03-14-2024

White Paper: Cross-stage Shader Optimization

Greg Fischer, Senior Engineer
LunarG Inc.

This white paper describes functionality in spriv-tools and glslang which allows the user to do dead code elimination in SPIR-V shaders across stages. Specifically, given two SPIR-V shaders A and B where B directly consumes the output of A, it will allow the user to eliminate output code in A that is not consumed by B.

This functionality also can be used to trim trailing dead components from aggregate input and output variables in the shaders.

Many drivers do these cross-stage optimizations automatically under the covers, but some drivers may not. These optimizations can have a significant impact on a shader's performance and size.

SPIRV-Tools Passes

This capability comprises several SPIRV-Tools passes.

First is a pass in spirv-tools which allows the user to eliminate trailing input variable components which are not used in the shader. This pass is called EliminateDeadInputComponents (EDIC). It was originally designed just for vertex shaders, but was later adapted to be applied to other shaders as well.

Next are passes in spirv-tools which allow the user to remove stores to output variables from a shader when the variables are not used in the subsequent shader in the graphics pipeline. They are called the EliminateDeadOutputStores (EDOS) passes.

Finally is a pass in spirv-tools which allows the user to eliminate trailing output variable components which are not used in the shader. This pass is called EliminateDeadOutputComponents (EDOC). It is best called after EDOS.

These passes currently work on vert, tese, tesc, geom and frag shaders only.

The EDOS passes can currently only be called from the spirv-tools API because of their more complex interface. EDOC also can only be called from the API. A "safe" version of EDIC (only works for vertex shaders) can be called from the spirv-opt Command Line Interface, but the general version can only be called from the API.

Vulkan Shader Input and Output Interface Rules

Changing the variables in the interface of a shader must be done carefully. There are rules in Vulkan concerning the validity of shader interfaces.

In Vulkan, when two shaders A & B interface, it is valid for B to be missing an input variable which would correspond to an output variable in A. However, it is invalid for A to be missing an output variable that corresponds to an input variable in B.

Finally, for each output variable in A, the matching input variable must have the identical type. So if the trailing members of an input variable are removed, the same members must be removed from its matching output variable in the preceding shader.

Aggressive Dead Code Elimination (ADCE) and Input and Output Variables

Given these Vulkan interface rules, ADCE eliminates unread Input variables by default. However, it cannot remove unstored output variables by default.

For this reason, we also added a new option to the ADCE pass which allows removal of unstored output variables when the user knows it is safe to do so. It is safe to do so if ADCE has also been called on the following shader in the pipeline.

Workflow

Because of the rules in Vulkan concerning interfaces, a special sequence of operations must be followed when changing the interfaces to retain interface validity.

The following steps can be used if a user wishes to perform cross-stage dead code and interface elimination in SPIR-V shader A followed by shader B in the pipeline.

Both shaders should first be optimized using spirv-opt and a comprehensive size reduction recipe including exhaustive inlining, local store/load elimination and dead code elimination. This will remove any dead code in B, especially dead uses of input variables and the dead input variables themselves. This also puts shader A in a canonical form for further analysis and optimization.

If EDIC has not been performed on shader B, now is a good time to do that. This needs to be done to ensure that the interfaces between A and B match, especially if EDOC is later applied to A.

There are two spirv-tools passes that together perform EDOS on a shader A given its following shader B. They will remove from A any stores to output variables not used in shader B.

The first is AnalyzeLiveInputPass(live_locs, live_builtins). It is called on the B shader. This creates two sets, live_locs and live_builtins, a set of input locations and a set of input builtin variables used by the B shader.

These live sets are then passed to EliminateDeadOutputStoresPass() called on the A shader. This eliminates stores to any output locations or builtins that are not in the live sets.

EDOC can then be effectively applied to the A shader.

ADCE (Aggressive Dead Code Elimination) should now be performed again on shader A. Not only will this remove dead code used by the removed dead output stores, but we can and should enable removal of dead output variables from the shader by ADCE at this time.

If A is a vertex shader, EDIC should now be applied to A.

If the user wishes to perform this optimization across three shaders A, B, and C, the steps above should be applied first to B & C, then A & B. A similar back-to-front sequence should be used for longer shader sets.

Additional Cleanup Workflow

ADCE can be done after EDIC to remove the old types that have been trimmed and replaced with new types.

spirv-tools API Functions

The spirv-tools passes named above can be registered and actuated using the following mapping to true spirv-tools API functions:

EDIC CreateEliminateDeadInputComponentsPass()
EDOS CreateAnalyzeLiveInputPass(live_locs, live_builtins)
CreateEliminateDeadOutputStoresPass(live_locs, live_builtins)
EDOC CreateEliminateDeadOutputComponentsPass()
ADCE CreateAggressiveDCEPass(false, true)

glslang API Functions

This functionality has been packaged and added to the glslang API as well to make this capability a little easier for glslang users. These glslang API functions are called SpirvToolsEliminateDeadInputComponents(), SpirvToolsAnalyzeDeadOutputStores(), and SpirvToolsEliminateDeadOutputStores(). Together they can be used to perform the workflow detailed above. SpirvToolsEliminateDeadOutputStores() performs EDOC and ADCE as well as EDOS.

Example

The following is an example of the workflow applied to the following two shaders:

```
// ========== t.vert ==========
#version 450

layout(location = 0) in   vec4 iva[2];
layout(location = 0) out vec4 ova[2];

void main()
{
   ova[0] = iva[0];
   ova[1] = iva[1];   // Not used by t.frag
}

// ========== t.frag ==========
#version 450

layout(location = 0) in   vec4 ifa[2];
layout(location = 0) out vec4 ofs;

void main()
{
   ofs = ifa[0];   // ifa[1] AKA ova[1] in t.vert is unused
}
```

It can be seen that the value assigned to ova[1] in t.vert, AKA ifa[1] in t.frag, is not used, which means the assignment to ova[1] in t.vert is dead. Ultimately, all the SPIR-V code associated with the dead statement in t.vert can be eliminated, which means that iva[1] in t.vert becomes unused. So iva and ova in t.vert, as well as ifa in t.frag can be reduced to arrays of length 1.

Following the workflow, EliminateDeadInputComponents (EDIC) is first applied to t.frag, reducing ifa to length 1 and giving us:

```
// ========== t.frag ==========
#version 450

layout(location = 0) in   vec4 ifa[1];
layout(location = 0) out vec4 ofs;
```

```
void main()
{
   ofs = ifa[0];   // ifa[1] AKA ova[1] in t.vert is unused
}
```

Next comes EliminateDeadOutputStores (EDOS). First, AnalyzeLiveInputPass is called on t.frag, giving live_locs = {0} and live_builtins = {}. Then EliminateDeadOutputStores (and AggressiveDeadCodeElimination) is called on t.vert, removing the store to ova[1].

```
// ========== t.vert ==========
#version 450

layout(location = 0) in   vec4 iva[2];
layout(location = 0) out vec4 ova[2];

void main()
{
   ova[0] = iva[0];
   // ova[1] = iva[1];
}
```

Next, EliminateDeadOutputComponents (EDOC) is applied to t.vert. Since ova[1] is no longer stored, ova can be reduced to length 1, yielding the following:

```
// ========== t.vert ==========
#version 450

layout(location = 0) in   vec4 iva[2];
layout(location = 0) out vec4 ova[1];

void main()
{
   ova[0] = iva[0];
}
```

Finally, EDIC can be applied to the vertex shader, t.vert. Since iva[1] is no long read, iva can be reduced to length 1, yielding the following:

```
// ========== t.vert ==========
#version 450

layout(location = 0) in   vec4 iva[1];
layout(location = 0) out vec4 ova[1];

void main()
{
   ova[0] = iva[0];
}
```

In summary, the final shaders are:

```
// ========== t.frag ==========
#version 450

layout(location = 0) in   vec4 ifa[1];
layout(location = 0) out vec4 ofs;

void main()
{
   ofs = ifa[0];   // ifa[1] AKA ova[1] in t.vert is unused
}

// ========== t.vert ==========
#version 450

layout(location = 0) in   vec4 iva[1];
layout(location = 0) out vec4 ova[1];

void main()
{
   ova[0] = iva[0];
   // ova[1] = iva[1];
}
```

This example was made minimalistic to easily see the changes and understand the workflow, but the elimination in typical shaders can be much more significant.