

Kraid, a new compiler for Panfrost

XDC 2026

Faith Ekstrand

Lorenzo Rossi

Christoph Pillmayer



COLLABORA

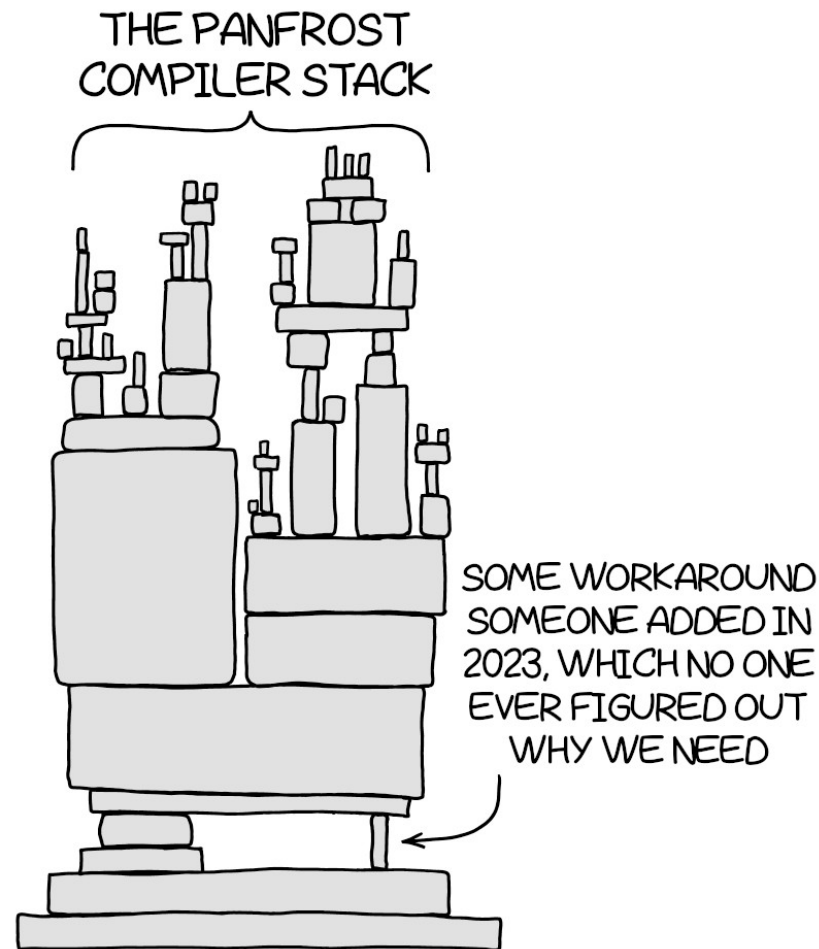
Open First



Why do we need a new compiler?

Technical debt

- It's a Bifrost compiler
- Valhall support was bolted on
 - Valhall is treated as a weird Bifrost extension
- And then we tried to add v15...
- It's a Gallium compiler
 - Assumed "vanilla" GLSL-style NIR
 - Vulkan was bolted on
- Maintained under time pressure



Valhall on Bifrost

- The whole compiler is built around bifrost/ISA.xml
 - Includes instruction definitions, disassembly formatting, and binary encoding
- valhall/ISA.xml is an extension of the Bifrost ISA
 - Includes instruction definitions, formatting, and some encoding
 - Most of the encoding is done separately in va_pack.c
- Everything follows Bifrost conventions
 - 64-bit sources use 2 sources



Built for Gallium and GLSL

- Attempts to consume “vanilla” NIR with little lowering
- Makes NIR → bi overly complex
 - Open-coded constant folding and back-end ALU for on-the-fly I/O lowering
- Limits how much NIR can optimize for us
- Drivers have to work with GLSL-level abstractions
 - Blend shaders use framebuffer fetch
 - Not everything fits into a GLSL paradigm

Technical debt keeps piling up

- The team's priority has been Vulkan, not the compiler
 - People get their feature working and move on
 - We haven't had much time for maintenance tasks
- Old decisions coming back to bite us
 - We split vectors in NIR → bi and try to keep everything 32-bit
 - BI_SWIZZLE accidentally conflated swizzling and widening
- No one knows how Bifrost RA works



Let's talk about **BI_SWIZZLE_***



**IT'S A
TRAP**

The pitfalls of BI_SWIZZLE_*

- These instructions all interpret r1 differently
 - S16_T0_F32 r3 r1.h0
 - IADD.v2i16 r3 r2 r1.h00
 - IADD.s32 r3 r2 r1.h0
 - IADD.u32 r3 r2 r1.h0
 - FADD.f32 r3 r2 r1.h0
- They're all BI_SWIZZLE_H00
- We don't have enough metadata to tell them apart
- This makes safely optimizing swizzles nearly impossible



Kraid design principals

Kraid design principals

- A back-end compiler should only do a few things:
 - Instruction selection
 - Swizzle and source/destination modifier propagation
 - Register allocation
 - Scheduling
 - Encoding
- Leave the optimization heavy lifting to NIR!

Kraid design principals

- Simple, focused design
- Correctness first
 - We don't want to repeat our BI_SWIZZLE mistake
- Separate IR definition from per-generation encoding
- Use XML from Arm for instruction encoding
- Hardware unit tests, similar to NAK
- SSA-based register allocation



Doing more in NIR...

The IR crossroads

- NIR $\rightarrow$ BIR conversion too complex
- Lots of hand-chiseled immediate folding overflow checks, instruction selection, ecc...



How to transfer

1) Create new vendored intrinsics in NIR

- As close as possible to what the hardware offers

2) Lower generic intrinsics to vendored ones in NIR

- All complexity goes here: version differences, instruction selection, code paths...

3) Delete the old path

4)... Profit!

Benefits of NIR

- We can reuse the more advanced optimizations of NIR
 - Folding, sinking, algebraic, copy-prop...
 - New optimizations using NIR utilities
- You always know what hardware does
- Keep complexity confined in a pass
- Your backend gets simpler (and easier to swap!)

Example: Blend shaders (!39367)

- Hardware cannot do complex Blend operations
- It instead has building blocks: LD_TILE / BLEND / ST_TILE
 - Complex blend operations must be broken down
 - Emulated through “blend shaders”:
 - LD_TILE
 - manually blend
 - write back result
- How do we write blend shaders?



Example: Blend shaders (Previously)

- 1) Create shader loading inputs and use `nir_store_output`
- 2) Call `nir_lower_blend`
 - Lowers the complex blend in `load_output` -> ops -> `store_output`
- 3) Rewrite `load_output/store_output`
 - Into other abstract NIR intrinsics
- 4) Backend translates those into `LD_TILE / BLEND`

Example: Blend shaders (Now)

Create blend shader using hardware intrinsics directly

1) Don't use `nir_lower_blend`

- Re-use internal bits to blend colors

2) Emit `ld_tile_pan / blend_pan` directly

3) Backend does nothing interesting



Example: Blend shaders (Benefits)

- We always know what hardware instruction we're using
- Later (WIP) optimization for multisample:
 - Run fragment shaders only once
 - Blend for each sample in a for-loop
 - Way easier now

Example: Varying loads

- Hardware has 5 families of LD_VAR*
 - Immediate or not
 - Read descriptors or have them inlined
 - Flat, special...

Example: Varying loads (Previously)

- Before (in the backend):
 - Check versions
 - Check if immediates fit
 - Immediate-fold manually (fragile)

Example: Varying loads (Now)

- After:
 - NIR pass only handles version differences
 - NIR Emits raw instructions that map 1-to-1 with HW
 - Immediate folding done by NIR
 - Backend only cares about immediate variants
 - NIR has more passes to fully leverage the hardware
 - Easier mediump lowering!

There can't be too many... right?

There can't be too many... right?

- FS outputs to Blend !39244
- Tile loads/stores (and PLS) !39367
- VS out / FS in varyings !40391
- Texture lowering !41036
- Texture/image queries !41352
- Image intrinsics !41391
- VS attributes !43431
- FS coordinates !43706



Back-end design

“A back-end compiler is pretty much just a fancy register allocator”

—Faith, probably

Modeling registers for Mali

- Registers are nominally 32-bit scalars



Modeling registers for Mali

- Registers are nominally 32-bit scalars
- Some instructions treat the 32-bits as a vector:
 - `FADD.v2f16 r0 r1.abs r2.h00`
 - `LSHIFT.v4u8 r0 r1 r2.b1`

Modeling registers for Mali

- Registers are nominally 32-bit scalars
- Some instructions treat the 32-bits as a vector: v2f16, v4i8
- Some instructions only read or write 8 or 16 bits
 - `LOAD.i8 r0.b0 r2 #48`
 - `STORE.i16 r0.b0 r2 #48`
 - `LSHIFT.u32 r0 r1 r2.b1`

Modeling registers for Mali

- Registers are nominally 32-bit scalars
- Some instructions treat the 32-bits as a vector: v2f16, v4i8
- Some instructions only read or write 8 or 16 bits
- Most ALU ops have swizzles that shuffle bytes or up-cast
 - `FADD.v2f16 r0 r1.neg r2.h00`
 - `FADD.f32 r0 r1.neg r2.h00`

Modeling registers for Mali

- Registers are nominally 32-bit scalars
- Some instructions treat the 32-bits as a vector: v2f16, v4i8
- Some instructions only read or write 8 or 16 bits
- Most ALU ops have swizzles that shuffle bytes or up-cast
- Most ALU ops allow a half-word write mask
 - `FADD.v2f16 r0.h0 r1.h00.neg r2.h00`



Modeling registers for Mali

- Registers are nominally 32-bit scalars
- Some instructions treat the 32-bits as a vector: v2f16, v4i8
- Some instructions only read or write 8 or 16 bits
- Most ALU ops have swizzles that shuffle bytes or up-cast
- Most ALU ops allow a half-word write mask
- 64-bit ops use pairs of 32-bit registers



The challenge: split/collect

There are a few different options in this design space:

- SSA values are vectors (the NIR design):
 - Straightforward for instruction selection, encode, and register allocation
 - You end up with piles of SPLIT and COLLECT that have to be managed
- SSA values are scalars (the NAK design):
 - No SPLIT and COLLECT instructions to manage
 - SPLIT is free and COLLECT becomes an RA constraint problem



Or we could ignore all that and
do it differently...

The kraid design: A hybrid approach

- An `SSAValue` in kraid is a scalar:
 - It can be 8, 16, or 32 bits
 - We have `MKVEC.v2i8/v4i8/v2i16` to collect
 - Source swizzles can “split” on-the-fly
- An `SSARef` is a vector of `SSAValue`:
 - “Collect” is handled as an RA constraint like in NAK
- Only 32-bit scalars can be put in a vector `SSARef`!



The kraid design: A hybrid approach

- For any bit size, there is one unique layout
 - ≤ 32 bits $\rightarrow$ scalar
 - > 32 bits $\rightarrow$ vector
 - We have no 24 or 48-bit type

The kraid design: A hybrid approach

- For any bit size, there is one unique layout
- We do need MKVEC to collect sub-32-bit values
 - This isn't common since most sub-32-bit is vectorized
 - Most MKVEC can be folded into a swizzle somewhere
 - RA tries to get rid of MKVEC when it can
 - It's all very best-effort

The kraid design: A hybrid approach

- For any bit size, there is one unique layout
- We do need MKVEC to collect sub-32-bit values
- We can implement 8 and 16-bit scalar ALU by masking
 - `FADD.v2f16 r0.h0 r1.h11.abs r2.h00`
 - `LSHIFT.v4u8 r0.h0 r1.b1111 r2.b2222`
 - For 8-bit, sometimes we instantaneously use 16 bits.



The kraid design: A hybrid approach

- For any bit size, there is one unique layout
- We do need MKVEC to collect sub-32-bit values
- We can implement 8 and 16-bit scalar ALU by masking
- Vectors (> 32 bits) are collected by RA
 - We have no COLLECT instructions outside of MKVEC
 - 64-bit ops are heavily dependent on RA



The kraid design: A hybrid approach

- For any bit size, there is one unique layout
- We do need MKVEC to collect sub-32-bit values
- We can implement 8 and 16-bit scalar ALU by masking
- Vectors (> 32 bits) are collected by RA
- Is this a good idea?
 - Shader-db says it's working out just fine...



Conquering swizzles



Swizzles on Mali GPUs

- Each source has a data type
 - i32, v2f16, v4i8, v2i8, etc.
 - Determines both the bit size and number of components consumed
- The swizzle maps source components to bytes
 - For v2f16, you have: `.h01 .h00 .h11 .h10`
 - For v2u8, you have: `.b00 .b01 .b03 .b32` etc.
 - For v4u8, you have: `.b0000 .b1111 .b2310 .b0123` etc.
- For 16, 32, and 64-bit, some swizzles also do type conversion
 - `FADD.f32 r0 r1 r2.h0`

Modeling swizzles

- The ISA names are relative to the type
 - For f16, .h0 is a swizzle
 - For f32, .h0 widens half to float

Modeling swizzles

- The ISA names are relative to the type
- BI_SWIZZLE conflates things even worse

Modeling swizzles

- The ISA names are relative to the type
- BI_SWIZZLE conflates things even worse
- Kraid's `Swizzle` struct does not
 - Each `Swizzle` uniquely refers to one way of viewing the source data
 - `Swizzle` encodes both byte shuffles and sign/zero/float extension
 - You can know what a `Swizzle` does without looking at the data type!

Modeling swizzles

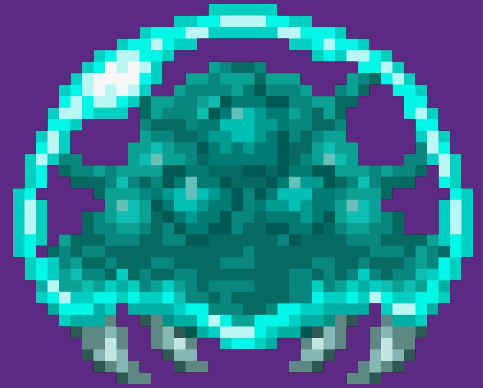
- The ISA names are relative to the type
- BI_SWIZZLE conflates things even worse
- Kraid's Swizzle struct does not
- Kraid's Swizzle is also easily composable
 - This is useful for things like copy propagation

Modeling swizzles

- The ISA names are relative to the type
- BI_SWIZZLE conflates things even worse
- Kraid's Swizzle struct does not
- Kraid's Swizzle is also easily composable
- Data types are used to validate swizzles



We've had almost no
swizzle bugs with Kraid!





COLLABORA

A new ISA XML from Arm

ISA Description – Old

- Written by hand
- Partially based on RE
- Doesn't contain all required info for de-/encode
- No assembly syntax

ISA Description - New

- genxml(-ish) format
 - `</instr arch=x unit=y ...>`
 - `</enum ...>`
- Contains everything
 - Encoding
 - Syntax
 - DataTypes



Rust Interface

- Everything parsed into one ISA struct
 - Supported modifiers + swizzles
 - Execution units
 - Variants
- HW datatypes → Rust enums
- HW instructions → Rust structs

Rust Interface

- Un-/Pack via TryDecode/TryEncode
 - Does the right thing based on architecture
 - Returns an Error for invalid inputs
- Transparent handling of non trivial encoding
 - Virtual fields: e.g. `offset` => `offset_hi`, `offset_lo`
 - Derived field: opposite of virtual field

Encoder - Old

- Coupled to the IR (IR ~= Bifrost)
- Valhall properties for most Bifrost instructions
- `va_swap_12`
- Valhall packer with lots of hard-coded offsets

Encoder - New

- `impl From<IREnum> for <HWEnum>`
- Fill struct with properly typed fields per instruction
- Encode with `HWInstr::TryEncode`
- Validates encoded instruction automatically

Decoder

- Fully defined by the xml
- Iteratively group instruction by shared fixed bits
 - LOAD, STORE, FMA, ...
- Resolve aliased instruction by #fixed bits
 - AND, LSHIFT, LSHIFT_AND



Disassembler

- Syntax from XML
- Usage like llvm-mc
 - `hexdump <binary> | pan-dis --arch 13`
 - `0x69400028002005 → ATOM1.i64.slot0.ainc [r4:r5].zext, #0x20`
- Reports encoding errors
 - `0x69400128102005 → Invalid (Unsupported MessageSlotIndexM)`



Results

Shader-DB Fossils

Totals from 584 (98.48% of 593) affected shaders:

Instrs: 106771 -> 94067 (-11.90%)

FMA cycles: 358.617179 -> 357.171872 (-0.40%)

CVT cycles: 368.17971 -> 275.4531 (-25.19%)

SFU cycles: 369.0625 -> 356.4375 (-3.42%)

Load/Store cycles: 7750.0 -> 7435.0 (-4.06%)

Spill instructions: 227 -> 222 (-2.20%)

Fill instructions: 606 -> 296 (-51.16%)

Spill cost: 14690 -> 8980 (-38.87%)

CL fp16

```
$ clpeak --opencl --compare ./clpeak-no-kraid.csv --mixed-precision-compute
```

```
Backend: OpenCL
```

```
Platform 0: rusticl
```

```
Device 0: Mali-G725 MC11 (Panfrost)
```

```
Driver version : 26.3.0-devel (git-93b1f69448)
```

```
Mixed-precision compute fp16xfp16+fp32 (GFLOPS)
```

```
mp      : 1320.01 (was 1009.49, +30.8%)
```

```
mp2     : 1243.12 (was 1066.33, +16.6%)
```

```
mp4     : 1096.21 (was 811.45, +35.1%)
```

```
mp8     : 1363.14 (was 1013.28, +34.5%)
```

```
mp16    : 1359.17 (was 1014.16, +34.0%)
```

App FPS

- Candy Crush +45%
- Star Rail +6%
- GFXBench Aztec +6%
- CoD Mobile +3%
- GFXBench Trex +3%
- Apex Legends +2%
- Genshin Impact +2%

Faith

NYPD



Alyssa



I rewrote your compiler!



Thank you!



We are hiring
col.la/careers



COLLABORA

Open First