FOSDEM 2024

Challenges of supporting multiple versions of LLVM in IGC

Mateusz Belicki

purpose of this talk:

- share experience of working on a downstream compiler using LLVM as „optimization framework”
- describe engineering choices and their results
- provide some insight into how LLVM is used
- gather comments from other maintainers

what is IGC

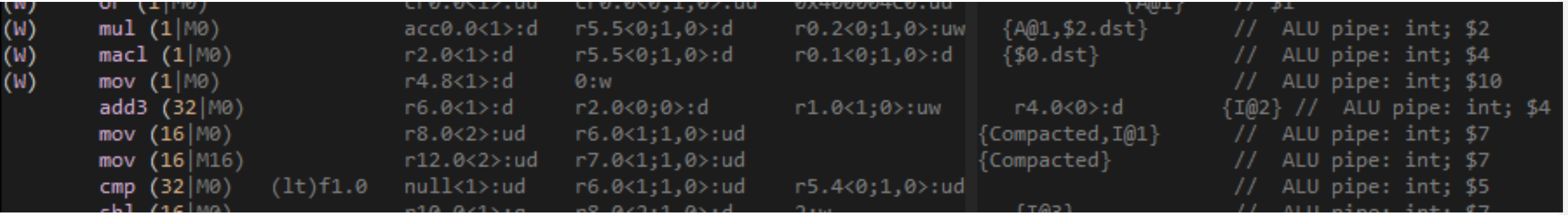
and its relationship with LLVM

what is IGC

- IGC stands for Intel Graphics Compiler:
  - a SPMD compiler for all Intel GPU targets;
  - supporting both compute (OpenCL, SYCL) and graphics API (DX, Vulkan, etc.).
- part of the userspace graphics driver:
  - library rather than a standalone executable;
  - transforms kernel/shader „bytecode” (e.g. SPIR-V) into final binary.

how IGC uses LLVM

- we take LLVM's pass manager and populate it with both custom IGC passes and generic LLVM transforms;
- LLVM codegen infrastructre is not currently used:
  - custom pass-based emitter
  - perfroms vectorization and lowering to another IR called VISA



how IGC uses LLVM

- all custom parts created using LLVM's C++ API
- why not use C API:
  - stable but limited;
  - allows building IR but does not help with transforms;
  - designed for frontends;
  - IGC defines it's own passes, needs all APIs that LLVM provides;

IGC – open source model

- distribution:
  - both closed source and open source releases;
  - sources and binary releases can be accessed on github: https://github.com/intel/intel-graphics-compiler
  - part of some GNU/Linux distros:
    - Arch, Ubuntu, Debian, etc.

- development:
  - (sadly) closed-source first
  - open source repository mirrors closed source repo
  - patches are auto-generated from closed source repo with sensitive parts removed
  - both repos in sync, open repository gets changes at the same time as closed one

IGC – supported LLVM versions

- closed source Windows compiler still on LLVM9 due to performance reasons:
  - LLVM upgrades bring both performance improvements and regressions
  - regressions usually can be addressed by changes on the IGC side, but require some effort

- open source GNU/Linux compiler currently on LLVM14 but supports older versions too:
  - preassure to keep up with LLVM version coming from OS distro;
  - some flexibility needed to support multiple OS and their LLVM versions;

supporting LLVM versions from 9 to 15

with the same codebase

supporting multiple LLVM versions – key issues:

1. changes in optimizations perfromed by generic LLVM passes:
   - can cause both performance improvements and regressions
   - sometimes passes start to emit IR unsupported by the backend
   - single most problematic pass: **InstructionCombiner**
   - those cases can ususally be addressed by transforming IR on our side before/after generic otimization pass
2. API changes:
   - more in following slides

LLVM wrapper – a way to address API changes

- LLVM version 9 to 15 changes are mostly API-level:
  - roughly the same things are possible, but either with different call or set of multiple calls
  - new types appearing in new versions (e.g. scalable vectors) can be mocked or implemented using previously avaialble
- can be handled with a wrapper layer:
  - gathers #ifdefs in a single place, so they are way easier to mantain
  - convention for wrapper functions is to be as close to new version of API as possible, this way when the support for older version is dropped the wrapper can be easily removed

LLVM wrapper

- https://github.com/intel/intel-graphics-compiler/tree/master/IGC/WrapperLLVM
- headers only
- mostly mirrors LLVM header structure
- each wrapper function maps to a function from highest currently supported LLVM version is placed in an analogous header

LLVM wrapper

- most cases are fairly simple:

```
inline llvm::Attribute getWithStructRetType
                            (llvm::LLVMContext &Context, llvm::Type *Ty) {
#if LLVM_VERSION_MAJOR <= 11
        return llvm::Attribute::get(Context, llvm::Attribute::StructRet);
#else
        return llvm::Attribute::getWithStructRetType(Context, Ty);
#endif
}
```

LLVM wrapper

- for some of them a whole implementation is needed:

```
 inline llvm::Constant *getSplatValue(llvm::ConstantVector *CV,
                                      bool AllowUndefs = false) {
#if LLVM_VERSION_MAJOR < 10
      if (!AllowUndefs)
        CV->getSplatValue();
      llvm::Constant *Elt = CV->getOperand(0);
      for (unsigned I = 1, E = CV->getNumOperands(); I < E; ++I) {
        llvm::Constant *OpC = CV->getOperand(I);
        if (llvm::isa<llvm::UndefValue>(OpC))
          continue;
        if (llvm::isa<llvm::UndefValue>(Elt))
          Elt = OpC;
        if (OpC != Elt)
          return nullptr;
      }
      return Elt;
#else
      return CV->getSplatValue(AllowUndefs);
#endif
    }
```

LLVM wrapper

- this can happen for whole classes:

```
namespace IGCLLVM {
#if LLVM_VERSION_MAJOR < 11
    class AddrSpaceCastOperator
            : public llvm::ConcreteOperator<llvm::Operator, llvm::Instruction::AddrSpaceCast> {
        friend class llvm::AddrSpaceCastInst;
        friend class llvm::ConstantExpr;

    public:
        llvm::Value *getPointerOperand() { return getOperand(0); }

        const llvm::Value *getPointerOperand() const { return getOperand(0); }

        unsigned getSrcAddressSpace() const {
            return getPointerOperand()->getType()->getPointerAddressSpace();
        }

        unsigned getDestAddressSpace() const {
            return getType()->getPointerAddressSpace();
        }
    };
#else
    using llvm::AddrSpaceCastOperator;
#endif
} // namespace IGCLLVM    }
```

LLVM wrapper

- but sometimes just a type alias is needed:

```
namespace IGCLLVM
{
#if LLVM_VERSION_MAJOR <= 10
    using llvm::CallSite;
    using CallSiteRef = IGCLLVM::CallSite;
#else
    using CallSite = llvm::CallBase;
    using CallSiteRef = IGCLLVM::CallSite&;
#endif
}
```

LLVM wrapper

- 52 headers currently including 224 wrapper functions and 32 type aliases

directory	type aliases	functions
ADT	7	15
Analysis	3	3
CodeGen	0	1
IR	15	162
MC	1	5
Option	0	2
Support	3	24
Target	1	3
Transforms	2	9

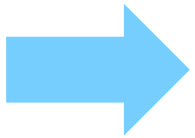
supporting future LLVM versions

and how we plan to deal with opaque pointers

opaque pointers

- LLVM14 introduces opaque pointers mode, with LLVM16 they are mandatory:

```
define i8* @test(i8* %p) {
  %p2 = getelementptr i8, i8* %p, i64
  ret i8* %p2
}
```



```
define ptr @test(ptr %p) {
  %p2 = getelementptr i8, ptr %p, i64 1
  ret ptr %p2
}
```

opaque pointers

- we unfortunately happen to rely heavily on the pointee type information:
  - for GPU targets we have types that are just passed around as pointers, e.g. samplers, textures etc.
  - presence of given type can alter code generation (e.g. implicit kernel arguments, different instructions), so we have to know exactly what is the pointee type in some cases

```
%spirv.SampledImage._void_2_0_0_0_0_0_0 = type opaque
%spirv.Image._void_2_0_0_0_0_0_0 = type opaque
%spirv.Sampler = type opaque
```

opaque pointers

- This change cannot be handled by the wrapper alone:
  - cannot deduce the pointee type from opaque pointer (at least not always and not without no runtime cost)
- Fortunately there is already a mechanism in LLVM16 that can help with this issues:
  - *„Target-extension types represent types that need to be preserved through optimization, but otherwise are not introspectable by target-independent optimizations.”*
  - `%val = alloca target("spirv.DeviceEvent")`

opaque pointers

- staged plan to move to opaque pointers:
  1. remove all places that use pointee type information, but either don't need it or can get it from elsewhere (e.g. instruction that uses the pointer);
     - basically this: https://llvm.org/docs/OpaquePointers.html#migration-instructions
  2. move to LLVM14, dropping support for older versions
  3. Internall apply TET patch:
     - patch for TET available only in LLVM16
     - the patch fortunatelly is relatively easy to port to LLVM14
  4. update all places that use pointee type information to use TET:
     - this allows us to work on TET support before upgrading to LLVM16
  5. move to LLVM16, dropping support for older versions

conclusions

conclusions

1. multiple versions of LLVM can be supported at the same time with some care;
2. even long streaches (9 – 15) are possible between versions that introduce small API changes;
3. however, there is a risk of fundamental upstream changes (opaque pointers), that may force abandoning support for older versions.


intel

target extension types

```
define target("spirv.DeviceEvent") @basic_alloc
                                        (target("spirv.DeviceEvent") %arg) {
    %val = alloca target("spirv.DeviceEvent")
    store target("spirv.DeviceEvent") %arg, ptr %val
    %ret = load target("spirv.DeviceEvent"), ptr %val
    ret target("spirv.DeviceEvent") %ret
}
```