DXC Update: HLSL to SPIR-V for Vulkan

Ehsan Nasiri, Google

SIGGRAPH, July 31, 2019

Overview

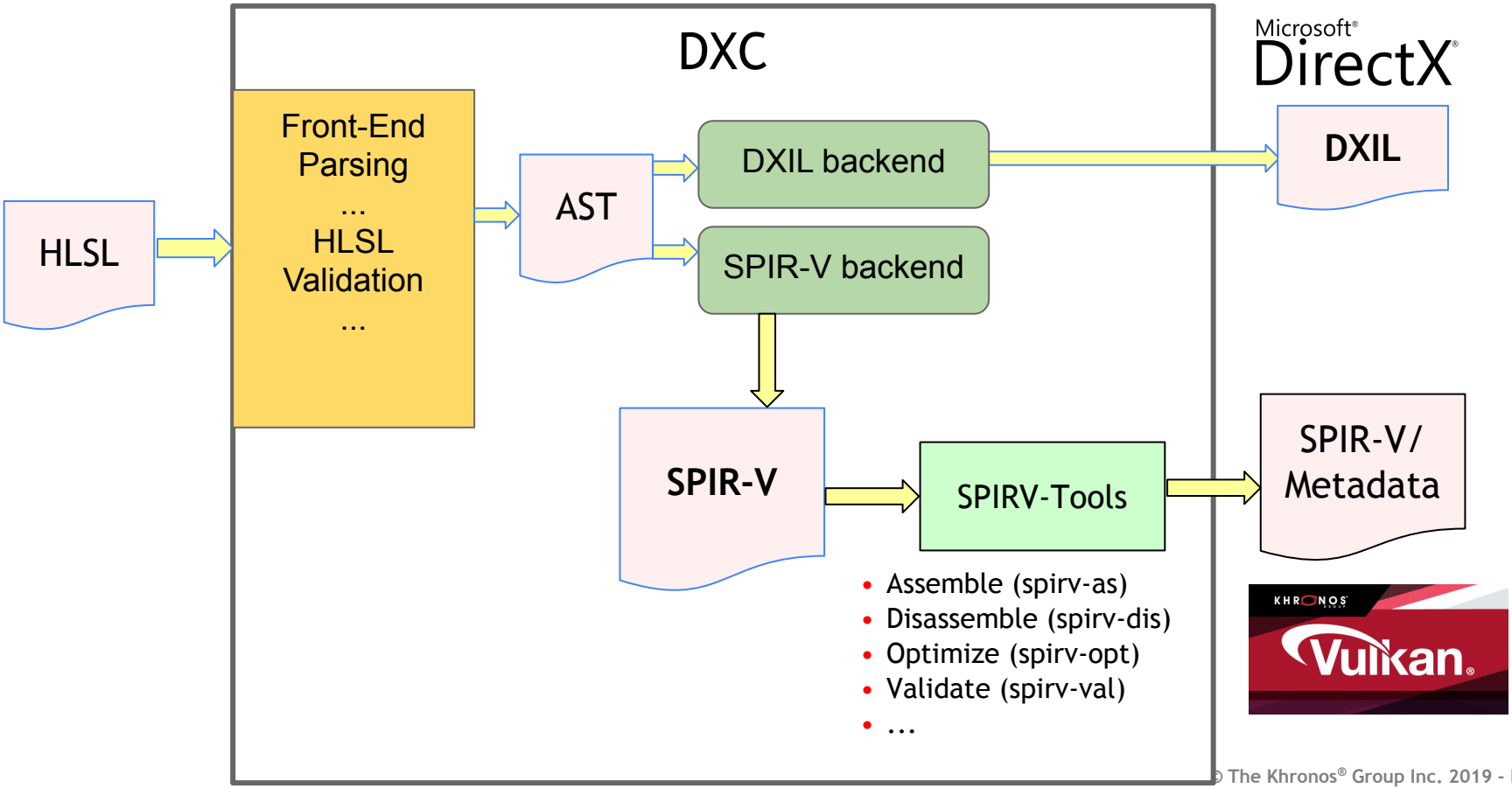
- DXC: Background
- Feature Coverage
- Downloads and Docs
- What's new

DXC: Background

- **DirectXShaderCompiler (DXC)**
  - Microsoft's next-gen HLSL compiler
  - Open sourced in January 2017
  - Based on LLVM/Clang

- **Spiregg: HLSL to SPIR-V compilation using DXC**
  - Google contributing SPIR-V CodeGen since April 2017
  - Share front-end parsing, HLSL validation
  - Recommended DXC for HLSL to SPIR-V compilation

DXC: Background



DXC: Goal

Make HLSL a first-class citizen for Vulkan

DXC: Current Status

- ✅ **Covers ~all native HLSL features**
  - C-ish features
    - Math types, Control flows, Functions, enums, etc.
  - C++-ish features
    - Resource types and methods, Namespaces, structs, etc.

- ✅ **Supports 16-bit and 32-bit types**

DXC: Current Status

- Supports Shader Model 6.2 and below
- Supports Vulkan 1.0 & 1.1

DXC: Current Status

- **Covers ~all Vulkan KHR/EXT extensions**
  - Up to SPV_EXT_descriptor_indexing
  - SPV_EXT_physical_storage_buffer not yet supported

- **Vendor extensions**
  - Mainly up to the vendor to contribute
  - Code reviews, testing, documentation required
  - E.g. Supports NV RTX for Vulkan (Contribution by NVIDIA)

DXC: Linux and macOS

- **Windows specific techniques**
  - Adapter code for non-Windows platforms (for COM, SAL, etc)

- **Master branch fully supported Linux and macOS now!**
  - Travis CI running for all commits and pull requests

Using DXC: Downloads and Docs

- **Pre-built binaries (Windows)**
  - Rolling release build from latest master branch:
  - http://khr.io/dxcappveyorbuild

- **User manual**
  - How HLSL and Vulkan language features are translated:
  - http://khr.io/hlsl2spirv

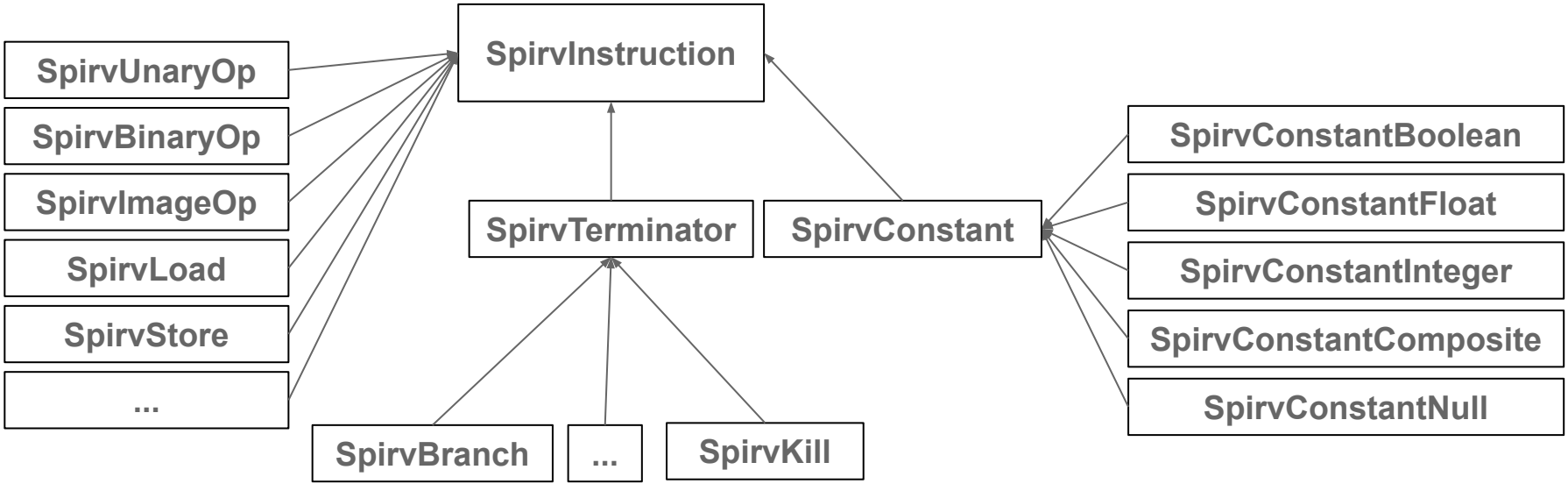
- **How to build**
  - Windows
  - Linux & macOS

What's New

In-Memory Representation of SPIR-V

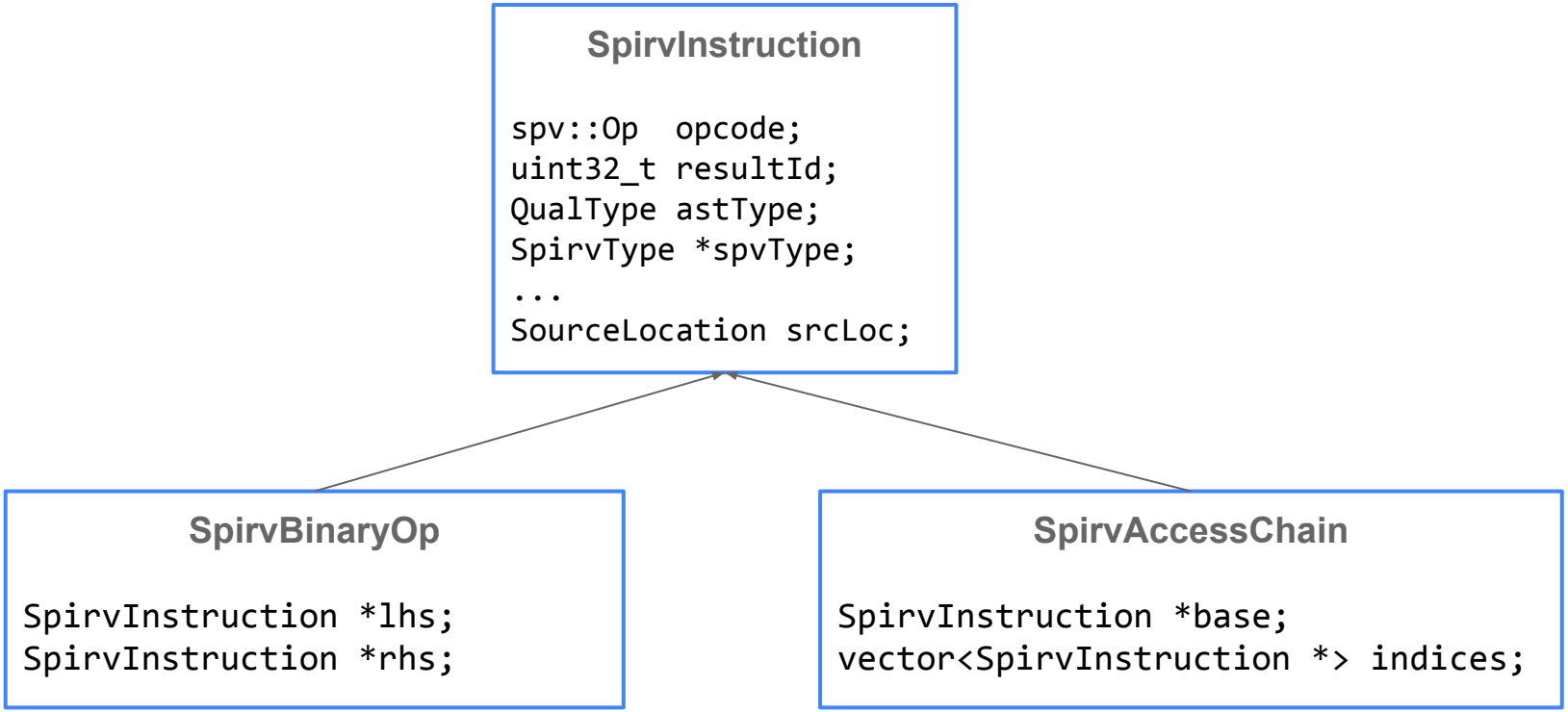
- **Faithful representation for**
  - Instructions
  - Basic blocks
  - Functions
  - Modules
  - etc.

IMR: SPIR-V Instructions



- Leverage polymorphism for module traversals
- LLVM-style RTTI: llvm::isa<SpirvConstant>(ptr)

IMR: SPIR-V Instructions



IMR: SPIR-V Basic Block

SpirvBasicBlock

```
uint32_t labelId;
string   labelName;
vector<SpirvBasicBlock *, 2> successors;
SpirvBasicBlock *mergeTarget;
SpirvBasicBlock *continueTarget;

// ...

vector<SpirvInstruction *> instructions;
```

IMR: SPIR-V Function

SpirvFunction

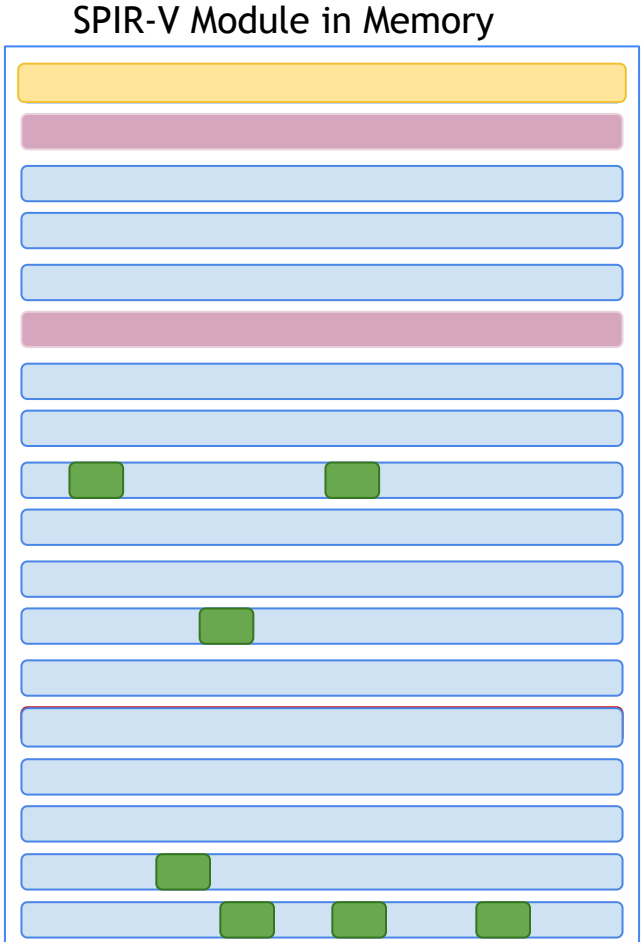
```
uint32_t functionId;
SpirvType *returnType;
SpirvType *functionType;
string functionName;
vector<SpirvFunctionParameter *, 8> parameters;
vector<SpirvVariable *> variables;

// ...

vector<SpirvBasicBlock *> basicBlocks;
```

IMR: SPIR-V Module

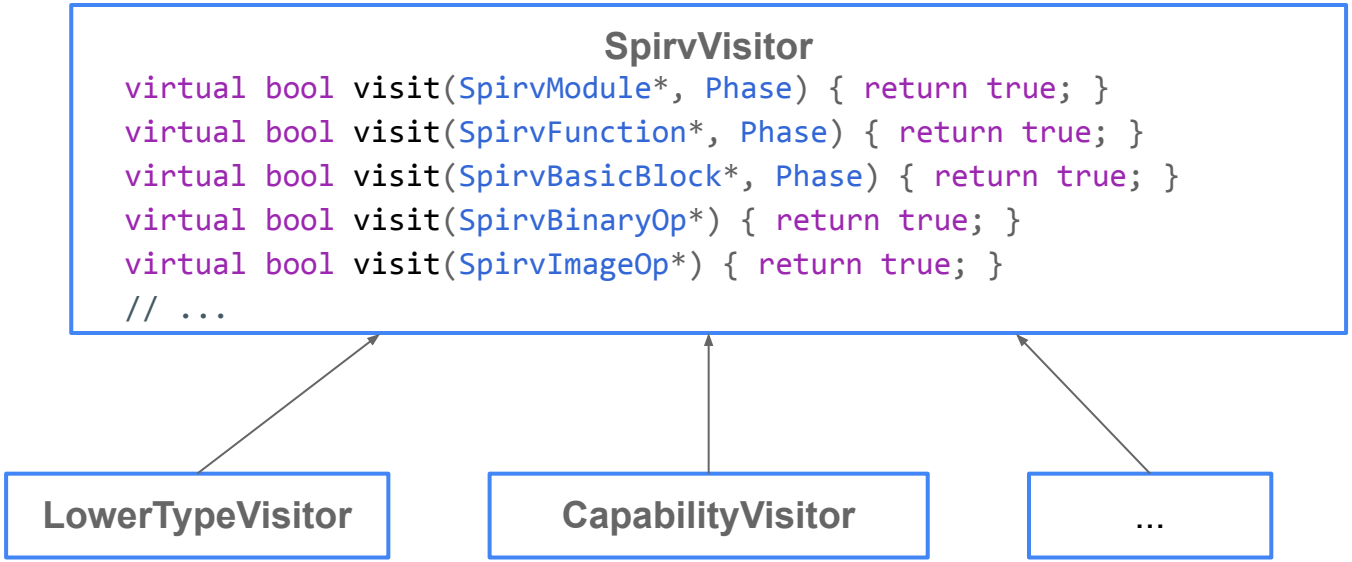
SpirvModule

```
vector<SpirvCapability*> capabilities;
vector<SpirvExtension*>  extensions;
vector<SpirvExtInstImport *> extInstSets;
SpirvMemoryModel *memoryModel;
vector<SpirvEntryPoint *> entryPoints;
vector<SpirvExecutionMode *> executionModes;
vector<SpirvModuleProcessed *> moduleProcesses;
vector<SpirvDecoration *> decorations;
vector<SpirvConstant *> constants;
vector<SpirvVariable *> variables;

// ...

vector<SpirvFunction *> functions;
```

IMR Traversals



IMR Traversals



IMR Traversals

CapabilityVisitor

```
class CapabilityVisitor : public SpirvVisitor {
public:
bool visit(SpirvImageOp *) {
  // Add capabilities related to SPIR-V Image instructions...
}

bool visit(SpirvEntryPoint *instr) {
  // Add Shader/Geometry/Tessellation capability based on the
execution mode
}

// Visitor for other instructions...

};
```

```
CapabilityVisitor visitor;
module->invokeVisitor(&visitor);
```

SpirvModule

```
bool invokeVisitor(const SpirvVisitor* visitor) {
//...
for (auto fn : functions)
  if (!fn->invokeVisitor(visitor, reverseOrder))
    return false;
//...
}
```

SpirvFunction

```
bool invokeVisitor(const SpirvVisitor* visitor) {
for (auto *bb : orderedBlocks)
  if (!bb->invokeVisitor(visitor))
    return false;
}
```

SpirvBasicBlock

```
bool invokeVisitor(const SpirvVisitor* visitor) {
for (auto *inst : instructions)
  if (!inst->invokeVisitor(visitor))
    return false;
}
```

SpirvInstruction

```
bool invokeVisitor(const SpirvVisitor* visitor) {
 return visitor->visit(this);
}
```

Support for min-types

- **HLSL** `min16int`, `min16float`, **etc.**
- `RelaxedPrecision` **is applied to the variable**
- **Must also be propagated forward to arithmetic ops using the variable**

```
OpDecorate %a RelaxedPrecision
OpDecorate %b RelaxedPrecision
; ...
%a = OpVariable %int32 Function
%b = OpVariable %int32 Function
%1 = OpIAdd %int32 %a %b
%2 = OpIAdd %int32 %1 %int_5
%3 = OpIAdd %int32 %2 %int_6
```

Support for HLSL `precise`

- Arithmetic operations that modify the value of a **`precise`** variable
- **`Propagate`** `NoContraction` **decoration**

```
float4 a, b, c, d;
float3 r = float3(a * b); // should be precise
float3 s = float3(c * d); // should be precise
precise float4 v;
v.xyz = r + s;            // should be precise
v.w = a.w * b.w + c.w;    // should be precise
```

Better Debug Information

- Starting with line number info (OpLine)
- Point to the HLSL source file and line number

main.hlsl

```
1  #include "new.hlsl"
2  [numthreads(4, 4, 4)]
3  void main(uint3 tid : SV_DispatchThreadID) {
4     foo(b);
5  }
```

new.hlsl

```
1  groupshared int b;
```

```
%main_hlsl = OpString "main.hlsl"
%new_hlsl = OpString "new.hlsl"
...
OpLine %new_hlsl 1 17
%b = OpVariable %_ptr_Workgroup_int Workgroup
...
OpLine %main_hlsl 4 4
OpFunctionCall %void %foo %b
```

Thank you!