How to *not* write a back-end compiler

Jason Ekstrand, XDC 2019

Who am I?

- Name: Jason Ekstrand
- Employer: Intel
- First freedesktop.org commit: wayland/31511d0e dated Jan 11, 2013
- Mesa commit count: 5365
- What I work on: Everything Intel but not OpenGL front-end
  - src/intel/*
  - src/compiler/nir
  - src/compiler/spirv
  - src/mesa/drivers/dri/i965

Why am I giving this talk?

- Many new contributors have joined the Mesa community
- Many new HW drivers are being added to Mesa:
  - Freedreno a2xx
  - Panfrost
  - Lima
  - Etnaviv
- They're all using NIR and actively writing their own back-end compilers
- More mature drivers have a lot of wisdom to share
  - We've already made most of the mistakes!

Everything hardware-specific should go in the back-end

Do as much lowering in NIR as practical

We used to put everything in the back-end for Intel compilers. Over the course of the last 4 years, we've moved most of our back-end lowering to NIR:

- Complex type handling (we used to handle arrays of structs!)
- I/O variable -> location slot lowering (nir_lower_io)
- Boolean clean-up on gen6 and earlier
- Handling large arrays with scratch (we still have code for this in vec4)
- Fragment input interpolation on gen11+
- Storage image format conversion
- ...

Do as much lowering in NIR as practical

- Lowering in NIR let's NIR's optimizer run on the lowered code
  - This is important for cases where lowering produces lots of ALU math
- Many passes can be re-used across drivers
  - Even though each vendor's HW is different, any individual lowering operation may be common to two or more platforms
  - Intel and ir3 share fragment input interpolation lowering
- NIR does allow HW-specific intrinsics if needed:
  - nir_intrinsic_image_deref_load_param_intel
  - nir_intrinsic_load/store_ssbo_ir3
  - nir_intrinsic_tlb_sample_color_v3d

We need an optimizer! Only the back-end knows best, right?

A brief history of optimization in Intel compilers

- Ian landed "Initial Commit. lol" for the GLSL compiler was Feb 2010
- GLSL IR optimizations start showing up around April 2010
- Eric Anholt started the Intel FS back-end in August 2010
- Matt Turner started going to town on back-end optimizations in 2013
  - Quickly discovered flat IRs work better than trees for many optimizations
- NIR landed in December of 2014 and just took over...
  - Quickly discovered that SSA is way better than our back-end for most optimizations
  - Quickly discovered that optimizing at a sligntly higher level is also better
- We've since deleted most of our complex back-end optimizations

Who needs an optimizer? NIR will do everything for me, right?

Who needs an optimizer? You need an optimizer!

Every compiler needs at least some optimizations if for no other reason than to clean up the mess left by NIR -> back-end translation.

- Copy propagation
- Dead code elimination
- Common sub-expression elimination
- Register coalescing

Who needs an optimizer? You need an optimizer!

You likely also will need some amount of optimization and lowering that is specific to the target hardware:

- Flag register handling
- SIMD width splitting (this is pretty intel-specific)
- Address register/indirect handling
- Non-ALU ops such as texturing or memory access

Current Intel compiler optimization philosophy

- Do as much optimization in NIR as possible
- Lower in NIR whenever possible so NIR can optimize the lowered code
- Back-end compilers still have some optimizations
  - NIR -> back-end translation emits lots of MOVs
  - Some lowering has to be back-end specific
  - That lowering produces a mess that needs an optimizer
- Generally, we try to keep the back-end optimizer minimal

Panfrost: A case study

The early Panfrost compiler was *not* designed with optimization in mind

- IR data structures were not designed with mutation in mind
- No unification between vector and scalar instructions
  - Everything had to be handled twice
- Data was encoded to make final binary generation easy
  - Channel mask encodings depended type of instruction and data type

In short, early Panfrost didn't have an IR so much as data structures for holding barely abstracted instructions.

Many apps don't need control flow; we can implement that later.

No, you need to think about control-flow early!

- glmark2 is not representative; real apps and the CTS *do* use control-flow
- Control-flow affects almost every aspect of your IR
  - Is it a list of instructions or a list of blocks?
  - If you're doing SSA, you now have phis
  - Scheduling needs to not move instructions across block boundaries
  - Passes like CSE can't combine things without thinking about blocks
  - You need real data-flow analysis for RA and several passes
- Yes, you need to think about loops too
  - Most of the interesting edge cases that break your entire design come from loops

“There are two ways to write a shader compiler: Without control flow, in 10 smiling days or with control flow, in 10 meh years. Except if you start without control flow and try to retcon it in it'll actually be more like 17 years and you'll be crying the whole time.”

– Alyssa Rosenzweig, Panfrost

I'm going to use static single assignment (SSA) form!

SSA is likely *not* as good an idea as you think

- SSA is great for many optimizations which is why NIR uses it
- SSA *looks* like a great idea in your back-end but...
- No hardware is SSA so you have to go out at some point
- Going out of SSA is not as easy as you think it is
  - If you don't have control-flow, you haven't thought about it
  - If you don't have loops, your out-of-SSA pass is wrong
  - If you have loops, it's probably still wrong
  - If it is right, it's probably not efficient
- There's a 90% chance you're better off using nir_convert_from_ssa()

But SSA is so useful! Why can't I have nice things?

- But SSA makes scheduling easier
  - Yes, but only pre-RA scheduling and you really need post-RA scheduling
- I want to do register allocation in SSA
  - Going out of SSA post-RA is even harder than going out of SSA pre-RA
  - Unless your allocator is *really* good at coalescing, you'll get piles of moves
  - Going out-of-SSA post-RA generally generates more moves than pre-RA
- But if I don't have SSA, I have to do data-flow analysis
  - You have to data-flow analysis anyway
  - SSA just splits live ranges and makes things a *little* bit easier

Freedreno: A case study

The early Freedreno compiler was SSA

- Freedreno used its own out-of-SSA pass instead of nir_convert_from_ssa()
- Arrays were implemented with NIR variables
- And then Rob implemented loops...
- As of now, Freedreno calls nir_convert_from_ssa(phi_webs_only=true)
  - Freedreno is still SSA as much as possible
  - All arrays and phi webs now use nir_register
  - All nir_registers are still treated like arrays by the back-end

Keep it simple: 32-bit types are all you need to get started.

Been there, made that mistake....

The i965 back-end compilers' 32-bit assumptions invaded every corner of the IRs

- Didn't have a concept of the size of a type
- Register allocation is in terms of whole 32B registers (32-bit SIMD8)
  - SIMD8 16-bit values didn't actually save us any space
- Strides were mostly reserved for special edge cases
  - Passes didn't handle them properly and broke when strides became common
- Instructions with a mix of type sizes have more HW restrictions
- Passes assumed !partial_write == read_matches_write

Been there, made that mistake....

The i965 back-end compilers' 32-bit assumptions invaded every corner of the IRs

- Anything strided or < 32B was considered a partial write
  - Copy propagation couldn't propagate 16-bit data in SIMD16 or 8-bit anywhere
  - Strides imposed by HW restrictions also prevented copy-prop
  - Data-flow analysis thought all these values were live for the entire program
    - This caused piles of RA fail and spilling on some Vulkan CTS tests
    - We added an UNDEF instruction to help with this
  - Other passes fall over on 8 and 16-bit data for the same reasons

How do we fix this mess?
Throw it away and start over?

IBC: **I**ntel **B**ack-end **C**ompiler
We're trying to do it right this time!

IBC: A new **I**ntel **B**ack-end **C**ompiler

- Started about 6 months ago (first serious commit dated March 13, 2019)
- Mostly from-scratch code-base
  - Shares final codegen (assembler) with the old back-end
- Trying to build it right this time:
  - Core IR design based heavily on NIR
  - Attempts to find balance between being close to hardware and nice to work with
  - Handled SIMD32 compute and fragment shaders on day 1
  - Handled scalar (non-divergent) values and 8 and 16-bit types on day 1
  - Flag and accumulators are allocated so no more unnecessary scheduling barriers

Focus on the weird corner cases in your hardware to ensure you get the IR design right.

IBC: A new **I**ntel **B**ack-end **C**ompiler

The git history of gitlab.freedesktop.org/jekstrand/ibc-tests is quite informative:

- Add the first working CS test
- Add a test for ieq and bcsel
- Add 8 and 16-bit math tests
- Add a simple test runner script
- Add a test for boolean OR
- Add a test for subscripting
- Add a simple subgroup broadcast test
- Add a test for fsign
- Add subgroup tests
- Add clustered subgroup tests
- Add a simple if statement test
- Convert ieq-bcsel to SPIR-V
- Convert bool-or to SPIR-V
- Add a simple phi test
- Add some comparsion tests
- Add a ballot test

IBC: A new **I**ntel **B**ack-end **C**ompiler

IBC is coming along rather nicely but is still an early prototype

- Supports vertex, fragment, and compute stages
- Supports ANV and iris; no i965 yet
- Passing ~95% of the Vulkan CTS
- Compiling ~90% of shader-db with iris
  - Currently about +15% instructions; likely mostly MOVs around SENDs
- Benchmark/perf: Unknown
- https://gitlab.freedesktop.org/jekstrand/mesa/tree/ibc

IBC: A new **I**ntel **B**ack-end **C**ompiler

Moving forward, we're focusing on performance followed by conformance

- IBC has a scheduler but it needs tuning
  - It can move flag and accumulator writes past each other!
- Implement spilling and tune scheduling to avoid spilling
- Investigate why IBC is generating piles of MOVs and eliminate them
- Benchmark and analyze any perf differences we see
- Implement all the other shader stages and features
  - Currently crashes when you hit an unsupported feature
  - This is an intentional choice and will not change in the foreseeable future

A quick recap:

- Do as much lowering and optimization in NIR as practical
- Write enough of an optimizer to clean up NIR -> back-end translation
- Don't do SSA in your back-end unless you really know what you're doing
- Don't avoid the complexity off; attack it head-on
  - Control-flow, including loops!
  - All the types and bit sizes you need
  - Flags and whatever other oddities your HW needs
- Think long-term and focus on getting your IR right, not getting glmark2 running