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

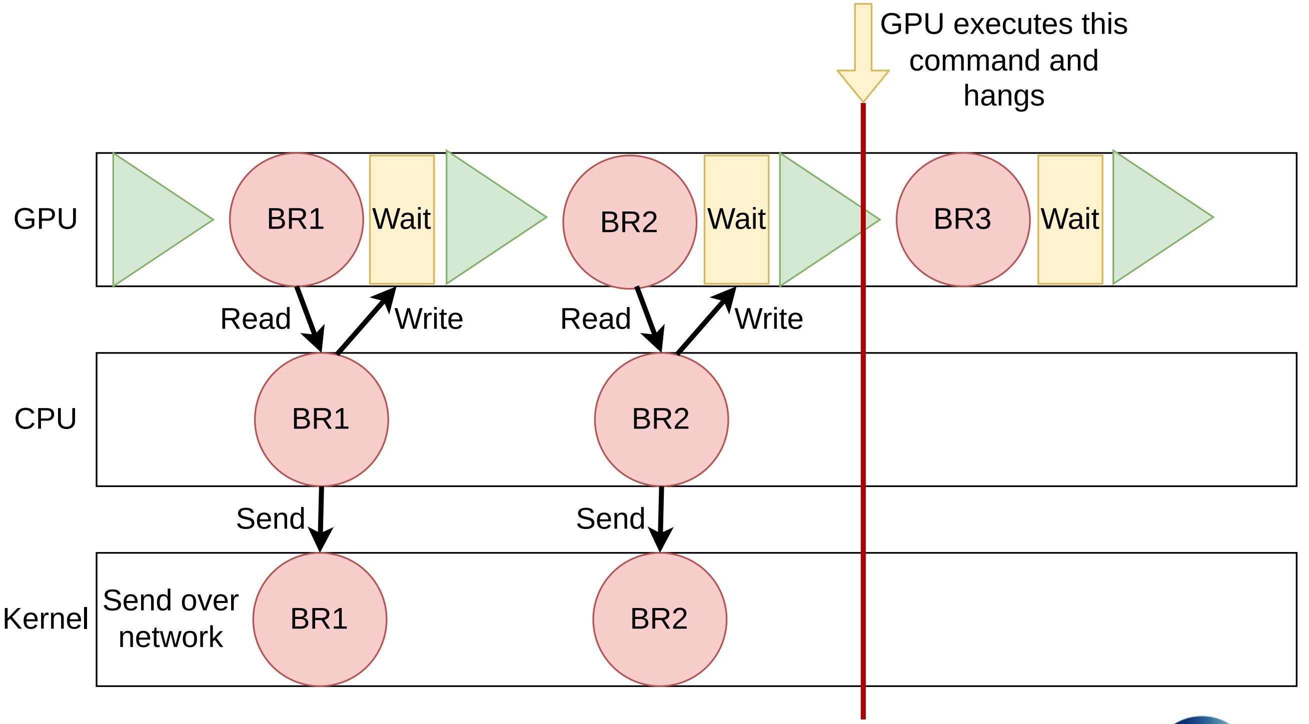
- Unrecoverable hangs are rarely caused by sync issues
- Cannot allow GPU to race ahead of the last know breadcrumb
- A hang may happen asynchronously to the GPU packet that triggered it, e.g.
  - A job is scheduled to another GPU unit
  - That GPU unit hangs some time afterwards
  - There may not be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



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=$BREAKPOINT:$BREAKPOI
```

- 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:

```
/* 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);
```

Editing Shaders

```
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
```

igalia

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:

```
<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="r
<reg32 offset="0xb985" name="HLSQ_CONTROL_4_REG" variants="A6XX" usage="r
```

- To stomp register you need to specify:

```
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/

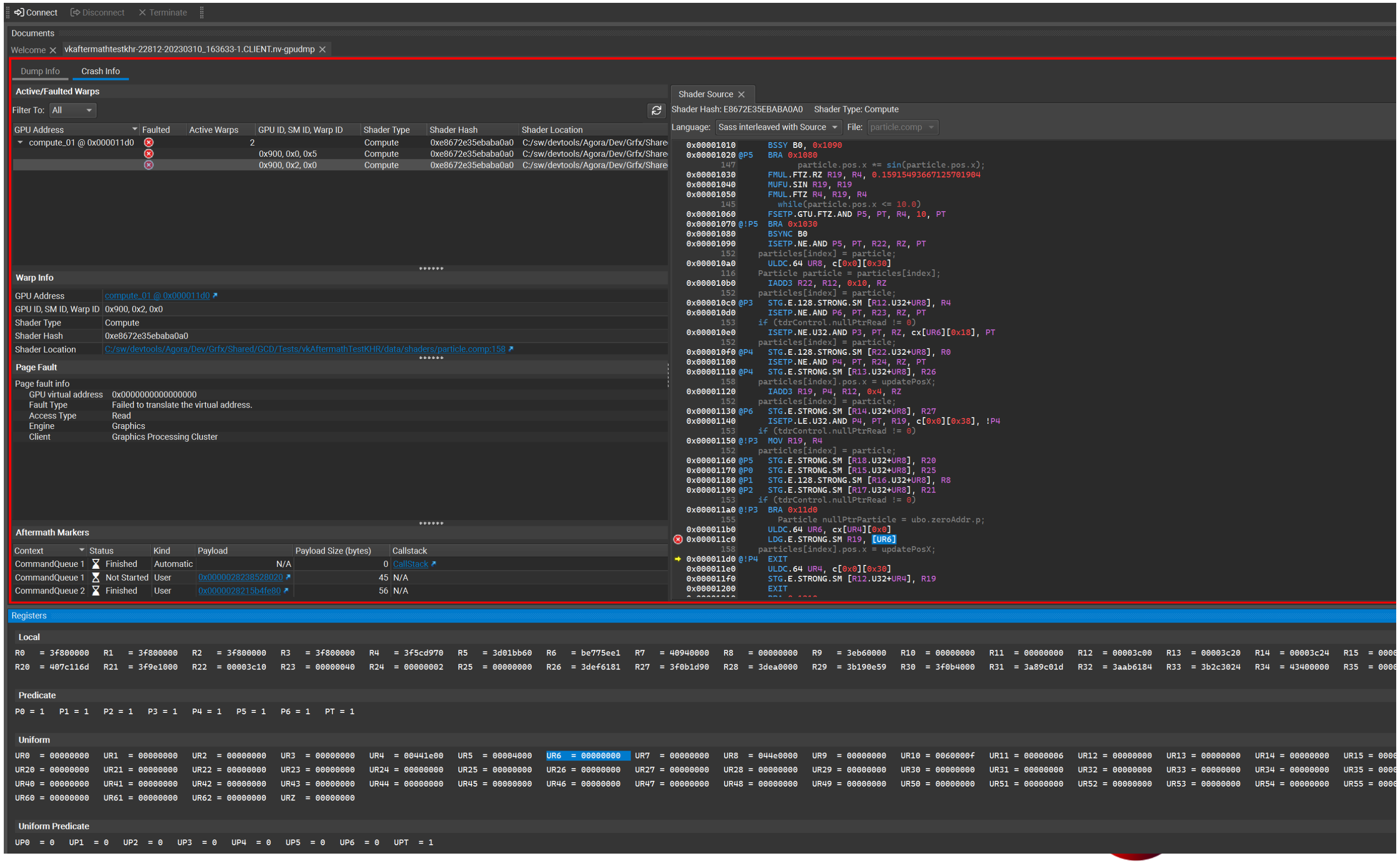
Unreleased - Radeon - Shader Debugger

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



Connect | Disconnect | Terminate

Documents

Welcome ✕ | vkaftermathtestkhr-22812-20230310_163633-1.CLIENT.nv-gpudmp ✕

Dump Info | Crash Info

Active/Faulted Warps

Filter To: All

GPU Address	Faulted	Active Warps	GPU ID, SM ID, Warp ID	Shader Type	Shader Hash	Shader Location
compute_01 @ 0x000011d0	☒	2		Compute	0xe8672e35ebaba0a0	C:/sw/devtools/Agora/Dev/Grfx/Share
	☒		0x900, 0x0, 0x5	Compute	0xe8672e35ebaba0a0	C:/sw/devtools/Agora/Dev/Grfx/Share
	☒		0x900, 0x2, 0x0	Compute	0xe8672e35ebaba0a0	C:/sw/devtools/Agora/Dev/Grfx/Share

Warp Info

GPU Address	compute_01 @ 0x000011d0
GPU ID, SM ID, Warp ID	0x900, 0x2, 0x0
Shader Type	Compute
Shader Hash	0xe8672e35ebaba0a0
Shader Location	C:/sw/devtools/Agora/Dev/Grfx/Shared/GCD/Tests/vkAftermathTestKHR/data/shaders/particle.comp:158

Page Fault

Page fault info

GPU virtual address	0x0000000000000000
Fault Type	Failed to translate the virtual address.
Access Type	Read
Engine	Graphics
Client	Graphics Processing Cluster

Aftermath Markers

Context	Status	Kind	Payload	Payload Size (bytes)	Callstack
CommandQueue 1	Finished	Automatic	N/A	0	CallStack
CommandQueue 1	Not Started	User	0x0000028238528020	45	N/A
CommandQueue 2	Finished	User	0x0000028215b4fe80	56	N/A

Shader Source ✕

Shader Hash: E8672E35EBABA0A0 Shader Type: Compute

Language: Sass interleaved with Source File: particle.comp

```
0x00001010       BSSY B0, 0x1090
0x00001020 @P5   BRA 0x1080
       147             particle.pos.x *= sin(particle.pos.x);
0x00001030       FMUL.FTZ.RZ R19, R4, 0.15915493667125701904
0x00001040       MUFU.SIN R19, R19
0x00001050       FMUL.FTZ R4, R19, R4
       145         while(particle.pos.x <= 10.0)
0x00001060       FSETP.GTU.FTZ.AND P5, PT, R4, 10, PT
0x00001070 @!P5  BRA 0x1030
0x00001080       BSYNC B0
0x00001090       ISETP.NE.AND P5, PT, R22, RZ, PT
       152     particles[index] = particle;
0x000010a0       ULDC.64 UR8, c[0x0][0x30]
       116     Particle particle = particles[index];
0x000010b0       IADD3 R22, R12, 0x10, RZ
       152     particles[index] = particle;
0x000010c0 @P3   STG.E.128.STRONG.SM [R12.U32+UR8], R4
0x000010d0       ISETP.NE.AND P6, PT, R23, RZ, PT
       153     if (tdrControl.nullPtrRead != 0)
0x000010e0       ISETP.NE.U32.AND P3, PT, RZ, cx[UR6][0x18], PT
       152     particles[index] = particle;
0x000010f0 @P4   STG.E.128.STRONG.SM [R22.U32+UR8], R0
0x00001100       ISETP.NE.AND P4, PT, R24, RZ, PT
0x00001110 @P4   STG.E.STRONG.SM [R13.U32+UR8], R26
       158     particles[index].pos.x = updatePosX;
0x00001120       IADD3 R19, P4, R12, 0x4, RZ
       152     particles[index] = particle;
0x00001130 @P6   STG.E.STRONG.SM [R14.U32+UR8], R27
0x00001140       ISETP.LE.U32.AND P4, PT, R19, c[0x0][0x38], !P4
       153     if (tdrControl.nullPtrRead != 0)
0x00001150 @!P3  MOV R19, R4
       152     particles[index] = particle;
0x00001160 @P5   STG.E.STRONG.SM [R18.U32+UR8], R20
0x00001170 @P0   STG.E.STRONG.SM [R15.U32+UR8], R25
0x00001180 @P1   STG.E.128.STRONG.SM [R16.U32+UR8], R8
0x00001190 @P2   STG.E.STRONG.SM [R17.U32+UR8], R21
       153     if (tdrControl.nullPtrRead != 0)
0x000011a0 @!P3  BRA 0x11d0
       155         Particle nullPtrParticle = ubo.zeroAddr.p;
0x000011b0       ULDC.64 UR6, cx[UR4][0x0]
0x000011c0       LDG.E.STRONG.SM R19, [UR6]
       158     particles[index].pos.x = updatePosX;
0x000011d0 @!P4  EXIT
0x000011e0       ULDC.64 UR4, c[0x0][0x30]
0x000011f0       STG.E.STRONG.SM [R12.U32+UR4], R19
0x00001200       EXIT
```

Registers

Local

```
R0   = 3f800000   R1   = 3f800000   R2   = 3f800000   R3   = 3f800000   R4   = 3f5cd970   R5   = 3d01bb60   R6   = be775ee1   R7   = 40940000   R8   = 00000000   R9   = 3eb60000   R10  = 00000000   R11  = 00000000   R12  = 00003c00   R13  = 00003c20   R14  = 00003c24   R15  = 000
R20  = 407c116d   R21  = 3f9e1000   R22  = 00003c10   R23  = 00000040   R24  = 00000002   R25  = 00000000   R26  = 3def6181   R27  = 3f0b1d90   R28  = 3dea0000   R29  = 3b190e59   R30  = 3f0b4000   R31  = 3a89c01d   R32  = 3aab6184   R33  = 3b2c3024   R34  = 43400000   R35  = 000
```

Predicate

```
P0 = 1   P1 = 1   P2 = 1   P3 = 1   P4 = 1   P5 = 1   P6 = 1   PT = 1
```

Uniform

```
UR0  = 00000000   UR1  = 00000000   UR2  = 00000000   UR3  = 00000000   UR4  = 00441e00   UR5  = 00004000   UR6  = 00000000   UR7  = 00000000   UR8  = 044e0000   UR9  = 00000000   UR10 = 0060000f   UR11 = 00000006   UR12 = 00000000   UR13 = 00000000   UR14 = 00000000   UR15 = 000
UR20 = 00000000   UR21 = 00000000   UR22 = 00000000   UR23 = 00000000   UR24 = 00000000   UR25 = 00000000   UR26 = 00000000   UR27 = 00000000   UR28 = 00000000   UR29 = 00000000   UR30 = 00000000   UR31 = 00000000   UR32 = 00000000   UR33 = 00000000   UR34 = 00000000   UR35 = 000
UR40 = 00000000   UR41 = 00000000   UR42 = 00000000   UR43 = 00000000   UR44 = 00000000   UR45 = 00000000   UR46 = 00000000   UR47 = 00000000   UR48 = 00000000   UR49 = 00000000   UR50 = 00000000   UR51 = 00000000   UR52 = 00000000   UR53 = 00000000   UR54 = 00000000   UR55 = 000
UR60 = 00000000   UR61 = 00000000   UR62 = 00000000   URZ  = 00000000
```

Uniform Predicate

```
UP0  = 0   UP1  = 0   UP2  = 0   UP3  = 0   UP4  = 0   UP5  = 0   UP6  = 0   UPT  = 1
```

Proprietary - NVIDIA Aftermath

- Collects GPU “mini-dumps”
- Visualizes GPU state at the moment of crash
- Collects breadcrumbs
- Shows crashing shader and it 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?

We're hiring!

igalia.com/jobs/

igalia