A year of ACO

from prototype to default

Timur Kristóf

2020

XDC
2020

Linux Open-Source
Graphics Drivers Group

Table of Contents

1. Our ACO story

2. How ACO works

3. Changes to ACO so that it can be the default compiler in RADV
   - New hardware support
   - Small bitsizes
   - Geometry shaders
   - Tessellation

4. Future plans

VALVE


Our ACO journey
VALVE

ACO motivation

VALVE

Fluid gaming
We'd like to give gamers a smooth, stutter-free experience, so we prioritize compilation speed.

Developed in mesa
Issues can be fixed within mesa releases, independently of the schedule of other projects.

Runtime performance
Good divergence analysis allows us to better optimize runtime performance.

Where we were last XDC

VALVE

Supported stages @ XDC 2019

Started with PS only.
Then added CS and VS.

VALVE

Supported hardware @ XDC 2019

Started on Polaris (GFX8)
Then Vega was added (GFX9)

VALVE

ACO 2019-2020 by numbers

VALVE

300+ merge requests
5 current contributors
15 supported shader stages
5 supported HW generations (all GCN/RDNA)

Full feature parity

- VK exts supported by RADV work with ACO
- Full conformance on newer HW

No benchmarks this time, but...

Runtime:

Phoronix has awesome benchmarks.

Compile time:

Consistent with what was shown in XDC2019.

VALVE

Where ACO fits in

VALVE

RADV/ACO shader compilation

VALVE

SPIR-V to NIR
Parse the SPIR-V shader into NIR.

NIR lowering and optimization
Lower the IR to be suitable for consumption by RADV / ACO.

ACO shader compilation
Compile the shader into a program the HW can execute.

Recap: how ACO works

VALVE

ACO compilation phases 1-3

VALVE

1. Instruction selection
Based around the NIR divergence analysis, emits ACO IR which is SSA.

2. Value numbering, optimization
Common subexpression elimination, constant propagation, instruction combining.

3. Setup reductions, insert exec mask
Ensure reductions work. Add instructions that control SIMD lanes.

ACO compilation phases 4-6

VALVE

4. Live variable analysis
Calculate register need, used for spilling and scheduling.

5. Spilling
Lower to CSSA, then try to decrease register usage.

6. Instruction Scheduling
Move loads as high up as possible.

ACO compilation phases 7-9

VALVE

7. Register Allocation
Works on SSA, emits shuffle code.

8. SSA Elimination
Insert copies to match phi node semantics.

9. Lower to HW instructions
Replace pseudo instructions with machine instructions.

ACO compilation phases 10-11

VALVE

10. Insert wait states and NOPs, resolve hazards
Ensure correct behaviour of memory instructions, eliminate HW hazards.

11. Assembler
ACO IR is already almost GCN/RDNA ASM, so only need to encode.

How we had to change it to make it the default

VALVE

New hardware support

New (to us) hardware support

Generation	Code names	Year
GFX6	Pitcairn, Oldand, etc	2012
GFX7	Hawaii, Bonaire, etc	2013
GFX10	Navi	2019

New (to us) hardware support

- Different assembly encodings
- Some instructions missing on old HW or removed from new HW
- Missing subgroup features on old HW
- New (to us) hazards
- Very helpful community

Small bitsizes

Small bitsizes

We had a lot of ideas.

Small bitsizes

We had a lot of bad ideas.

Lots of edge cases, more exceptions than rules.

Small bitsizes

Then we agreed to push the heavy lifting into RA, but...

Small bitsizes

Maintaining the register file

- Previously:
  256 VGPRs + 106 SGPRs = 362 registers
- Now:
  We manage every byte of every register

Yes, every single byte.

Small bitsizes

RA shuffles are now crazy

- Previously:
  We had to move contents of registers
- Now:
  We move individual bytes between registers
  Sometimes even within the same register

Small bitsizes

HW support is inconsistent

- Some new HW: keeps high bits
- Other new HW: overwrites high bits
- Old HW: manual coding needed

Geometry shaders

Geometry shaders

- New intrinsics
- I/O: LDS, VRAM
- Figure out merged shaders
- GS copy shader

Geometry shaders: 4 new stages

- `vertex_es` (GFX6-8)
- `geometry_gs` (GFX6-8)
- `vertex_geometry_gs` (GFX9+)
- `gs_copy_vs`

Without geometry shaders: just VS

SW Vertex shader (HW VS):

```
VS go brrrr
export to where PS can read from
```

Geometry shaders: separate stages

SW Vertex shader (HW ES):

```
VS go brrrr
store outputs to VRAM
```

Geometry shader (HW GS):

```
load inputs from VRAM
GS does its thing
store outputs to VRAM
```

GS copy shader (HW VS):

```
load GS outputs from VRAM
export to where PS can read from
```

Geometry shaders: merged VS+GS stage

SW VS+GS (HW GS):

```
if (am I a SW VS invocation?)
     VS go brrrr
     store outputs to LDS
endif
if (am I a SW GS invocation?)
     load inputs from LDS
     GS does its thing
     store outputs to VRAM
endif
```

GS copy shader (HW VS):

```
load GS outputs from VRAM
export to where PS can read from
```

Geometry shaders: GS copy shader

Hardware limitations:

- GS store their outputs in VRAM
- PS read their inputs from elsewhere
- So, we need to copy

(These limitations are eliminated by NGG on Navi)

Tessellation

Tessellation

Tessellation shaders don’t actually tessellate.

Tessellation

- Lots of new intrinsics
- Lots of new stages
- Brain-twister I/O

Tessellation: merged stages

On GFX9+:

VS and TCS are merged

TES and GS are merged, too.

Tessellation: 6 new stages

When you have just tessellation:

- VS: `vertex_ls` (GFX6-8)
- TCS: `tess_control_hs` (GFX6-8)
- VS+TCS: `vertex_tess_control_hs` (GFX9+)
- TES: `tess_eval_vs`

When you also have GS:

- TES: `tess_eval_es` (GFX6-8)
- TES+GS: `tess_eval_geometry_gs` (GFX9+)

Tessellation: optimizing merged stages

Sometimes the number of VS and TCS invocations are the same.

This happens when the input and output patch size are 3.
Turns out they almost always are.
In this case, VS outputs can be eliminated and transformed to temps.

Tessellation: I/O

Shaders can now read their own output.

So, sometimes we have to store them in both LDS and VRAM.
Using different layouts.

Wave32

Wave32

Optionally,

you can have 32 SIMD lanes instead of 64

This breaks a lot of assumptions we had.

Wave32: special registers

We had to change every usage of `exec` and `vcc`.

Wave32: booleans

Before:

Uniform (32-bit) vs. divergent (64-bit)

After:

All booleans are now per-lane

Uniforms are optimized back

Wave32: subgroups

Subgroup size?

Wave32: subgroups

Subgroup size?

We always report 64

but pretend that 32 of those are disabled

Conforms to the spec, but still troublesome for naive, non-conformant apps
eg. some expect to use the `(subgroup_size - 1)`th lane

Wave32: when to use?

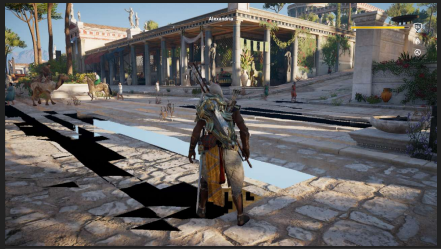
It's off by default (unless you use an env var)

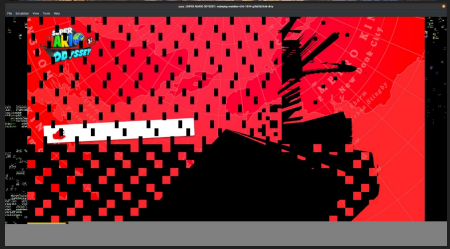
CS: optionally available

VS, TCS, TES, GS: decide based on divergence?

PS: performs bad due to interpolation

Btw, we fixed some bugs, too





Btw, we fixed some bugs, too

And we added a unit testing framework to make sure they don't happen again.

Our plans for the future

VALVE

RDNA 2 Support

In progress

- We can look at RadeonSI
- We can browse LLVM code

RadeonSI (OpenGL) support

Long road, but already started

- Unify shader arguments - Connor
- Unify I/O handling - Marek
- but ACO still depends on RADV internals

Ray Tracing

- There is a draft KHR extension
- And some LLVM code
- ...but no public RDNA 2 HW docs yet

Mesh shaders

- Possibly doable even on Navi 10 NGG
- ...but no KHR extension yet

More optimizations

- Rapid-packed math

More optimizations

- Clauses on Navi
- Post-RA scheduler
- NGG GS
- Others are also on the way
- And more
- And then some

Thanks

Questions, suggestions, discussion?

A year of ACO
Timur Kristóf

Venemo @ #dri-devel, #radeon, ...
github.com/Venemo/xdc2020-aco

VALVE

A year of ACO

from prototype to default

Timur Kristóf

2020

XDC
2020

Linux Open-Source
Graphics Drivers Group