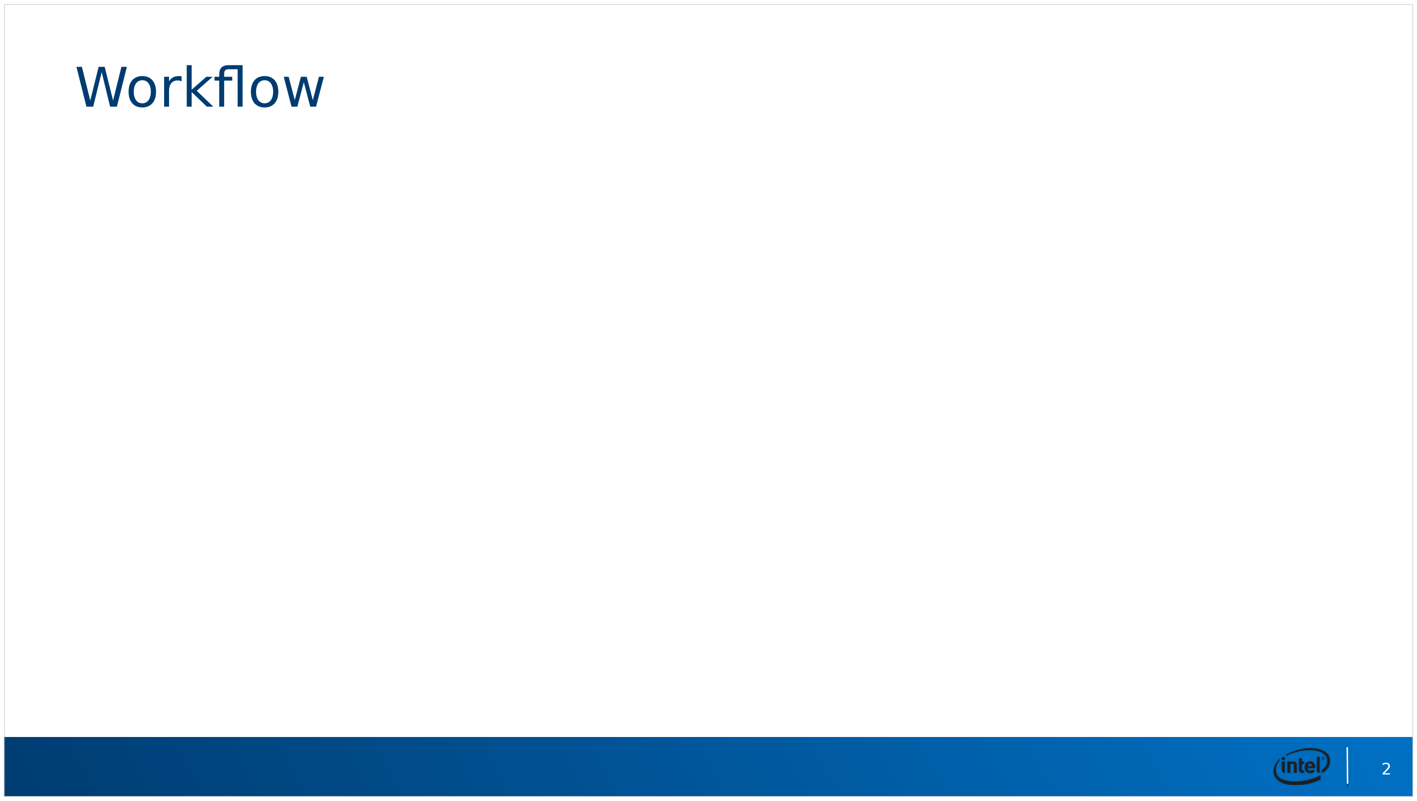
Implementing Optimizations in NIR

Ian Romanick

XDC 2019

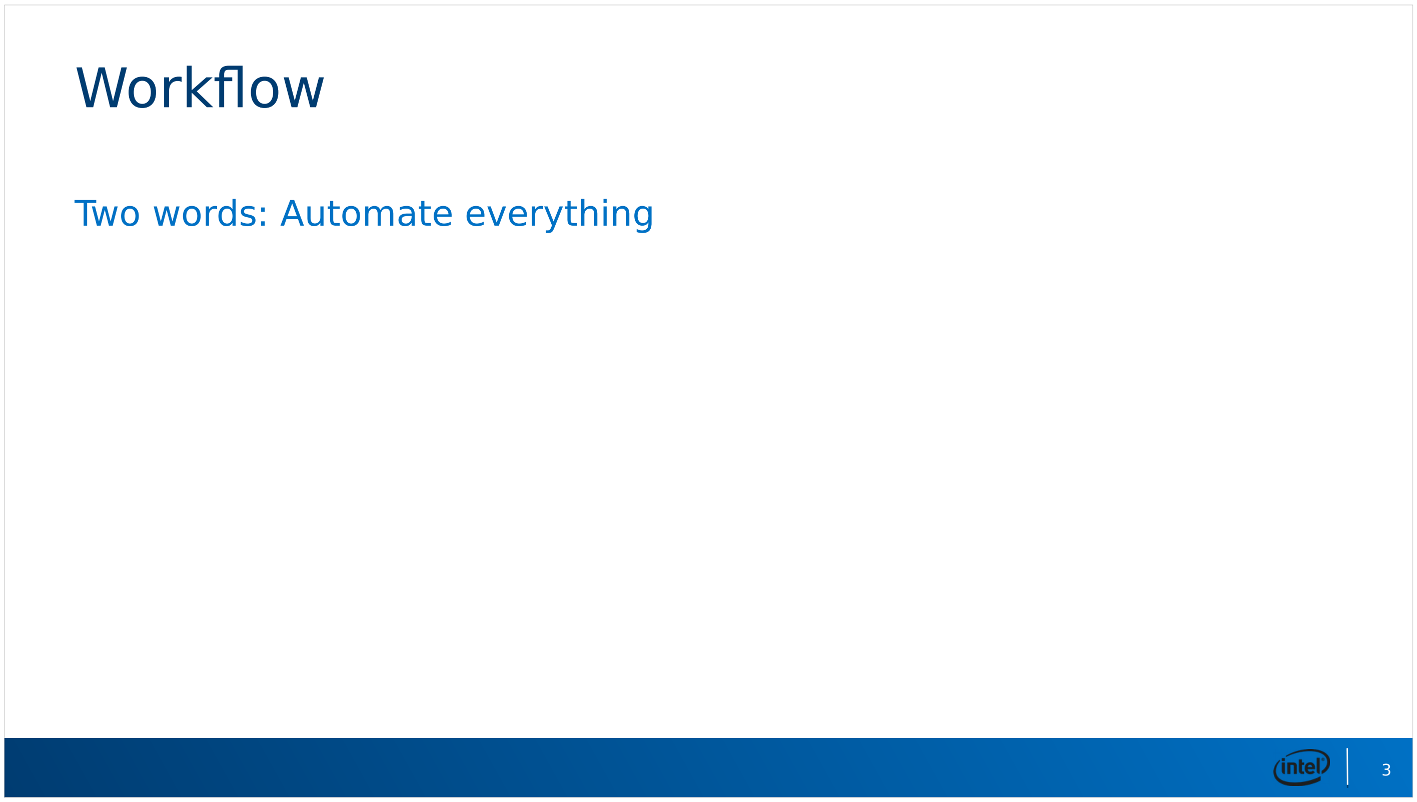
I’ve spent a good part of the last year and a half working on various sorts of NIR optimizations. These include both opt_algebraic optimizations and optimizations that required new passes. I’m going to present some things that I’ve learned through that process.



Workflow

2

Before I talk developing optimizations, I’m going to talk a bit about optimizing the developer.



Workflow

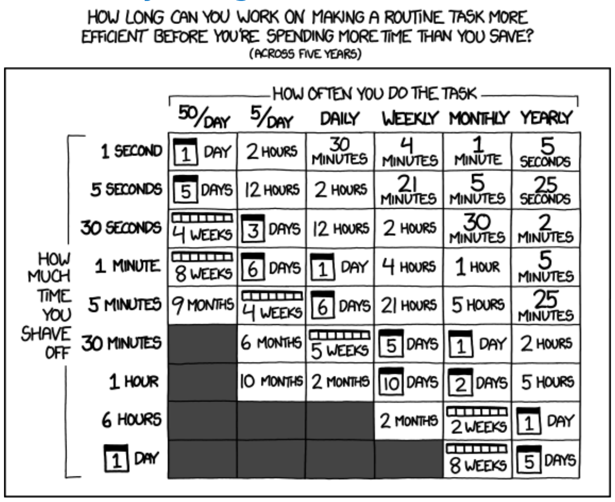
Two words: Automate everything

Automate your automation. Queue “I put automation in your automation” joke.

Under-appreciated and often procrastinated aspect of developer work.

Workflow

Two words: Automate everything



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE? (ACROSS FIVE YEARS)

HOW MUCH TIME YOU SHAVE OFF	HOW OFTEN YOU DO THE TASK: 50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES	■	6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR	■	10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS	■	■	■	2 MONTHS	2 WEEKS	1 DAY
1 DAY	■	■	■	■	8 WEEKS	5 DAYS

https://xkcd.com/1205/

It is possible to go too far. There is an Amdahl's Law type of analysis to do here.

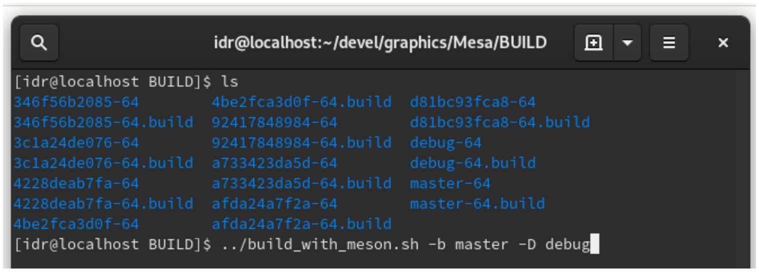
A big factor in productivity of this work is how many edit-compile-benchmark iterations can you do in a day. Same applies to edit-compile-test or edit-compile-debug.

The comic doesn't tell the whole story. Say you have two computational tasks that take 5 minutes each, and it takes 5 seconds to start each one. You're not going to sit and watch the first one complete. If you don't come back to it for 10 minutes (because you were busy reading web comics), that's 5 minutes wasted. Do that twice, and that's entire missed edit-compile-benchmark cycle.

Workflow

Two words: Automate everything

- Script your whole build and install process
  - Install each build to a unique location... probably named after the GIT SHA

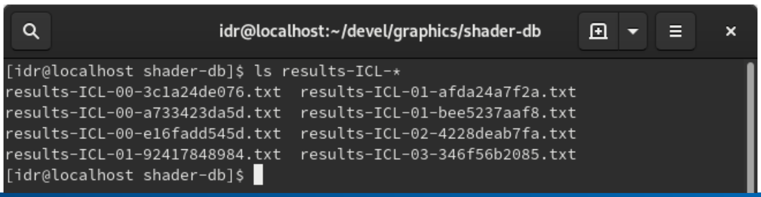


```
[idr@localhost BUILD]$ ls
346f56b2085-64        4be2fca3d0f-64.build  d81bc93fca8-64
346f56b2085-64.build  92417848984-64        d81bc93fca8-64.build
3c1a24de076-64        92417848984-64.build  debug-64
3c1a24de076-64.build  a733423da5d-64        debug-64.build
4228deab7fa-64        a733423da5d-64.build  master-64
4228deab7fa-64.build  afda24a7f2a-64        master-64.build
4be2fca3d0f-64        afda24a7f2a-64.build
[idr@localhost BUILD]$ ../build_with_meson.sh -b master -D debug
```

Workflow

Two words: Automate everything

- Script your whole build and install process
  - Install each build to a unique location... probably named after the GIT SHA
- Make a wrapper script to set environment variables to use local builds
- Script running your test suite *and* scraping data



```
[idr@localhost shader-db]$ ls results-ICL-*
results-ICL-00-3c1a24de076.txt  results-ICL-01-afda24a7f2a.txt
results-ICL-00-a733423da5d.txt  results-ICL-01-bee5237aaf8.txt
results-ICL-00-e16fadd545d.txt  results-ICL-02-4228deab7fa.txt
results-ICL-01-92417848984.txt  results-ICL-03-346f56b2085.txt
[idr@localhost shader-db]$
```

Workflow

Two words: Automate everything

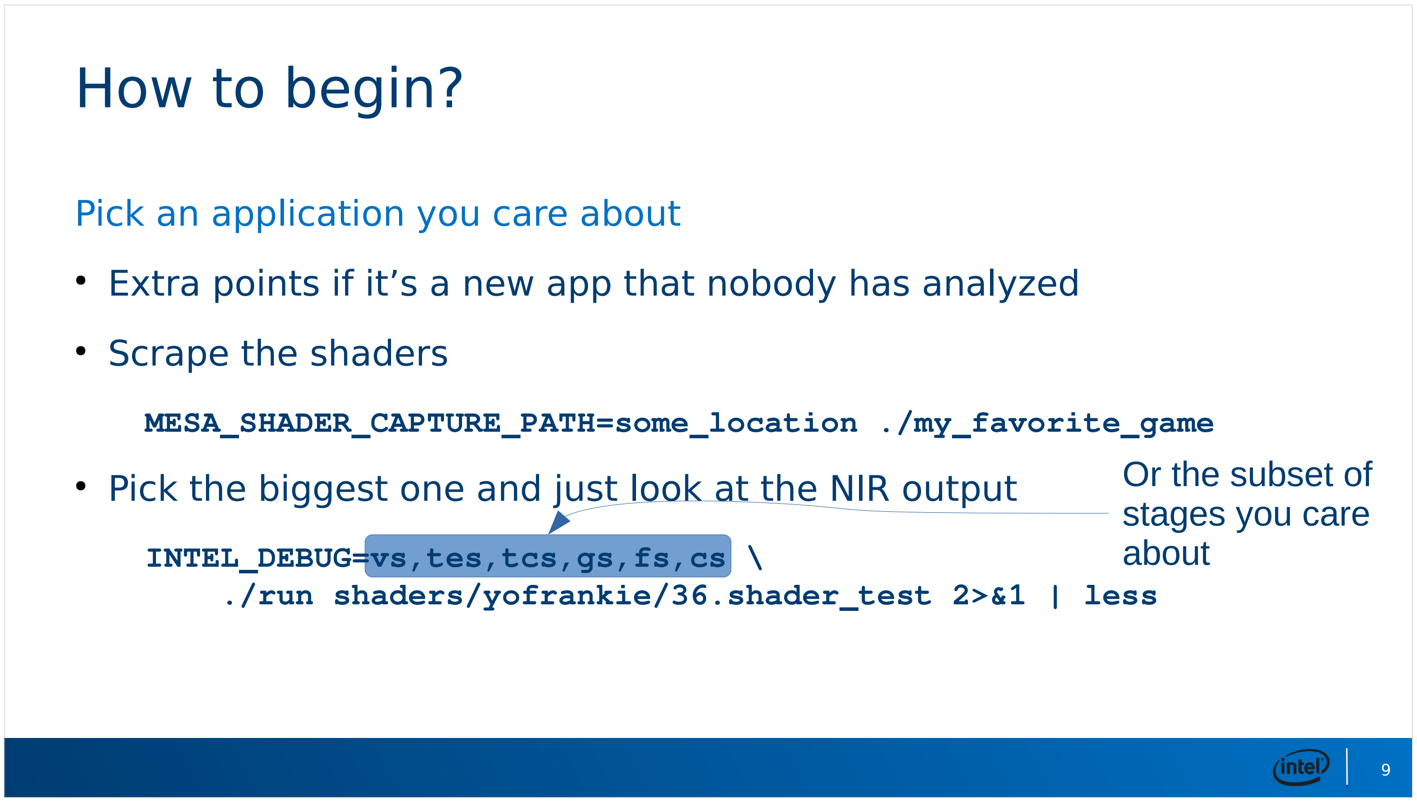
- Script your whole build and install process
  - Install each build to a unique location... probably named after the GIT SHA
- Make a wrapper script to set environment variables to use local builds
- Script running your test suite *and* scraping data
- Final script: determine the current SHA, run all the other scripts
  - **`git rebase -i -x`** is your new best friend

Workflow

Get a second machine

- Many rebuilds, shader-db runs, benchmark runs take a lot of time
- It’s hard to do other work when you system is bogged down
- shader-db especially benefits from as many cores as possible
  - See recent Phil’s Computer Lab YouTube videos about Ivy Bridge Xeons

Building a lower clock 10-core / 20-thread shader-db system on the cheap would be interesting.



How to begin?

Pick an application you care about

- Extra points if it’s a new app that nobody has analyzed
- Scrape the shaders

```
MESA_SHADER_CAPTURE_PATH=some_location ./my_favorite_game
```

- Pick the biggest one and just look at the NIR output

```
INTEL_DEBUG=vs,tes,tcs,gs,fs,cs \
    ./run shaders/yofrankie/36.shader_test 2>&1 | less
```

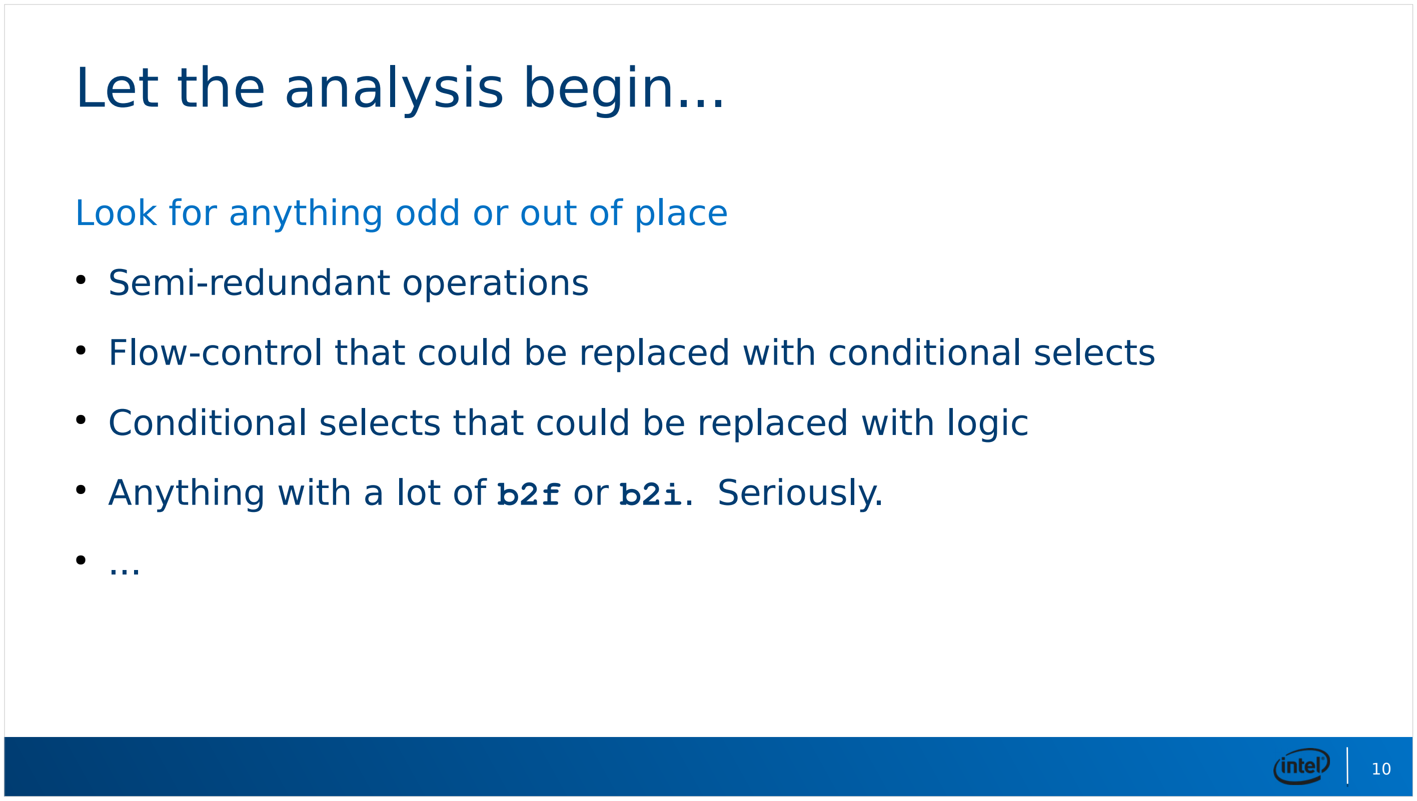
Or the subset of stages you care about

NIR_PRINT=true environment variable is useful at some later steps. This dumps the NIR instructions after ever optimization or lowering pass that makes any progress.

NIR_PRINT=true only works in debug builds, but you want to do most shader-db runs on -march=native release builds.

I usually look at the NIR instead of the GPU assembly because instruction scheduling makes the assembly a lot harder to follow.

Looking at the GPU assembly can help find other kinds of optimizations. This is especially true if you want to massage NIR into a form easier to generate machine instructions for.



Let the analysis begin...

Look for anything odd or out of place

- Semi-redundant operations
- Flow-control that could be replaced with conditional selects
- Conditional selects that could be replaced with logic
- Anything with a lot of **`b2f`** or **`b2i`**. Seriously.
- ...

Semi-redundant patterns like **`x - y`** vs **`x > y`**. This kind of optimization is looking to make CSE and other optimizations more effective.

`b2f` and, to a lesser extent, **`b2i`** appear due to the way the HLSL compiler handles Booleans for some shader models. Most shaders that we see are some how ported from HLSL... often by decompiling the output of the HLSL compiler.

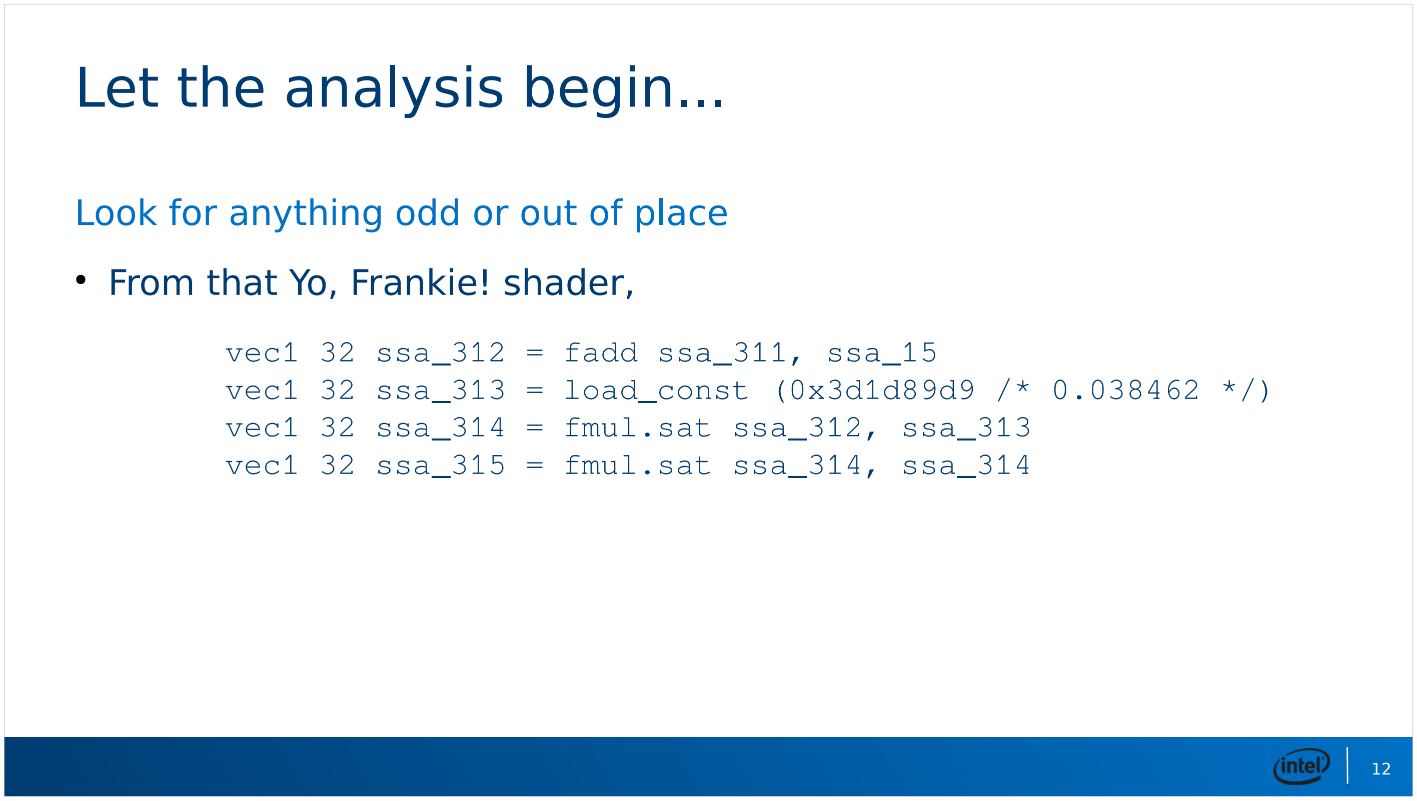
When we start looking at these shaders, what are we likely to find?

Let the analysis begin...

https://commons.wikimedia.org/wiki/File:Starr-091115-1255-Psidium_guajava-lots_of_fruit_on_ground-Olinda-Maui_(24622476339).jpg

Remember kids, the lowest hanging fruit is the stuff sitting on the ground.

I’m not kidding when I say that I practically can’t look at the NIR from a shader without seeing something to improve.



Let the analysis begin...

Look for anything odd or out of place

- From that Yo, Frankie! shader,

```
vec1 32 ssa_312 = fadd ssa_311, ssa_15
vec1 32 ssa_313 = load_const (0x3d1d89d9 /* 0.038462 */)
vec1 32 ssa_314 = fmul.sat ssa_312, ssa_313
vec1 32 ssa_315 = fmul.sat ssa_314, ssa_314
```

I literally picked this shader from the public shader-db at random, and I think that illustrates my point.

I saw a few odd things, but nothing obviously jumped out as being improvable until I got near the end of the shader.

Let the analysis begin…

Look for anything odd or out of place

- From that Yo, Frankie! shader,

```
vec1 32 ssa_312 = fadd ssa_311, ssa_15
vec1 32 ssa_313 = load_const (0x3d1d89d9 /* 0.038462 */)
vec1 32 ssa_314 = fmul.sat ssa_312, ssa_313
vec1 32 ssa_315 = fmul.sat ssa_314, ssa_314
```

Redundant!

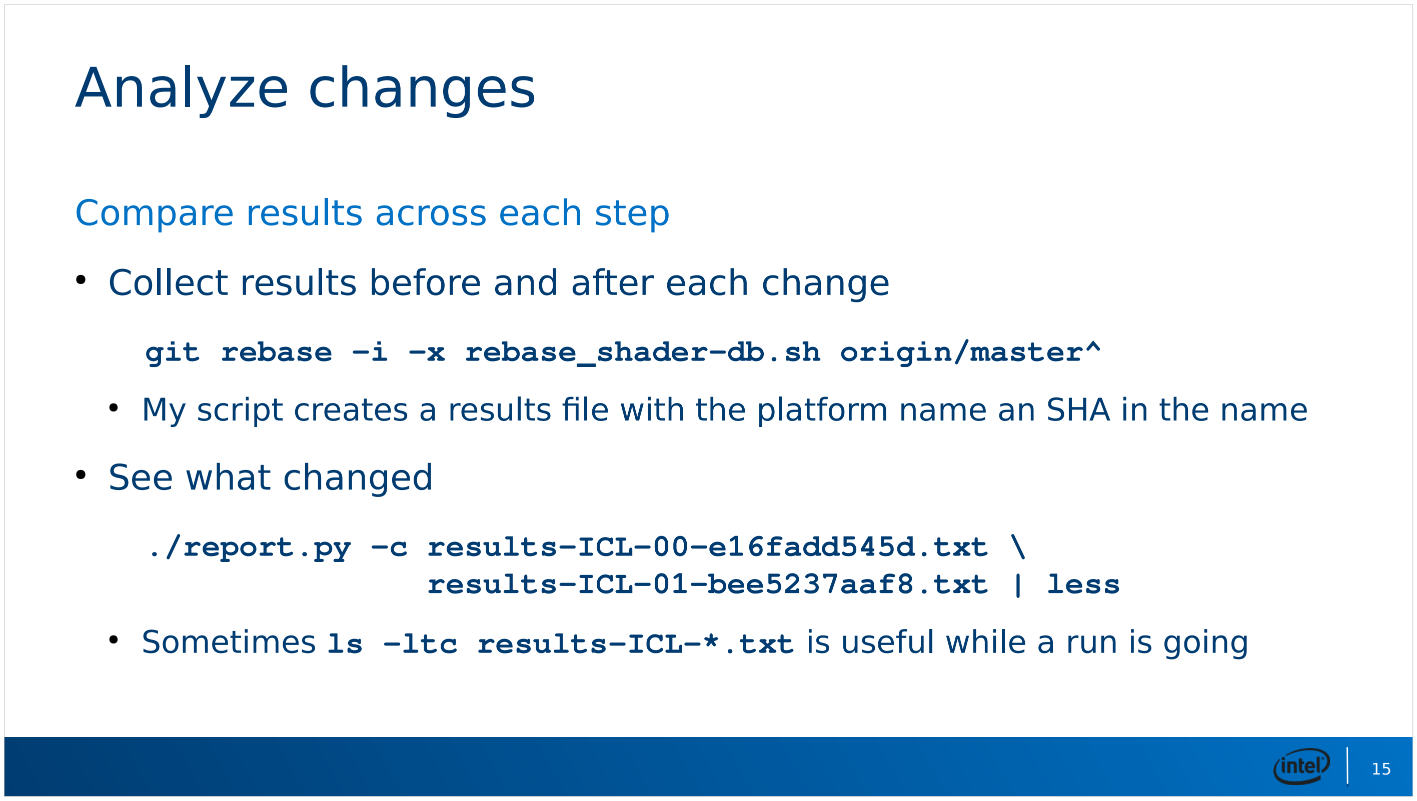
Implement a fix

Most optimizations of this class are algebraic

- Simple match a pattern, replace with a new pattern

```
(('fsat', ('fmul', ('fsat', a), ('fsat', b))),
 ('fmul', ('fsat', a), ('fsat', b))),
```

- More complex optimizations my require a new pass
  - **`nir_opt_comparison_pre`** was added for the **`x < y`** vs. **`x - y`** cases
  - Find a pass that does something similar and borrow from it
  - Maybe enhance an existing pass



Analyze changes

Compare results across each step

- Collect results before and after each change

```
git rebase -i -x rebase_shader-db.sh origin/master^
```

  - My script creates a results file with the platform name an SHA in the name

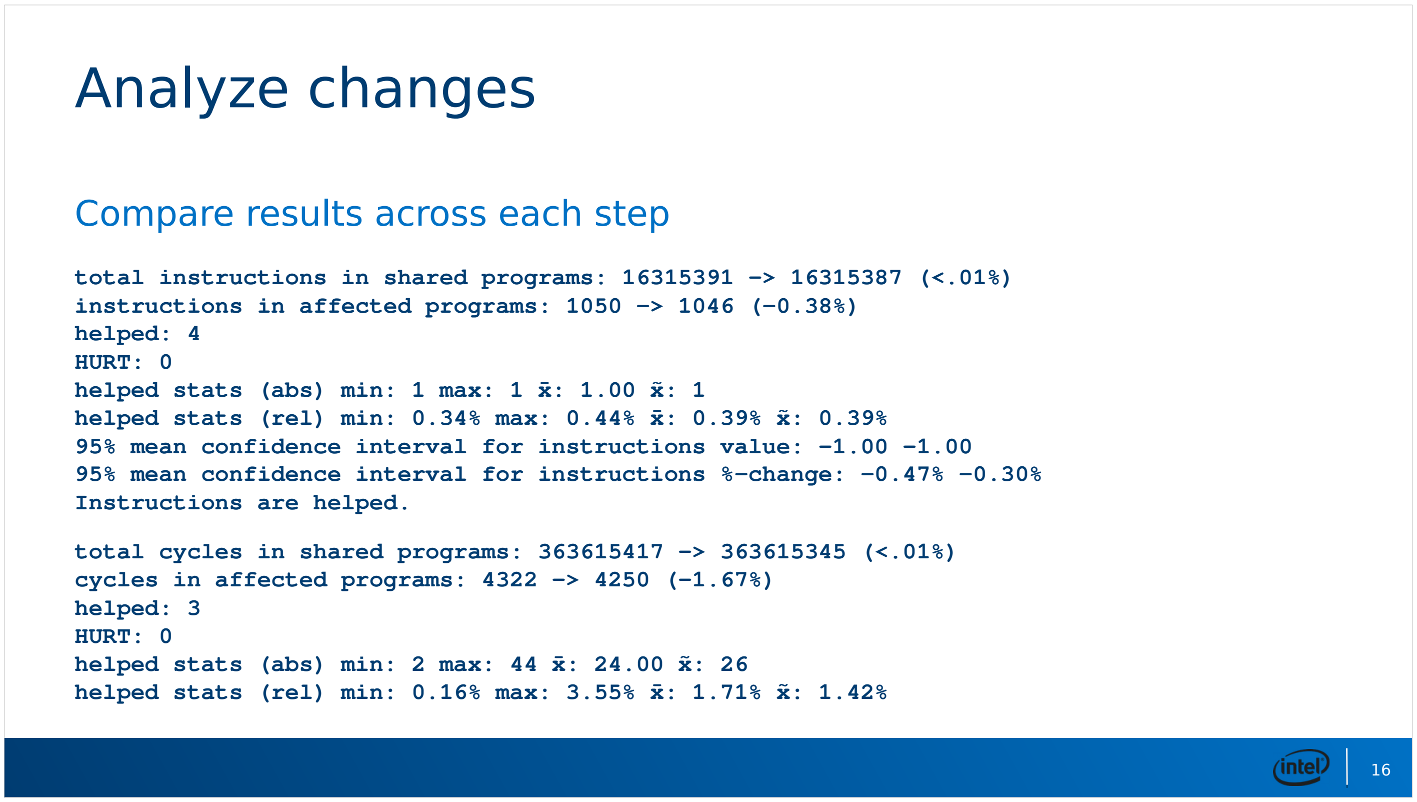
- See what changed

```
./report.py -c results-ICL-00-e16fadd545d.txt \
               results-ICL-01-bee5237aaf8.txt | less
```

  - Sometimes `ls -ltc results-ICL-*.txt` is useful while a run is going

rebase_shader-db runs shader-db for each Intel platform. Catches cases where a change helps a modern platform but regresses an older platform that has more limitations, lacks certain instructions, etc.

Creates output files that contain the platform name and the SHA. The number before the SHA is the number of steps in the branch since origin/master.



Analyze changes

Compare results across each step

```
total instructions in shared programs: 16315391 -> 16315387 (<.01%)
instructions in affected programs: 1050 -> 1046 (-0.38%)
helped: 4
HURT: 0
helped stats (abs) min: 1 max: 1 x̄: 1.00 x̃: 1
helped stats (rel) min: 0.34% max: 0.44% x̄: 0.39% x̃: 0.39%
95% mean confidence interval for instructions value: -1.00 -1.00
95% mean confidence interval for instructions %-change: -0.47% -0.30%
Instructions are helped.

total cycles in shared programs: 363615417 -> 363615345 (<.01%)
cycles in affected programs: 4322 -> 4250 (-1.67%)
helped: 3
HURT: 0
helped stats (abs) min: 2 max: 44 x̄: 24.00 x̃: 26
helped stats (rel) min: 0.16% max: 3.55% x̄: 1.71% x̃: 1.42%
```

Fun fact: the shaders helped by that change weren’t even in Yo Frankie! That shader did change, but the instruction count did not.

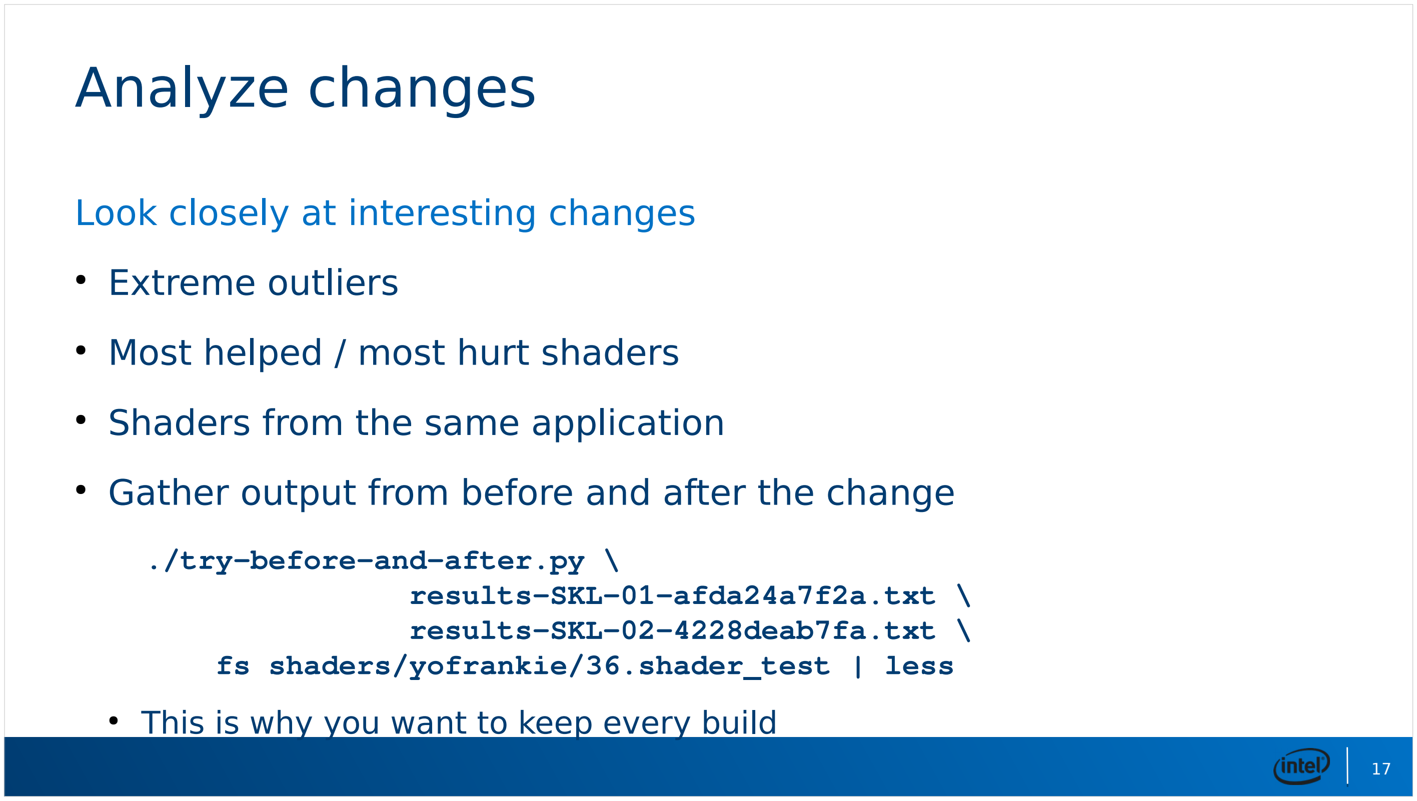
A couple of interesting things to note in each bit.

1. Min and max helped and hurt.

2. Mean versus median. This can be especially interesting. If the mean is significantly higher than the median, it means that there are some extreme outliers that may be worth examination.

3. The total instructions in shared program percentage. If this says “< .01%” (without regressions that need fixing), I usually won’t continue won’t continue working on a change.

This opt didn’t really help, so I’d save it off somewhere for later.



Analyze changes

Look closely at interesting changes

- Extreme outliers
- Most helped / most hurt shaders
- Shaders from the same application
- Gather output from before and after the change

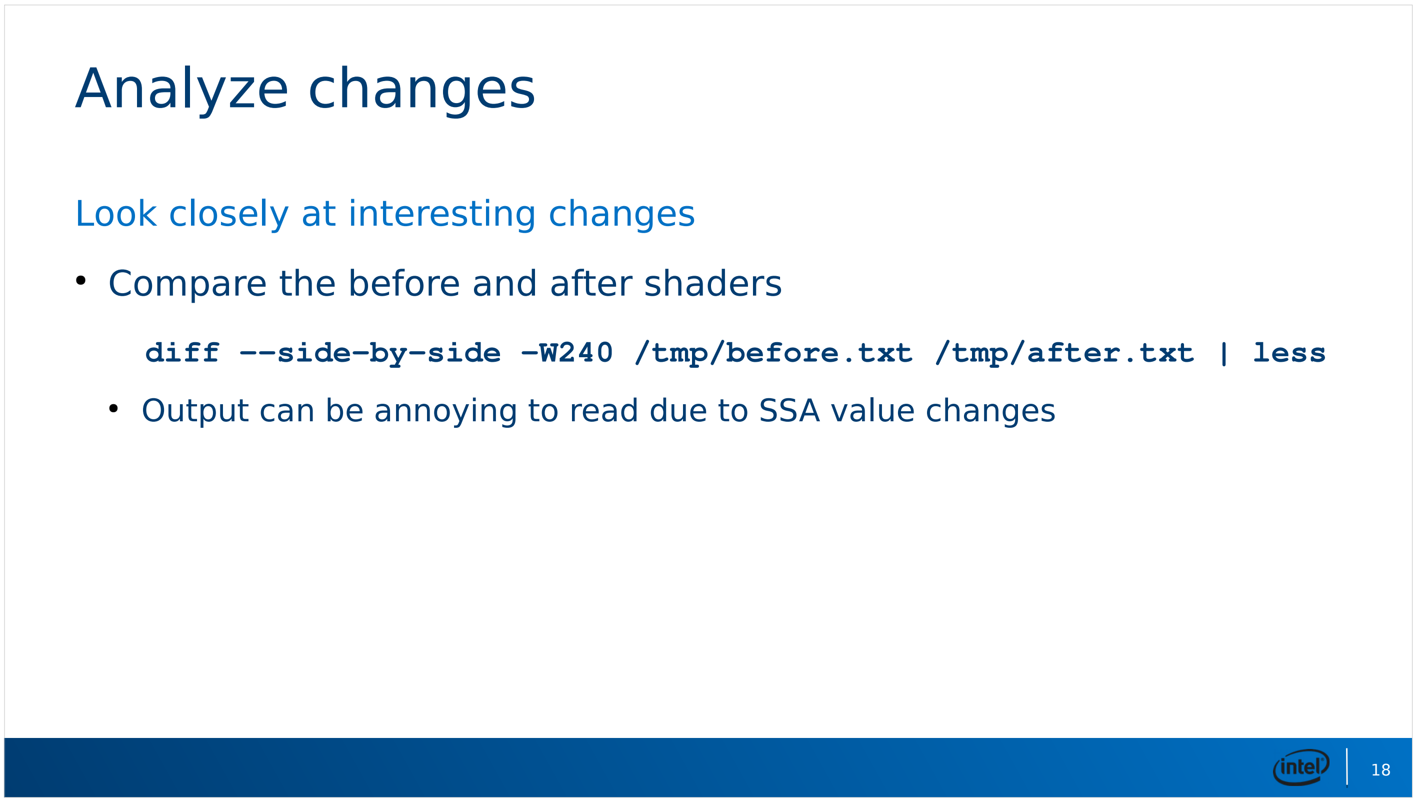
```
./try-before-and-after.py \
               results-SKL-01-afda24a7f2a.txt \
               results-SKL-02-4228deab7fa.txt \
    fs shaders/yofrankie/36.shader_test | less
```

  - This is why you want to keep every build

Outliers include shaders with large change in number instructions (but may not be largest percent change), shaders with a large change in spills / fill, shaders with changes in number of loops (very uncommon).

When you review patches like this, look at the shader-db results. If the commit message doesn't mention the hurt shaders or outliers, ASK if the author looked at them. No matter how new you are, this is reasonable review feedback to give.

The "try-before-and-after" script scrapes the SHA of the build to use and the name of the platform from the names of the result files. This makes it easy to just edit the old report.py command line.



Analyze changes

Look closely at interesting changes

- Compare the before and after shaders

```
diff --side-by-side -W240 /tmp/before.txt /tmp/after.txt | less
```

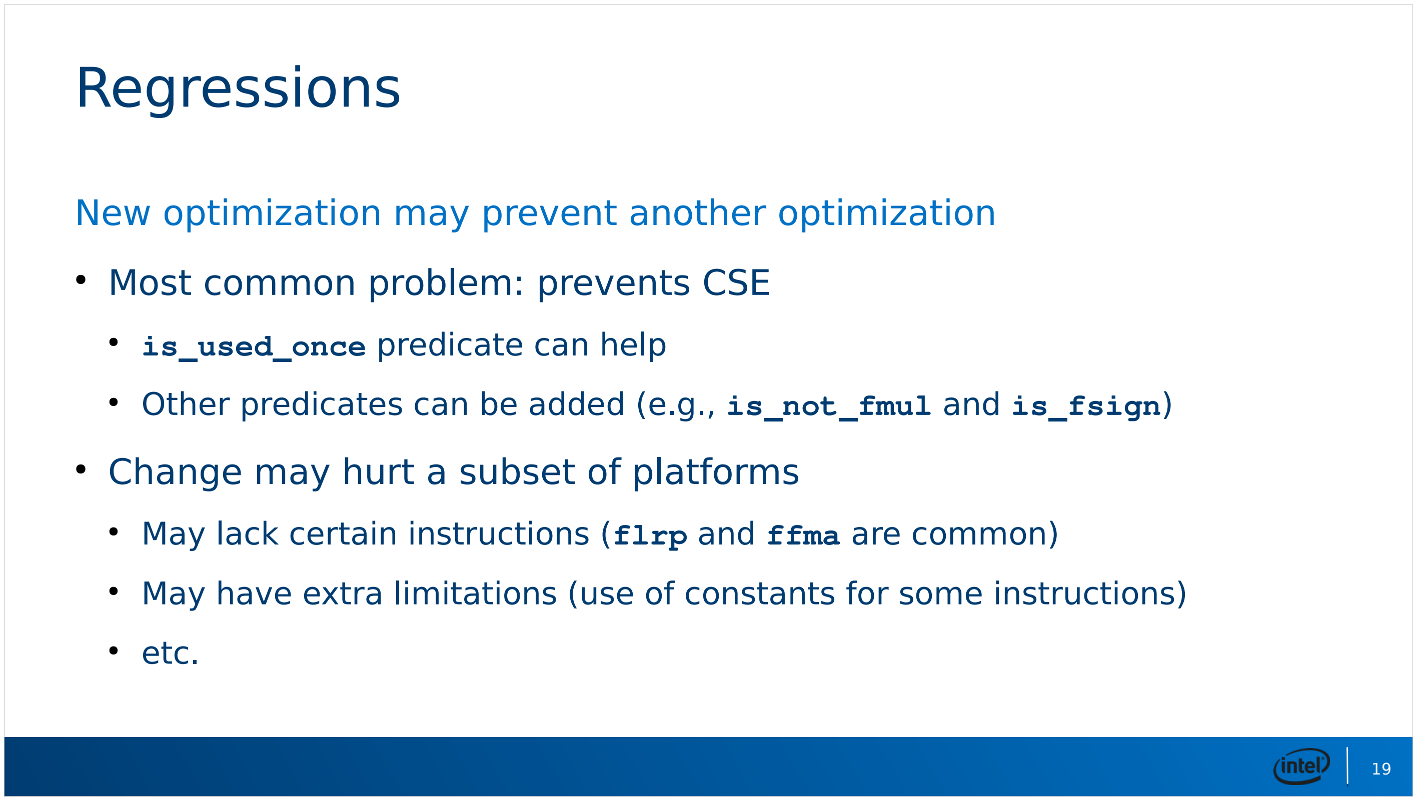
  - Output can be annoying to read due to SSA value changes

If **`git rebase -x`** is your best friend, **`diff -side-by-side`** is who you hang out with when they're busy.

Removing one NIR instruction early in the shader causes all of the SSA values to be renumbered, and that causes most of the rest of the shader to appear changed.

Shouldn't be hard to write a script to "add N to all ssa_X where X >= Y".

Make notes. I have a long text file of weird things that I've observed in shaders. After maintaining that file for quite some time I realized the importance of tracking which shader I saw each thing in.



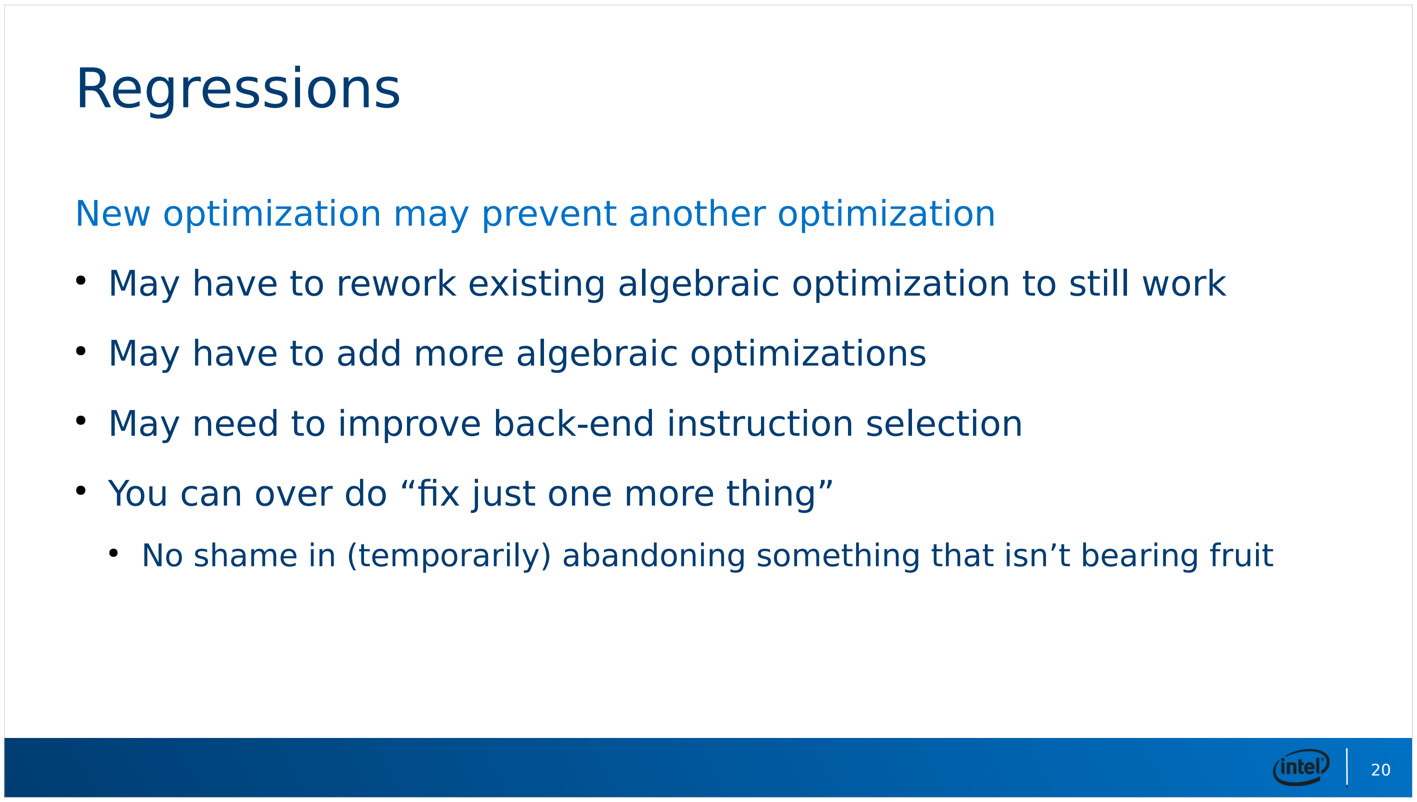
Regressions

New optimization may prevent another optimization

- Most common problem: prevents CSE
  - `is_used_once` predicate can help
  - Other predicates can be added (e.g., `is_not_fmul` and `is_fsign`)
- Change may hurt a subset of platforms
  - May lack certain instructions (`flrp` and `ffma` are common)
  - May have extra limitations (use of constants for some instructions)
  - etc.

All of the predicates live in nir_search_helpers.h.

Especially for Intel GPUs, some optimizations hurt our ability to generate `FMA` instructions. Adding the right predicates can help avoid most of the hurtful cases while still allowing most of the helpful cases.



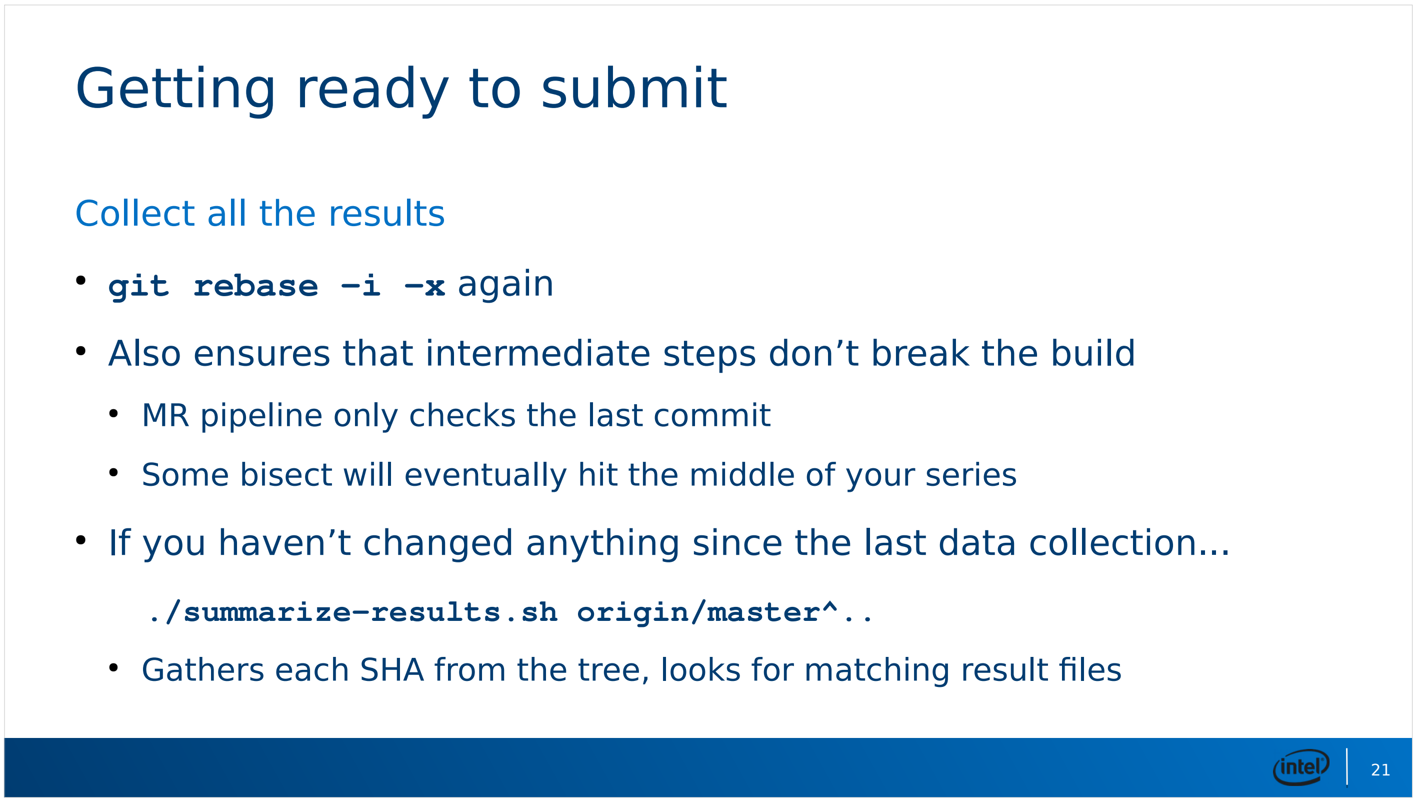
Regressions

New optimization may prevent another optimization

- May have to rework existing algebraic optimization to still work
- May have to add more algebraic optimizations
- May need to improve back-end instruction selection
- You can over do “fix just one more thing”
  - No shame in (temporarily) abandoning something that isn’t bearing fruit

It’s like the old joke about the programmer who died in the shower. He read the shampoo bottle “lather, rinse, repeat” and got stuck in an infinite loop.

It can also start to feel like the old woman who swallowed a fly. A fix on a fix on a fix on a fix... Either reevaluate the fundamental approach or bail. As many patches as I’ve landed to opt_algebraic, I’ve tossed out twice as many.



Getting ready to submit

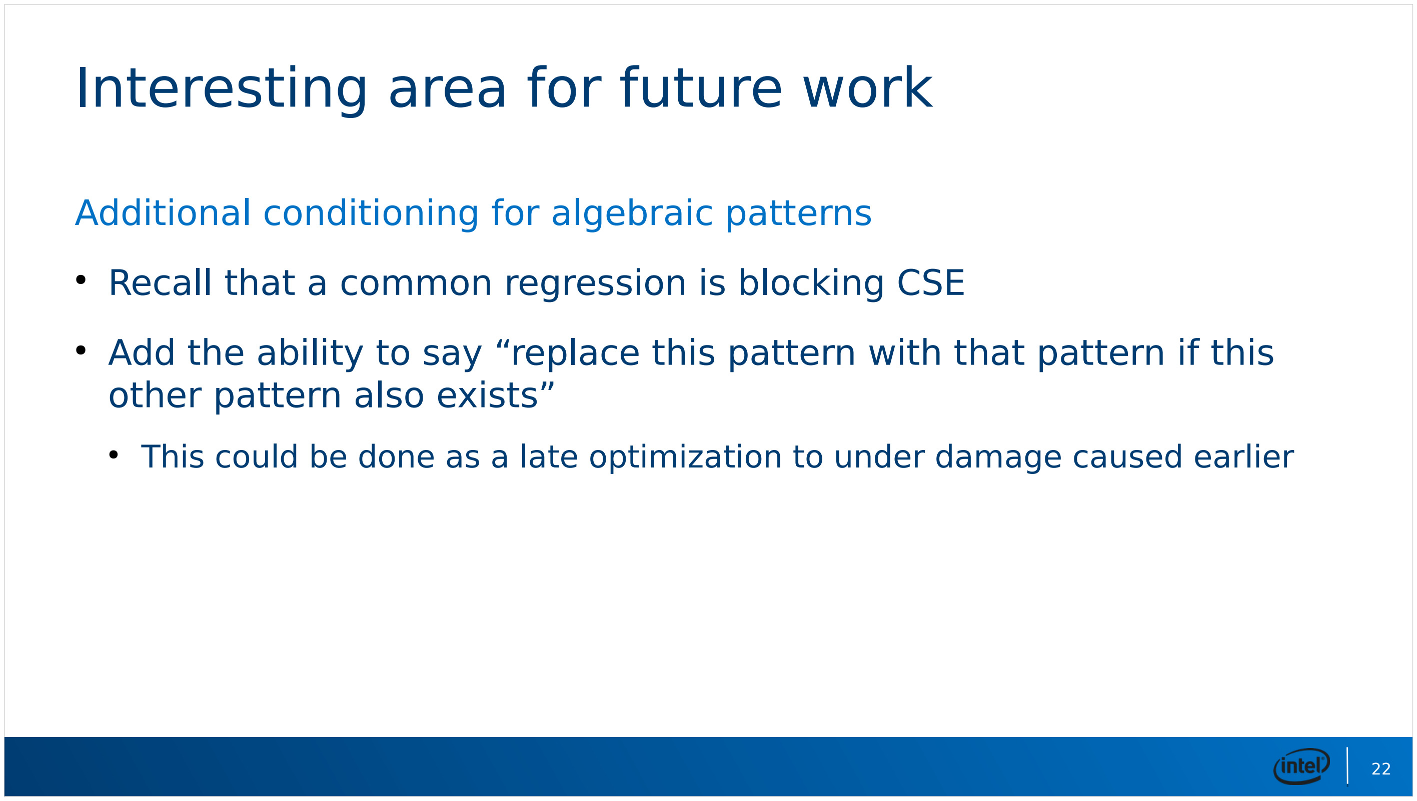
Collect all the results

- **`git rebase -i -x`** again
- Also ensures that intermediate steps don’t break the build
  - MR pipeline only checks the last commit
  - Some bisect will eventually hit the middle of your series
- If you haven’t changed anything since the last data collection...

```
./summarize-results.sh origin/master^..
```

  - Gathers each SHA from the tree, looks for matching result files

Collect results to put in each commit message. A summary of the results across the whole series is good for the “cover letter” part of the MR.



Interesting area for future work

Additional conditioning for algebraic patterns

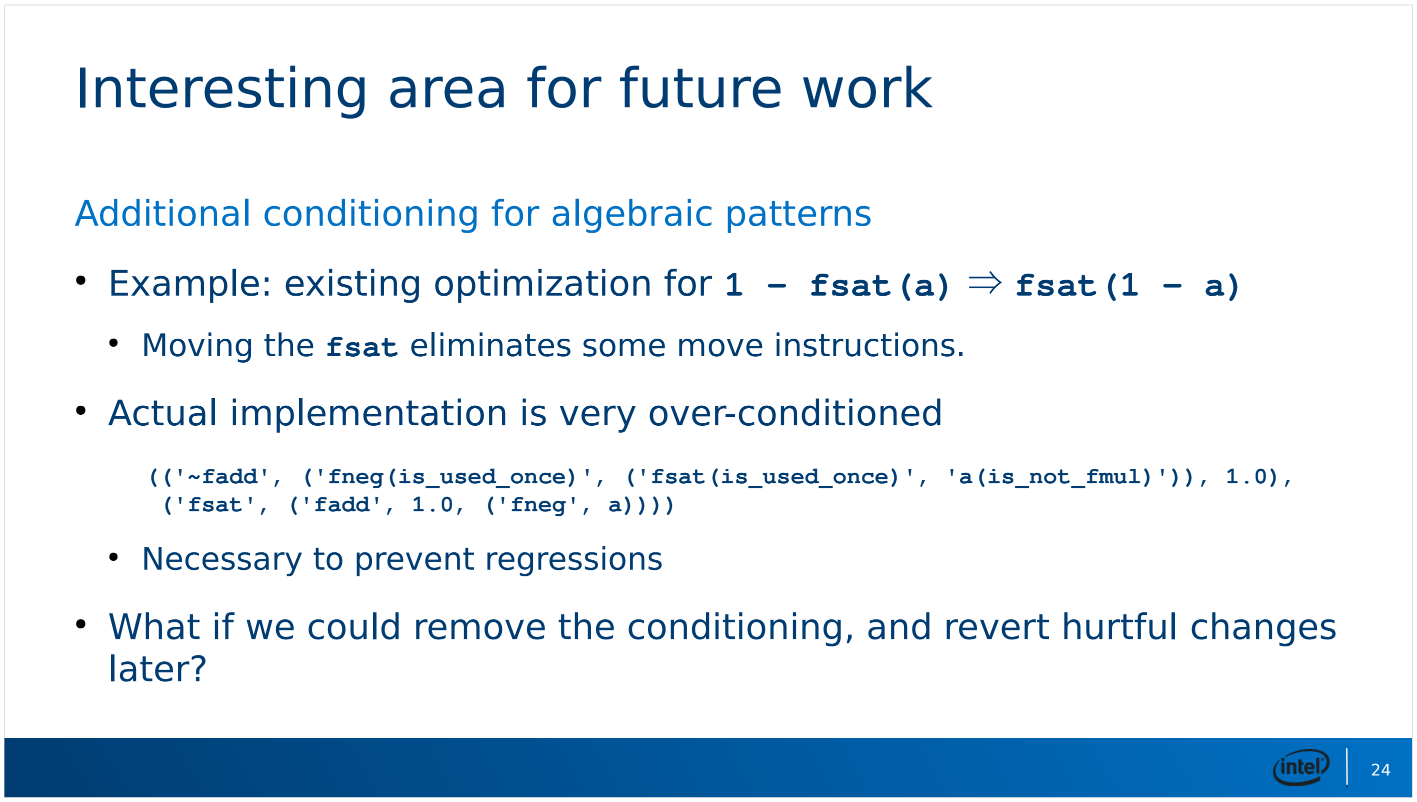
- Recall that a common regression is blocking CSE
- Add the ability to say “replace this pattern with that pattern if this other pattern also exists”
  - This could be done as a late optimization to under damage caused earlier

This is an idea that I had on Monday while preparing for this presentation, so it’s not fully thought out yet.

Interesting area for future work

Additional conditioning for algebraic patterns

- Example: existing optimization for **`1 - fsat(a)`** ⇒ **`fsat(1 - a)`**
  - Moving the **`fsat`** eliminates some move instructions.



Interesting area for future work

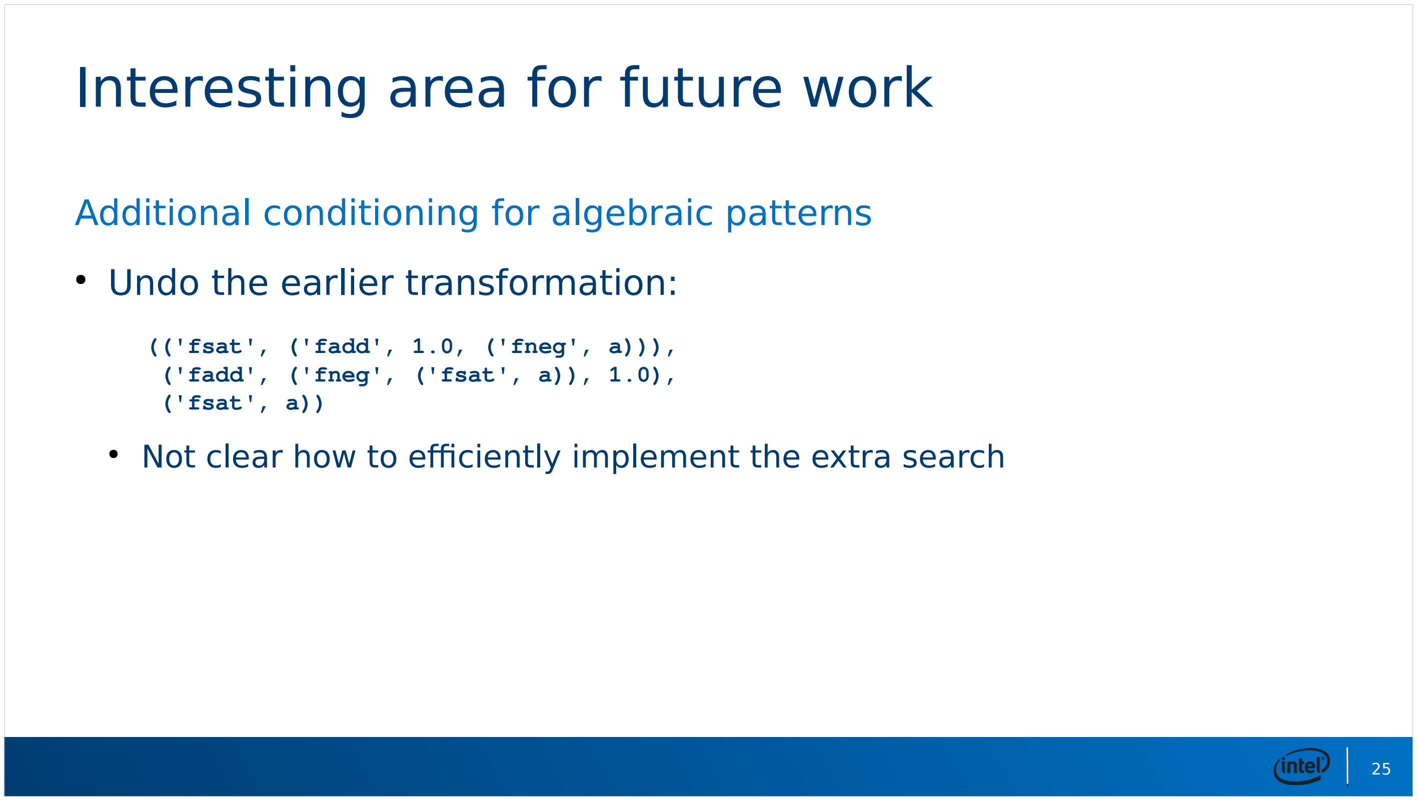
Additional conditioning for algebraic patterns

- Example: existing optimization for **`1 - fsat(a)`** ⇒ **`fsat(1 - a)`**
  - Moving the **`fsat`** eliminates some move instructions.
- Actual implementation is very over-conditioned

```
(('~fadd', ('fneg(is_used_once)', ('fsat(is_used_once)', 'a(is_not_fmul)')), 1.0),
 ('fsat', ('fadd', 1.0, ('fneg', a))))
```

  - Necessary to prevent regressions
- What if we could remove the conditioning, and revert hurtful changes later?

When I originally implemented this optimization, it helped a lot of shaders, but it also caused a lot of regressions. To compensate, I added a lot of constraints on the application of the optimization.



Interesting area for future work

Additional conditioning for algebraic patterns

- Undo the earlier transformation:

```
(('fsat', ('fadd', 1.0, ('fneg', a))),
 ('fadd', ('fneg', ('fsat', a)), 1.0),
 ('fsat', a))
```

  - Not clear how to efficiently implement the extra search

25

Remember when I said there’s no shame in temporarily abandoning some optimizations? After adding an optimizer feature like this, it may be worth revisiting some of those earlier attempts.

Links

- Scripts for building and testing Mesa:

  https://gitlab.freedesktop.org/idr/mesa-scripts

- shader-db with added scripts:

  https://gitlab.freedesktop.org/idr/shader-db/commits/scripts

Questions?