Writing compilers in Rust?

Faith Ekstrand

XDC 2023
About me

- Faith Ekstrand (@gfxstrand@mastodon.gamedev.place)
- First freedesktop.org commit: wayland/31511d0e, Jan 11, 2013
- Worked at Intel from June 2014 to December 2022
  - NIR, Intel (ANV) Vulkan driver, SPIR-V → NIR, ISL, other Intel bits
- Now at Collabora since January 2022
  - Work across the upstream Linux graphics stack, wherever needed
  - Currently the lead developer / maintainer of NVK
Introducing NAK: The Nvidia Awesome Kompiler
NAK: The Nvidia Awesome Kompiler

- Brand new back-end compiler for NVIDIA hardware
- Written in Rust
- Tries to be a model NIR user
  - NIR passes are written in C
  - Lower in NIR, keep the back-end simple
- Fully SSA until register allocation
  - This register allocator actually works! 😅
Why Rust?
Why Rust?

- C is kinda terrible. We all know this...
  - Those of us who prefer C know it best
- C++ is also terrible.
- Rust is less terrible?
  - Powerful type system that doesn’t rely on virtual dispatch
  - Has a large, well thought out standard library
  - Borrow checker that catches real bugs
Wait, you don’t hate the borrow checker?!?

Nope! Once you learn to work with it, the borrow checker becomes a code review buddy, pointing out serious bugs.

Structuring your code to be borrow-checker-friendly often results in better, more obviously correct code.
Why is NAK a good candidate for Rust?

• NAK is self-contained and doesn’t need to call out
  - There are a few utilities we could maybe use
  - Mostly, it just consumes NIR and produces a blob of bytes

• The Rust/C interface for NAK is 7 functions
  - Not much to bindgen

• We only really need to read NIR
  - More on that...
Can you write a compiler in it?
But can you write a compiler in it?

• That’s the question I set out to answer with NAK.
• And... Yes, yes you can!
• You do need to make friends with the borrow checker...
  - Lots of hash maps
  - Lots of SoA where you might do AoS in C/C++
Challenge 1: Reading NIR from Rust

- Not too bad if the Rust NIR usage is read-only
- NIR lowering passes and optimization loop are written in C
- Rust NIR wrappers take a \&nir_shader (not mut!)
  - Uses traits to add methods to NIR types
  - You can do nb.iter_instrs() on a \&nir_block
  - Iterator for exec_list (thanks, Karol)
  - srcs_as_slice() allows safe srcs[] array access
  - NirSrc::as_uint() \rightarrow Option<u64> gets a u64 if a source is const
Challenge 2: Instruction sources

There are a lot of things we want:

- **Clear meaning of sources**
  - This gets worse when there are indirects

- **Fast, generic access to sources**
  - Passes like copy propagation don’t care what most sources mean

- **Avoid extra array allocations**

- **Special sources like predicates and indirects**

- **Source modifiers** (because, of course there are...)
Prior art: NIR

- Sources are in arrays
- Depend on documentation to know what’s what
  - We all know just how well that works...
- We avoid extra allocation by using unsized arrays
  - Rust can’t do that!
- Everything is an SSA def or const_index
  - No modifiers
  - No types
- We depend on nir_validate to ensure correctness
Prior art: ACO

- Each instruction is its own type
  - This helps a bit with metadata like rounding modes

- Sources are still arrays
  - Each instruction documents source meanings

- Avoid extra allocation using aco::span<T>
  - Basically, it’s unsized arrays except way better and you can have more than one
  - O(1) source access
  - Supports all the usual C++ iterator stuff
  - Can have large numbers of sources AND destinations
  - Not implementable in Rust...
How to do this in Rust...

If I could have everything I want....

- Want to use Rust enums...
  - Rust enums are tagged unions
  - Safer than NIR’s enum and pointer cast

- Want descriptive names for sources

- Want typesafe metadata

- Want Rust’s type system to check stuff for me

- Want generic O(1) access to sources
You can't always get what you want
But if you try sometimes, well, you just might find
You get what you need...
Let's look at some NAK code....

```rust
#[repr(C)]
#[derive(SrcsAsSlice, DstsAsSlice)]

pub struct OpSel {
    pub dst: Dst,

    #[src_type(Pred)]
    pub cond: Src,

    #[src_type(ALU)]
    pub srcs: [Src; 2],
}
```
Let’s look at some NAK code....

```rust
pub struct OpSel {
    pub dst: Dst,
    pub cond: Src,
    pub srcs: [Src; 2],
}
```

**Named Destination**
Let’s look at some NAK code....

```rust
#[repr(C)]
#[derive(SrcsAsSlice, DstsAsSlice)]

pub struct OpSel {
    pub dst: Dst,
    #[src_type(Pred)]
    pub cond: Src,
    #[src_type(ALU)]
    pub srcs: [Src; 2],
}
```
Let’s look at some NAK code....

```rust
#[repr(C)]
#[derive(SrcsAsSlice, DstsAsSlice)]
pub struct OpSel {
    pub dst: Dst,
    #[src_type(Pred)]
    pub cond: Src,
    #[src_type(ALU)]
    pub srcs: [Src; 2],
}
```

Type Decorations
Let's look at some NAK code....

```
2771  #[repr(C)]
2772  #[derive(SrcsAsSlice, DstsAsSlice)]
2773  pub struct OpSel {
2774      pub dst: Dst,
2775  
2776      #[src_type(Pred)]
2777      pub cond: Src,
2778  
2779      #[src_type(ALU)]
2780      pub srcs: [Src; 2],
2781  }
```
Let’s look at some NAK code....

```rust
pub trait SrcsAsSlice {
    fn srcs_as_slice(&self) -> &[Src];
    fn srcs_as_mut_slice(&mut self) -> &mut [Src];
    fn src_types(&self) -> SrcTypeList;
}

pub trait DstsAsSlice {
    fn dsts_as_slice(&self) -> &[Dst];
    fn dsts_as_mut_slice(&mut self) -> &mut [Dst];
}
```
Let’s look at some NAK code....

```rust
#[derive(Display, DstsAsSlice, SrcsAsSlice,
    pub enum Op {
        FAdd(OpFAdd),
        FFma(OpFFma),
        FMnMx(OpFMnMx),
        FMul(OpFMul),
        MuFu(OpMuFu),
        FSet(OpFSet),
        FSetP(OpFSetP),
        FSwzAdd(OpFSwzAdd),
        DAdd(OpDAdd),
```
Let's look at some NAK code.

```python
113 if self.is_instr_live(instr) {
114     if let PredRef::SSA(ssa) = &instr.pred.pred_ref {
115         self.mark_ssa_live(ssa);
116     }
117
118     for src in instr.srcrefs() {
119         self.mark_src_live(src);
120     }
121 } else {
122     self.any_dead = true;
123 }
```
This gets us most of what we want

- Good use of Rust enums
- Descriptive names for sources
- Typesafe metadata
- Rust’s type system to check stuff for me
- Generic O(1) access to sources

- Works with all Rust’s iterator stuff
- Only per-op code is a lookup table
Let’s look at some NAK code....

```rust
#[repr(C)]
#[derive(SrcsAsSlice, DstsAsSlice)]
pub struct OpFAdd {
    pub dst: Dst,
    #[src_type(F32)]
    pub srcs: [Src; 2],
    pub saturate: bool,
    pub rnd_mode: FRndMode,
}
```

We can do metadata, too
Challenge 3: Representing values

There are a lot of things we want:

- SSA and registers
  - We need registers for final code-gen

- Vectors and 64-bit values
  - This one has hidden and very subtle SSA-based RA implications

- Predicates and GPRs

- Uniform and non-uniform values

- Immediates and cbufs
SSA Values

• Each SSAValue represents a single 32-bit value
  - Has a RegFile (GPR, Pred, UGPR or UPred), and 29-bit index
  - Packs into 32 bits so it’s cheap to copy around
  - Implements Eq+Hash so it can be a HashMap key

• Each SSARef contains 1-4 SSAValues
  - Packs into 128 bits so it’s cheap(ish)
  - Implements Deref<[SSAValue]>

• Register allocation automatically collects into consecutive register ranges as-needed
Other value types

• A **RegRef** represents a register range
  - Has a RegFile, an index, and a number of components.

• A **CBufRef** represents a constant buffer value
  - 32 or 64 bits, depending on opcode

• An **Imm32** represents a 32-bit immediate
  - Or the top 32 bits of a 64-bit source

• Special case immediates: **Zero, True, and False**
  - Often allowed when Imm32 is not
Sources and Destinations

```rust
#[derive(Clone, Copy, Eq)]
pub enum SrcRef {
    Zero,
    True,
    False,
    Imm32(u32),
    CBuf(CBufRef),
    SSA(SSARef),
    Reg(RegRef),
}
```

```rust
#[derive(Clone, Copy)]
pub enum Dst {
    None,
    SSA(SSARef),
    Reg(RegRef),
}
```
Challenge 4: Instruction lists

- NIR uses linked lists of instructions
  - O(1) insertion, intrusive so no extra allocation, they’re great!
  - Rust really doesn’t like linked lists...
- Proper container types are essential to Rust’s safety model
- We use `Vec<Box<Instr>>`
  - Everything is safe. Yay!
  - You can’t insert in the middle. Booo...
- The biggest challenge is mutability
  - You can’t look at one element while modifying another (at least not easily)
Mutability challenges

Vec<T> doesn’t let you have mutable references to multiple elements simultaneously.

There are a few workarounds:

- `slice::split_at_mut()` to split the slice
- Use indices like you would pointers and `v[i]` everywhere
  - This gets sketchy fast!
- Re-structure your pass to avoid mutability
A safe pattern: Map

- `map_instrs()` takes a callback mapping an instruction to zero or more instructions.
- Provided by Shader, Function, and BasicBlock
- Most simple optimization or lowering passes use `map_instrs()` to avoid mutability headaches
A safe pattern: Map

```rust
pub fn lower_vec_split(&mut self) {
    self.map_instrs(|instr: Box<Instr>, _| -> MappedInstrs {
        match instr.op {
            Op::INeg(neg) => MappedInstrs::One(Instr::new_boxed(OpIAdd3 {
                dst: neg.dst,
                srcs: [Src::new_zero(), neg.src.ineg(), Src::new_zero()]
            })),
            _ => MappedInstrs::One(instr),
        }
    }
```

More complicated passes hand-roll...

```rust
let mut instrs = Vec::new();
for (ip, mut instr) in bb.instrs.drain(..).enumerate() {
    match &mut instr.op {
        |
        |
    }
    instrs.push(instr);
}
bb.instrs = instrs;
```

It’s not ideal but it works
The gather/modify pattern

Most passes happen in two separate steps:

• Step 1: Gather information about SSA values
• Step 2: Transform the IR based on the gathered information

• This keeps Rust’s borrow checker happy...
• And it prevents bugs!
  - Lots of NIR bugs have crept in because of accidentally looking at the IR you’ve already modified and assuming it’s the original.
Working with SSA values

• **HashMap is your friend…**
  - Gather pass builds a HashMap<SSAValue,Data>
  - Modify pass uses the map to update instructions

• **Examples:**
  - Dead code builds a HashSet<SSAValue> of live SSA values
  - Copy prop builds a HashMap<SSAValue,Copy> of copies
Challenge 5: Control-flow graphs

- Graphs are a PITA with Rust
  - You inevitably end up with tables and indices
- Hide all the insanity!
- CFG<T> is a generic container type
  - NAK uses CFG<BasicBlock> but you can use any type
  - Stores nodes and edges (each node has &[usize] preds and succs)
  - Contains dominance and loop nesting information
  - Re-orders by reverse post-order DFS
  - Implements Deref<[T]> so you can iterate it
Challenge ∞: Spilling and RA

Yeah... This presentation is already long enough. 😅
Final thoughts: Do I like it?

Yes! I’ve loved working on NAK in Rust

- Rust enums (tagged unions) are awesome
- Proc macros are tricky but really useful
- Rust’s traits and generics work well
- I like having a standard library
- Over-all, abstractions are just as powerful but more explicit in Rust than with C++
Final thoughts: Advice for others

- HashMap<K, V> is your friend
- Implement From<T> for everything
- The borrow checker is your friend, not your enemy
  - Re-structure your code to be borrow-checker friendly
  - Don’t just Rc everything. Interior mutability isn’t as cool as it looks
Thank you!
We are hiring
col.la/careers
There are five styles built into the template. To apply:

- Highlighting the text
- Press TAB to demote to the next level; or
- Press SHIFT + TAB to promote a level
There are five styles built into the template. To apply:

- Highlighting the text
- Press TAB to demote to the next level; or
- Press SHIFT + TAB to promote a level
Developing better technologies
Developing better technologies
Slide content paragraphs
With or without bullets
<table>
<thead>
<tr>
<th>Date</th>
<th>2010</th>
<th>2011</th>
<th>2012</th>
<th>2013</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
</tr>
<tr>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
</tr>
<tr>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
</tr>
<tr>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
</tr>
<tr>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
</tr>
<tr>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
<td>Content</td>
</tr>
</tbody>
</table>