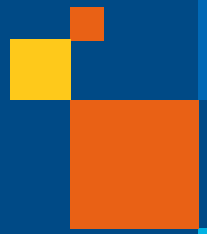
EuroLLVM 2025

Bridging LLVM and SPIR-V for Heterogeneous Computing

Vyacheslav Levytskyy
Michal Paszkowski



intel®

Current Status

- The SPIR-V backend began 2025 with two major advances
  - it has become an official target, exiting experimental status in LLVM, and
  - proved its value for heterogeneous computing by successfully passing the SYCL CTS test suite
- Focus and goals in 2025 are:
  - in the domain of compute applications
    - to harden presence in Intel deep learning workloads and extensions/backends for popular AI compiler codebases (e.g., OpenAI Triton, OpenXLA)
    - to become the default tool for LLVM IR to SPIR-V translation in the SYCL/ DPC++ compiler and pave the way for integration with the AdaptiveCpp SYCL implementation
  - in general
    - to continue optimization of the internal representation of the translation process
    - to refactor the sequence and duties of a part of translation passes
    - to enhance the test suite with SYCL and Khronos LLVM/SPIR-V Translator cases

Agenda

- Scope and driving forces
- Actors and cases
- From technical details to tangible improvements
- Applications and dependencies
- Quality assurance

SPIR-V in Heterogeneous Computing

- SYCL: the compilation process is both complex and complicated
  - host and device code are to be compiled and linked into a single application binary
  - an implementation is to distribute transformation actions along the data flow, and balance compiler and runtime architecture design
- Implications of targeting multiple backends
  - reasoning in terms of unified representation of the source code
  - materialization of compute kernels across a wide range of backend APIs
  - decisions about when, how and what is to be converted on the way from backend-independent to device-specific

SPIR-V in Heterogeneous Computing

- SPIR-V and its native LLVM backend to manage the complexity
  - SPIR-V: a portable, standardized, cross-vendor unifying IR for programming heterogeneous accelerators
  - LLVM SPIR-V Backend vs. Khronos LLVM/SPIR-V Translator
  - why do we need any extra to the Khronos LLVM/SPIR-V Translator
- Adaptation of the LLVM SPIR-V Backend to the SYCL model
  - functional improvements
  - bridging the standard, environmental specifications, and LLVM concepts
- What about HLSL and Vulkan
  - improving overall maturity of the backend

SPIR-V Backend as a Crossroad: Actors and Cases

- The SPIR-V standard specification
  - computational vs. graphical flavor of SPIR-V
  - high expectations to a semantics flow from well established instructions
- Users of SPIR-V in the role of a portable IR
  - multiple frontends, run-time environments and backends (C/C++, Fortran, HLSL, SYCL, OpenMP, OpenCL, Level Zero, Vulkan, ...)
    - binding of builtins to intrinsics
  - reverse translation to LLVM IR as a part of a hardware driver
  - heterogeneous computing workflows (HPC, GPU programming for neural networks)
    - specialized types (e.g., LLM quantization)
- Concepts of LLVM IR and utilities/frameworks for code lowering
  - SPIR-V is a semantically rich language
  - LLVM concepts and utilities (IR, available passes, virtual registers and low-level types) alone are not enough to pass semantics
  - being a part of upstream closes access to some proprietary up-to-date features

Between SPIR-V and Execution Environments

- Kernel vs. Shader: overlapped domains and implicit restrictions
  - Vulkan implies Shader and the Logical addressing model, compute environments are behind Kernel and the non-Logical addressing (physical pointers)
  - SPIR-V capabilities may normatively “implicitly declare” Kernel or Shader
  - bit operations are enabled by either Shader or BitInstructions (that doesn’t implicitly declare Kernel); OpImageRead/OpImageWrite
- The LLVM SPIR-V backend implementation
  - has a common for Kernel/Shader core logics of translation and overall stability
    - however, SPIRVSubtarget::isVulkanEnv()/isOpenCLEnv() are massively used to branch away Kernel and Shader-related features
  - helps to improve future versions of the SPIR-V specification, linking SPIR-V and execution environments
- A case study: the SPIR-V specification required Shader-coupled capabilities to read/write images in the OpenCL SPIR-V environment
  - https://github.com/KhronosGroup/SPIRV-Headers/issues/487

Between SPIR-V, Frontends and Runtimes

- Fragility of execution environments regarding SPIR-V pointer types
- The SPIR-V spec doesn’t govern how a frontend is to resolve a problem of targeting SPIR-V instructions with LLVM instructions or intrinsics
- Typed SPIR-V instructions have (too) high expectations to a quality of semantics flow during translation
- Frontend and optimizer produce a lot of patterns influencing type inference
- Type inference is unconditionally needed and not always feasible

Type Inference

- The SPIR-V language has a developed type system
  - Most instructions in SPIR-V have a type identifier attached
  - Types are OpTypeXXX instructions, built up hierarchically, that is parameterized by results of dependent type definition instructions
- Type Inference
  - a Module pass
  - look for known IR patterns, reveal dependencies, and use prior knowledge to deduce types
  - dispatch meaning via internal intrinsics and is unfortunately intertwined with the “emit intrinsics” pass
  - desperately try not to rely on mangling

Type Inference: Steps of the Pass

- Optional: Parse and store argument types of function declarations.
- Analyze bodies of Module's functions, keeping a worklist of uncomplete type deduction to postpone some of records until we evaluate the Module in full
  - fix GEP result types ahead of type inference
  - process parameters by the function header, checking explicit type tips and call sites
  - forward traversal of function's instructions: use operand to deduce instruction’s result
  - backward traversal of function's instructions: analyze instruction’s result and operand to specify, or update, or cast other instruction's operands
    - forward traversal of PHIs

Type Inference: Examples of IR Patterns

- Use operand to deduce instruction's result
  - `AllocaInst`: `getAllocatedType()`
  - `LoadInst`: `getPointerOperand()`
  - `GlobalValue`: `getValueType()`
    - and check nested types for `StructType`, `ArrayType`, and `VectorType`
  - `AddrSpaceCastInst`: `getPointerOperand()`
  - `BitCastInst`
  - `AtomicCmpXchgInst`: `getNewValOperand()`
  - `AtomicRMWInst`: `getValOperand()`
  - `PHINode`: by majority of `getIncomingValue(i)`
  - `SelectInst`: `getTrueValue()`/`getFalseValue()`
  - `CallInst`: well-known functions
    - e.g., pointer type conversions between address spaces: `to_global()`, `to_local()`, `to_private()`
  - and also builtins and `IntrinsicInst`
    - `spirv.Event`
    - `spirv.Image`

Type Inference: IR Patterns and Validation

- Analyze instruction's result and operand to specify, or update, or cast other instruction's operands
  - evaluate a known type and its completeness from the perspective of type inference, and create a list of operands to apply the type to
  - Examples:
    - `PHINode`: all value operands must have the same type as the result
    - `ICmpInst`: operands must have the same type
    - `AddrSpaceCastInst`, `BitCastInst`, `LoadInst`, `SelectInst`: known relations between result/operands
    - `CallInst`: well-known builtins/SPIR-V opcodes, e.g.: Src/Dest pointers in `OpGroupAsyncCopy`, in `OpAtomicXXX` the result has the same type as the value pointed to by the Pointer operand
- Modes/Options
  - process Module's functions vs. post-processing
  - insert a final type vs. mark as a temporary guess (uncomplete type)
  - build a new type definition vs. update the existing definition vs. build a pointer type cast and propagate changes further to affected operand's users
  - maybe restore original function return type for the analysis

Type Inference: Steps of the Pass

- Specify types of function parameters
  - check function's call sites to evaluate actual argument operand types and formal parameter of the outer function
- Process the worklist of uncomplete pointer types
  - try to deduce a better type having full information about the Module's IR patterns and dependencies between values.
- Optionally: Support the function pointers extension
- Continuously
  - a user instruction may require an explicit pointer type conversion to remain valid: propagate type update information where required
  - modify an LLVM type to conform with future transformations in IRTranslator: replace <1 x Type> vector type by the element type
    - <1 x Type> is not a legal vector type in LLT: IRTranslator represents it as the scalar

Type Inference: Case Study

- OpGroupAsyncCopy
  - an asynchronous group copy between pointer from Source to Destination
  - described in terms of number of elements rather than number of bytes
    - to implement this correctly we absolutely must know the pointee type
- SPIR-V builtins: established approach
  - bring a stable interface to express a meaning missing in LLVM
  - unmangled name looks like __spirv_OpCode [_OptionalPostfix]
- __spirv_GroupAsyncCopy(..., opaque dest ptr, opaque src ptr, ...)
  - doesn't provide any hint as for the pointee type: try to deduce types from other relations when possible but use mangling as a last-ditch effort

```
define spir_kernel void @foo(ptr addrspace(1) %src, ptr addrspace(3) %local) {
  %e = tail call spir_func target("spirv.Event")
       @__spirv_GroupAsyncCopy(i32 2, ptr addrspace(3) %local, ptr addrspace(1) %src,
                               i64 1, i64 1, target("spirv.Event") zeroinitializer)
```

Type Inference: Case Study

- Concerns: OpGroupAsyncCopy and __spirv_GroupAsyncCopy
  - “number of bytes” is more on the spirit of SPIR-V than “number of elements”
  - pointee type hint: mangling is a poor way to harvest semantical information, it is dangerous to rely on mangling
- Ideas
  - the link between the SPIR-V standard and the LLVM backend implementation is bidirectional
  - a new OpUntypedGroupAsyncCopyKHR works with untyped pointers
    - SPV_KHR_untyped_pointers is required
  - the SPIR-V spec doesn’t govern “SPIR-V friendly” builtins
    - a frontend chooses how to expose the binding
    - a new __builtin_spirv_... builtin may convey more information

Logical Layout of a Module

- Normative: a linear list of instructions in the prescribed order
- Doesn’t match well LLVM concepts and utilities
  - no easy mapping with LLVM Module as the top-level container of objects
  - explicit module scope sub-sections (names, annotations, types, constants, etc.) referred to and reused by function scope instructions
- Uniqueness
  - two different type identifiers mean two different types
  - the same opcode and operands of a non-aggregate, non-pointer type require reusing a single type definition
- Lack of support for module scope definitions during translation
  - module vs. function scope mismatch: no good place to keep definitions
    - duplicated type definitions are to be manually constructed
    - constants are created and duplicated from IRTranslator and on

Translation Time Performance vs. Module Layout

- Reference point
  - measure only speed, ignore memory (for now)
  - take as the input a SYCL test case: ~1Mb LLVM IR binary
  - realize the difference between a sequence of actions of “llvm-spirv …” and “llc -O0 -mtriple=spirv64-unknown-unknown …”
  - the Khronos LLVM/SPIR-V Translator: ~0.38 seconds
- Starting position (Dec 2024)
  - about 213 seconds, ~x560 worse than Khronos Translator
  - main culprit (as simple as “perf record …” / “perf report …”)
    - deduplicating definitions and gathering module scope instructions
    - inefficient data structures and general approach of tracing dependencies SPIR-V identifiers via an explicitly built graph
- Changes in PR #120415 result in ~x5 speed up (~42s)

Translation Time Performance vs. Module Layout/gMIR

- Problem: observed compile-time performance for a reference binary LLVM IR is still ~x105 worse than Khronos Translator time
- PR #130605 overhauls definitions deduplication, passing semantic info between passes and tracking of IR values and types (March 2025)
  - improved performance: ~x5 speed up (~7.5 seconds)
    - total ~x25 speed up comparing with the starting position
  - less bloated intermediate representation of internal translation steps
    - internal intrinsics: eliminate spv_track_constant, improve spv_assign_name
    - remove gMIR GET_XXX pseudo codes
    - generate ASSIGN_TYPE pseudo code only when required by instruction selection pattern matching rules
  - implementation has actually become simpler, meaning easier maintenance

Translation Time Performance: Reflections

- Future Work
  - representative set of test cases: use SYCL CTS and OpenCL CTS
  - address memory usage
  - isolate Type Inference into a separate pass and rethink it
- “emit-intrinsics” pass
  - pack IR names and decoration to restore eventually as SPIR-V opcodes
  - starts lowering of inline asm, switch
  - preprocess the Module before IRTranslator transformations
  - interlaces Type Inference IR traversals with generation of internal intrinsics
- The intent is to separate Type Inference from general lowering to internal intrinsics and encoding other kinds of semantical info
  - make it reusable for LLVM codebase and downstream projects
  - refactor and improve; compare to the Khronos Translator implementation

Translation Time Performance: a Bonus Win

- Problem
  - correct caching of LLVM IR entities for the sake of tracing and eventual deduplication for the module scope and reuse
  - the backend has no control and is not updated on changes in IR
- Motivation: previous approach led to stale or incorrect records
  - removal of instructions from gMIR
  - modifications during instruction selection
- Reworked cache for objects deduplication

Reworked Deduplication

- Key/value descriptors parameterized by components and provide redundancy to ensure eventual consistency
  - `using IRHandle = std::tuple<const void *, unsigned, unsigned>;`
  - `using MIHandle = std::tuple<const MachineInstr *, Register, size_t>;`
- Bi-directional mappings between IR entities and SPIR-V definitions to allow for efficient add, find, erase operations with fixing
  - IRHandle x MachineFunction → MIHandle and MachineInstr → IRHandle x MachineFunction
- Custom hashing
  - `size_t to_hash(const MachineInstr *MI)`
    - MI->getOpcode() and MI->getNumOperands()
    - combine operands after MI->getNumDefs()
      - MachineOperand::MO_Cimmediate: getType() and getCImm()
      - MachineOperand::MO_FPImmediate: getType() and getFPImm()
      - otherwise: getType()

Reworked Deduplication

- Add a record (LLVM IR/gMIR entity → MachineInstr)
  - protect against rewriting (keep actual or invalidate stale records)
  - map LLVM entity to unique IR Handle (use distinctive features of the object)
  - extend MachineInstr record with its def virtual register and custom hash value
- Find a record (LLVM IR/gMIR entity, MachineFunction) → MachineInstr
  - get from the map
  - invalidate if it's a stale or incorrect record
    - record is valid if there is a definition for the v-reg, and this definition is the same as the stored instruction
  - otherwise, reuse the definition
- Erase (MachineInstr)
  - a way to keep the cache valid in cases when code manipulations are controlled by the backend

Instruction Selection and TableGen

- Motivation: encode Instruction Selection in TableGen when feasible
- Problem: SPIR-V language is not an ISA
  - semantically rich, approximately at the same level as LLVM IR
  - emitted code includes type information as references to type instructions
- Register: a match (not perfect) to the SPIR-V notion of identifier
  - no relation to physical registers, doesn't care about name, size or alignment
  - specialize to express a type
- Register Classes: fine-tuning to correspond LLT to SPIR-V types
  - 1:1 relation with registers: 6 typed identifiers, a type def, any type and any id
- TableGen pattern matching uses a pseudo-instruction to carry types
  - simplification of pseudo-instructions: from 8 to a single ASSIGN_TYPE
  - instruction semantics normally is rich, requiring coding apart from TableGen

A pragmatic approach to GlobalISel and Machine Verifier

- Machine Verifier does not recognize SPIR-V's OpPhi as a PHI node
  - Recall the issue as discussed at the 2024 LLVM Dev Mtg

```
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
```

Initial observations:

- OpPhi is indeed a PHI node
  - starts a basic block
  - has pairs of incoming value and labels
- GenCode hardcodes what is PHI
- Option A: change GenCode to allow overriding the check whether the instruction opcode represents a PHI
- Option B: change the SPIR-V backend to postpone generation of OpPhi rather than to patch CodeGen

A pragmatic approach to GlobalISel and Machine Verifier

- Machine Verifier does not recognize SPIR-V's OpPhi as a PHI node
  - Recall the issue as discussed at the 2024 LLVM Dev Mtg

```
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 = G_PHI %12:iid, %bb.2, %8:iid, %bb.3
```

More observations:

- ~300 references to MI.isPHI(), including opt. passes, may break SPIR-V CFG in a lot of ways
- E.g.: MachineSink opt. pass:

```
bool MachineSinking::AllUsesDominatedByBlock(...) const {
  if (all_of(MRI->use_nodbg_operands(Reg), [&](MachineOperand &MO) {
      ...
      return UseBlock == MBB && UseInst->isPHI() &&
      ...
    BreakPHIEdge = true;
    return true;
```

- We apply non-elegant but pragmatic solution
  - Generate TargetOpcode::PHI instead of SPIRV::OpPhi after Instruction Selection
  - Patch opcodes on finalizing the Module's logical layout

A pragmatic approach to GlobalISel and Machine Verifier

- Consistent pointer types requires bitcasts, but G_BITCAST must change the type, and there are no typed pointer types

```
define void @foo(i1 %arg) {
  %r1 = tail call ptr @f1()
  %r2 = tail call ptr @f2()
  ...
  %ret = phi ptr [ %r1, %l1 ], [ %r2, %l2 ]
  ret void
}

define ptr @f1() {
  %p = alloca i8
  store i8 8, ptr %p
  ret ptr %p
}

define ptr @f2() {
  %p = alloca i32
  store i32 32, ptr %p
  ret ptr %p
}
```

Patching GlobalISel vs. overheads of early instruction selection

- the notion of typed pointers is important for SPIR-V: we need bitcasts between pointers with different pointee types
- low level types of source and destination hold no details to improve validation of G_BITCAST
- the solution is to run instruction selection for G_BITCAST immediately after IR Translation, lowering to OpBitcast
- the only noticed consequence is that CombinerHelper cannot transform known patterns around G_BUILD_VECTOR

Widening Application Areas in Compute

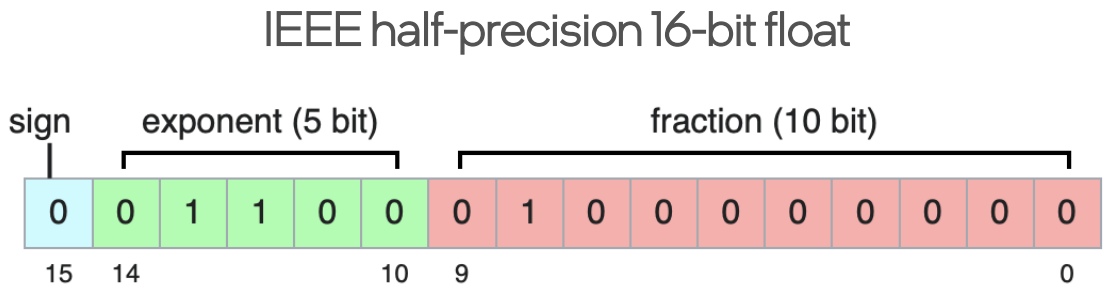
- The SPIR-V Backend is in SYCL/DPC++ CI workflows
- OpenAI Triton backend for Intel GPUs uses the SPIR-V backend API

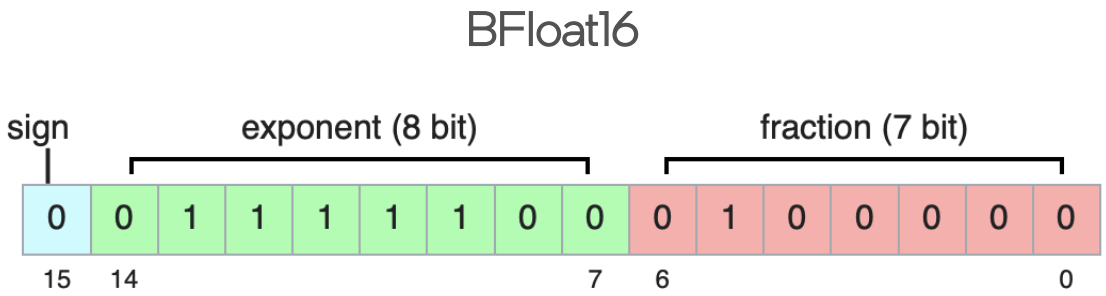
```
extern "C" LLVM_EXTERNAL_VISIBILITY bool
SPIRVTranslate(Module *M, std::string &SpirvObj, std::string &ErrMsg,
               const std::vector<std::string> &AllowExtNames,
               llvm::CodeGenOptLevel OLevel, Triple TargetTriple)
```

- Intel MLIR-based Graph Compiler
  - proper support of GPUToLLVMSPV requires sync in terms of OpenCL builtins
  - MLIR lowering emits assembly and binary code via general LLVM's interface
  - SPIRVTargetMachine is to be in sync with the SPIR-V backend API
- Intel Extension for OpenXLA
  - In XLA CPU/GPU backends use LLVM for code generation, and the SPIR-V backend is a way to emit "native code" for Intel hardware
- Good news: more use cases and dependencies are expected
  - e.g.: BFloat16 and XLA support

BFloat16 (mis)representation in GlobalISel

- SPIR-V backend and many other targets require correct and efficient handling of non-standard floating-point types like BFloat16.
- GlobalISel has limited ability to represent non-IEEE floating-point types. The current LLT model in GlobalISel primarily encodes bit widths but not the semantic differences among various FP types (BF16, FP16, etc.).
- Floating-point vs. integer distinctions are often inferred from the instruction opcode (e.g. G_FMUL implies a float), but 16-bit BF16 vs. 16-bit IEEE half is ambiguous.





Current solutions and problems

- Many projects introduce custom handling by encoding BF16 as “16-bit float” or “int16” + special intrinsics. Targets with separate int/float register banks attempt to deduce float usage from the surrounding operations. Some backends rely on extra metadata, flags, or frequent bitcasts instructions to signal “this is BF16!”.
- All these solutions are hacky, lead to fragmentation in the LLVM world, and are error-prone!

Solutions proposed by the community

a) Extend LLT with extra bits
   - Add a small number of bits (2–4) to indicate whether a 16-bit scalar is BF16, IEEE-16, or another "variant float."
   - Straightforward for new types like BF16 but requires updating legalization/regbank code to respect the new type info.

b) Redefine LLT Kinds (Integer, Float, Pointer, Vector, etc.)
   - Introduce richer Kind enumerations (e.g., FLOAT vs. INTEGER), plus a small "FPInfo" field for BF16 / IEEE16 / etc.
   - Makes float vs. integer explicit, removing guesswork. Larger refactorings in GlobalISel passes needed, helps future expansions (FP8, TF32, etc.).

c) Attach FP-Type Operand or Metadata
   - Keep LLT as is, but store a type-enum operand in FP operations (e.g. "BF16 operand" for G_FMUL).
   - Requires minimal changes to the LLT structure, but MIR becomes more verbose; passes that interpret types must now read an additional operand.

d) Use Metadata / Analysis

https://discourse.llvm.org/t/rfc-globalisel-representing-fp-types-in-llt
https://discourse.llvm.org/t/rfc-globalisel-adding-fp-type-information-to-llt

Quality Assurance

- The SPIR-V Backend test suite is constantly growing
  - 609 LIT tests as of March 2025
- Reflects necessity, priority and what's important to develop in the not quite well-established domain
- Instrumented not only with FileCheck, but with SPIR-V tools, most notably spirv-val
  - automatically run using Github Actions
- Mainly focused on
  - features, including 37 SPIR-V extensions
  - regression testing
  - general SPIR-V validity, including fragments from much bigger Conformance Test Suites
- Future Work
  - more compatibility testing in terms of the SPIR-V specification version?
  - identify weakly tested sub-systems/passes (a structured fuzzer? code coverage analysis)
  - automation of non-functional testing

The Roadblocks After Going Official/Non-experimental

- More buildbots and environments helped to reveal weak points
  - some quick post-promotion problem solving efforts were required to support the move
- Mainly minor issues
  - improve portability of the code (kind of a missing #include or select another library call)
  - address sanitizer complaints (an uninitialized variable to pick up a proper InstructionSet in the emit-intrinsics pass)
  - interaction with LLVM unit tests (move unit tests resources, a Module pointer, from the class-scope to a local scope of the class member function to be sure that before the test env is teared down the pointer is released)
- Buildbots testing SPIR-V backend revealed issues in our implementation of user-facing options parsing
  - different concurrency conditions in running unit tests
  - API and usage of the extensions list is reworked to remove writes to the global cl::opt variable; no calling cl::ParseCommandLineOptions() in multi-threaded context

QA for Applications in Heterogeneous Computing

- The LLVM SPIR-V backend is OpenCL 3.0 and SYCL conformant
- OpenCL CTS
  - flaky tests are addressed by recent rework of caching of LLVM IR entities for tracing, deduplication and reuse
- SYCL CTS
  - as of March 2025 all known issues are addressed, including backports to LLVM 14-19 branches of Khronos Translator and type of GEP results in type inference
- Intel XPU backend for Triton: high pass rates in unit tests: 96-98%
  - a way to extend use cases and add a perspective (e.g., support of <1 x Type> vector type)
- SYCL end-to-end test suite: stable high pass rates: 94-99%
  - unsupported features
    - by intent as deprioritized, e.g.: FPGA
    - temporarily, e.g.: AddressSanitizer
  - features in progress, e.g.: recently added extensions, like SPV_INTEL_joint_matrix

QA for Applications in Heterogeneous Computing

- Complicated workflow: downstream intel/llvm, upstream LLVM/SPIR-V Backend, the Khronos LLVM/SPIR-V Translator, CPU/GPU OpenCL and Level Zero run-times and drivers
  - a time lag is always present
  - it’s hard to achieve actual pass rates due to mismatch between components version
- Continuous testing out of LLVM
  - scheduled SYCL CTS runs as one of https://github.com/intel/llvm CI workflows
  - precommit CI workflow in https://github.com/intel/llvm running the SYCL end-to-end test suite
  - Intel XPU backend for Triton: a CI workflow in https://github.com/intel/intel-xpu-backend-for-triton

Non-functional Testing: Run-Time Performance

- On the stage of initial probes: a thorough/automated approach is to do
- GROMACS benchmark set
  - No OpenMP: a statistically significant but really small difference between means
    - SPIR-V Backend: time elapsed: ~1342s, perf (ns/day): 1.288
    - Khronos Translator: time elapsed: ~1378s , perf (ns/day): 1.254
    - t-Test, $\alpha = 0.05$: p-value = 0.03 for both time and performance
  - With OpenMP: no statistically significant difference between means
    - SPIR-V Backend: time elapsed: ~835s, perf (ns/day): 2.069
    - Khronos Translator: time elapsed: ~835s , perf (ns/day): 2.069
    - t-Test, $\alpha = 0.05$: p-value = 0.98/0.99
  - Run-time performance of the SPIR-V Backend's code is on-par with Khronos Translator's
- Future Work
  - part of planned developments of non-functional testing procedures
  - enhance along axes of benchmarks, options, environments and automation

Thank you!

Questions?