Introduction

• Who am I?
  • Long time LLVM contributor
  • HLSL team at Microsoft
• What am I talking about?
  • Ongoing effort to add HLSL support to Clang
• Where am I?
What is HLSL?

• High Level Shader Language was introduced with DirectX 9
• Initially supported vertex and pixel “shading”
• Started as a C-like language, but has evolved to be more C++-like
• Largely source compatible with other commonly used shader languages
• First graphics-focused language coming to Clang!
Language Features

• Supports enough C syntax to be familiar

• Has enough differences to be strange

• Implicitly parallel programming model

• Some C/C++ features just don’t make sense
Implicit Parallelism

• GPU hardware is not all the same
  • Wide SIMD
• Implicit parallelism enables source portability
• Vectors are vectors of vectors
Where is HLSL used?

- HLSL has a rich ecosystem
- Can target every major graphics API
- Used everywhere modern 3d games run
- DXC is shipped in the DirectX and Vulkan SDKs
Missing Features

• Key C/C++ features are missing

• No support for pointers or references

• User defined templates were not supported until 2021

• No C++ 11 anything…

• Organic language growth led to gaps in features
Compiler Performance Concerns

Loading...
Compiler Performance Concerns

• Shader compilers sometimes run at runtime

• Re-parsing standard library headers can be slow

• Re-loading or initializing full serialized ASTs can be slow too

• Lazy AST initialization is a _big_ win

HLSSL’s Library

- Pre-defined typedefs for common data types
- Built-in vector and matrix types
- Large collection of built-in functions
- Some complex data types
Balancing Priorities

- Scalability & Maintainability
- Compiler speed
- Robust Tooling
hlsl.h

- Header implementations are easy to write and test
- Re-parsing is slow
- Only typedefs and mapping functions to builtins
- Limited to older language features
Built-in vector Type

- HLSL Vectors behave like clang’s vector extension
- User-defined templates aren’t supported in older language modes
  - HLSL 2018 can’t parse this code

```cpp
namespace hls1 {
    template <typename element_type, int element_count>
    using vector = element_type __attribute__((__ext_vector_type__(element_count)));

    using float4 = vector<float, 4>;
} // namespace hls1
```
AST Initialization

- Define trivial types on AST initialization
- Types with no methods
- Types that are frequently used
- Makes the type available immediately
- Allows us to bypass parsing unsupported features

```c
void HLSLExternalSchema::defineHLSLVectorAlias() {
    ASTContext &AST = SemaPtr->getASTContext();
    llvm::SmallVector<NamedDecl> => TemplateParams;
    auto *TypeParam = TemplateTypeParmDecl::Create(
        AST, HLSLNamespace, SourceLocation(), 0, 0,
        AST.Idents.get("element");
        tok:TokenKind::identifier), false, false);
    TypeParam->setDefaultArgument(AST.getTrivialTypeSourceInfo(AST.FloatTy));
    TemplateParamsemplace_back(TypeParam);
    auto *SizeParam = NonTypeTemplateParmDecl::Create(,
        AST, HLSLNamespace, SourceLocation(), 0, 1,
        AST.Idents.get("element_count"),
        tok:TokenKind::identifier), AST.IntTy,
        false, AST.getTrivialTypeSourceInfo(AST.IntTy));
    Expr *LiteralExpr =
        IntegerLiteral::Create(AST, llvm::APInt(AST.getIntWidth(AST.IntTy), 4),
        AST.IntTy, SourceLocation());
    SizeParam->setDefaultArgument(LiteralExpr);
    TemplateParamsemplace_back(SizeParam);
    auto *ParamList =
        TemplateParameterList::Create(AST, SourceLocation(), SourceLocation(),
        TemplateParams, SourceLocation(), nullptr);
    IdentifierInfo *II = AST.Idents.get("vector");
    tok:TokenKind::identifier);
    QualType AliasType = AST.getDependentSizedExtVectorType(,
        AST.getTemplateTypeParmType0, 0, false, TypeParam),
    DeclRefExpr::Create(,
        AST, NestedNameSpecifierLoc(), SourceLocation(), SizeParam,
        false, DeclarationNameInfo(SourceName->getDeclName(), SourceLocation()),
        AST.IntTy, VK_LValue,
        SourceLocation());
    auto *Record = TypeAliasDecl::Create(AST, HLSLNamespace, SourceLocation(),
        SourceLocation(), II,
        AST.getTrivialTypeSourceInfo(AliasType));
    Record->setImplicit(true);
    auto *Template =
        TypeAliasTemplateDecl::Create(AST, HLSLNamespace, SourceLocation(),
        Record->getIdentifier(), ParamList, Record);
    Record->setDeclaredAliasTemplate(Template);
    Template->setImplicit(true);
    Template->setLexicalDeclContext(Record->getDeclContext());
    HLSLNamespace->addDecl(Template);
}
```
AST On Demand
clang::ExternalASTSource

- Forward declare types
- Populate definitions on use
- Solves exactly this problem!
External ASTs

- Basis for precompiled headers and modules
- Designed to enable lazy deserialization of bitcode ASTs
- Also used by lldb and Tooling APIs
BuiltinTypeDeclBuilder

- Forward declare on initialization
- Hook through ExternalSemaSource::CompleteType
- Called whenever the language requires completed types

```cpp
void HLSLEnternalSemaSource::forwardDeclareHLSLTypes() {
    CXXRecordDecl *Decl;
    Decl = BuiltinTypeDeclBuilder(*SemaPtr, HLSLNamespace, "RWBuffer")
        .addTemplateArgumentList()
        .addTypeParameter("element_type", SemaPtr->getASTContext().FloatTy)
        .finalizeTemplateArgs()
        .Record;
    if (!Decl->isCompleteDefinition())
        Completions.insert(
            std::make_pair(Decl->getCanonicalDecl(),
                std::bind(&HLSLEnternalSemaSource::completeBufferType,
                    this,
                    std::placeholders::_1)));}

void HLSLEnternalSemaSource::completeBufferType(CXXRecordDecl *Record) {
    BuiltinTypeDeclBuilder(Record)
        .addHandleMember()
        .addDefaultHandleConstructor(*SemaPtr, ResourceClass::UAV)
        .addArraySubscriptOperators()
        .annotateResourceClass(HLSLResourceAttr::UAV,
            HLSLResourceAttr::TypedBuffer)
        .completeDefinition();}
```
Everything in the AST

- Extending HLSL with internal attributes
- Complete ASTs for methods
- Minimize codegen changes
- Better tooling experience!
Internal Attributes

- Attributes have no spelling
- Never string-match type names
- Model special behaviors of built-in types
- Special code generation
- Initialization behavior

```llvm
!llvm.ident = !{!0}
!llvm.version = !{!1}
!llvm.valver = !{!2}
!llvm.shaderModel = !{!3}
!llvm.resources = !{!4}
!llvm.entryPoints = !{!12}

!0 = !{"dxcrast\(private\) 1.7.0.3682"}
!1 = ![i32 1, i32 0]
!2 = ![i32 1, i32 7]
!3 = !["cs", i32 6, i32 0]
!4 = ![i5, i8, null, null]
!5 = ![i6]
!6 = ![i32 0, "%class.Texture2D<float>"* undef, !"", i32 0, i32 0, i32 1, i32 2, i32 0, !7]
!7 = ![i32 0, i32 9]
!8 = ![i9, !11]
!9 = ![i32 0, "%class.RWBuffer<int>"* undef, !"", i32 0, i32 1, i32 1, i32 10, i1 false, i1 false, i1 false, !10]
!10 = ![i32 0, i32 4]
!11 = ![i32 1, "%class.RWBuffer<int>"* undef, !"", i32 0, i32 0, i32 1, i32 10, i1 false, i1 false, i1 false, !10]
!12 = ![void ()* @CSMain, "%CSMain", null, !4, !13]
!13 = ![i32 4, !14]
!14 = ![i32 8, i32 8, i32 1]```
Future Directions
Even more in the AST

• Moving IR-based analysis to AST & Clang CFG
• Augment with internal attributes
• Provide higher quality diagnostics
• earlier & more consistently

namespace hls

template<typename element_type>
[[hls::resource(UAV, TypedBuffer)]] struct RWBuffer {
    element_type *h;
    [[hls::uninitialized]] RWBuffer() {
        h = reinterpret_cast<element_type *>(__builtin_hls_create_handle());
    }
    [[hls::initializer]] RWBuffer(const RWBuffer &) = default;
    [[hls::initializer]] RWBuffer &operator=(const RWBuffer &) = default;

    element_type operator[](size_t Idx) { return h[Idx]; }
};

} // namespace hls
Are HLSL features valuable to C++?

- New attributes might enable expressing API constraints
- HLSL matrix syntax might be nice for C++
Balancing Priorities

- Scalability & Maintainability
  - Do as much as possible in HLSL
- Compiler speed
  - Lazy AST population
  - Works with PCH
- Robust Tooling
  - Complete ASTs
  - Source available
HLSSL Future Directions

• Working hard on HLSL Support in Clang

• Want to have clangd support in clang-16

• Public language design process
  • https://github.com/microsoft/hlsl-specs

• Actively working to make HLSL more like C++
More Resources

• Join the monthly HLSL Working Group meetings
• https://github.com/orgs/llvm/projects/4
• https://clang.llvm.org/docs/HLSL/HLSLDocs.html
• Find us on Discord, Discourse and IRC
Consolation Prize