Debugging GPU faults: QoL tools for your driver

Danylo Piliaiev
2023-10-17
Who Am I?

- Currently implementing Adreno 7XX GPU generation in Turnip
- My blog: blogs.igalia.com/dpiliaiev

In the past

- Worked on mobile video games
- Debugging unruly games since 2018
- At Igalia since November 2020
The Problem

- "What if I was able to quickly edit this GPU packet?"
- "What if I was able to dump this buffer here?"
- Or "It would have been nice to print that shader register!"
- "I'll implement it later...."
Unrecoverable Hangs - Roadblocks

- Computer completely locks up and has to be rebooted
- Last few seconds of logs/anything else are lost
- Existing tooling isn't of much use with such constraints
  - GFR (Graphics Flight Recorder):
    - VK layer for breadcrumbs
    - Dumps command buffers with commands' status
Unrecoverable Hangs - Solution

More BREADCRUMBS!

- GFR writes results to the disk
- GFR logging is far behind what's actually runs on GPU
- GFR could be too high level:
  - Blits/BeginRenderPass/EndRenderPass could internally use a lot of different 2d and 3d blits
Unrecoverable Hangs - Solution

Observations

- Unrecoverable hangs are rarely caused by sync issues
- Cannot allow GPU to race ahead of the last know breadcrumb
- A hang may happen asynchronously to the GPU packet that triggered it, e.g.
  - A job is scheduled to another GPU unit
  - That GPU unit hangs some time afterwards
  - There may not be a way to synchronize it
Unrecoverable Hangs - Solution

The current solution in Turnip is:

• Breadcrumbs are inserted after each GPU command
• GPU writes a breadcrumb and immediately waits for this value to be acknowledged
• CPU in a busy loop checks the breadcrumb value
  ○ If new one is found, it is sent over the network
• The CPU acks the breadcrumb and GPU continues execution
How Our Breadcrumbs Work

GPU
- BR1
- Wait
- BR2
- Wait
- BR3
- Wait

Send

Wait

Read

Write

GPU executes this command and hangs

CPU
- BR1
- BR2

Send

Kernel
- BR1
- BR2

Send over network
Asynchronous hangs?

- Require explicit input in tty for each breadcrumb

```
GPU is on breadcrumb 18, continue? y
GPU is on breadcrumb 19, continue? y
GPU is on breadcrumb 20, continue? y
GPU is on breadcrumb 21, continue?
```
Breadcrumbs In Practice

- Increase GPU hang timeout
- Receive breadcrumbs on another machine via bash spaghetti

```
nc -lvup $PORT | stdbuf -o0 xxd -pc -c 4 | \
awk -Wposix '{printf("%u:%u\n", "0x" $0, a[$0]++)}
```

- Launch workload with `TU_BREADCRUMBS` envvar

```
TU_BREADCRUMBS=$IP:$PORT, break=$BREAKPOINT:$BREAKPOINT
```

- Launch workload and break on the last known
Further Material

https://blogs.igalia.com/dpiliaiev/debugging-unrecoverable-hangs/

https://gitlab.freedesktop.org/mesa/mesa/-/merge_requests/15452
Faster Way To Debug Hangs
Breadcrumbs Shortcoming

- Breadcrumbs are good for finding a command that hangs
- They cannot tell which part of the GPU state caused it
- They are useless for misrenderings
- Some issues are not reproducible with breadcrumbs
Reproducing Hangs

- Drivers are already able to capture command streams
  - And all used buffers
- With this it is trivial to replay the submissions back
- Ideally requires user-space specified GPU addresses
Replaying - Caveats

- Multiple queues
  - When to re-upload memory?
  - Just force a single queue?
- Timeline semaphores?
- Recordings may be huge
Editing The Command Stream

• Even the most minimalistic editing is useful:

```c
/* pkt4: GRAS_2D_RESOLVE_CNTL_2 = { X = 63 | Y = 63 } */
pkt(cs, 4128831);
/* pkt4: RB_BLIT_SCISSOR_TL = { X = 0 | Y = 0 } */
pkt4(cs, 0x88d1, (2), 0);
/* pkt4: RB_BLIT_SCISSOR_BR = { X = 63 | Y = 63 } */
pkt(cs, 4128831);
pkt7(cs, CP_MEM_WRITE, 20);
/* { ADDR_LO = 0x1f7580 } */
pkt(cs, 2061696);
/* { ADDR_HI = 0x40 } */
pkt(cs, 64);
pkt(cs, 1216352390);
pkt(cs, 1107296256);
```
const char *source = R"(
  shps #l37
  getone #l37
  cov.u32f32 r1.w, c504.z
  cov.u32f32 r2.x, c504.w
  cov.u32f32 r1.y, c504.x
  ....
  end
)"
;
upload_shader(&ctx, 0x100200d80, source);
emit_shader_iova(&ctx, cs, 0x100200d80);
Replaying Edited Command Stream

- The decompiler emits C code with raw commands
- The replay tool takes original submissions capture:
  - Finds unused memory range
  - Emits edited command stream there
  - Overrides target submission
Dumping GPU Memory

- Dumping GPU memory is simple to implement
- Act of copying may disturb GPU caches
- Kernel cooperation is needed to implement it properly:
  - GPU interrupts execution e.g. by faulting
  - Now memory could be read undisturbed
Dumping Shader's Registers

- `print %tmp_regs, %src_reg`
  - `%tmp_regs` - 3 consecutive free regs
    - For 64b address and 32b tmp offset
  - `%src_reg` - a single register to print

Shader Log Entries: 6
[0] 00000004 0.0000
[1] 00000000 0.0000
[2] 00000000 0.0000
[3] 00000000 0.0000
[4] beadc429 -0.3394
[5] beadc429 -0.3394
Dumping Shader's Registers

- Want a nicer print? Just `print $src_reg`?
- You still need to allocate temporary registers
  - What if there are no free regs?
  - Spilling regs may not be that easy at this stage
- Too much trouble for a little gain...
Short Summary

- A tool to replay command stream submissions
- A tool to decompile a command stream into C code
- An option to replay edit command stream
- Helpers to dump GPU memory from the command stream
- Helpers to dump shader registers
Stale Regs In Command Stream
Debugging Stale Registers

- It could be hard to spot stale reg usage:
  - It may appear as a random geometry flicker
  - Game hanging at a random moment
  - Rare CTS test failure
Stomping Registers - Caveats

- Could be a bit tricky if a combination of regs causes an issue
- VK pipelines could be set outside a renderpass
- Doesn't help if stale regs are between draw calls
- Default invalid value may be valid for some registers
Stomping Registers

- We mark each register with where it is used:

```xml
<reg32 offset="0x9600" name="VPC_DBG_ECO_CNTL" usage="cmd"/>
<reg32 offset="0xb987" name="HLSQ_CS_CNTL" variants="A6XX" usage="cmd"/>
<reg32 offset="0xb984" name="HLSQ_CONTROL_3_REG" variants="A6XX" usage="rp"/>
<reg32 offset="0xb985" name="HLSQ_CONTROL_4_REG" variants="A6XX" usage="rp"/>
```

- To stomp register you need to specify:

```bash
export TU_DEBUG_STALE_REGS_RANGE=0x0c00,0xbe01
export TU_DEBUG_STALE_REGS_FLAGS=cmdbuf,renderpass
```
Turnip Tooling - Summary

• Unique tooling:
  ◦ Driver breadcrumbs
  ◦ Command stream replaying and editing
    ■ GPU memory dumping
    ■ Shader register dumping
  ◦ Debug option to find stale reg usage
Other Drivers and Tooling
Generic

- GFR - Graphics Flight Recorder
  - Instruments command buffers with completion tags
    - Uses `VK_AMD_buffer_marker` (nothing vendor specific)
- In vkd3d-proton:
  - Breadcrumbs
  - Shader printf
  - Descriptor debugging
Other Mesa Drivers

- Feature toggles and debug flags
- Shader assembly replacement for debugging
- GPU submissions decoding
- GPU crash dumps decoding
Radeon - UMR

- GPU register dumps
- SGPR / VGPR shader register dumps
- Shader wavefront Debugging
- Shader disassembly around the crash site
- See Maister's blog post for it in action

https://themaister.net/blog/2023/08/20/hardcore-vulkan-debugging-digging-deep-on-linux-amd-gpu/
Unreleased - Radeon - Shader Debugger
Proprietary - Radeon™ GPU Detective

- Postmortem analysis of GPU crashes
- Information about page faults
- Breadcrumbs reflecting done and in-progress GPU work

Command Buffer ID: 0x107c

[>] "Frame 1040 CL0"
  ├─[X] "Depth + Normal + Motion Vector PrePass"
  ├─[>] "DownSamplePS"
  │  └─[X] Draw
  │  ├─[>] Draw
  │  └─[>] "Bloom"
  │     ├─[>] "BlurPS"
  │     │  └─[>] Draw
  │     │     └─[>] Draw
  │     └─[ ] Draw
  │         └─[ ] "BlurPS"
Proprietary - NVIDIA Aftermath

- Collects GPU “mini-dumps”
- Visualizes GPU state at the moment of crash
- Collects breadcrumbs
- Shows crashing shader and its registers
Q&A

- Any good tools I haven't mentioned?
- Maybe you tried something before?
- Maybe you have an idea for a tool to implement?