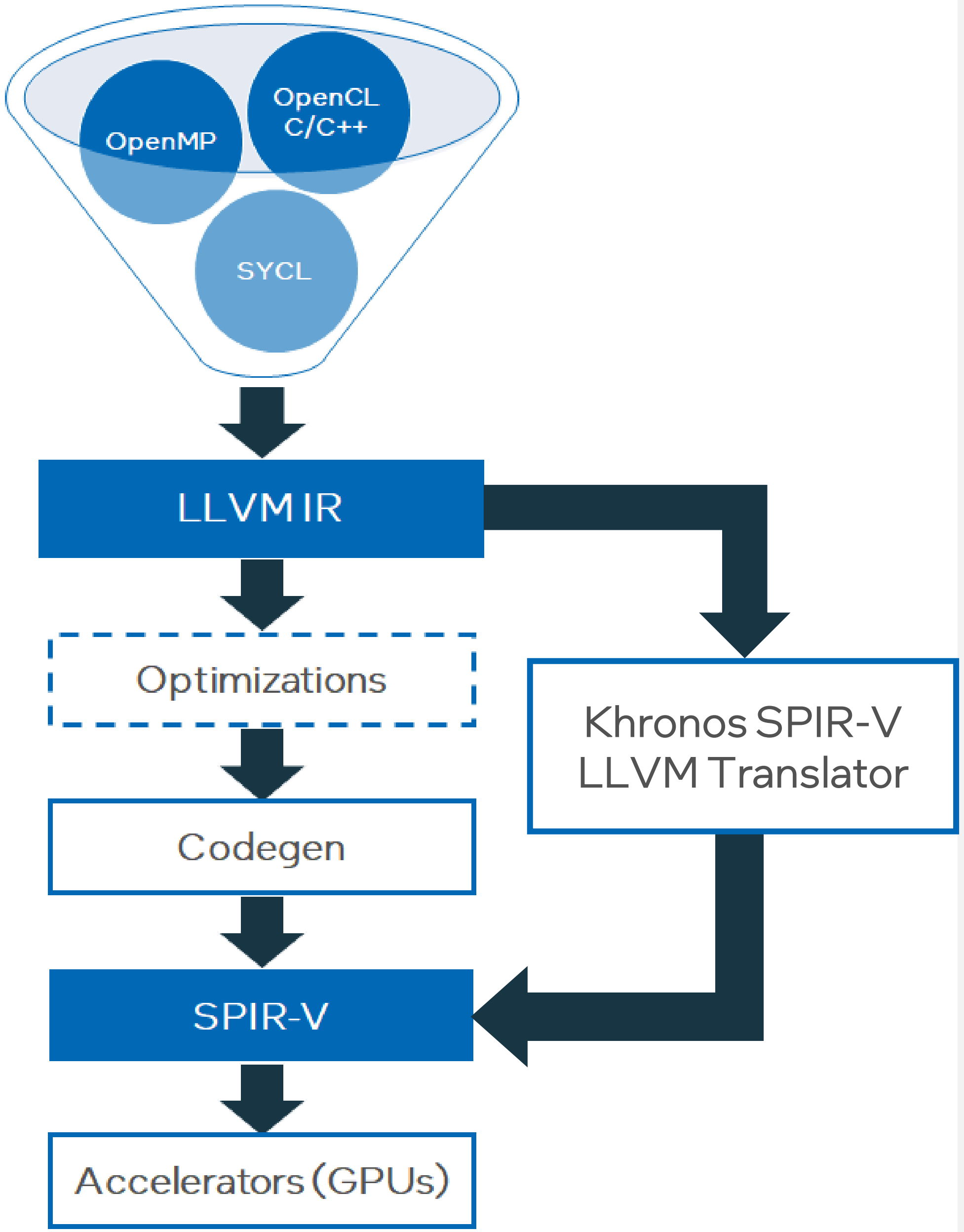
2022 LLVM Developers’ Meeting

SPIR-V Backend in LLVM: Upstream and Beyond

Alex Bezzubikov, Michal Paszkowski

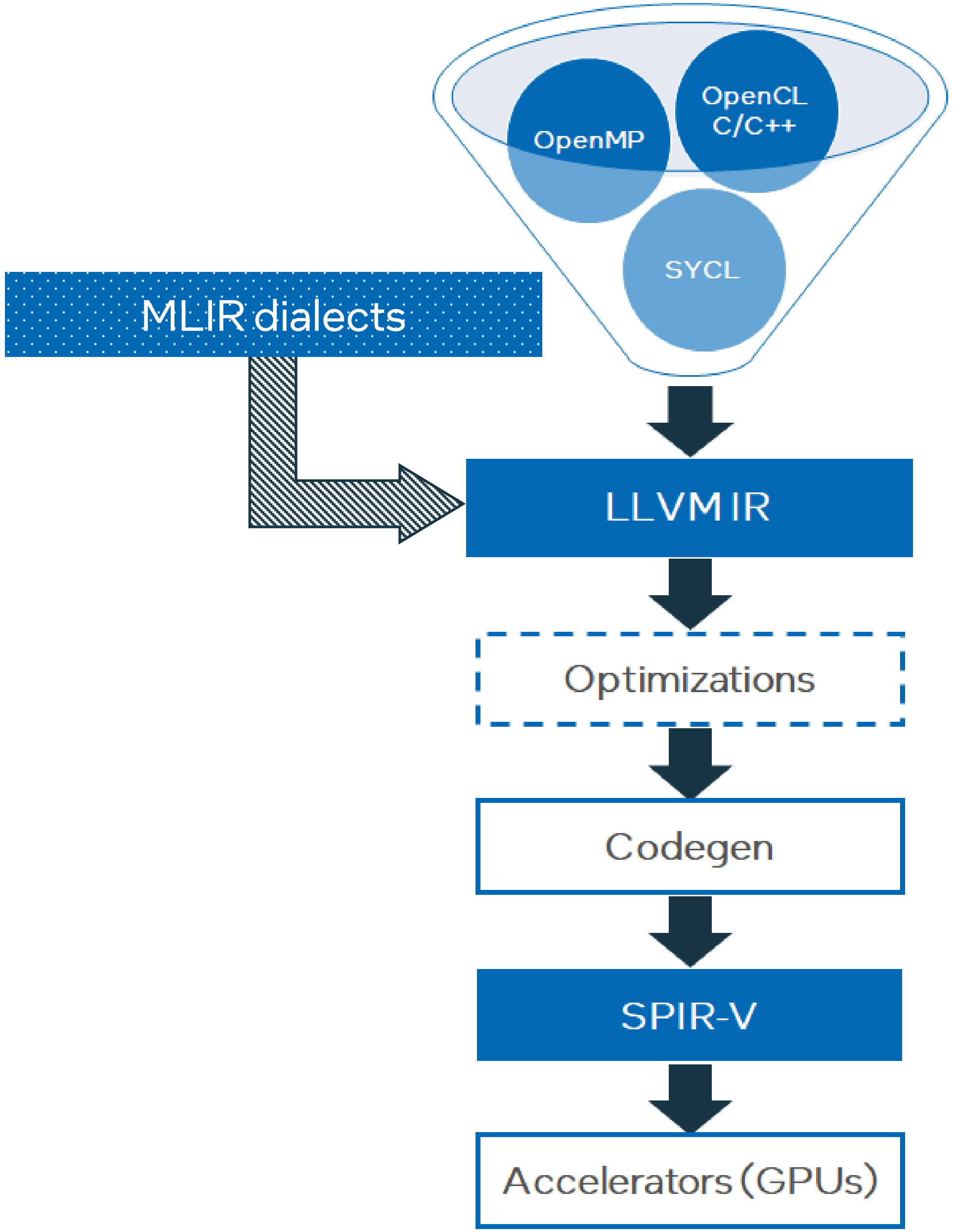
SPIR-V in a nutshell

- 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, Vulkan)
- Always consumed in a specific environment
  - Core specification defined by Khronos Group
  - Environment defined by client APIs (can apply more restrictions than core specification and define extra instructions)
- Khronos SPIR-V LLVM Translator is a production-ready bi-directional “bridge” between LLVM IR and SPIR-V
  - Potential inconsistencies between the translator and OpenCL/OpenMP Clang
- SPIR-V backend is a native solution to targeting SPIR-V from LLVM



Motivation for a new backend

- Accelerators (GPUs) consume SPIR-V and there is no native solution bridging LLVM IR with SPIR-V
- There are already other GPU backends in LLVM (such as AMDGPU, NVPTX)
- Makes MLIR-to-LLVM translation sensible for more accelerators



```
__kernel void foo(__global float *a, __global float *b,
                  __global float *out) {
  size_t idx = get_global_id(0);
  out[idx] = a[idx] + b[idx]+2.f;
}
```

OpenCL

```
define spir_kernel void @foo(ptr addrspace(1) %a, ptr addrspace(1) %b,
                             ptr addrspace(1) %out)  {
%call = tail call spir_func i64 @_Z13get_global_idj(i32 0)
%arrayidx = getelementptr inbounds float, ptr addrspace(1) %a, i64 %call
%0 = load float, ptr addrspace(1) %arrayidx
%arrayidx1 = getelementptr inbounds float, ptr addrspace(1) %b, i64
%call
%1 = load float, ptr addrspace(1) %arrayidx1
%add = fadd float %0, %1
%add2 = fadd float %add, 2.000000e+00
%arrayidx2 = getelementptr inbounds float, ptr addrspace(1) %out, i64
%call
store float %add2, ptr addrspace(1) %arrayidx2
ret void
}
```

LLVM IR

```
%ulong = OpTypeInt 64 0
%v3ulong = OpTypeVector %ulong 3
%_ptr_Input_v3ulong = OpTypePointer Input %v3ulong
%void = OpTypeVoid
%float = OpTypeFloat 32
%_ptr_CrossWorkgroup_float = OpTypePointer CrossWorkgroup %float
%9 = OpTypeFunction %void %_ptr_CrossWorkgroup_float
%_ptr_CrossWorkgroup_float %_ptr_CrossWorkgroup_float
%__spirv_BuiltInGlobalInvocationId = OpVariable %_ptr_Input_v3ulong Input
%float_2 = OpConstant %float 2
%foo = OpFunction %void None %9
%a = OpFunctionParameter %_ptr_CrossWorkgroup_float
%b = OpFunctionParameter %_ptr_CrossWorkgroup_float
%out = OpFunctionParameter %_ptr_CrossWorkgroup_float
%entry = OpLabel
%15 = OpLoad %v3ulong %__spirv_BuiltInGlobalInvocationId Aligned 32
%call = OpCompositeExtract %ulong %15 0
%arrayidx = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %a %call
%18 = OpLoad %float %arrayidx Aligned 4
%arrayidx1 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %b %call
%20 = OpLoad %float %arrayidx1 Aligned 4
%add = OpFAdd %float %18 %20
%add2 = OpFAdd %float %add %float_2
%arrayidx2 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %out %call
OpStore %arrayidx2 %add2 Aligned 4
OpReturn
OpFunctionEnd
```

SPIR-V

Backend design overview

- Based on the GlobalISel framework
  - Much simpler structure, easier to understand
  - Easy to use C++ code for selection when necessary
- Not a real hardware (no scheduling, register allocation, etc.)
- SPIR-V is much closer to LLVM IR than to MIR
- Bridging the gap between SPIR-V and MIR:
  - Heavy use of target-specific intrinsics
  - Pre-CodeGen passes
  - Global entries tracking infrastructure
  - Some C++ in InstructionSelector

Scope differences: LLVM & MIR vs SPIR-V

- Types, constants, globals (MIR only)
  - LLVM & MIR: function-scoped (structure types are still global)
  - SPIR-V: module-scoped, defined once
- Not enforcing that in MIR for consistency, tracking all the instances to be processed later

  Type*|Constant*|… ↔ pair<MachineFunction*, Register>
- Processed later at AsmPrinter

```
__kernel void foo(__global float *a, __global float *b,
                  __global float *out) {
  size_t idx = get_global_id(0);
  out[idx] = a[idx] + b[idx]+2.f;
}
```

OpenCL

```
define spir_kernel void @foo(ptr addrspace(1) %a, ptr addrspace(1) %b,
                             ptr addrspace(1) %out)  {
%call = tail call spir_func i64 @_Z13get_global_idj(i32 0)
%arrayidx = getelementptr inbounds float, ptr addrspace(1) %a, i64 %call
%0 = load float, ptr addrspace(1) %arrayidx
%arrayidx1 = getelementptr inbounds float, ptr addrspace(1) %b, i64
%call
%1 = load float, ptr addrspace(1) %arrayidx1
%add = fadd float %0, %1
%add2 = fadd float %add, 2.000000e+00
%arrayidx2 = getelementptr inbounds float, ptr addrspace(1) %out, i64
%call
store float %add2, ptr addrspace(1) %arrayidx2
ret void
}
```

LLVM IR

```
%ulong = OpTypeInt 64 0
%v3ulong = OpTypeVector %ulong 3
%_ptr_Input_v3ulong = OpTypePointer Input %v3ulong
%void = OpTypeVoid
%float = OpTypeFloat 32
%_ptr_CrossWorkgroup_float = OpTypePointer CrossWorkgroup %float
%9 = OpTypeFunction %void %_ptr_CrossWorkgroup_float
%_ptr_CrossWorkgroup_float %_ptr_CrossWorkgroup_float
%__spirv_BuiltInGlobalInvocationId = OpVariable %_ptr_Input_v3ulong Input
%float_2 = OpConstant %float 2
%foo = OpFunction %void None %9
%a = OpFunctionParameter %_ptr_CrossWorkgroup_float
%b = OpFunctionParameter %_ptr_CrossWorkgroup_float
%out = OpFunctionParameter %_ptr_CrossWorkgroup_float
%entry = OpLabel
%15 = OpLoad %v3ulong %__spirv_BuiltInGlobalInvocationId Aligned 32
%call = OpCompositeExtract %ulong %15 0
%arrayidx = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %a %call
%18 = OpLoad %float %arrayidx Aligned 4
%arrayidx1 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %b %call
%20 = OpLoad %float %arrayidx1 Aligned 4
%add = OpFAdd %float %18 %20
%add2 = OpFAdd %float %add %float_2
%arrayidx2 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %out %call
OpStore %arrayidx2 %add2 Aligned 4
OpReturn
OpFunctionEnd
```

SPIR-V

Bypassing IRTranslator lowering

- IRTranslator usually lowers the level of abstraction, which is sometimes too low for SPIR-V generation
  - Aggregates: types, loads/stores, extract/insertvalue
  - GEPs
- SPIR-V is designed to be platform-agnostic, that's why no overoptimization is expected to happen
- Accelerator-focused compiler consumes SPIR-V and then is entitled to do whatever it wants
- Needed mostly for OpenCL-specific types, Clang seems to avoid generating aggregate loads/stores for ordinary kernel code

Bypassing IRTranslator lowering

- Value = Result<id> + ResultType<id>
- Want to avoid `getOrCreateVRegs(…).size() > 1`

Input LLVM IR:

```
%struct.ndrange_t = type { i32, [3 x i64], [3 x i64], [3 x i64] }

%ndrange = alloca %struct.ndrange_t, align 8  %3 = alloca %struct.ndrange_t, align 8
call spir_func void @OpBuildNDRange_i64_i64_i64(%struct.ndrange_t* %3, i64 3, i64 0, i64 0)
%4 = load %struct.ndrange_t, %struct.ndrange_t* %3, align 8
store %struct.ndrange_t %4, %struct.ndrange_t* %ndrange, align 1
```

Output from default IRTranslator:

```
%34:id = OpFunctionCall %5:type(s32), @OpBuildNDRange_i64_i64_i64, %21:_(p0), %22:_(s64), %23:_(s64),
%23:_(s64)
%35:_(s32) = G_LOAD %21:_(p0)
%46:_(s64) = G_CONSTANT i64 8
%45:_(p0) = G_PTR_ADD %21:_, %46:_(s64)
%36:_(s64) = G_LOAD %45:_(p0)
%48:_(s64) = G_CONSTANT i64 16
...
```

Bypassing IRTranslator lowering

- Want to avoid `getOrCreateVRegs(…).size() > 1`

Input LLVM IR:

```
%struct.ndrange_t = type { i32, [3 x i64], [3 x i64], [3 x i64] }

%ndrange = alloca %struct.ndrange_t, align 8  %3 = alloca %struct.ndrange_t, align 8
call spir_func void @OpBuildNDRange_i64_i64_i64(%struct.ndrange_t* %3, i64 3, i64 0, i64 0)
%4 = load %struct.ndrange_t, %struct.ndrange_t* %3, align 8
store %struct.ndrange_t %4, %struct.ndrange_t* %ndrange, align 1
```

After LLVM IR preparing:

```
call spir_func void @OpBuildNDRange_i64_i64_i64(%struct.ndrange_t* %4, i64 %5, i64 %6, i64 %7)
%8 = call i32 @llvm.spv.load.p0s_struct.ndrange_ts(%struct.ndrange_t* %4, i16 1, i8 8)
call void @llvm.spv.store.i32.p0s_struct.ndrange_ts(i32 %8, %struct.ndrange_t* %ndrange, i16 2, i8 1)
```

Output from default IRTranslator:

```
%47:id = OpFunctionCall %5:type(s32), @OpBuildNDRange_i64_i64_i64, %31:_(p0), %32:_(s64), %34:_(s64),
                                                                                             %36:_(s64)
%48:_(s32) = G_INTRINSIC_W_SIDE_EFFECTS intrinsic(@llvm.spv.load), %31:_(p0), 1, 8
G_INTRINSIC_W_SIDE_EFFECTS intrinsic(@llvm.spv.store), %48:_(s32), %7:_(p0), 2, 1
```

Keeping type information

- SPIR-V types are similar to LLVM IR types, LLTs are not
- Can be deduced in simple cases (integers, floats)
- Aggregates, pointers, API-specific types?

```
%add = fadd float %2, %4
call void @llvm.spv.assign.type.f32(float %add, metadata float 0.000000e+00)

%27:_(s32) = G_FADD %23:_, %25:_
G_INTRINSIC_W_SIDE_EFFECTS intrinsic(@llvm.spv.assign.type), %27:_(s32), <0x79ece28>

%12:type(s32) = OpTypeFloat 32
%46:fid(s32) = G_FADD %40:fid, %41:fid
%47:anyid(s32) = ASSIGN_TYPE %46:fid(s32), %12:type(s32)
```

llvm::ConstantAsMetadata*
(key in the tracking map)

Instruction Selection

- Relying on TableGen as much as possible
- Type information folding:

```
%2:type(s32) = OpTypeFloat 64, 0
%13:_(s32) = G_FNEG %8:_(s32)
%14:anyid(s64) = ASSIGN_TYPE %2:_(s32), %2:type(s32)
                  ↓

%14:id = OpFNegate %2:type, %8:id
```

```
class UnOpTyped<string name, bits<16> opCode, RegisterClass CID, SDNode node>
                : Op<opCode, (outs ID:$dst), (ins TYPE:$src_ty, CID:$src),
                  "$dst = "#name#" $src_ty $src", [(set ID:$dst, (assigntype (node CID:$src),
                                                                               TYPE:$src_ty))]>;

def OpFNegate: UnOpTyped<"OpFNegate", 127, fID, fneg>;
```

Instruction Selection

- Register types mismatch:
  - There are no “real” registers in SPIR-V
  - LLT do not matter for SPIR-V as result type is calculated from LLVM IR and is a separate operand
  - Example: *OpFAdd* operand can be any FP value or vector of FP values
  - Don’t want to introduce casts or create huge multiclasses (want them small)
  - Solution: pseudo-casts of everything to s32 which are removed later

```
%37:fid(s32) = GET_fID %20:anyid(s16)
%38:fid(s32) = GET_fID %22:anyid(s16)
%36:fid(s32) = G_FADD %37:fid, %38:fid

%23:id = OpFAddS %28:type, %37:fid, %38:fid

%10 = OpTypeFloat 32
...
%23 = OpFAdd %10 %20 %22
```

Instruction Selection

```
%31:_(s32) = G_LOAD %19:anyid(p1)
%20:anyid(s32) = ASSIGN_TYPE %31:_(s32), %28:type(s32)
   …
%37:fid(s32) = GET_fID %20:anyid(s32)
%38:fid(s32) = GET_fID %22:anyid(s32)
%36:fid(s32) = G_FADD %37:fid, %38:fid
%28:type(s32) = OpTypeFloat 32
%23:anyid(s32) = ASSIGN_TYPE %36:fid(s32), %28:type(s32)
```

Removing duplicates

```
float bar(float *a, int idx) {
  return a[idx] * 2.f;
}

__kernel void foo(__global float *a, __global float *out) {
  size_t idx = get_global_id(0);
  out[idx] = (idx % 2) ? a[idx] + 2.f : bar(a, idx);
}
```

```
# Machine code for function _Z3barPU3AS4fi: IsSSA, TracksLiveness, Legalized, Selected

bb.1.entry:
  %2:type = OpTypeFloat 32
  %22:id = OpConstantF %2:type, 1073741824

# Machine code for function foo: IsSSA, TracksLiveness, Legalized, Selected

bb.1.entry:
  %2:type = OpTypeFloat 32
  %47:id = OpConstantF %2:type, 1073741824
```

```
__kernel void foo(__global float *a, __global float *b,
                  __global float *out) {
  size_t idx = get_global_id(0);
  out[idx] = a[idx] + b[idx]+2.f;
}
```

OpenCL

```
define spir_kernel void @foo(ptr addrspace(1) %a, ptr addrspace(1) %b,
                             ptr addrspace(1) %out)  {
%call = tail call spir_func i64 @_Z13get_global_idj(i32 0)
%arrayidx = getelementptr inbounds float, ptr addrspace(1) %a, i64 %call
%0 = load float, ptr addrspace(1) %arrayidx
%arrayidx1 = getelementptr inbounds float, ptr addrspace(1) %b, i64
%call
%1 = load float, ptr addrspace(1) %arrayidx1
%add = fadd float %0, %1
%add2 = fadd float %add, 2.000000e+00
%arrayidx2 = getelementptr inbounds float, ptr addrspace(1) %out, i64
%call
store float %add2, ptr addrspace(1) %arrayidx2
ret void
}
```

LLVM IR

```
%ulong = OpTypeInt 64 0
%v3ulong = OpTypeVector %ulong 3
%_ptr_Input_v3ulong = OpTypePointer Input %v3ulong
%void = OpTypeVoid
%float = OpTypeFloat 32
%_ptr_CrossWorkgroup_float = OpTypePointer CrossWorkgroup %float
%9 = OpTypeFunction %void %_ptr_CrossWorkgroup_float
%_ptr_CrossWorkgroup_float %_ptr_CrossWorkgroup_float
%__spirv_BuiltInGlobalInvocationId = OpVariable %_ptr_Input_v3ulong Input
%float_2 = OpConstant %float 2
%foo = OpFunction %void None %9
%a = OpFunctionParameter %_ptr_CrossWorkgroup_float
%b = OpFunctionParameter %_ptr_CrossWorkgroup_float
%out = OpFunctionParameter %_ptr_CrossWorkgroup_float
%entry = OpLabel
%15 = OpLoad %v3ulong %__spirv_BuiltInGlobalInvocationId Aligned 32
%call = OpCompositeExtract %ulong %15 0
%arrayidx = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %a %call
%18 = OpLoad %float %arrayidx Aligned 4
%arrayidx1 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %b %call
%20 = OpLoad %float %arrayidx1 Aligned 4
%add = OpFAdd %float %18 %20
%add2 = OpFAdd %float %add %float_2
%arrayidx2 = OpInBoundsPtrAccessChain %_ptr_CrossWorkgroup_float %out %call
OpStore %arrayidx2 %add2 Aligned 4
OpReturn
OpFunctionEnd
```

SPIR-V

Opaque types and kernel attributes

- OpenCL defines a family of opaque types such as *image2d_t*, *sampler_t*…
- They are represented as pointers to opaque structs in LLVM IR:

```
%opencl.image2d_ro_t = type opaque
%opencl.sampler_t = type opaque
```

- Various frontends provide these types and additional semantics for kernel functions in metadata:

```
define spir_kernel void @foo(i64 %sampler) #0 !kernel_arg_type !9 { ... }
!9 = !{!"sampler_t"}
```

- Generating SPIR-V without this extra information is very limited.
- There are many issues with this approach…

Opaque types and kernel attributes

- There are inconsistencies in how the type information is conveyed in opaque struct names and metadata between various frontends.
- Removing metadata should not cause the program to break.
- Some passes perform illegal optimizations (ptrtoint/inttoptr…).
- Very limited usability in LLVM IR (impossible to bitcast to/from these types).

Additionally, in “opaque pointer mode”…

- These types need to be inferred (demangling and parsing function names, analyzing instructions)
- Return types are not encoded in Itanium name mangling.

Parsing types and the new OpaqueType

- Currently, the SPIR-V backend translates information conveyed in metadata into new/existing opaque structs.
- The structs are later parsed using TableGen records and lowered into proper SPIR-V types.

Phabricator patch D135202 provides a better solution:

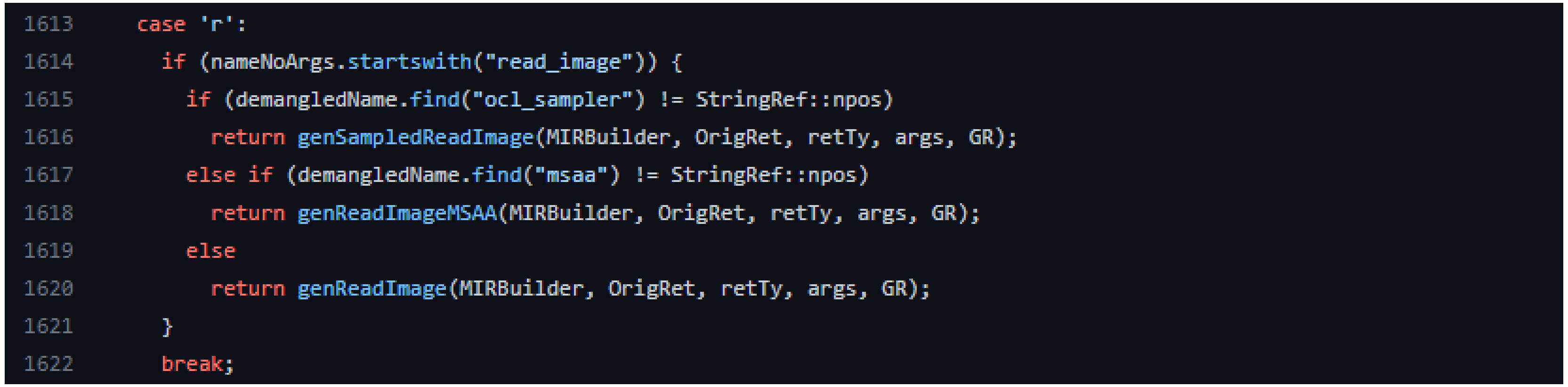
- Types like *image2d_t* could be represented using a new OpaqueType in LLVM IR:

```
opaque("image", void, i32 2, i32 0, …)
```

- This would eliminate the need for complex parsing in the backends, provide useful representation for IR transformations, solve the problem of inferring return and argument types in “opaque pointer mode”.

Lowering builtin functions

- Clang implements OpenCL/SPIR-V/GLSL builtin name mangling according to the Itanium C++ ABI and passes the mangled names to the backend.
  - Backends need to rely on string lookup for builtin lowering, parameter type compatibility is only checked during CodeGen, each backend essentially provides its own implementation even for simple builtins
  - Initial implementation in the SPIR-V backend was based on a huge *switch* matching demangled strings to 30 incomplete *gen<builtin name>* functions. Writing additional 600+ did not seem like a good idea:



```
case 'r':
  if (nameNoArgs.startswith("read_image")) {
    if (demangledName.find("ocl_sampler") != StringRef::npos)
      return genSampledReadImage(MIRBuilder, OrigRet, retTy, args, GR);
    else if (demangledName.find("msaa") != StringRef::npos)
      return genReadImageMSAA(MIRBuilder, OrigRet, retTy, args, GR);
    else
      return genReadImage(MIRBuilder, OrigRet, retTy, args, GR);
  }
  break;
```

TableGen'erated builtins implementation

- Organized builtin functions into ~18 groups, provided requirements + implementation for each group, and defined 600+ builtins in TableGen:
  - Groups are API-agnostic (one implementation for equivalent OpenCL, GLSL, SPIR-V builtins).
  - Some lowering details are hard-coded in the records (demangled name string, group, target opcode):

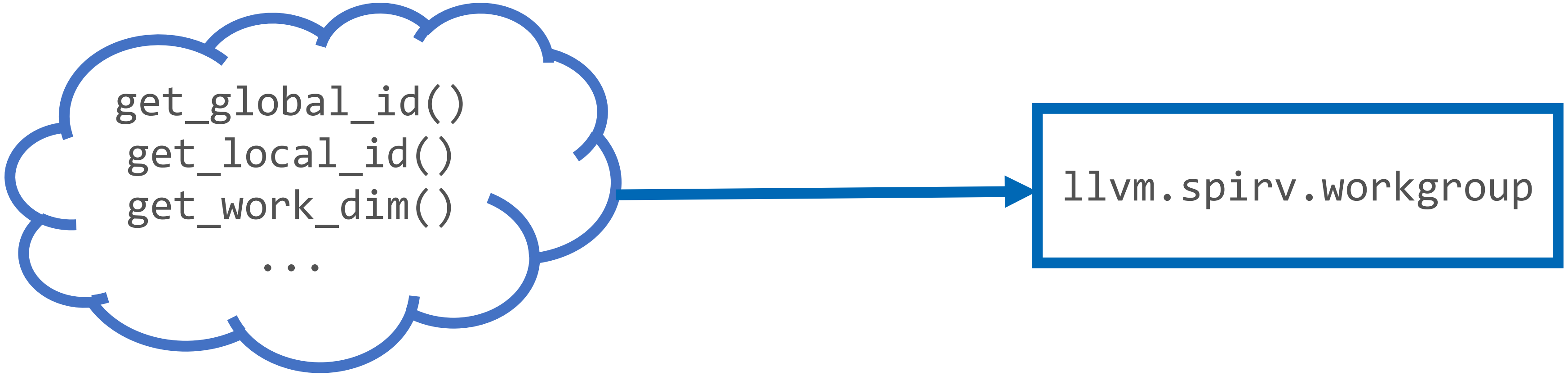
```
defm : NativeBuiltin<"isequal", Relational, OpFOrdEqual>;
defm : NativeBuiltin<"__spirv_FOrdEqual", Relational, OpFOrdEqual>;
```

  - Some are parsed/read in TableGen from the demangled names (e.g. destination sign):

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

Clang IR library with builtins definitions

- Instead of parsing demangled names, expand builtins into either native IR instructions or new SPIR-V target intrinsics.
  - Backend will only need to lower the new intrinsics and the group lowering functions from the current solution can be reused.
- Newly defined intrinsics will not match 1:1 with either OpenCL or SPIR-V builtins (We can't have 10 000+ intrinsics).
  - Instead, they will be more general and cover a group of functions:



```
get_global_id()
get_local_id()
get_work_dim()
...
```

```
llvm.spirv.workgroup
```

Extending the backend and new APIs

- Currently we are targeting the compute "flavor" of SPIR-V and support the OpenCL API.
- However, the backend can be easily extended for other APIs:
  - SPIR-V capabilities, extensions, decorations, and symbolic operands are defined in: llvm/lib/Target/SPIRV/SPIRVSymbolicOperands.td
  - Adding support for most builtins should only involve defining additional TableGen records in:
    llvm/lib/Target/SPIRV/SPIRVBuiltins.td

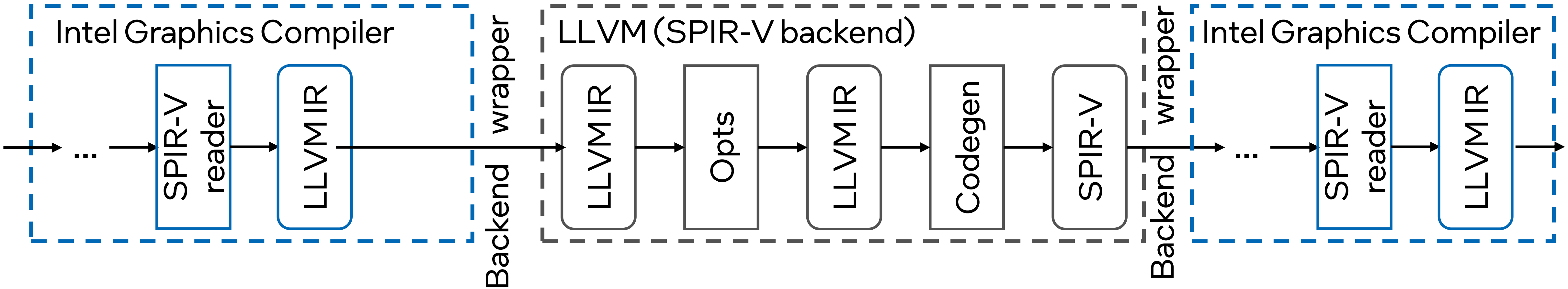
    Some GLSL builtins are defined as an example following the OpenCL records:

    ```
    defm : DemangledExtendedBuiltin<"u_min", OpenCL_std, 159>;

    defm : DemangledExtendedBuiltin<"UMin", GLSL_std_450, 38>;
    ```
  - Instruction printer is also based on TableGen.
- Support for 3D shaders would require introducing structured control flow (including OpSelectionMerge/OpLoopMerge instructions).

Conformance testing and maintaining the target in-tree

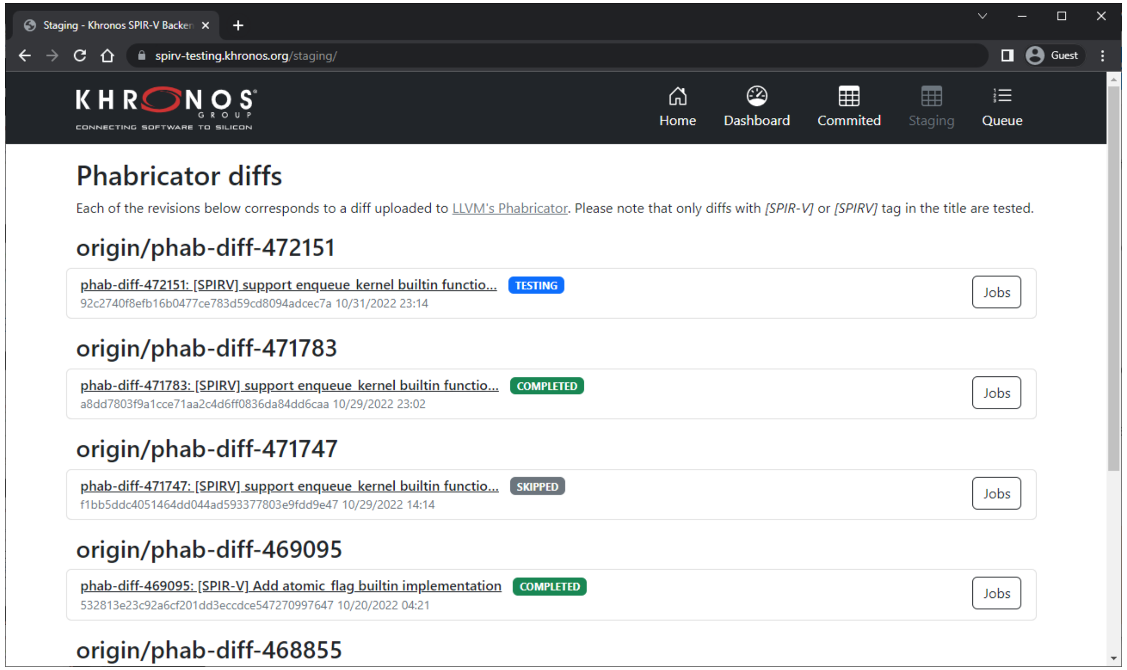
- About 220 passing LIT tests in the LLVM repository.
- Khronos + Intel also provide runtime testing infrastructure based on the *Khronos OpenCL Conformance Test Suite*.
  - The backend is integrated in the custom built open-source *Intel Graphics Compiler* + *Intel Compute Runtime* compilation flow.



  - The whole testing infrastructure is open source and available in the KhronosGroup/SPIRV-Backend-Testing repository.

Conformance testing and maintaining the target in-tree

- Khronos provides the spirv-testing.khronos.org website showing runtime testing results, dumps, and regression reproduction scripts.
- Each relevant commit to the LLVM main branch and Phabricator diffs with *[SPIR-V]* in the commit title are tested automatically.


Staging - Khronos SPIR-V Backer
spirv-testing.khronos.org/staging/
Guest
KHRONOS GROUP
CONNECTING SOFTWARE TO SILICON
Home
Dashboard
Commited
Staging
Queue

Phabricator diffs

Each of the revisions below corresponds to a diff uploaded to LLVM's Phabricator. Please note that only diffs with *[SPIR-V]* or *[SPIRV]* tag in the title are tested.

origin/phab-diff-472151

phab-diff-472151: [SPIRV] support enqueue_kernel builtin functio... TESTING
92c2740f8efb16b0477ce783d59cd8094adcec7a 10/31/2022 23:14
Jobs

origin/phab-diff-471783

phab-diff-471783: [SPIRV] support enqueue_kernel builtin functio... COMPLETED
a8dd7803f9a1cce71aa2c4d6ff0836da84dd6caa 10/29/2022 23:02
Jobs

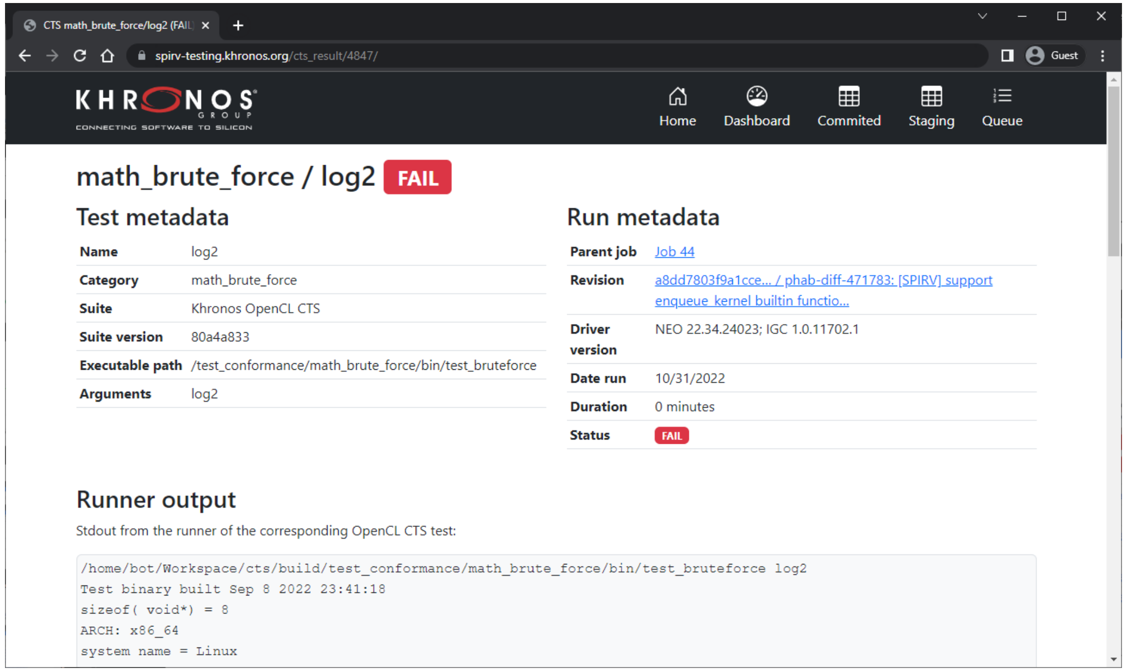
origin/phab-diff-471747

phab-diff-471747: [SPIRV] support enqueue_kernel builtin functio... SKIPPED
f1bb5ddc4051464dd044ad593377803e9fdd9e47 10/29/2022 14:14
Jobs

origin/phab-diff-469095

phab-diff-469095: [SPIR-V] Add atomic_flag builtin implementation COMPLETED
532813e23c92a6cf201dd3eccdce547270997647 10/20/2022 04:21
Jobs

origin/phab-diff-468855


CTS math_brute_force/log2 (FAIL
spirv-testing.khronos.org/cts_result/4847/
Guest
KHRONOS GROUP
CONNECTING SOFTWARE TO SILICON
Home
Dashboard
Commited
Staging
Queue

math_brute_force / log2 FAIL

Test metadata

Name	log2
Category	math_brute_force
Suite	Khronos OpenCL CTS
Suite version	80a4a833
Executable path	/test_conformance/math_brute_force/bin/test_bruteforce
Arguments	log2

Run metadata

Parent job	Job 44
Revision	a8dd7803f9a1cce... / phab-diff-471783: [SPIRV] support enqueue_kernel builtin functio...
Driver version	NEO 22.34.24023; IGC 1.0.11702.1
Date run	10/31/2022
Duration	0 minutes
Status	FAIL

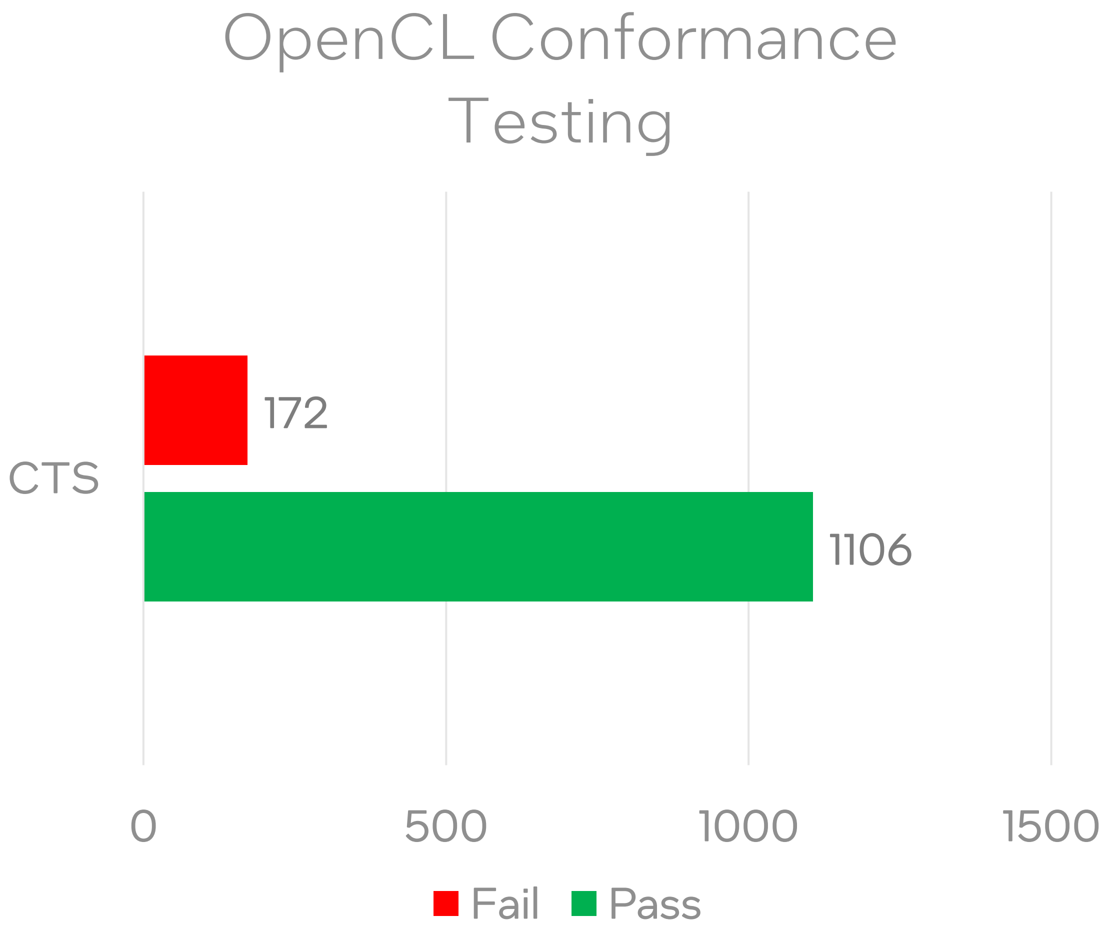
Runner output

Stdout from the runner of the corresponding OpenCL CTS test:

```
/home/bot/Workspace/cts/build/test_conformance/math_brute_force/bin/test_bruteforce log2
Test binary built Sep 8 2022 23:41:18
sizeof( void*) = 8
ARCH: x86_64
system name = Linux
```

Conformance testing and maintaining the target in-tree

- OpenCL Conformance Testing yields approximately 87% pass rate in Debug mode and 92% in Release mode (no assertions).
- Benchmarks show performance similar to the SPIR-V LLVM Translator (with –O0 input IR).



Roadmap / Future work

- Reach full OpenCL conformance
- Support opaque pointers in a way compatible with the Khronos LLVM/SPIR-V Translator
- Add full documentation: user/developer guide, describe conventions, supported intrinsics, builtin functions, and types.
- Make the backend a “non-experimental target” in LLVM
- Move builtins implementation from CodeGen to Clang