Shaderpulse: GLSL Frontend for MLIR SPIR-V **Open MLIR Meeting**

Ahmed Harmouche, 2024.10.17.



Parsing GLSL Overview of the parser

- Takes the token stream generated by the Lexer, and constructs an AST
 Tan. down percer: Requirely a Decent Percer
- Top-down parser: Recursive Decent Parser
- translation unit -> external declarations -> variable declarations, function prototypes, etc.
- The result of the parsing is a pointer to the root AST node, TranslationUnit

Parsing GLSL Traversing the AST

- Multiple ways for traversing
- 1. Take the root node, and explicitly traverse it "manually"
- 2. Use an ASTVisitor, and implicitly traverse it "automatically"
- AST constructs implement the ASTNode interface, thus they accept an ASTVisitor
- ASTVisitors can be used for various tasks, such as semantic analysis, ast printing, or code generation

Generating SPIR-V dialect Variable declarations

- MLIRCodeGen implements ASTVisitor interface
- Variable declarations:
 - Convert shaderpulse::Type to mlir::Type - If we are in global scope, generate spiry::GlobalVariableOp, if in a function
 - generate spirv::VariableOp
 - Apply initialiser expression.
 - Use IIvm: ScopedHashTableScope for the symbol table



Generating SPIR-V dialect Function declarations

- done visiting
- Collect the argument types
- Set insertion point to start of function, then visit the body of the function
- Insert the created function into a Ilvm::StringMap<spirv::FuncOp>



Create a new scope upon visiting a FunctionDeclaration, destroy scope when

Generating SPIR-V dialect Calling functions

- Look up the function in the function Map lacksquare
- If found, loop through the arguments, and visit them
- Generate a <spirv::FunctionCallOp> operation
- Push the function call result onto the expression stack
- Returning from a function:

 - function has result: pop expression stack, generate <spirv::ReturnValue> - function does not have a result: <<u>spirv</u>::ReturnOp>

Generating SPIR-V dialect Control Flow

- Generate spirv::LoopOp
- Structured in the following way: -> entry: jumps to header -> header: contains the BranchConditionalOp to determine where to loop: body, or merge

 - -> body: contains the code to be executed if the condition is satisfied -> merge block: header branches here when exiting the loop



Generating SPIR-V dialect Control Flow - break, continue

- Scenario: break in an if block
- Can't access the loop merge block directly from the selection block
- Break and continue are implemented using hidden control variables
- The respective control vars are set as break and continue statements are visited
- After the statement that contained break/continue, we insert a conditional branch, which determines to branch to a "post-gate block" (i.e. rest of the loop body), or to the merge/continue block

Generating SPIR-V dialect Selection

- To generate code for if-else statements we use spiry::SelectionOp
- Structured similarly to spirv::LoopOp: to branch: thenBlock (true part) or elseBlock (false part) -> thenBlock
 - -> elseBlock
 - -> merge block: both then and else block converge here



-> Selection header block: contains BranchConditionalOp to determine where

Generating SPIR-V dialect Handling expressions

- Many constructs in the AST contain an Expression
- We can resolve them in a recursive way
- Example: visiting a BinaryExpression
 - lhs->accept(visitor);
 - rhs->accept(visitor);
 - visitor->visit(this);
- encountering an operation that makes use of an expression, pop the expression stack to get the mlir::Value that the Operation uses.



Intermediary results of expressions are pushed to an expression stack. Upon

Generating SPIR-V dialect Expressions - handling types

- We check the operand types to determine which operation to pick
- isIntLike(), isFloatLike() expands to composites as well, i.e. an ivec3, uvec3 and int, uint are both "int like", this simplifies handling the operand types





Generating SPIR-V dialect Type conversions

- Type conversions are ConstructorExpressions
- Explicit conversion are supported
 - if (isUIntLike(fromType) && isFloatLike(toType)) { expressionStack.push_back(builder.create<spirv::ConvertUToF0p>(builder.getUnknownLoc(), toType, val)); } else if (isIntLike(fromType) && isFloatLike(toType)) { expressionStack.push_back(builder.create<spirv::ConvertSToF0p>(builder.getUnknownLoc(), toType, val)); } else if (isFloatLike(fromType) && isUIntLike(toType)) { expressionStack.push_back(builder.create<spirv::ConvertFToU0p>(builder.getUnknownLoc(), toType, val)); } else if (isFloatLike(fromType) && isIntLike(toType)) { expressionStack.push_back(builder.create<spirv::ConvertFToS0p>(builder.getUnknownLoc(), toType, val)); } else if ((isSIntLike(fromType) && isUIntLike(toType)) || (isUIntLike(fromType) && isSIntLike(toType))) { expressionStack.push_back(builder.create<spirv::Bitcast0p>(builder.getUnknownLoc(), toType, val));

Generating SPIR-V dialect Composites: arrays

Constructed using spirv::CompositeConstruct

float[3] myArr = float[3](1.0, 2.0, 3.0);

%cst_f32 = spirv.Constant 1.000000e-01 : f32 %cst_f32_0 = spirv.Constant 2.000000e-01 : f32 %cst_f32_1 = spirv.Constant 3.000000e-01 : f32 %0 = spirv.CompositeConstruct %cst_f32, %cst_f32_0, %cst_f32_1 : (f32, f32, f32) -> !spirv.array<3 x f32>

float[2][3] multiArr = float[2][3](arr1, arr2);

%17 = spirv.CompositeConstruct %15, %16 : (!spirv.array<3 x f32>, !spirv.array<3 x f32>) -> !spirv.array<2 x !spirv.array<3 x f32>>



Generating SPIR-V dialect Operations on arrays

- Indexing using spirv::AccessChainOp
- constant index

float elemFromMulti = multiArr[0][1];

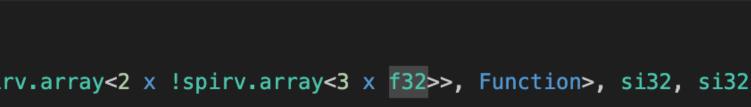
```
%cst0_si32_14 = spirv.Constant 0 : si32
%cst1_si32_15 = spirv.Constant 1 : si32
%22 = spirv.AccessChain %18[%cst0_si32_14, %cst1_si32_15] : !spirv.ptr<!spirv.array<2 x !spirv.array<3 x f32>>, Function>, si32, si32
```

variable index float someVar = myArr[varldx];



 \bullet

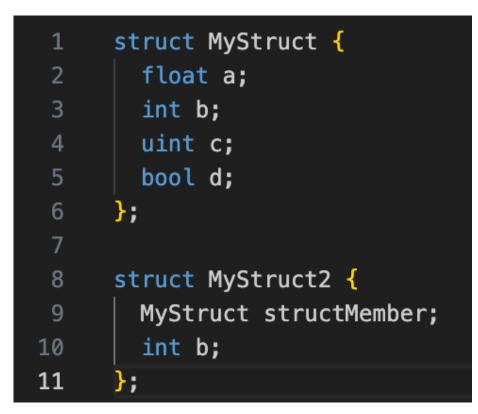




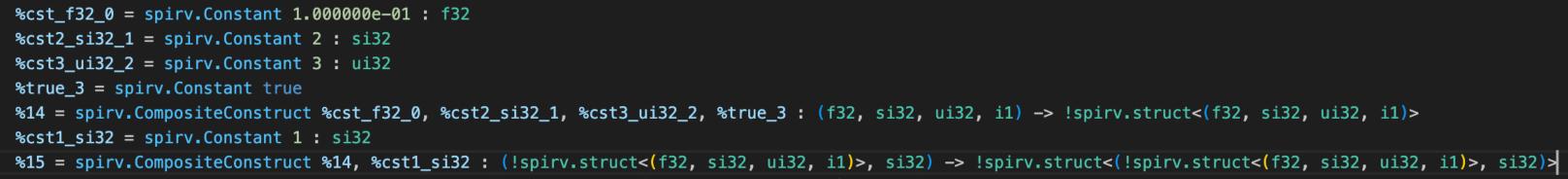
We can load the access chain to read from the array, and store into the access chain to write to the array

Generating SPIR-V dialect Composites: structs

Constructed using spirv::CompositeConstruct



MyStruct2 myStruct2 = MyStruct2(MyStruct(0.1, 2, 3u, true), 1);





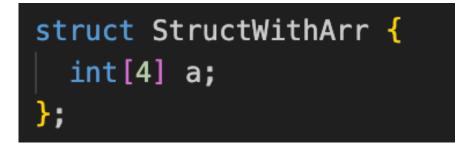
Generating SPIR-V dialect Operations on structs

Accessing a member using spirv::AccessChainOp

bool boolMember = myStruct2.structMember.d;

%cst0_i32_4 = spirv.Constant 0 : i32 %cst3_i32_5 = spirv.Constant 3 : i32 %17 = spirv.AccessChain %16[%cst0_i32_4, %cst3_i32_5] : !spirv.ptr<!spirv.struct<(!spirv.struct<(f32, si32, ui32, i1)>, si32)>, Function>, i32, i32 %18 = spirv.Load "Function" %17 : i1

Struct with an array member



int arrElemFromStruct = structWithArr.a[2];

```
%cst0_i32_8 = spirv.Constant 0 : i32
%cst2_si32_9 = spirv.Constant 2 : si32
%22 = spirv.AccessChain %21[%cst0_i32_8, %cst2_si32_9] : !spirv.ptr<!spirv.struct<(!spirv.array<4 x si32>)>, Function>, i32, si32
%23 = spirv.Load "Function" %22 : si32
```

Generating SPIR-V dialect Composites: Vectors

- Constructed using spirv::CompositeConstruct
- vec3 vec3 = vec3(1.0, 1.0, 1.0);•

%cst_f32_1 = spirv.Constant 1.000000e+00 : f32 %cst_f32_2 = spirv.Constant 1.000000e+00 : f32 %cst_f32_3 = spirv.Constant 1.000000e+00 : f32 %2 = spirv.CompositeConstruct %cst_f32_1, %cst_f32_2, %cst_f32_3 : (f32, f32, f32) -> vector<3xf32>

 $vec4 _vec4_1_2_1 = vec4(1.0, _vec2, 1.0);$ \bullet

%cst_f32_11 = spirv.Constant 1.000000e+00 : f32 %23 = spirv.Load "Function" %1 : vector<2xf32> %cst_f32_12 = spirv.Constant 1.000000e+00 : f32 %24 = spirv.CompositeConstruct %cst_f32_11, %23, %cst_f32_12 : (f32, vector<2xf32>, f32) -> vector<4xf32>







Generating SPIR-V dialect Operations on vectors: accessing an element

Implemented using spirv::CompositeExtractOp

vec3 myVec = vec3(1.0, 2.0, 3.0);float elemX = myVec.x;

%20 = spirv.Load "Function" %3 : vector<3xf32> %21 = spirv.CompositeExtract %20[0 : i32] : vector<3xf32>

float elemB = myVec.b;

%25 = spirv.Load "Function" %3 : vector<3xf32> %26 = spirv.CompositeExtract %25[2 : i32] : vector<3xf32>





Generating SPIR-V dialect Operations on vectors: swizzle

Implemented using spirv::VectorShuffleOp

vec3 myVec = vec3(1.0, 2.0, 3.0);vec3 reversed = myVec.zyx;

%11 = spirv.Load "Function" %6 : vector<3xf32> %12 = spirv.VectorShuffle [2 : i32, 1 : i32, 0 : i32] %11, %11 : vector<3xf32>, vector<3xf32> -> vector<3xf32>

• Swizzle can be chained:

float elem = reversed.zyx.xy.y $(3.0, 2.0, 1.0) \rightarrow (3.0, 2.0) \rightarrow 2.0$

%11 = spirv.Load "Function" %6 : vector<3xf32> **%12** = spirv.VectorShuffle [2 : i32, 1 : i32, 0 : i32] **%11**, **%11** : vector<3xf32>, vector<3xf32> -> vector<3xf32> %13 = spirv.VectorShuffle [0 : i32, 1 : i32] %12, %12 : vector<3xf32>, vector<3xf32> -> vector<2xf32> %14 = spirv.CompositeExtract %13[1 : i32] : vector<2xf32>

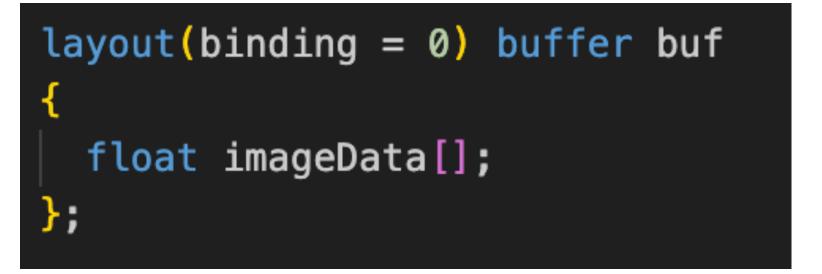
GLSL to SPIR-V dialect Compute shaders - built-ins

- Built-in compute vars are supported:
 - gl_GlobalInvocationID
 - gl_WorkGroupID
 - gl_WorkGroupSize
 - gl_LocalInvocationID
- built_in attribute is set on these variables
- mlir::spirv::StorageClass::Input is added



GLSL to SPIR-V dialect Compute shaders - SSBOs

To support them we need InterfaceBlocks



spirv.GlobalVariable @gl_GlobalInvocationID built_in("GlobalInvocationId") : !spirv.ptr<vector<3xui32>, Input> spirv.GlobalVariable @gl_WorkGroupID built_in("WorkgroupId") : !spirv.ptr<vector<3xui32>, Input> spirv.GlobalVariable @gl_WorkGroupSize built_in("WorkgroupSize") : !spirv.ptr<vector<3xui32>, Input> spirv.GlobalVariable @gl_LocalInvocationID built_in("LocalInvocationId") : !spirv.ptr<vector<3xui32>, Input> spirv.GlobalVariable @imageData {binding = 0 : i32} : !spirv.ptr<!spirv.rtarray<f32>, StorageBuffer>

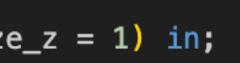


GLSL to SPIR-V dialect Compute shaders - execution mode

- spirv::ExecutionModeOp
- spirv::ExecutionMode::LocalSize is used to apply local size from:

layout (local_size_x = 32, local_size_y = 1, local_size_z = 1) in;

- execution_mode attribue
- values attribute



Let's try out shaderpulse



Thank you for your attention

	🖍 Unpin	Our Starred 9 →
អំ main 👻 អំ 1 Branch 🕟 0 Tags	Q Go to file t Add file -	- <> Code - About
wpmed92 Use simplified const creation in more	A GLSL compiler targeting SPIR-V mlir compiler cpp shaders llvm glsl	
.github/workflows	Conversion ops (#26)	2 months ago spirv mlir
compiler	Compile to spirv	2 weeks ago 🛱 Readme
example	Working mandelbrot example with Vulkan cpp sample code	
include	Fix buildIntConst param name	☆9 stars5 days ago○2 watching
🖿 lib	Use simplified const creation in more places	4 days ago 양 1 fork
→ Ilvm-project @ 76347ee	Update LLVM submodule (#32)	last month Releases
test	Implement do-while loops	last week No releases published
utils/include	Delete magic_enum util	last month
🗋 .clang-format	Apply clang format based on llvm style	last year Packages
🗋 .gitignore	Working mandelbrot example with Vulkan cpp sample code	2 weeks ago Publish your first package
🗋 .gitmodules	Integrate MLIR	last year
CMakeLists.txt	Compile to spirv	2 weeks ago
README.md	Replace example with a compute one	 C++ 91.0% GLSL 7.5% Other 1.5%