LunarGLASS: An Open Modular Shader Compiler Architecture

John Kessenich   LunarG, Inc.

Introduction

This proposes a long-term compiler stack architecture, based on establishing common intermediate representations (IRs) allowing modularity between stack layers. Each source language front end would benefit from a common set of high- and mid-level optimizations, as would each back end, without the need to invent additional IRs. The short-term goal is to leverage investments in existing IRs while the long-term goal is to reduce the number of IRs and not require optimization difficulties caused by losing information going through an IR.

Although a variety of IRs could be used for this purpose, a major part of this proposal is to base the IRs on LLVM. It is commonly understood that LLVM has a shortcoming with respect to targets that benefit from executing multiple shader instances in the same instruction stream (SIMD with SoA or hybrid SoA forms). A key assumption in this proposal is that this short coming can and will be addressed, likely through a variety of techniques, and that the effort to do so is outweighed by the benefits of leveraging the large body of LLVM transforms.

Finally, new languages like OpenCL C, as well as ever more complex uses of GLSL, are raising the level of sophistication needed in compiler stacks. This calls for a solution more in line with handling the full-fledged language features and optimizations that LLVM is already prepared for.

Proposed Long-Term Architecture

Diagram 1 presents the basics of the proposed compiler stack, with components discussed below.

Diagram 1: Block Architecture

Source Language
GLSL
Other Source Language
Additional Front End
Front End
Additional Front End
Many FEs, Light on optimizations
Device A
Device B
Top LLVM IR (Source-vector form)
Standard Middle End
Alternate Middle End
State for Code-Gen
Target Policies
One ME, Broad optimizations
Device B
Device C
Bottom LLVM IR (Target-register form)
Back End
Back End
Back End
Many BEs, Target-specific optimizations
Device C
Target
Machine Code
Machine Code
Machine Code

Front Ends. There can be any number of independent front-ends translating a source language or custom IR to the Top LLVM IR. They don't necessarily need to contain optimizations, though they can. (Sometimes needed for semantic checking and optimizing early can be benificial, if algorithms are of low time complexity.) However, any optimizations that can be shared are better maintained in fewer places, in the middle of the stack.

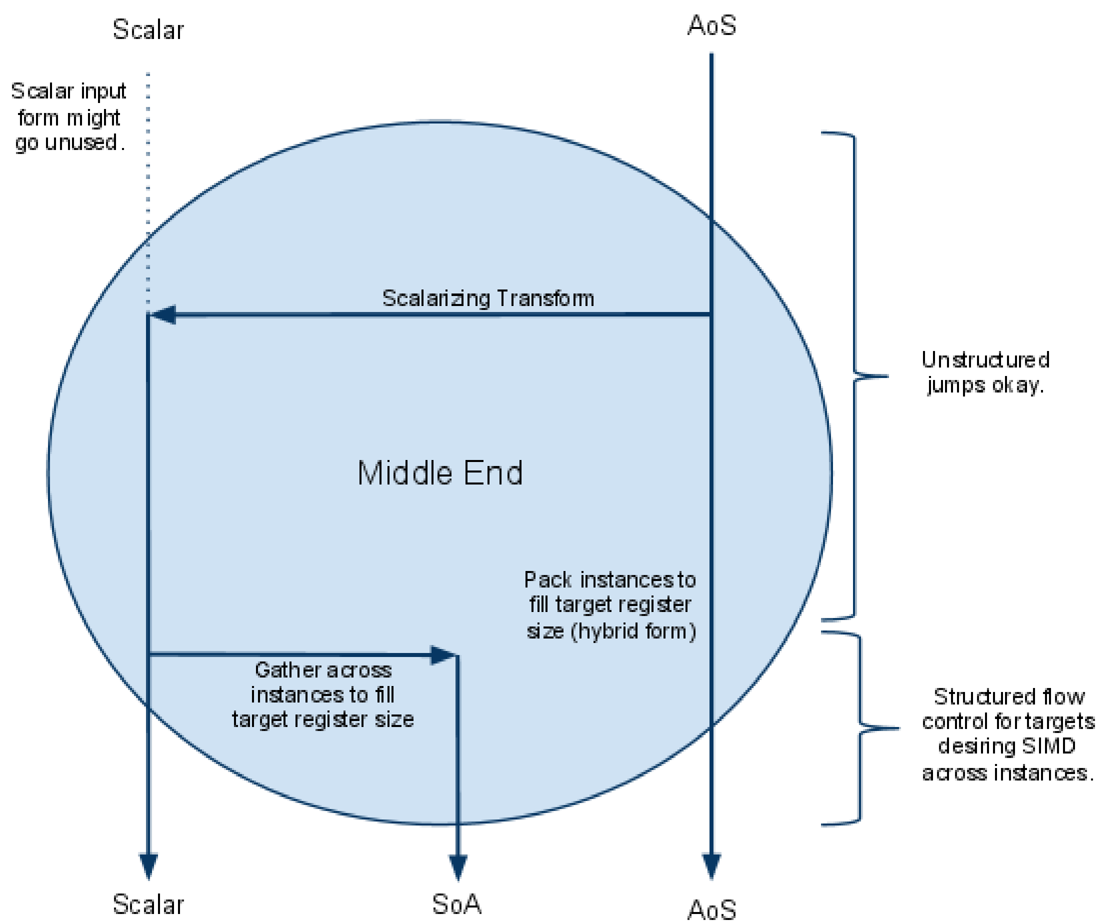
Top LLVM IR. The Top LLVM IR is in AoS form, as is the source code, with simple direct translation of source vectors to LLVM vectors, preserving the AoS form in the source. LLVM intrinsics would be added to directly represent built-in functions and graphics idioms.

Middle End. The middle end would hold the bulk of high-level and mid-level optimizations. Many optimizations that are specific to a back end can also take place here when only the policy is back-end specific, while the mechanism itself is still sharable code.

The standard middle end would contain a scalar-izer, useful for component optimizations and converting to non-AoS forms (see Conversion of Forms diagram below). This is the case if either the final target is scalar or the final target is SoA and early optimizations will be done on a scalar, implicitly parallel, representation. For SoA forms (or forms where the target architecture requires structure flow control), some combination of flow-control preserving transforms and re-transform back to structured flow-control will be needed.

Conversion between scalar, AoS, and SoA forms would be handled by common code as shown in Diagram 2:

Diagram 2: Conversion of Forms



An alternate middle end could be created and plugged in when someone wants to try something different than base LLVM with the above; either something simpler, more specialized, or more powerful, that might be called for in a particular situation. With the two IRs well-defined, this is a natural capability of this architecture.

Returning back to describing the components in Diagram 1:

Bottom LLVM IR. The bottom LLVM IR form uses scalar code for scalar targets, direct vector code for AoS targets, and transposed vector code for SoA targets. Or, hybrids of these. The defining point is that a vector register in the bottom LLVM IR form corresponds directly to a target architecture register. This reduces the amount of back-end specific code.

In addition to the top LLVM IR intrinsics, the bottom LLVM IR will include intrinsics for state-based code generation. This allows back ends to emit optimal target-specific code while still having the middle-end translate and optimize state-based code generation.

Back Ends. Any particular back end only has to deal with the scalar/SoA/AoS form it wants. For example, scalar form for a scalar CPU or hardware implicit parallelism, AoS for 4-wide SIMD, and SoA for general SIMD. The back end can focus on scheduling, register allocation, and other highly-specific target optimizations, given that the IR is already in the correct register form and has given the middle end its policies regarding function inlining, loop unrolling, and other sharable optimizations driven by target-specific policies.

A back-end could either choose to translate Bottom LLVM IR to its own IR, or use LLVM infrastructure to lower all the way to machine code.

How to Get There

There are certainly many ways to get from where we are today to the above architecture. Here is one possible sequence of steps to do so, within Mesa.

1. Define in detail the form and conventions needed to adapt LLVM IR to shader compilation. This is an open area of work, and could benefit from collaboration and study of previous uses of LLVM for shader compilation. It would include the vector conventions and intrinsics already described above, as well as how to handle different data types used to interface shaders to the pipeline.

2. Provide converters for proof of concept of the forms of LLVM IR defined in step 1.
   - a GLSL2 IR -> Top LLVM IR converter.
   - a top LLVM IR -> bottom LLVM IR (potentially a null converter for AoS targets).
   - a bottom LLVM IR -> TGSI IR or Mesa IR. (Using a simple stand-alone translator, not by

configuring LLVM’s back-end code generator.)

A working system would continue to work exactly the same way after plugging in this stack of converters, verifying correct expression in the Top and Bottom LLVM IRs.

3. The door is now opened for
- GLSL2 directly generating LLVM IR.
- Insertion of a middle end.
- Any back-end could directly consume Bottom LLVM IR and optionally use LLVM’s back-end code generator.
- Any new front-end directly generating Top LLVM IR can just plug in.

4. Expanding the middle end (or adding a new middle end), leveraging already written LLVM optimizations to do the bulk of optimizations. Then
- Front ends no longer have to do optimizations (a few will still make sense), increasing modularity. A new front end automatically leverages whatever middle end is plugged in.

Diagram 3: An example step migrating to this architecture with Mesa.

GLSL

GLSL2
emit top LLVM IR

Top LLVM IR (Source-vector form)

State for Code-Gen

transform to bottom LLVM IR, adopt LLVMPipe state-gen

Back end guided optimizations

Bottom LLVM IR (Target-register form)

IR Converter

TGSI

Pipe Driver

Machine Code

Tweaked LLVMPipe

x86/SSE