

LLVM Developers' Meeting 2024

Advancing SPIR-V Backend Stability: Navigating GlobalSel Compromises

Vyacheslav Levytskyy

Michal Paszkowski

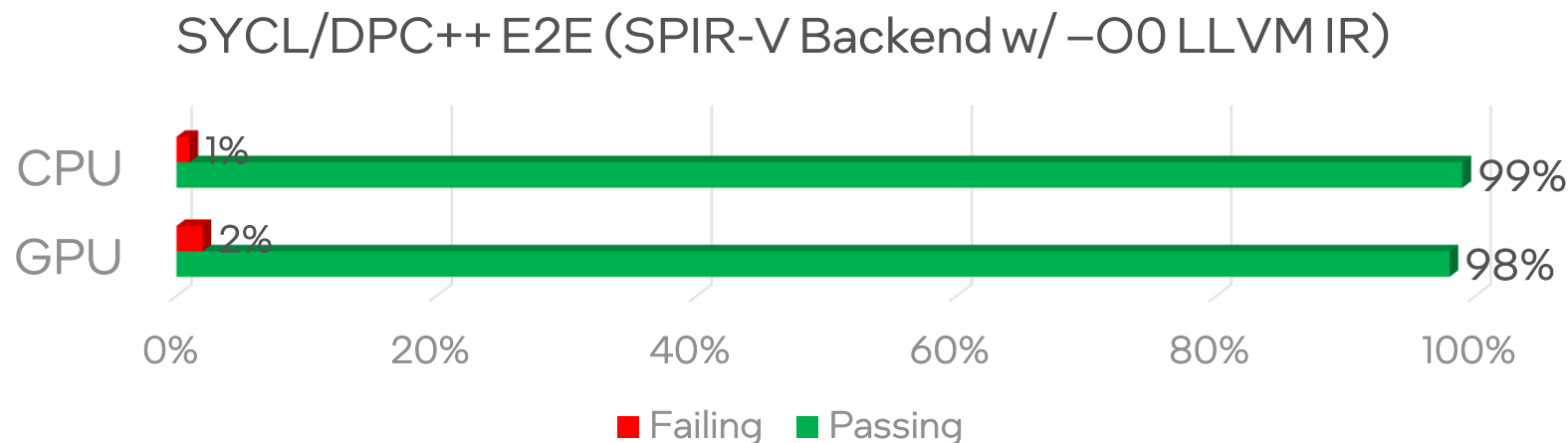


Agenda

- Brief recap of key concepts
- The key challenges in mapping LLVM IR to SPIR-V
- Reflecting on technical problems
- Correctness maintenance
- Future work

Immediate Takeaways

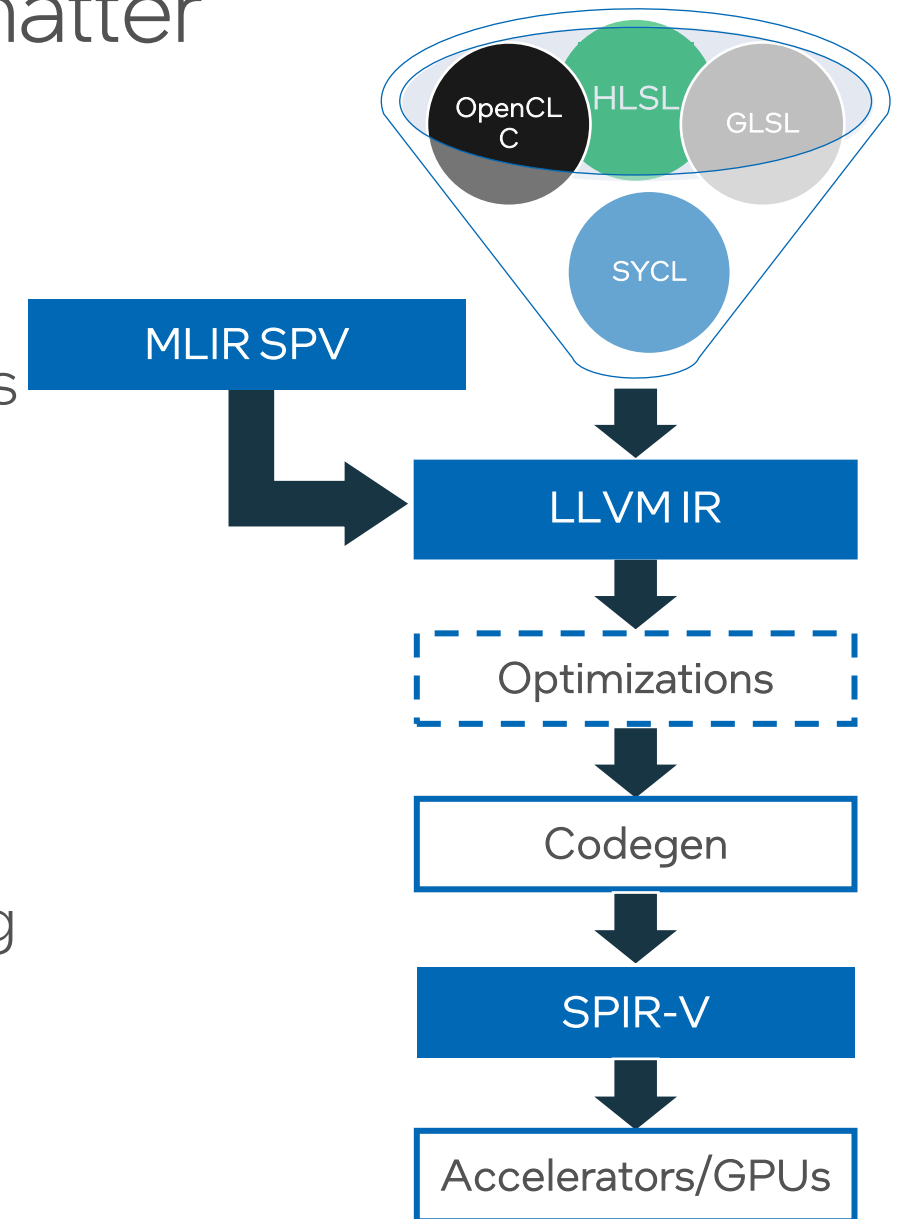
- SPIR-V backend got a serious uplift! ~500 LITs, OpenCL 3.0 conformant, nearly SYCL conformant (93-99% depending on optimization level), and massive improvements in code generation correctness and validity!



- Work in progress support for Vulkan and HLSL, 26 SPIR-V extensions implemented, and better compatibility with Khronos LLVM/SPIR-V Translator.
- Integration with external tools and libraries.

Why SPIR-V and SPIR-V backend matter

- SPIR-V is both an IR and portable binary format serving as a programming interface for heterogeneous accelerators.
- Rich ecosystem of high-level languages and APIs (e.g. OpenCL, SYCL, GLSL, HLSL)
- A SPIR-V module is always consumed under a specific environment. The core specification is defined by Khronos Group, vendors define client API environment which can apply more restrictions or provide spec extensions.
- This makes SPIR-V a great cross-vendor unifying IR!
- SPIR-V backend is developed by many Khronos member companies (Intel, Microsoft, Google, ...)



Challenges in Mapping LLVM IR to SPIR-V

SPIR-V is a semantically rich language!

- While semantically rich languages make it easy to lower to low-level representations, SPIR-V is nearly at the same (or sometimes higher) level than LLVM IR.
- SPIR-V concepts cannot be easily represented in Machine IR or translated following standard GlobalSel translation schema with the condition of meeting Machine Verifier requirements.

Challenges in Mapping LLVM IR to SPIR-V

- No real accelerator hardware
 - portable, hardware and vendor agnostic
 - the SPIR-V target (behavior/properties) and the SPIR-V representation (format) are matching products of the SPIR-V specification
- There will be a reverse translation from SPIR-V to an accelerator instruction set (FPGA, GPU, NPU), for example, via back encoding into LLVM IR
 - there will be a hardware-dependent optimization
 - no need to reason in terms of physical registers, allocation, scheduling, etc.
 - LLT types are not enough

How does it map to technical problems

- Types in general and definition scope
 - virtual registers and SPIRV identifiers
 - low-level types are very far from being able to express SPIRV types
- Before LLVM 17, IR pointer types contained an element type
 - SPIR-V pointers are always typed!
 - Schrödinger's TypedPointerType
- Control flow
 - structured control flow graph
 - a label is an instruction that starts a logical basic block
- Semantics of statements / instructions
 - subtle differences between LLVM IR and SPIRV; GISEL's MIR and SPIRV
 - e.g.: phi and bitcast

Type Inference

- Mismatch between LLVM notion of a valid virtual register and SPIR-V concept of <id>
 - GISEL virtual registers do not perfectly match SPIRV notion of identifier
 - GISEL does not keep track of how newly created virtual registers are related to original LLVM IR values
 - SPIR-V backend needs to do this however, to conform with the requirement to track types in general (not LLT types)
 - SPIR-V backend needs to calculate (infer) types

Type Inference

- Sometimes it is possible to infer types in a Module pass
 - look for known patterns in Instruction values
 - result element type of GEP, allocated type of alloca, pointer op of addrspacecast, ...
 - set result by an argument in a well-known function, by incoming values of phi, ...
 - deduce nested types of composites
 - deduce pointer operand types or insert a bitcast for known patterns
 - pointer ops of LoadInst, StoreInst, AtomicRMWInst, AtomicCmpXchgInst
 - align value types for ReturnInst, incoming values of phi, ICmplInst
 - use well-known builtins information in CallInst: OpGroupAsyncCopy, OpAtomic*
 - function parameters: analyze function's call sites
 - keep record of unknown types and re-visit them after all functions in the module are processed

Aggregates Lowering Mechanism

- GlobalSel facilitates code lowering when we are thinking in terms of instruction selection and real hardware
 - we don't expect a physical register to be able to store an arbitrary aggregate
- SPIR-V doesn't represent any actual instruction set architecture
 - we expect to see aggregate types and constants
 - explicitly preserved
 - listed in the module scope
 - not disintegrated by GlobalSel pipeline into low-level types and registers

Aggregates Lowering Mechanism

- Initial approach: remove aggregates from calls to avoid a crash
 - Mutate a function with aggregates in the signature
 - Replace aggregates with i32
 - The change in types is noted in metadata for later restoration
 - This helps survive IRTranslator
 - During call lowering the original function signature is restored
- Drawbacks
 - Impossible to deduce and store a correct type before IRTranslator
 - spirv-val reports invalid SPIR-V
 - A mismatch of object/ptr types in OpStore: incorrect aggregate types tracing
- Upgrade of the “remove aggregates” approach
 - Register the mutation and access original function type

Aggregates Lowering Mechanism

- Root cause
 - SPIR-V: use a single identifier (virtual register) per an aggregate value
 - IRTranslator: Aggregates get flattened
- Troubles with intrinsics after optimization
 - no place to rewrite function signature
 - translate call » not a known intrinsic » create v-regs » fail on multiple v-regs

```
%e = phi i32 [ %a, %entry ], [ %i, %body ]  
%i = add nsw i32 %e, 1  
%f1 = icmp eq i32 %i, 0  
br i1 %f1, label %exit, label %body
```

```
%e = phi i32 [ %a, %entry ], [ %math, %body ]  
%0 = call { i32, i1 } @llvm.uadd.with.overflow.i32(i32 %e, i32 1)  
%math = extractvalue { i32, i1 } %0, 0  
%ov = extractvalue { i32, i1 } %0, 1  
br i1 %ov, label %exit, label %body
```

- Problems with explicitly called intrinsics
 - call { i32, i1 } @llvm.uadd.with.overflow.i32(i32 %e, i32 1)
 - if rewrite return type: Machine Verifier fails due to wrong function signature, because it expects it to be {i32, i1} not i32
 - if do not rewrite return type: fail at IRTranslator on multiple v-regs

Aggregates Lowering Mechanism

- Options
 - SPIRV-specific unfolding/lowering; reinventing the wheel
 - Changes in GlobalSel logic would be too intrusive, chances are slim
 - Translating function calls inside GlobalSel
 - Rewrite to an internal `__spirv_ builtin` and try a custom call lowering
 - Improve existing approach and stop struggling with GlobalSel
- How to make friends with IRTranslator
 - Clearly communicate objectives
 - `void @llvm.fake.use(...)` to map virtual registers to the original value
 - `void @llvm.spv.value.md(metadata valAttrs)` to preserve name and data type
 - Reuse general logic of the shared code and lessen maintenance burden for target-independent changes

Control Flow

- Like LLVM IR: Modules of Functions, Functions of BBs, BBs of Instructions, and BB is terminated by a control flow instruction
- Unlike LLVM IR
 - GPU-specific compilers would like to see an information being structured
 - computational flavor figures this out by itself (OpenCL)
 - shaders require that their control flow is structured (Vulkan)
 - OpLoopMerge declare a structured loop
 - OpSelectionMerge declares a structured selection
 - In SPIR-V a label is an instruction that starts a logical basic block
 - It's not possible to delete instructions after the unconditional branch, because this instruction must be the last instruction in a block
 - There is no instruction to encode "if (Cond) then Stmt" logic
 - OpBranchConditional is a full if-then-else
- No optimization of branching like Branch Folding and If Conversion

Example 1: SPIR-V Control Flow Meets AsmPrinter

[#107013](#)

- AsmPrinter is hardcoded to always create a symbol for the end of a function if valid debug info is present
 - needFuncLabels() is a static function inside OR and returns true because
 - hasDebugInfo() is inside OR and returns true because
 - DbgInfoAvailable is a private class member and it's true because
 - valid debug info is present in the Module
- In SPIR-V each label is an instruction
 - a block always starts with an OpLabel instruction
- Solution

```
llvm/lib/CodeGen/AsmPrinter/AsmPrinter.cpp
- if (EmitFunctionSize || needFuncLabels(*MF, *this)) {
+ // SPIR-V supports label instructions only inside a block, not after the
+ // function body.
+ if (TT.getObjectFormat() != Triple::SPIRV &&
+     (EmitFunctionSize || needFuncLabels(*MF, *this))) {
```

Example 2: Machine Verifier, G_BITCAST and G_PHI

[#110270](#)

```
define void @foo(i1 %arg) {
entry:
    %r1 = tail call ptr @f1()
    %r2 = tail call ptr @f2()
    br i1 %arg, label %l1, label %l2
l1:
    br label %exit
l2:
    br label %exit
exit:
    %ret = phi ptr [ %r1, %l1 ], [ %r2, %l2 ]
    ret void
}
define ptr @f1() {
entry:
    %p = alloca i8
    store i8 8, ptr %p
    ret ptr %p
}
define ptr @f2() {
entry:
    %p = alloca i32
    store i32 32, ptr %p
    ret ptr %p
}
```

```
%r1 = OpFunctionCall %_ptr_Function_uchar %f1
%16 = OpFunctionCall %_ptr_Function_uint %f2
%r2 = OpBitcast %_ptr_Function_uchar %16
      OpBranchConditional %arg %24 %25
%24 = OpLabel
      OpBranch %26
%25 = OpLabel
      OpBranch %26
%26 = OpLabel
%ret = OpPhi %_ptr_Function_uchar %r1 %24 %r2 %25
...
%f1 = OpFunction %_ptr_Function_uchar None %8
%27 = OpLabel
    %p = OpVariable %_ptr_Function_uchar Function
      OpStore %p %uchar_8 Aligned 1
...
%f2 = OpFunction %_ptr_Function_uint None %10
%28 = OpLabel
%p_0 = OpVariable %_ptr_Function_uint Function
      OpStore %p_0 %uint_32 Aligned 4
...
```


Example 3: Building my own PHI

[#110019](#) and [#110507](#)

- OpPhi in SPIR-V is a PHI node
 - starts BB, pairs of incoming value and labels, etc.
- A subtle difference
 - OpPhi: exactly one entry for each parent block of the current block in the CFG
 - phi: one entry for each instance of the predecessor

```
%r = phi i32 [0, %l1], [1, %l2], [1, %l2]
but one record per 2 original %l2 predecessors:
%r = OpPhi %uint %const_0 %l1 %const_1 %l2
```

- Machine Verifier does not recognize OpPhi as PHI

```
bb.1.entry:
  successors: %bb.2, %bb.3
  OpBranchConditional %5:iid, %bb.2, %bb.3
bb.2.true_label:
; predecessors: %bb.1
  successors: %bb.4(0x80000000); %bb.4(100.00%)
  %12:iid = OpFunctionCall %2:type @foo
  OpBranch %bb.4
bb.3.false_label:
; predecessors: %bb.1
  successors: %bb.4(0x80000000); %bb.4(100.00%)
  %8:iid = OpFunctionCall %2:type @bar
  [...]
  OpBranch %bb.4
bb.4.merge_label:
; predecessors: %bb.3, %bb.2
  %15:id = OpPhi %2:type, %12:iid, %bb.2, %8:iid,
  %bb.3

*** Bad machine code: Virtual register defs don't
dominate all uses. ***
- v. register: %8
- v. register: %12
```

Discussion of Examples

- AsmPrinter: labels against OpLabel
 - Discussed and change merged, adding a hardcoded condition to the hardcoded if
- Machine Verifier's Procrustean bed of G_BITCAST & G_PHI
 - Suggested to generate OpBitcast immediately, without re-using G_BITCAST
 - OpBitcast is not a no-op bitcast
- Machine Verifier does not recognize OpPhi as a PHI
 - Suggested to re-use a general opcode to denote SPIR-V phi-node all along the way until a late final encoding when it may get converted to OpPhi
- SPIR-V backend's approach...
 - OpBitcast and OpPhi are generated where they are supposed to be, during instruction selection, and conform to the principle of reusing shared code inside GlobalSel
- Walls vs. Bridges
 - A time to guard the library and a time to refrain from overprotecting
 - Be more open to non-invasive cases? Allow to override default behavior?
 - The higher-level backend has also less trivial needs

Opaque Pointers: Challenges

- The transition from typed to opaque pointers in LLVM 17 simplified some aspects of LLVM IR but presented challenges for SPIR-V backend which requires pointer types for code generation.
- Before, pointer element types were used not only for emitting required type declarations, but also lowering nested types (structs or arrays), creating function declarations, resolving OpenCL builtin calls, creating pointer casts, and emitting SPIR-V/OpenCL builtin types.
- Now, SPIR-V backend needs to *correctly* deduce types early in the pipeline.

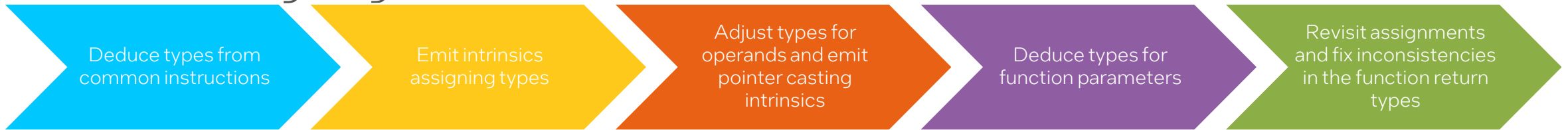
```
define void @foo(ptr %a) {  
    %gep = getelementptr inbounds i32, ptr %a, i64 0  
    ret void  
}  
  
define void @bar(ptr %b) {  
    call spir_func void @foo(ptr %b)  
    ret void  
}
```

```
%s = type { i32 }  
%w = type { [7 x %s] }  
  
define void @foo(ptr noundef byval(%w) align 4 %a) {  
    %val = load i32, ptr %a  
    ret void  
}
```

Opaque Pointers: Solution

Three passes work together to assign (pointer) types: 1. *SPIRVEmitIntrinsics*, 2. *SPIRVCallLowering*, and 3. *SPIRVPreLegalizer*

First, *SPIRVEmitIntrinsics* tries to infer and assign the relevant pointer types in LLVM IR using target intrinsics:



- This is a non-linear, nested, and often recursive process.
- Calls complicate type inference (e.g. function pointers and indirect calls must be resolved solely through uses, some builtin functions have well-known/set return or argument types and are hard-coded)
- PHI nodes can have incoming values with different pointer element types (if types differ, the most frequently occurring one is assigned).

The pass assigns all SSA values a type with *assign_ptr_type* (or *assign_ptr_type* in case of SPIR-V/OpenCL types) or replaces their uses with a *ptrcast* intrinsic.

Opaque Pointers: Solution

Second, *SPIRVCallLowering* creates function declarations and lowers formal arguments. SPIR-V function and parameter declarations also contain types:

```
%foo = OpFunction %float None %1
%bar = OpFunctionParameter %ptr_float
...
%baz = OpLoad %float %bar
```

If a call argument is of a pointer type in LLVM IR, the lowering takes a type from (in the order of precedence): *byval/byref* attributes, *assign_type* call (hence it must be a SPIR-V builtin type), or *assign_ptr_type* intrinsic call.

OpenCL/GLSL/... builtin function calls are lowered differently and have their types hard coded and/or parsed with TableGen definitions (*SPIRVBuiltins.td*):

```
defm : DemangledConvertBuiltin<"convert_char", OpenCL_std>;
...
class ConvertBuiltin<string name, InstructionSet set> {
  bit IsDestinationSigned = !eq(!find(name, "convert_u"), -1);
  ...
```

Itanium mangling does not encode return type information!

Opaque Pointers: Solution

Third, *SPRVPreLegalizer* removes all *assign_ptr_type/assign_type* intrinsic calls and assigns each MIR register relevant types. The mappings between registers and types are stored in *GlobalRegistry* to ensure each type declaration is printed once per module.

TargetExtType

SPIR-V has a family of types (such as *OpTypeImage*, *OpTypeEvent*...) which were previously represented as pointers-to-opaque-structs:

```
%opencl.event_t = type opaque  
define void @foo(i8 addrspace(1)* noundef %src) {  
  ...
```

In LLVM 16, a new *TargetExtType* was added for types that need to be preserved, but otherwise are not introspectable by target-independent optimizations (used by SPIR-V, DX, RISC-V, AArch64...):


```
define void @foo(target("spirv.Event") %src) {  
  ...
```

Unfortunately, this means that SPIR-V backend is not compatible with IR coming from older versions of LLVM due to the opaque pointer transition!

To reduce complexity, *TargetExtType* is also used as a substitute for *TypedPointerType* to represent nested types deduced in *SPIRV EmitIntrinsics* (LLVM values of *TypedPointerType* cannot be created!)

Testing, Conformance, and Quality

- LIT remains our most important method for catching regressions quickly and sketching easy to follow test cases – in contrast to complex external conformance test suites often requiring specific hardware and driver stack.
- Most of our LITs have been extended with additional *spirv-val* runs checking that the output SPIR-V binary adheres to the specification (using external Khronos SPIR-V Tools)

✓  SPIR-V Tests / Test SPIR-V / Lit Tests (ubuntu-latest) (pull_request) Successful in 45m

- Google has contributed *spirv-sim* tool for testing the SPIR-V structurizer as it is developed (control-flow and basic cross-lane interactions).
 - The tool provides more flexibility and is not as fragile as simple *FileCheck* lines.
 - *FileCheck* **could** be used for these tests, but the order of the *CHECK* lines must match the output CFG, not the input IR. The potential contributor would have to be knowledgeable and correct to make modifications to the tests.
 - *spirv-sim* helps avoid having a “ripple effect” of some change reaching the backend and necessitating modifications to the tests.

Future work

- SPIR-V has matured as a target in the last two years and more often is a better alternative to Khronos LLVM/SPIR-V Translator (which is bi-directional!)
- LLVM and many dependent project would benefit from having a SPIR-V consumer in tree as well.
 - Much of the code could be shared between the SPIR-V backend and consumer.
 - Could be used as a testing and production solution for the upstreamed SYCL/DPC++ project.

Thank you!
Questions?