NVIDIA AFTERMATH:

A NEW WAY OF DEBUGGING CRASHES ON THE GPU

Alex Dunn, 2nd March 2017

NVIDIA AFTERMATH

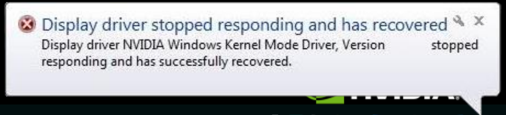
What is it?

- New tool to diagnose GPU crashes, available on GeForce!
- Coming to D3D for broad availability
- Ability to *classify* GPU crashes by location and type
- Can be shipped in game – catch crashes “from the wild”

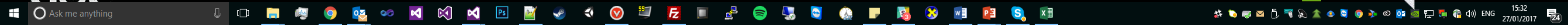
GPU CRASH?

a.k.a. TDR / Hang / Device Removed / Crash/ ?

Annoying → What can we do?


Display driver stopped responding and has recovered

Display driver NVIDIA Windows Kernel Mode Driver, Version stopped responding and has successfully recovered.



GPU DEBUGGING 101

Preventative
Changes timing
Development-use Only
Limited coverage

1st line of defense: MSFT Debug Layer

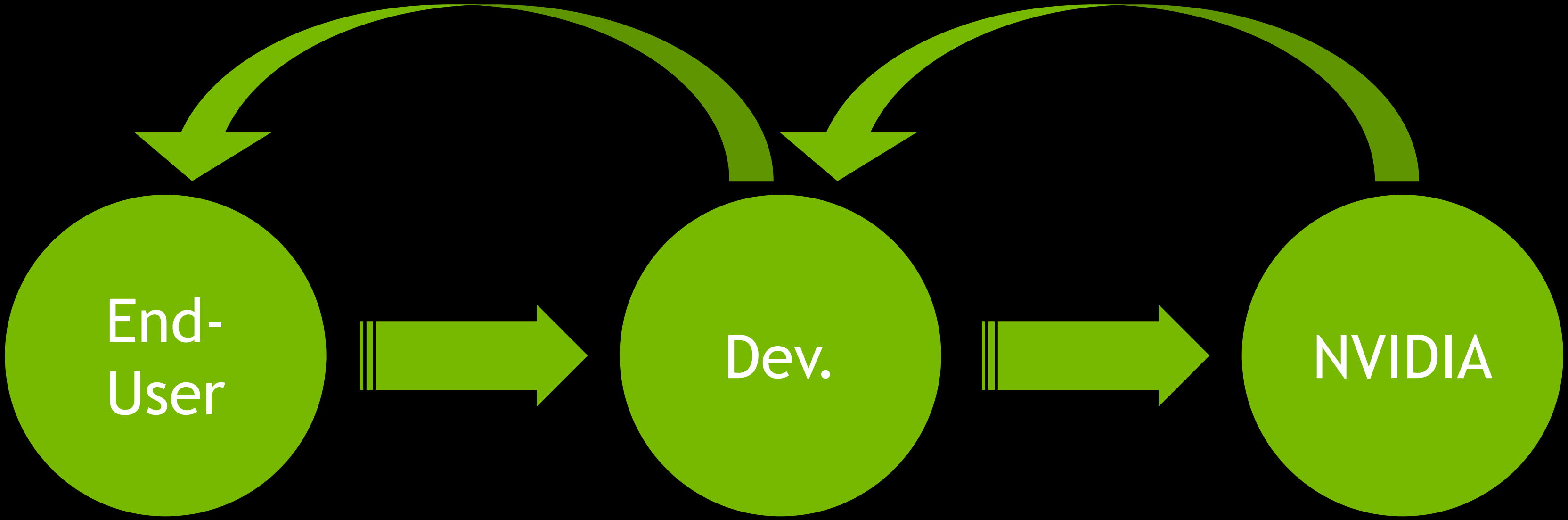
2nd line of defense: MSFT GPU-Based Validation

Final line of defense:
- Catches issues that fall through
- Minimal impact
- Shippable

OBSERVATION

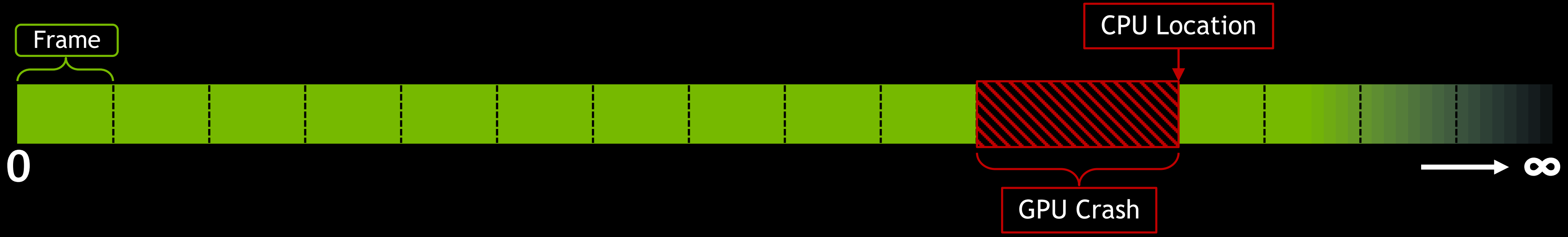
Current state of the art in GPU crash debugging isn’t enough

- There’s no simple way to debug crashes after the fact
- Some bugs take months to resolve (really!!)



DETECTING GPU CRASH #2

1. Crash detected based on error code from API (CPU)
2. Crash happened sometime in the last N frames of GPU commands…
3. CPU call stack is likely a red-herring



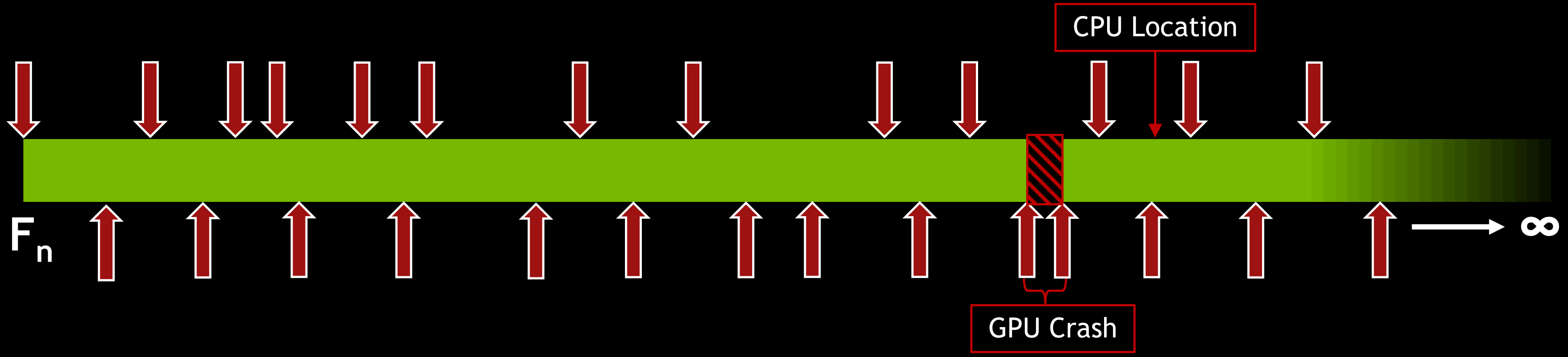
<u>Not useful for debugging!</u>

POC IMPLEMENTATION

KO: Increase accuracy of GPU crash location

Plan:

- Game inserts user-defined markers in the command stream
- GPU signals each marker once reached
- Last marker reached indicates GPU crash location



POC IMPLEMENTATION #2

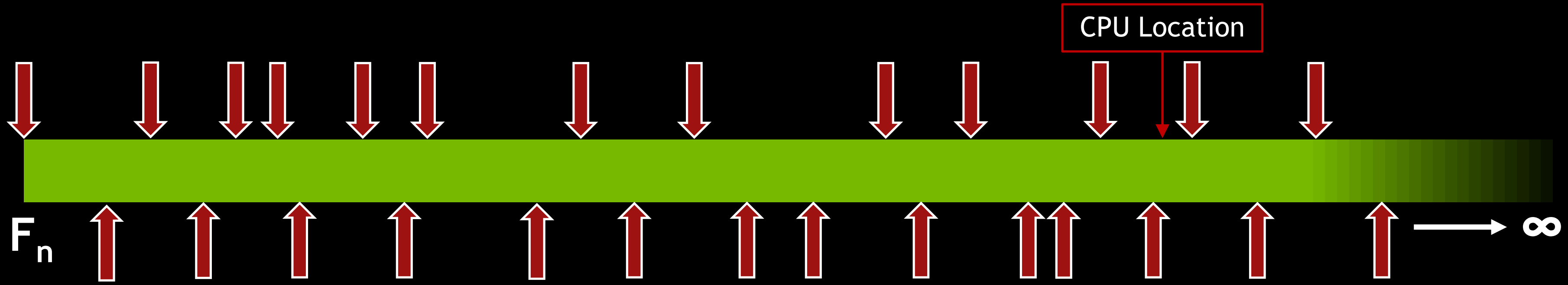
Implemented exclusively via DX12

- CopyBufferRegion inserts markers on GPU timeline
- Write to single memory location per queue
- Globally shared heap → post-crash accessible data

```
void SetMarker(char* markerName) {
      renamingOffset = (renamingOffset + kMarkerSize) % kRingBufferSize;

      const D3D12_RANGE readRange = { 0 };
      const D3D12_RANGE writeRange = { renamingOffset, renamingOffset + min(kMarkerSize,strlen(markerName)) };

      void* mappedDataBegin = nullptr;

      uploadHeap->Map(0, &readRange, &mappedDataBegin);
      {
            memcpy(((uintptr_t)mappedDataBegin + writeRange.Begin), &markerName[0], writeRange.End);
      }
      uploadHeap->Unmap(0, &writeRange);

      commandList->CopyBufferRegion(sharedHeap, 0, uploadHeap, writeRange.Begin, writeRange.End);
}
```

POC IMPLEMENTATION #3

How it looks in practice;



sharedHeap Contents: tiledLighting_Cull

POC IMPLEMENTATION #4

Case Study: IO Interactive

- IO Interactive facing *very* stubborn GPU crash
- Issue was open for >2 months, main focus of weekly meetings with NVIDIA
- With POC, issue was identified and fixed in a single afternoon
- “This tool is excellent” ☺

Conclusion:

- Discovering *where* a hang occurs in GPU timeline is valuable & **actionable**

POC (MINI) POST-MORTEM

Pros:

- Simple API → simple to integrate
- Enabled classification of GPU crashes
- Insight into where GPU crashes occur

Cons:

- GPU copies are super slow for this purpose
- Timing related behavior altered
- Separate process for marker read-back
- Serializes order of GPU work (wait-for-idle)
- Only supports DX12 – DX11 driver too smart

MOVING FORWARD (AFTERMATH)

And so, Aftermath was born…

Take all the *Pros*, leave the *Cons*; polish and improve from there

Make available in C++ library form

Key differences from the POC:

- Marker insertion uses low-level HW features inside driver
- GPU crash reason provided, { timeout, page-fault, … }

GAME INTEGRATION #1

Before other library calls are made:

- `GFSDK_Aftermath_DXxx_Initialize(…)`
- NB. Must return ‘GFSDK_Aftermath_Result_Success’

GAME INTEGRATION #2

To inject an event:

- `GFSDK_Aftermath_DXxx_SetEventMarker(T*, void*, UINT)`

GAME INTEGRATION #3

On a TDR/hang:

- `GFSDK_Aftermath_DXxx_GetData(…)`
- Fetches the last GPU-processed event marker
- Can also fetch the execution state for each GPU!

GAME INTEGRATION #4

```
enum GFSDK_Aftermath_Status
{
    GFSDK_Aftermath_Status_Active = 0,
    GFSDK_Aftermath_Status_Timeout,
    GFSDK_Aftermath_Status_OutOfMemory,
    GFSDK_Aftermath_Status_PageFault,
    GFSDK_Aftermath_Status_Unknown,
};
```

HOW TO ENABLE YOUR GAME*?

1. Grab the Aftermath package from (available on next driver posting):
   https://developer.nvidia.com/nvidia-aftermath

2. Integrate header + DLL into game → compile

3. Rename executable to: “NvAftermath-Enable.exe”

*(to ship in game, contact us)

WORKFLOW - TIPS

- Emit *regime* name as marker:

```
extern ID3D12CommandList* const m_commandList;
extern char* m_marker;

GFSDK_Dx12_SetEventMarker(m_commandList, (void*)m_marker, strlen(m_marker)+1);
```

- Track currently bound PSO?:

```
extern ID3D12CommandList* const m_commandList;
extern ID3D12PipelineState* const m_desiredPSO;

m_commandList->SetPipelineState(m_desiredPSO);

GFSDK_Dx12_SetEventMarker(m_commandList, (void*)m_desiredPSO, 0);
```

- Emit CPU backtrace on every/any API call:

```
extern ID3D12CommandList* const m_commandList;

PVOID stackPtrs[16] = { 0 };
CaptureStackBackTrace(1, 16, stackPtrs, NULL);

GFSDK_Dx12_SetEventMarker(m_commandList, &stackPtrs[0], sizeof(stackPtrs));
```

ROADMAP

What's next? (proposals)

- Expand API support
- Push/Pop marker style
- Page-fault? Supply resource identified!
- ? (feel free to make requests during questions)

NVIDIA working with Microsoft to develop an industry standard

QUESTIONS?

Thank you!

\0

Ref.

1. https://msdn.microsoft.com/en-gb/windows/uwp/gaming/handling-device-lost-scenarios
2. https://msdn.microsoft.com/en-us/library/windows/desktop/bb509553(v=vs.85).aspx
3. http://nvidia.custhelp.com/app/answers/detail/a_id/3335/~/tdr-(timeout-detection-and-recovery)-and-collecting-dump-files
4. https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html#devsandqueues-lost-device
5. https://developer.nvidia.com/nvidia-aftermath