Implementing a linker for SPIR-V binaries

Pierre Moreau

13th October 2017

Some background on the project

Goal adding OpenCL support to Nouveau (open-source driver for NVIDIA cards, on Linux)

How using SPIR-V as a representation of OpenCL to be consumed by Nouveau

Time frame
- Started on consuming SPIR-V in July 2015;
- Started on the SPIR-V linker in January 2017.

Some background on the library: SPIRV-Tools

Available at `https://github.com/KhronosGroup/SPIRV-Tools`. Improved daily by employees from Google and LunarG, among others.
Tools available:

- Assembler, from textual to binary representation;
- Disassembler, from binary to textual;
- Validator (WIP);
- Optimiser (WIP);
- Linker (WIP);
- Some utilities.

Introducing SPIR-V

SPIR-V: An introduction

Some history

Intermediate Language from the Khronos Group, represented as a stream of 32-bit words, used as input for various APIs:

- **OpenCL** Kernels in OpenCL ≥ 2.1 should be created from SPIR-V modules.
- **OpenGL** Available through the extension ARB_gl_spirv, core since 4.6.
- **Vulkan** Shaders should be created from SPIR-V modules.

Possible to compile from OpenCL, GLSL, HLSL, among others, to SPIR-V.

SPIR-V instruction: `insn_descr operand1 operand2 ...`
where `insn_descr` contains the instruction opcode and the number of words it is made of.

SPIR-V: An introduction

Format, Part 1

The binary is made of a header, followed by all the instructions.
Those instructions are grouped in different categories:

1. Capabilities required by the binary;
2. Declaration of use of extensions;
3. Import of extended set of instructions;
4. Declaration of the memory model used;
5. Declaration of all entry points;
6. Declaration of all execution modes;
7. Debug information;
8. Annotations;
9. Declaration of types, constants and global variables;
10. Declaration of function prototypes;
11. Definition of functions.

SPIR-V: An introduction

Format, Part 2

```
; SPIR-V
; Version: 1.0
; Generator: Khronos LLVM/SPIR-V Translator; 14
; Bound: 10
; Schema: 0
                      OpCapability Addresses
                      OpCapability Kernel
               %1 = OpExtInstImport "OpenCL.std"
                      OpMemoryModel Physical64 OpenCL
                      OpEntryPoint Kernel %6 "test"
                      OpSource OpenCL_C 102000
                      OpName %res "res"
            %uint = OpTypeInt 32 0
         %uint_42 = OpConstant %uint 42
            %void = OpTypeVoid
%_ptr_CrossWorkgroup_uint = OpTypePointer CrossWorkgroup %uint
               %5 = OpTypeFunction %void %_ptr_CrossWorkgroup_uint
               %6 = OpFunction %void None %5
             %res = OpFunctionParameter %_ptr_CrossWorkgroup_uint
               %8 = OpLabel
                      OpStore %res %uint_42 Aligned 4
                      OpReturn
                      OpFunctionEnd
```

Figure: The SPIR-V of a simple OpenCL kernel which writes the value 42 at some location in global memory.

About that linker

Linking: Overview

The different linking steps are:

1. Convert all binaries to the same endianness (using the one from the host);
2. Generate the header of the resulting module;
3. Shift the IDs in each module;
4. Combine each category from the different modules;
5. Extract the import and export pairs, and check their signature;
6. Remove duplicate information;
7. Remove no longer needed linkage information;
8. Remove imported definitions and replace their usage by the exported declarations;
9. Compact the IDs.

Linking: Generate the new header

Header format:

- Magic word;
- SPIR-V version number;
- Generator's magic number;
- ID bound ($0 < id < bound$);
- Reserved (should be 0);

Merging issues:

Version Currently $max(\$version_1, \$version_2, \dots)$, but it could fail if deprecation is introduced in SPIR-V.

IDs Need to shift IDs of each module to prevent them from overlapping each other.

Linking: Matching imports and exports

How to identify imported and exported symbols?

- Use the LinkageAttribute annotation: gives the type of linkage, and a name;
- An import/export pair shares the same name.

There is a match if

- they have the same decorations;
- they have the same types/signature; types need to have the same decorations to match;
- their arguments have the same decorations (maybe?); only valid for functions.

Linking: Cleaning up

The following is done automatically:

- Remove duplicates:
  - Capabilities;
  - Import of extended set of instructions;
  - Annotations: an import and the corresponding export have the same ones;
  - Types: idem.
- Remove declaration of imported global variables and functions;
- Change code to use IDs of exported symbols, rather than imported ones;
- Compact IDs: the previous operation will leave some unused IDs;

You might want to manually run some optimisations on the resulting module, such as function inlining, dead code removal, etc.

To conclude

Conclusion: Current status

Done:

- Combine several SPIR-V modules together;
- (Implement a very basic annotation manager);
- (Remove various duplicates due to combining modules).

To do:

- (Remove duplicate extensions and names);
- (Remove dead instructions);
- Support OpenCL maths optimisation flags;
- Optimise the code (quadratic behaviours);
- Keep the DefUseManager up-to-date;
- Other code clean-ups.

Why is an annotation manager helpful?

Some facts

- Annotations can be applied to a single target (type, variable, function, annotation group).
- Annotation groups can be applied to multiple targets at once.

Groups make it complicated to interact with decorations: you could have T1 with decorations A, B and G1{C, D}, and T2 with A, G2{B}, C, D.

Conclusion: Current status

Done:

- Combine several SPIR-V modules together;
- (Implement a very basic annotation manager);
- (Remove various duplicates due to combining modules).

To do:

- (Remove duplicate extensions and names);
- (Remove dead instructions);
- Support OpenCL maths optimisation flags;
- Optimise the code (quadratic behaviours);
- Keep the DefUseManager up-to-date;
- Other code clean-ups.