommended.

Local Store/Load Elimination - Single Store (--eliminate-local-single-store)

Like the single block load/store elimination above, this pass also optimizes a simple, specific case: a local variable stored to only once. All loads in the same function which the store dominates can simply be replaced with the store's value.

Access chain loads and stores are not optimized and may inhibit optimization, so access chain conversion is recommended beforehand. Function calls also may inhibit dataflow analysis and optimization, so inlining is also recommended beforehand.

Insert/Extract Elimination (--eliminate-insert-extract)

After access chain conversion and store/load removal, sequences similar to the following may appear:

```
%20 = OpCompositeInsert %S_t %18 %19 0
...
%22 = OpCompositeInsert %S_t %21 %20 1
...
%24 = OpCompositeInsert %S_t %23 %22 2
…
%26 = OpCompositeExtract %v4float %24 1
```

```
%29 = OpFMul %28 %26
```

These sequences are typical in shaders which pack and then unpack a composite object across a function call which has been inlined.

Extracts such as the one above may be simply replaced with the corresponding inserted value in the insertion sequence. For example, the Extract and FMul above can be replaced with:

```
%29 = OpFMul %28 %21
```

Similar to store/load elimination, insert/extract elimination both reduces code size and aids analysis through propagation of values.

Dead Branch Elimination (--eliminate-dead-branches)

It is possible for a shader to contain significant sections of code which are never executed because they are control dependent on a conditional branch on a value that is always false. After inlining, store/load and insert/extract elimination, some such opportunities may be exposed as constant boolean values are propagated through the shader into conditional branch instructions.

This pass finds conditional branches on constant boolean values, converts the conditional branch into the correct unconditional branch and eliminates all possible resulting dead code. For example:

```
...
OpBranchConditional %true %21 %22
%21 = OpLabel
OpStore %v %14
OpBranch %20
%22 = OpLabel
OpStore %v %16
OpBranch %20
%20 = OpLabel
%23 = OpLoad %v4float %v
…
```

Would be replaced with:

```
…
OpBranch %21
%21 = OpLabel
OpStore %v %14
OpBranch %20
%20 = OpLabel
```

```
%23 = OpLoad %v4float %v
…
```

Such dead code elimination, besides reducing the number of instructions, simplifies control flow and thus creates additional opportunities for analysis and optimization. For example, a local variable assigned to twice before dead branch elimination might only be assigned once after dead branch elimination, creating an additional opportunity for single store load/store elimination. So repetition of passes may be beneficial, depending on the shaders.

Block Merge (--merge-blocks)

After dead branch elimination, sequences of single blocks are often left, such as the "after" sequence in the dead branch elimination section above. The Block Merge pass cleans up such sequences, creating a single block from them. Specifically, this pass searches for a first block with a branch to a second block which has no other predecessors. When this is found, the first and second blocks can be combined into a single block. For example, the final sequence in the dead branch section above becomes:

```
…
OpStore %v %14
%23 = OpLoad %v4float %v
…
```

Besides eliminating instructions, this has the benefit of creating new opportunities for single block store/load elimination as it moves instructions previously in different blocks into the same block. For example, in the case above, executing single block store/load elimination would delete the load, replace %23 everywhere with %14, and the store could potentially be removed as well.

Local Store/Load Elimination - Multiple Store (--eliminate-local-multi-store)

This pass is used to eliminate all remaining local variables which are only accessed directly with loads and stores. Variables with access chain references are not optimized. Thus, this phase is most effective when it follows exhaustive inlining and access chain conversion.

The algorithm tracks each variable and its stored value through the program. If at any point multiple values for a single variable reach a block, a phi function is generated and that value is used for the variable from that point (until it is stored or another merge point is reached). If the variable is loaded, the load is deleted and the loaded value is replaced with the stored value. All stores of candidate variables are finally deleted.

For example, the following sequence:

```
…
OpBranchConditional %22 %24 %25
%24 = OpLabel
%27 = OpVectorTimesScalar %v4float %26 %float_0_5
OpStore %v %27
OpBranch %23
%25 = OpLabel
%29 = OpFAdd %v4float %28 %18
OpStore %v %29
OpBranch %23
%23 = OpLabel
%30 = OpLoad %v4float %v
OpStore %gl_FragColor %30
```

would be changed to:

```
…
OpBranchConditional %22 %24 %25
%24 = OpLabel
%27 = OpVectorTimesScalar %v4float %26 %float_0_5
OpBranch %23
%25 = OpLabel
%29 = OpFAdd %v4float %28 %18
OpBranch %23
%23 = OpLabel
%31 = OpPhi %v4float %27 %24 %29 %25
OpStore %gl_FragColor %31
```

Note all stores and loads of %v have been removed and the phi value is stored to gl_FragColor.

As stated earlier, this pass will run more efficiently if local loads and stores that can be eliminated with simpler passes are eliminated. It will also run more efficiently if any dead control flow is eliminated.

Currently this pass will generate phi functions which are not used. It is therefore beneficial to run a dead code elimination pass after this pass. Avoiding these unused phi functions are left for future work.

Dead Code Elimination - Aggressive (--eliminate-dead-code-aggressive)

This pass (aka ADCE) detects and deletes instructions in a function that are not used in computing any output value from that function. It does this by marking as live all the function’s output instructions, that is, all instructions that directly make changes outside the scope of the function. It then iteratively marks as live all instructions that these instructions use until no more instructions are marked live. All remaining instructions are dead and can be deleted. For example, consider the following sequence with dead variable %dv:

```
...
%17 = OpLoad %v4float %Dead
%18 = OpExtInst %v4float %1 Sqrt %17
OpStore %dv %18
%19 = OpLoad %v4float %v
OpStore %gl_FragColor %19
OpReturn
```

Assuming that the store to gl_FragColor is the only output instruction, there is no live load of %dv, thus it's store instruction (and the instructions that it uses) are not marked live. So they are considered dead and deleted:

```
...
%19 = OpLoad %v4float %v
OpStore %gl_FragColor %19
OpReturn
```

This version of dead code elimination is particularly good at removing dead def-use cycles. One version of these cycles is generated by the local access chain conversion pass. An example of such a def-use cycle is:

```
%19 = OpLoad %S_t %s
%20 = OpCompositeInsert %S_t %18 %19 0
%22 = OpCompositeInsert %S_t %21 %20 1
%24 = OpCompositeInsert %S_t %23 %22 2
OpStore %s %24
```

After insert/extract elimination and all extracts are eliminated these insert sequences remain. Neither the single block or single store store/load elimination passes will eliminate the store and thus the insert sequence remains. However, the ADCE algorithm will detect that this cycle is not used as part of any output computation and will delete it. For this reason, ADCE should be run after insert/extract elimination to remove these dead cycles.

Common Uniform Elimination (--eliminate-common-uniform)

Loads of uniform values is a potential source of redundant code and thus creates an opportunity for additional size reduction.

Often uniform values are packed into composite objects and they are loaded using access chains. This pass first converts uniform access chain loads into loads and extracts. Just as was true for local variables, this form allow loads from the same composite object to be shared and repeating loads to be eliminated, which is done next. If the first load does not dominate all remaining loads, it is hoisted to the nearest dominating block. Finally, common extracts are shared and the repeat extracts are eliminated.

For example, the following uniform loads:

```
%34 = OpAccessChain %_ptr_Uniform_float %u %int_0
%35 = OpLoad %float %34
OpStore %o0 %35
%36 = OpAccessChain %_ptr_Uniform_float %u %int_1
%37 = OpLoad %float %36
OpStore %o1 %37
...
%40 = OpAccessChain %_ptr_Uniform_float %u %int_1
%41 = OpLoad %float %40
OpStore %o2 %41
```

will be converted to:

```
%50 = OpLoad %U_t %u
%51 = OpCompositeExtract %float %50 0
OpStore %o0 %51
%52 = OpCompositeExtract %float %50 1
OpStore %o1 %52
…
OpStore %o2 %52
```

Two forms of common uniform loads are the images and samplers used in texture references. However, these cannot be removed from the blocks which contain the texture reference. So this pass does a special traversal just to eliminate these common uniform loads within a single block.

This pass is fairly independent from the other passes. It does not depend on any of them and they do not depend on it.

Limitations

Besides kernels and physical addressing, there are a few other features which are not currently supported and will cause these passes to return silently without making changes.

Most passes currently do not support the extension KHR_variable_pointers. While not as big of an effort, most passes also currently do not support OpGroupDecorate, and --convert-local-access-chains and --eliminate-common-uniform do not support modules which contain non-32-bit integers.

These limitations exist because of cost/benefit calculation and the desire to make these passes available as soon as possible for the restricted functionality. Support may be added in the future as priorities change.

Future Work

A version of inlining has been requested which only inlines very small functions and functions that are only called one time. This assures that the final size of the function is no larger than the original size.

Several other optimizations could be beneficial in reducing SPIR-V size. Constant folding could allow more constant branches to be detected. Common Subexpression Elimination could be beneficial for some shaders. These are left to future work.

At some point we should subsume spirv-remap's module-level dead type and dead function elimination into spirv-opt.

Acknowledgements

Thanks to Dan Ginsburg and Valve for their support and assistance. Thanks also to David Neto and his team at Google for the spirv-opt infrastructure which gave this effort a good head start, and for their thorough and detailed reviews of this code. Thanks also to Dan Baker at Oxide Games for assistance with sample shaders.