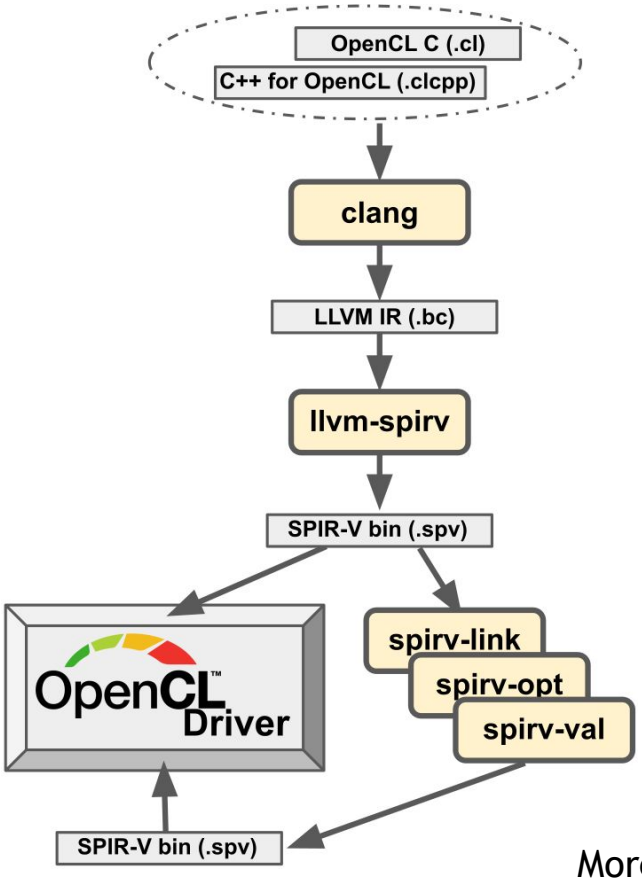
SPIR-V support in LLVM and Clang

2021 LLVM Developers' Meeting

SPIR-V in a nutshell

- **Intermediate representation for compute libraries very similar to LLVM IR**
- **Different “flavors” with specific validation rules**
  - SPIR-V for compute kernel
  - SPIR-V for shaders etc.
- **A SPIR-V module is always consumed under a specific environment**
  - Core specifications, defined by the Khronos SPIR-V group
  - Environment specifications, defined by the client API
    - Can apply more restrictions to core specifications
  - Extended instruction sets
    - Extra instructions a client API can define

Current state - example OpenCL kernel compilation



1. Use *clang* from https://github.com/llvm/llvm-project

   clang -target spir -c -emit-llvm -o test.bc test.cl

2. Use *llvm-spirv* from https://github.com/KhronosGroup/SPIRV-LLVM-Translator

   llvm-spirv test.bc -o test.spv

3. Optionally use *spirv-link*, *spirv-opt*, *spirv-val* from https://github.com/KhronosGroup/SPIRV-Tools

   spirv-link -o app.spv test.spv othertest.spv

More details at https://github.com/KhronosGroup/OpenCL-Guide

Q: What is the motivation for SPIR-V generation in LLVM?

- **Desirable interface for clang (taken from https://github.com/KhronosGroup/SPIRV-LLVM-Translator/wiki/SPIRV-Toolchain-for-Clang)**

clang -c test.cl -target spirv[32|64] -o test.spv

- **Stick to conventional 'three-layer' approaches as much as possible:**
  - Frontend -> opt -> codegen
    - "Frontend" being any tool producing LLVM IR
  - Dedicated 'spirv' triple
  - Separate 'spirv' target
    - Slowly improved/better tailored architecture
  - Away from 'spir' triple (spir-*-*)
    - Will likely remain for a while until all tools switch away from it (e.g. clspv)

Q: What issues do we see right now?

I. Confusing to application developers

- **What tools are needed and what is the flow?**
- **How to synchronize/align separate tools?**
- **Where to report bugs or ask for help?**

II. Hard to maintain and extend

- **Keeping in sync with new features**
- **LLVM stack after frontend doesn't know anything about SPIR-V execution or memory models**
  - No specific generic optimizations tailored to GPGPU (SIMT) functionality
  - Hard to fix issues or even report bugs for out-of-tree vendors
    - Example: convergent operations semantics is still WIP, absence of GPU memory model hierarchy

Consequences: upstream compilation flow is hardly usable for GPGPU and other accelerators!!!

Q: What are the potential users of SPIR-V in LLVM?

- **Clang based: OpenCL C, C++ for OpenCL, SYCL, OpenMP GPU offload, HIP**
- **Other:**
  - Rust, Julia, Halide?
- **‘spv’ dialect might be preferred for new languages that generate MLIR straight from the beginning**

Q: What are the approaches of integrating SPIR-V into LLVM?

- **Use SPIRV-LLVM translator as an external tool in Clang toolchain**
  - Mainly addressed issues for application developers
- **Use native SPIR-V Backend in LLVM**
  - https://github.com/KhronosGroup/LLVM-SPIRV-Backend
- **Use MLIR infra for SPIR-V generation**
  - Do both approaches fully address all issues?
  - Directly produce MLIR from source code (CIL to ‘spv’ dialect)
  - Gradual integration of MLIR ‘spv’ dialect into backend

Q: What is the proposed roadmap?

1. **Add SPIR-V support via translator as a quick working solution**
   a. Clang toolchain will likely need to be modified anyway to be tailored towards MLIR and non-LLVM based backends
   b. External tools are already used for CUDA and proved to be working
2. **Add SPIR-V backend into LLVM**
   a. Reuse some Clang functionality from Step 1
   b. Start integrating with MLIR (backend will reuse components from ‘spv’ dialect)
      i. Add compute OpenCL flavor to ‘spv’ dialect
3. **? Use CIL (Clang Intermediate Language) to directly generate SPIR-V**
   a. Reuse some Clang functionality from Step 1 and MLIR functionality from Step 2.b.i

Thank You!

(more) Questions?