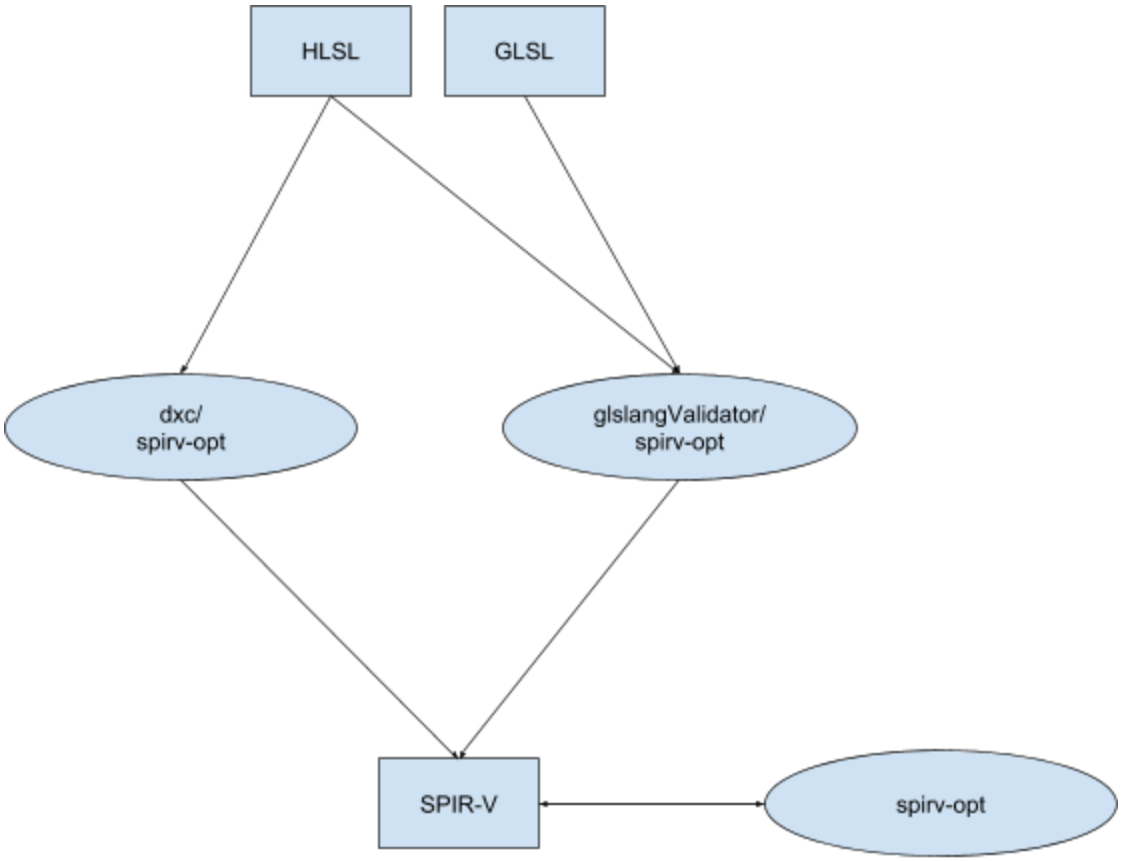
SPIR-V Legalization and Size Reduction with spirv-opt

Greg Fischer, LunarG Inc.
May 25, 2018

Summary

This paper discusses how spirv-opt can be used to “legalize” and reduce the size of SPIR-V shaders for the Vulkan graphics API . It also describes specific passes in spirv-opt and gives a recommended “recipe” for users who wish to customize the optimization of their shaders.



Proposed SPIR-V Workflow

Introduction

Since the inception of SPIR-V, there has been an interest in optimizing its code, particularly with the goal of reducing its size. Understandably, the raw SPIR-V emitted from the frontends glslangValidator and dxc is verbose, particularly with regard to function scope variables and their loads and stores. Significant opportunities for size reduction come from 1) eliminating these loads and stores, 2) eliminating dead code due to branches with constant conditionals and 3) eliminating redundant computations.

Additionally, as HLSL support was added to the Vulkan tool-chain, it was noted that several capabilities in HLSL, while expressible in SPIR-V, are not directly supported by Vulkan. However, the classic code optimization of propagation of values in these shaders can produce “legal” Vulkan shaders.

We have introduced passes to spirv-opt that use classic code optimization techniques to address these inefficiencies and “illegalities” in a SPIR-V module. These passes can reduce raw SPIR-V size and bring SPIR-V sizes within 10% of DX byte code. They can also sufficiently propagate values in SPIR-V shaders to meet Vulkan specifications.

Many of 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 regrettably may not derive much benefit from some of these passes 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. Complete support of kernels is left for future work and the status of this work is beyond the scope of this paper.

There are several other features which these passes do not support: please refer to the **Limitations** section of this document for more information. Running these passes on modules with unsupported features will cause the pass to return silently without changing the module.

Legalization

Several features in HLSL present capability which is not directly supported by modern graphics hardware. One such feature is structures with opaque members such as textures and samplers. Another such feature is local structured buffers.

DirectX users rely on fxc to legalize such code through modern optimization transformations such as value propagation. These users also rely on fxc to eliminate unused and unbound objects such as textures and samplers from shaders. This role is filled by spirv-opt in the Vulkan toolchain.

The dxc frontend by default will run spirv-opt propagation and dead code elimination passes sufficient to legalize all SPIR-V. However, glslangValidator must be purposefully built with spirv-opt in order for it to legalize by default. Otherwise, the following message will be emitted when “illegal” SPIR-V is generated:

WARNING: AST will form illegal SPIR-V; need to transform to legalize

The user may also perform legalization directly with spirv-opt using the --legalize-hlsl option or with RegisterLegalizationPasses() through the API. Default legalization passes can be disabled in the front ends using -Od.

SPIR-V Size Reduction using -Os

For those developers who wish derive the benefits of SPIR-V size reduction without specifying individual passes, the -Os option is available in dxc and glslangValidator as well as spirv-opt. This option invokes a recommended “recipe” of passes for size reduction. One advantage of this option is that developers will always get the “latest and greatest” passes and recipe. Another advantage is that developers will not need to continually update their build scripts as new passes and recipes become available. Those developers that wish to have finer control over which passes are invoked are invited to continue reading.

A Recipe

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

--merge-return
--inline-entry-points-exhaustive
--eliminate-dead-functions
--scalar-replacement
--eliminate-local-single-block
--eliminate-local-single-store
--simplify-instructions
--vector-dce
--eliminate-dead-inserts
--eliminate-dead-code-aggressive
--eliminate-dead-branches
--merge-blocks
--eliminate-local-multi-store
--simplify-instructions
--vector-dce
--eliminate-dead-inserts
--redundancy-elimination
--eliminate-dead-code-aggressive
--strip-debug

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.

Pass Descriptions

We will now discuss each pass individually to help users make adjustments to the recipe above.

Note that each pass is applied to all entry point functions and recursively to all functions called by the entry point functions after the pass has been applied to them until all functions in all the entry point function call trees have been processed.

Early Return Elimination (--merge-return)

Early returns in functions are a very handy tool for programmers which reduce code size and improve readability. However, they can cause difficulty during inlining as they do not map nicely into structured control flow in the calling function. In some cases they can actually prohibit the inlining of a function.

This pass transforms the control flow of a function so that early returns are no longer needed. Running this pass improves the ability of inliner to inline that function at the cost of a small increase in the size of the function.

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

Shaders often are broken into separate subroutines to increase modularity. But these subroutine calls create a barrier to data flow 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.

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.

Dead Function Elimination (--eliminate-dead-functions)

This pass eliminates functions in a module which are no longer part of the call tree of any entry point in the module. These are often created by function inlining but can also be created by dead code elimination. Calling this function reduces the size of the module. This does not reduce optimization time as most passes only optimize functions in an entry point call tree.

Scalar Replacement of Aggregates (--scalar-replacement)

Shaders often have code that packs data into and unpacks data out of structures, typically across function calls. After inlining, this copying code is a significant opportunity for store/load elimination.

This pass replaces such structures with individual variables for each of their members and replaces stores and loads of the structure members with stores and loads of the variables. This greatly eases the work of subsequent passes that eliminate stores and loads by allowing them to concentrate on stores and loads of variables only.

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 variables) 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, load/load and store/store 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 scalar replacement 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 that 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 scalar replacement is recommended beforehand. Function calls also may inhibit dataflow analysis and optimization, so inlining is also recommended beforehand.

Simplification (--simplify-instructions)

This pass replaces certain combinations of instructions with a simpler set. One class of combinations is replacing extract instructions with the actual value that is being extracted. For example, given the following instruction sequence, %26 can be replaced throughout the function with %21 and the extract can be deleted.

%22 = OpCompositeInsert %S_t %21 %20 1
...
%26 = OpCompositeExtract %v4float %24 1

Another class is arithmetic simplification and constant folding. For example, given the following instruction sequence, %31 can be replaced with %30 throughout the function and the FAdd can be deleted.

```
%5  = OpConstant %float 0
...
%31 = OpFAdd %float %30, %5
```

Such simplification reduces code size and aids in subsequent analysis.

Vector Dead Code Elimination (--vector-dce)

This code recognizes when a value inserted into a vector is never used and removes the insertion. Besides elimination unused OpCompositeInsert, it recognizes when values in an OpVectorShuffle or an vector typed OpCompositeConstruct is unused and replaces those values with an OpUndef. This pass should be followed with aggressive dead code elimination to eliminate the unused values.

Dead Insert Elimination (--eliminate-dead-inserts)

This code recognizes when a value inserted into a vector or structure is never used and removes the insertion. Vector DCE is more efficient but does not optimize structure insertions. This pass should be followed with aggressive dead code elimination to eliminate the unused values.

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

This pass, also known as 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 its 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 slightly more expensive style of dead code elimination is able to detect and remove cycles of dead code which look live to simpler DCE algorithms. A common example of such a cycle is a summation inside of a loop whose resulting value is no longer live.

This algorithm also detects entire dead structured control flow sequences (ie if-then-else, loop and switch constructs which do not generate live values) and eliminates them as well.

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

It is possible for a shader to contain significant sections of code that are never executed because they are control dependent on a conditional branch whose value is always false. After inlining, store/load elimination and simplification, 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 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 that 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 that 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 scalar replacement.

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, and a phi operation is generated, which merges the multiple values into one value and that value is used for the variable from that point until it is assigned again 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.

Common Subexpression Elimination (--redundancy-elimination)

This pass eliminates instructions whose opcode and operand are identical to an instruction that dominates it and replace all uses of its result id with the dominating instruction's result id. This pass reduces shader size. It is most effective when run after value propagation passes which reveals common subexpressions.

Debug Instruction Elimination (--strip-debug)

This pass eliminates all debug instructions, for example, OpName, which do not have any bearing on the output of the shader module. This pass reduces shader size and can generally be run at any time.

Limitations

Besides kernels and physical addressing, there are a few other features that 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 several passes do not support modules that 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.

Testing and Issues

While reasonable effort is made to test this code, bugs are possible and the quality of this code is dependent on users trying it and reporting issues. Issues and feature requests can be reported here.

Acknowledgements

Thanks to Dan Ginsburg and Valve for their support and assistance. Thanks also to David Neto, Steven Perron, Diego Novillo, Alan Baker, Lei Zhang and the rest of the team at Google for the spirv-opt infrastructure and their contributions to and reviews of this work. Thanks also to Dan Baker at Oxide Games for assistance with sample shaders.