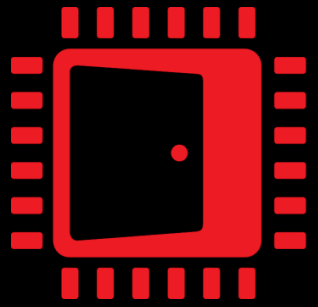


AMD
GPUOpen

# INTRODUCING GPU RESHAPE

## API-AGNOSTIC INSTRUMENTATION & INSTRUCTION LEVEL VALIDATION

MIGUEL PETERSEN

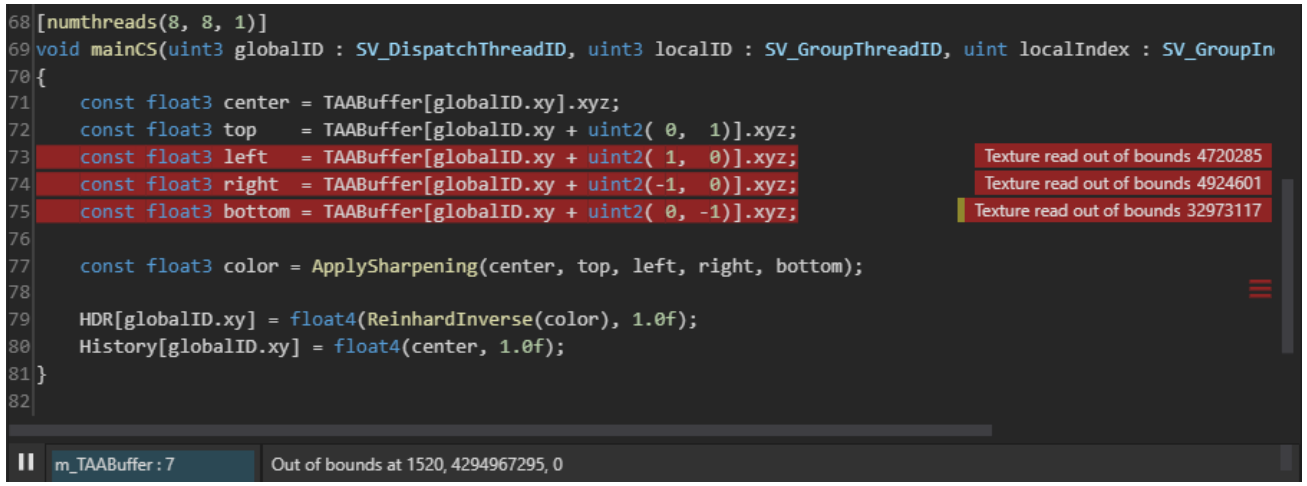
AMD
together we advance_

# THE PROBLEM

- Modern APIs are powerful, but highly complex
- Something inevitably goes wrong
  - What went wrong?
  - Where did it go wrong?
  - How do we know?
- **DXGI_ERROR_DEVICE_REMOVED / VK_DEVICE_LOST**
  - Sometimes not so obvious

# THE PROBLEM

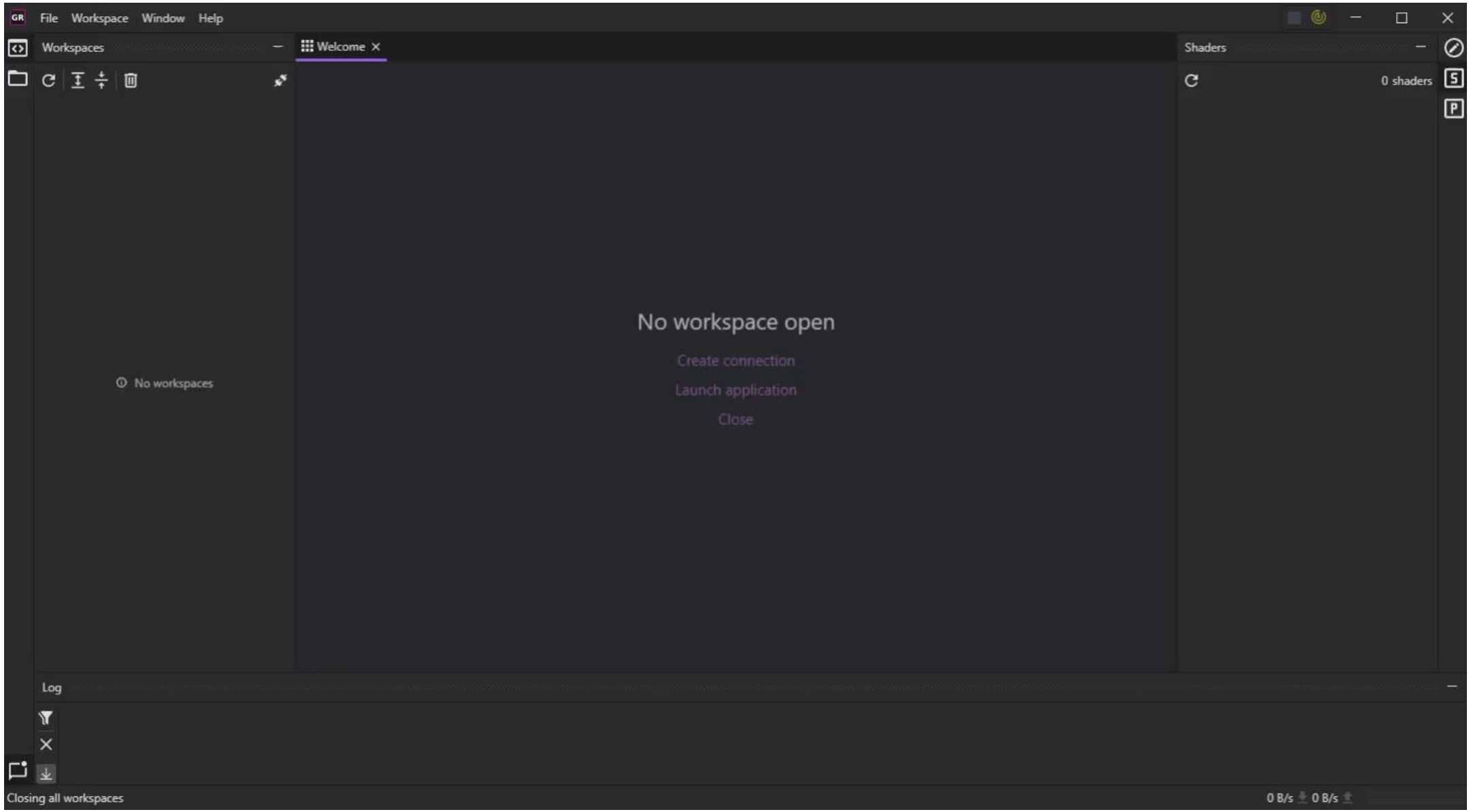
- Excellent validation tooling on the CPU timeline
  - Standard validation layers
  - Limited by available data
- What if the issue occurs on the GPU timeline?
  - May result in undefined behaviour, crashes, or worse
  - Caused by dynamic data not visible on the CPU timeline




  - **This is what GPU Reshape is all about!**

# THE IDEA

- Conceptually, GPU Reshape is simple
  - Before something bad can happen, validate it
  - If something bad did happen, inform the user


File
Workspace
Window
Help
Workspaces
Welcome
Shaders
0 shaders
No workspaces
No workspace open
Create connection
Launch application
Close
Log
Closing all workspaces
0 B/s
0 B/s

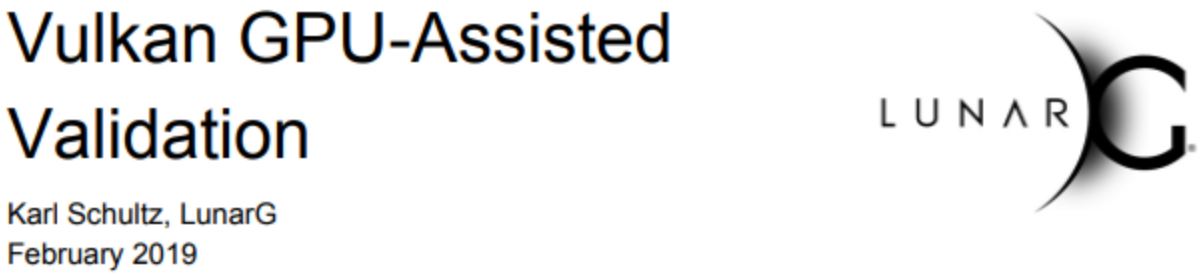
# THE IDEA

- So, what can go wrong? A lot!

| | | |
|---|---|---|
| Element / Texel Out Of Bounds | Exporting Inf / NaN | Invalid Descriptor Indexing |
| Uninitialized Data | Mismatched Descriptors | Race Conditions |
| Infinite Loops (TDR) | Hardware Slow Paths | **And a lot more!** |

# THE IDEA

- Validation takes many forms
  - Static analysis
  - Symbolic analysis
  - Source instrumentation
  - Binary instrumentation
- GPU Reshape is an integration-free framework
  - Leaves only binary instrumentation
- Smarter people have already proved the point



# THE IDEA

- Binary instrumentation transforms code

```
%image = OpLoad %imagePtr
%texel = OpImageRead %f4 %image %index None
```

```
%image = OpLoad %imagePtr
%oob   = OpUGreaterThanEqual %b %index %size
OpSelectionMerge %resume None
OpBranchConditional %oob %fail %resume

%fail = OpLabel
… failure code …
OpBranch %resume

%resume = OpLabel
%texel   = OpImageRead %f4 %image %index None
```

- Inject user programs with validation code
- No modifications needed from the user

# THE IDEA

- Easier to think about with source code

```
float4 texel = image[coordinates];
```

```
if (any(coordinates >= imageSize)) {
    ReportFault();
}

float4 texel = image[coordinates];
```

- Injected validation of image load coordinates
- Numerous projects employ hand-written validation
  - Fully automated through GPU Reshape
  - Not all faults are immediately visible in source code

# THE IDEA

- Certain features may safe-guard operations
  - Faulting operations can cause general instability
  - Limits our ability to stream validation data back

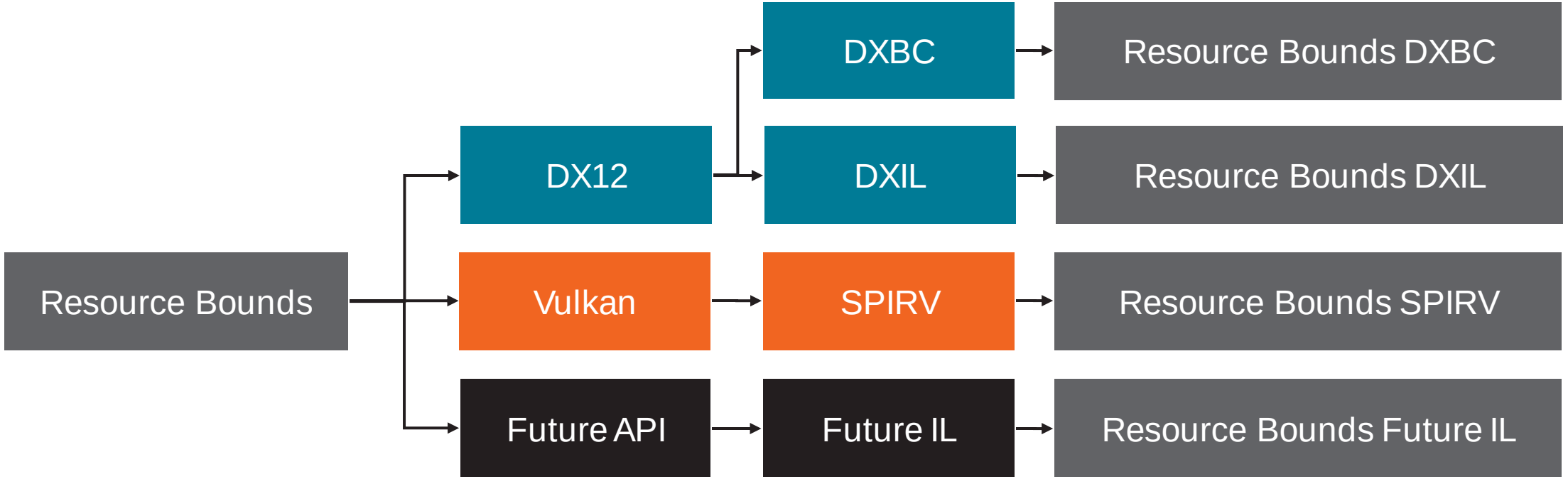
```
if (any(coordinates >= imageSize)) {
   ReportFault();
}

float4 texel = image[coordinates];
```

```
float4 texel;

if (any(coordinates >= imageSize)) {
   ReportFault();

   texel = 0.0f.xxxx;
} else {
   texel = image[coordinates];
}
```

- Guard faulting instructions in a separate branch

# THE IDEA

- Multiple backends, multiple intermediate languages
  - Permutation problem



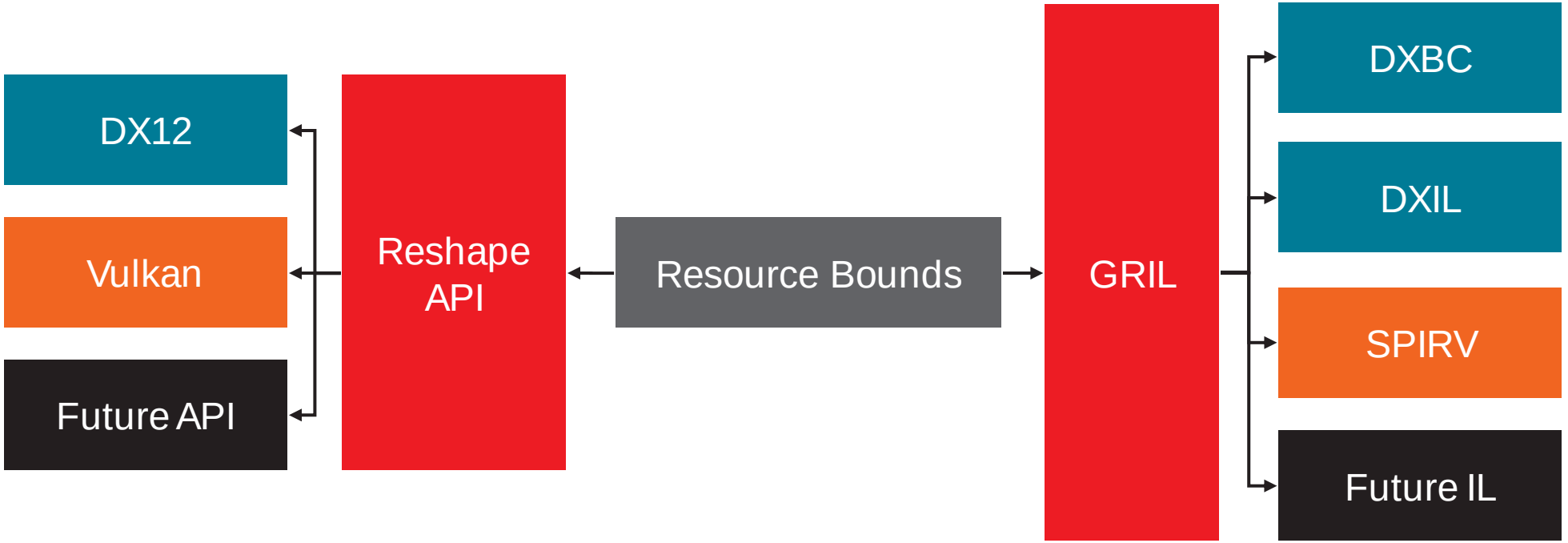
# INTERMEDIATE LANGUAGES / GRIL

- Implementation per backend/intermediate-language infeasible
- Representations may be different between the ILs
  - Concepts are mostly the same
  - We need a common form

**Write once instrument everywhere**

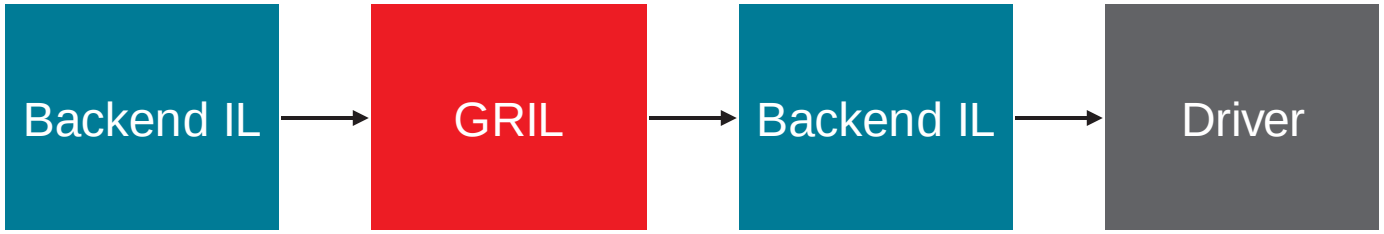
# INTERMEDIATE LANGUAGES

- Shared abstraction
  - Intermediate Languages
  - APIs



# INTERMEDIATE LANGUAGES

- GRIL is heavily LLVM™ inspired
  - Single-static assignment
  - Strong typing system
  - Basic blocks (stream of instructions)
  - Similar programming model
- **All instrumentation happens on GRIL**
  - Bi-directionally translated to and from backend languages
- Native parsing and recompilation




  - Single layer translation
    - No intermediate representations from binary to GRIL
    - Highly performant

# INTERMEDIATE LANGUAGES

- Feature parity with backend languages is not the goal
  - Too much work
  - GRIL only exposes a sub-set of each language
  - Behaviour of unexposed constructs maintained
    - Instructions
    - Constants
    - Etc.
- **Trivial differences** in instructions abstracted away
  - Difference in address spaces
  - Specialized instruction operands
  - Etc.
- Language **paradigm differences** need to be addressed
  - Scalarization/vectorized representations
  - Structured/unstructured control flow
- **Infer when we can, expose when we cannot**

# INTERMEDIATE LANGUAGES

- SPIRV is a vectorized representation
- DXIL is a scalarized representation
- **GRIL follows a vectorized form**
- More work to scalarize SPIRV than to scalarize (instrumented) GRIL
  - DXIL scalarization inferred in the backend

```
float4 a = ...;
float4 b = ...;
a += b;
```

```
float a[4] = ...;
float b[4] = ...;
a[0] += b[0];
a[1] += b[1];
a[2] += b[2];
a[3] += b[3];
```

  - Applies to any vectorized operation (binary, unary, etc.)

# INTERMEDIATE LANGUAGES

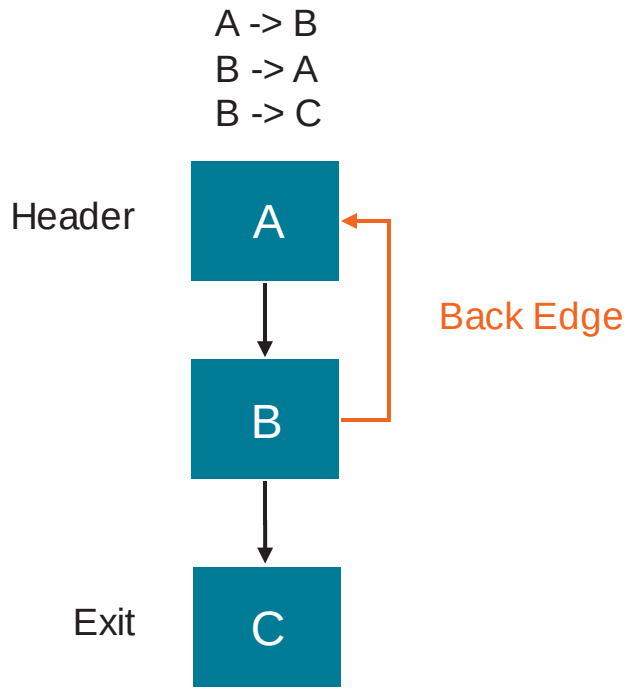
- Structured control flow puts a strict set of requirements on branching
  - SPIRV is fully structured
  - DXIL is unstructured (e.g., goto)
- Inferring structured control flow is difficult, and dangerous
  - Inclusively exposed in the intermediate language
  - Backends may rewrite shaders for relaxed control flow

```
pre.BranchConditional(
    pre.Equal(terminationID, pre.UInt32(1u)),
    terminationBlock,
    selectionMergeBlock,
    // SPIRV Selection Merge Construct
    IL::ControlFlow::Selection(selectionMergeBlock)
);
```

  - Features written with structured control-flow in mind
    - Backends may discard information

# INTERMEDIATE LANGUAGES

- Features may rely on structured control flow constructs
  - Such as Loop manipulation
- **What is a loop really?**
  - It's a for statement! A while statement! In source code ☹
  - What about in ILs? A set of blocks branching to each other
    - Headers represent the entry point
    - Back edges represent the cyclical branching
- Backend ILs may not preserve this information (DXIL)
  - Metadata stripped out
  - Requires reconstruction
- Reshape provides tooling to reconstruct such constructs
  - Lots of literature on this!



# INTERMEDIATE LANGUAGES

- Numerous additional differences
  - Instruction sets
  - Binding models
  - Type representation
  - Constant representation
  - Addressing mechanisms
  - Metadata representation
  - And so forth!
- Not all that fun to talk about
- **Given compliance, translation is seamless**

# BUILDING BLOCKS

- Instrumentation is half the battle
- Features never interact with the APIs

# BUILDING BLOCKS

- GPU Reshape is a collection of building blocks
- API abstractions
  - Data streaming and synchronization
  - Resource management
  - Descriptor management
- Standardized functionality
  - GRIL manipulation
    - Instruction emitters
    - Basic block splitting
  - Analysis passes
    - Dominator/loop trees
    - Conditional constant propagation
- Some more interesting than others

# BUILDING BLOCKS

- Validation data streaming
  - Something bad happened, stream back the details
  - Backends handle state management and synchronization
- Streaming data from GRIL is a one-liner

```
// Export the message
ResourceRaceConditionMessage::ShaderExport msg;
msg.SGUID = oob.UInt32(sguid);
msg.LUID  = eventDataID;
oob.Export(exportID, msg); // Send it!
```

  - Full interoperability with GPU, CPU, and networking friendly
  - No post processing needed, send straight to the UI for presentation
  - Binding code generated from schema files
    - GRIL
    - C++
    - C#

# BUILDING BLOCKS

- Descriptor management
  - One of the biggest differences between APIs
  - Features mostly want to discern handles with ids and metadata
- Abstracted as **Resource Tokens**
  - Physical Unique ID
  - Resource Type (Texture, Buffer, CBuffer, Sampler)
  - Sub-resource Base (Slices, Mips, Etc.)
- Exposed in GRIL as a one-liner

```
IL::ResourceTokenEmitter token(pre, resourceHandle);

// Get token details
IL::ID PUID = token.GetPUID();
IL::ID SRB  = token.GetSRB();
```

  - Single (register) vectorized instruction with a couple scalarized

# BUILDING BLOCKS

- Feature programs
  - Shaders written entirely in GRIL
  - Translated to backend language

```
void SRBMaskingShaderProgram::Inject(IL::Program &program) {
   ... omitted few setup lines
   IL::Emitter<> emitter(program, *basicBlock, basicBlock->GetTerminator());

   // Get current mask
   IL::ID srbMask = emitter.Extract(emitter.LoadBuffer(bufferID, puidEventDataID), 0u);

   // Bit-Or with desired mask
   IL::ID bufferID = emitter.Load(initializationMaskBufferDataID);
   emitter.StoreBuffer(bufferID, puidEventDataID, emitter.BitOr(srbMask, maskEventDataID));
}
```

- Features can manipulate state independent of shader operations
  - Same programming model as instrumentation
  - Minimal work to support it

# BUILDING BLOCKS

- Command abstraction

```
CommandBuilder builder(context->buffer);
builder.SetShaderProgram(srbMaskingShaderProgramID);
builder.SetEventData(srbMaskingShaderProgram->GetPUIDEventID(), static_cast<uint32_t>(puid));
builder.SetEventData(srbMaskingShaderProgram->GetMaskEventID(), ~0u);
builder.Dispatch(1, 1, 1);
```

  - Inject arbitrary commands prior to user operations
    - Supply instrumentation data to pending dispatch/draw
      - “User called you with 13 vertices!”
      - Push/root constants, descriptor data, etc.
    - Execute feature programs
  - Anything the feature needs
- Submit commands independent of user operations

```
scheduler->Schedule(Queue::Compute, buffer);
```

# FEATURES

- So now that we have everything
- How are we using it?

# FEATURES

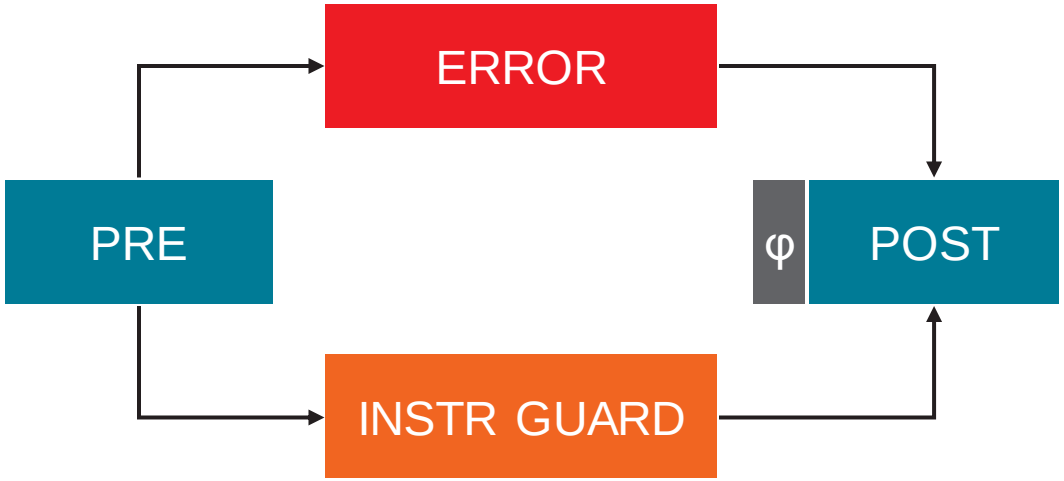
- Most features follow the same doctrine
  - Find all potentially faulting instructions


  - Validate operands prior to instruction
  - Split the basic block according to needs


- Simple splitting allocates an ERROR block
  - Conditionally branched to if a fault was detected
  - ERROR exports validation data
  - POST acts as structured merge block

# FEATURES

- Safe-Guarding splitting requires an additional block
  - Migrate dangerous instruction to guarded block
  - Allocate dummy values in case of an error




- POST block merges instruction result $\varphi(ERROR, GUARD)$
  - $\varphi$ selects a value based on the control flow predecessor
  - $value = wasError\ ?\ dummyValue : instrValue$

# RESOURCE BOUNDS

- Validation of texel/element addressing in bounded resources

```
71    const float3 center = TAABuffer[globalID.xy].xyz;
72    const float3 top    = TAABuffer[globalID.xy + uint2( 0,  1)].xyz;
73    const float3 left   = TAABuffer[globalID.xy + uint2( 1,  0)].xyz;
74    const float3 right  = TAABuffer[globalID.xy + uint2(-1,  0)].xyz;
75    const float3 bottom = TAABuffer[globalID.xy + uint2( 0, -1)].xyz;
```

```
Texture read out of bounds 2672405
Texture read out of bounds 3340465
Texture read out of bounds 35745133
```

  - [RW]Buffer / [RW]StructuredBuffer / [RW]Texture[…]
- Most functionality supplied by hardware/ILs

```
IL::ID cond = pre.Any(pre.GreaterThanEqual(index, pre.ResourceSize(instr->buffer)));
```

  - SPIRV **OpImageQuerySize**
  - DXIL **@dx.op.getDimensions**
- Let GRIL handle the heavyweight work
  - Just assume vectorization
  - Export data on errors

# EXPORT STABILITY

- Validation of floating-point stability on export operations

```
85 #ifdef ID_TANGENT
86     Output.Tangent = normalize(vec3(transMatrix * vec4(a_Tangent.xyz, 0.0)));
87     Output.Binormal = cross(Output.Normal, Output.Tangent) * a_Tangent.w;
88 #endif
```

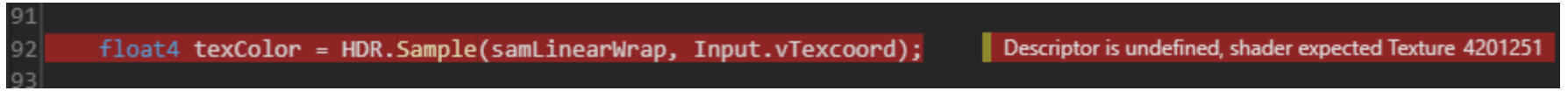
Exporting NaN 2171
Exporting NaN 2171

  - Writes to unordered access views
  - Writes to render targets
  - Writes to inter-stage structures (e.g., vertex exports)
- Very simple test

```
IL::ID isInf = pre.Any(pre.IsInf(value));
IL::ID isNaN = pre.Any(pre.IsNaN(value));
```

# DESCRIPTORS

- Validation of descriptor validity




  - Undefined
  - Out of bounds indexing
  - Compile-time to runtime mismatch
  - Missing table bindings
- **Resource Token** abstraction provides all the data needed
  - Fully guarded
  - Reports exact descriptor present

```
IL::ID runtimeType = IL::ResourceTokenEmitter(pre, resourceHandle).GetType();
IL::ID mismatch    = pre.NotEqual(compileType, runtimeType);
```

  - Feature validates the runtime descriptor type against instruction
- Guarding of instruction using descriptor data

# INITIALIZATION

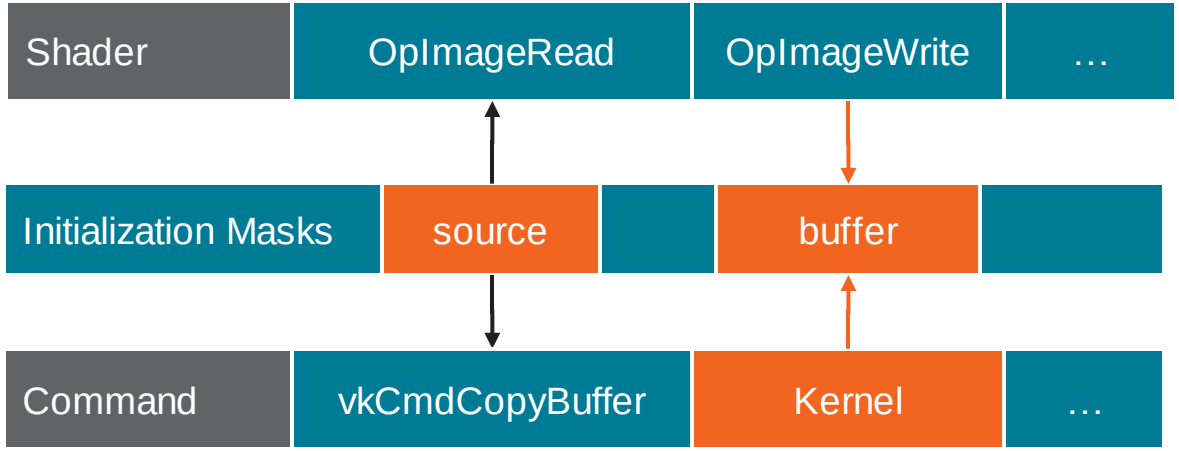
- Validation of resource writes prior to reads

```
43        for (int x = -1; x <= 1; ++x)
44        {
45            const float2 st = uv + float2(x, y) * texelSize;
46            const float depth = DepthBuffer.SampleLevel(DepthSampler, st, 0.0f).x;   Uninitialized resource read 11251
47            if (depth < closestDepth)
```

  - Myriad of ways resources can be initialized
    - Command buffers: Clears/Render Pass flags/Copies/…
    - Shaders: UAV writes
- Tracked initialization masks in a persistent buffer
  - Indexed by resource token physical UID
  - Granularity on a per-resource level
    - Sub-resource tracking coming later

# INITIALIZATION

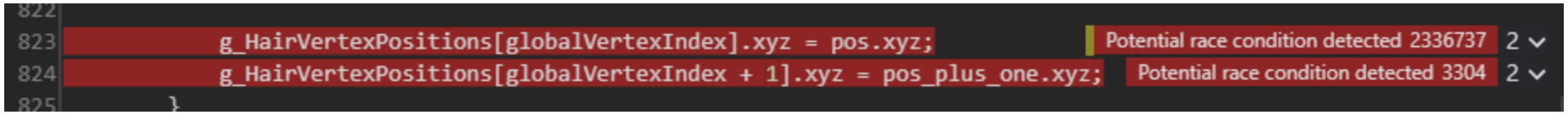
- Mask initialization must occur in shader




  - Reads validate mask against expected state
  - Writes atomically assign mask bits
  - Command buffer writes (e.g., copies) launch a separate kernel for initialization logic
- Transfer/copy queues are emulated
  - Cannot execute compute kernels on native queues
  - Transparent to the application

# CONCURRENCY

- Validation of single-producer/multiple-consumer relations

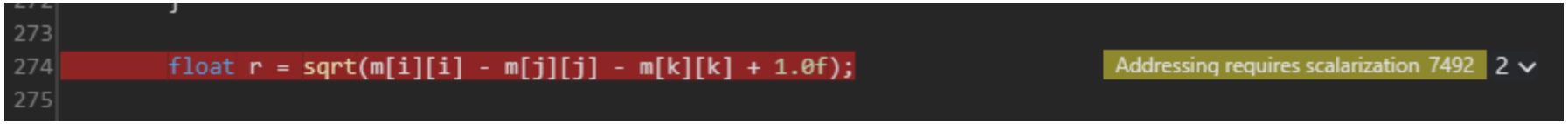



  - Granularity between events (draw, dispatch, etc.) and queues
- Atomic guards on resource operations (writes, loads, samples, etc.)


  - Single atomic CAS operation
  - If lock failed, and not the current event lock id, potential race condition
- Persistent resource lock states across the device
  - Events (e.g., draws) allocate a 32-bit identifier representing a lock (push/root constant)
  - Same mechanism as initialization validation
    - Command buffer induced race conditions not implemented yet
- Not a hazard check

# WATERFALLING

- Validation of waterfalling conditions




  - Serialization of **dynamic register indexing** (S/VGPR)
  - Architecturally specific (AMD)
  - Performance implications
- Local addressing is serialized if both
  - The data accessed cannot be deduced at compile time
  - The indexing requested cannot be deduced at compile time
- Constant data can be moved to memory (`global_load_dword`)
- Constant indexing can (try to) inline the element

# WATERFALLING

- Serialization commonly takes two forms
  - Set of conditional masking instructions for small data types, **not free**

```
  v_cndmask_b32  v2, v3, v2, vcc_lo
  v_cmp_eq_i32  vcc_lo, 0, v4
  v_cndmask_b32  v1, v2, v1, s0
  v_cndmask_b32  v0, v1, v0, vcc_lo
```

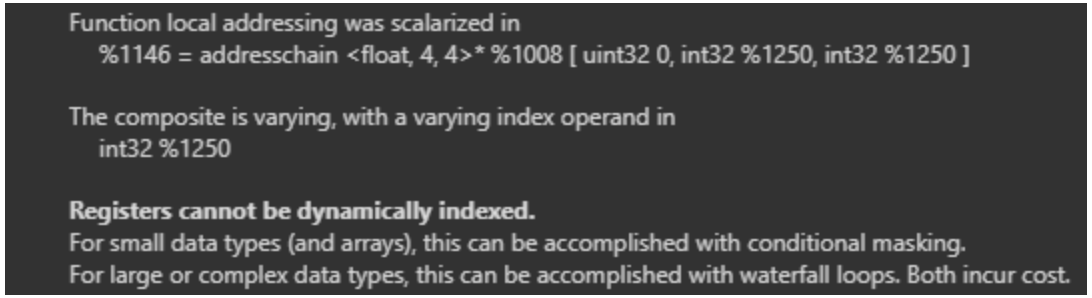
  - "Waterfall" loop for large data types and descriptors, **expensive**

```
  s_mov_b32     exec_lo, s1
label_00B4:
  s_mov_b32     vcc_lo, exec_lo
  v_readfirstlane_b32  s2, v12
  s_mov_b32     m0, s2
  v_cmpx_eq_i32  exec_lo, s2, v12
  v_movrels_b32  v0, v0
  s_andn2_b32   exec_lo, vcc_lo, exec_lo
  s_cbranch_execnz  label_00B4
```

    - Actual loop, reduces execution mask by unique value grouping until done

# WATERFALLING

- Validation is non-trivial ☹
- Determine if either can be constant-folded
  - Compilers resolve this through a chain of optimization passes
  - SSA-Rewrite > Loop-Unrolling > CCP > …
- Conditional Constant Propagation (CCP) with Constant Folding
  - *D. Novillo, "A propagation engine for GCC", GCC Developers Summit, pages 175–185, 2005*
  - *Mark N. Wegman and F. Kenneth Zadeck, "Constant propagation with conditional branches", ACM Trans. Program. Lang. Syst. 13, 2 (April 1991), 181–210. 10.1145/103135.103136*
  - Conservative Load/Store Propagation
  - Simulated Loop Propagation
- Validation checks if either the data or indexing may be constant folded
  - Exceptions apply, but good estimate!

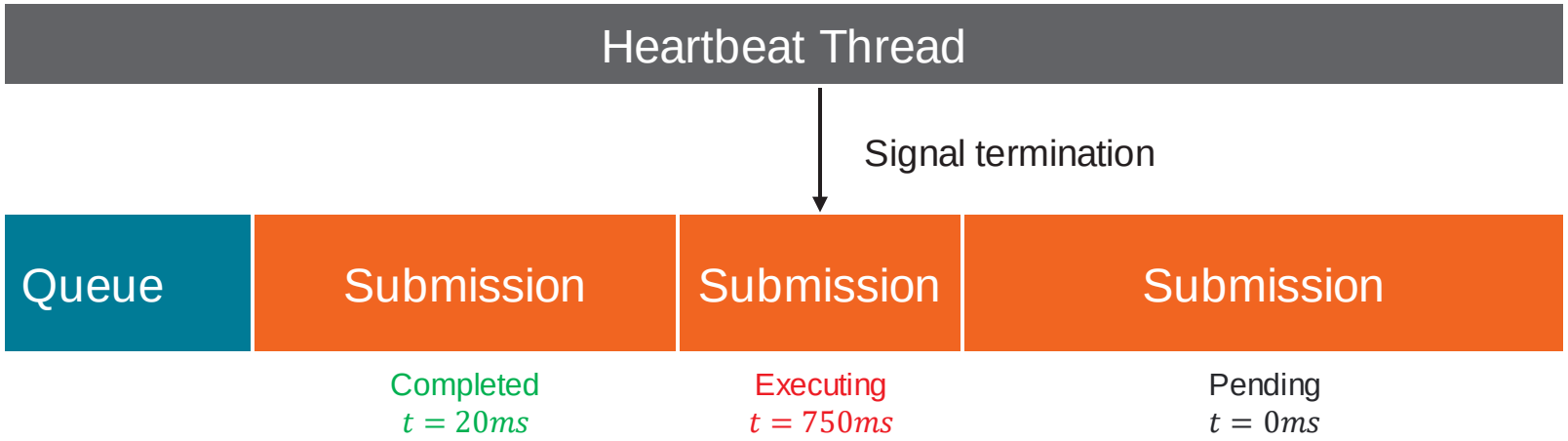


# LOOPS (EXPERIMENTAL)

- Guarding of potentially infinite/TDR loops

```
1598    %1558 = extract { int32, int32, int32, int32 } %1557 3
1599    %1559 = mul int32 %1558 int32 25234234
1600    %1560 = lessthan int32 %557 int32 %1559
1601    branchconditional bool %1560 label %175 label %286        Loop timeout 263567
```

  - Escape loops before potential driver timeouts
- CPU heartbeat thread
  - Monitors all active submissions
  - Signals termination if elapsed time exceeds threshold

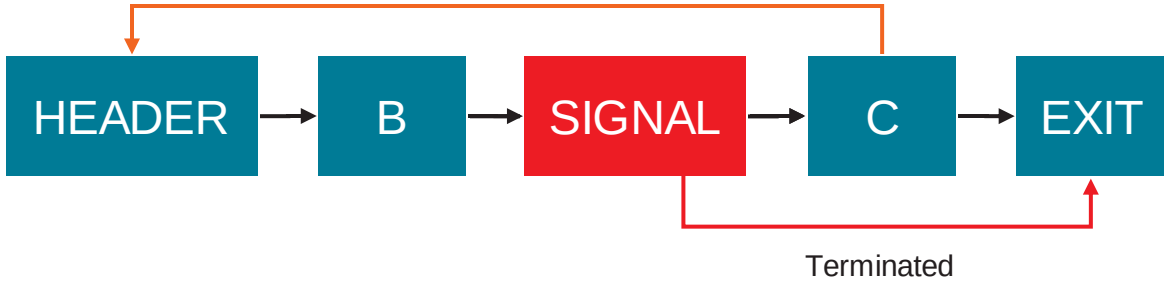


# LOOPS (EXPERIMENTAL)

- Loop headers atomically read signal each iteration

```
IL::ID signal = pre.AtomicAnd(pre.AddressOf(buffer, submissionID), pre.UInt32(1u));
```

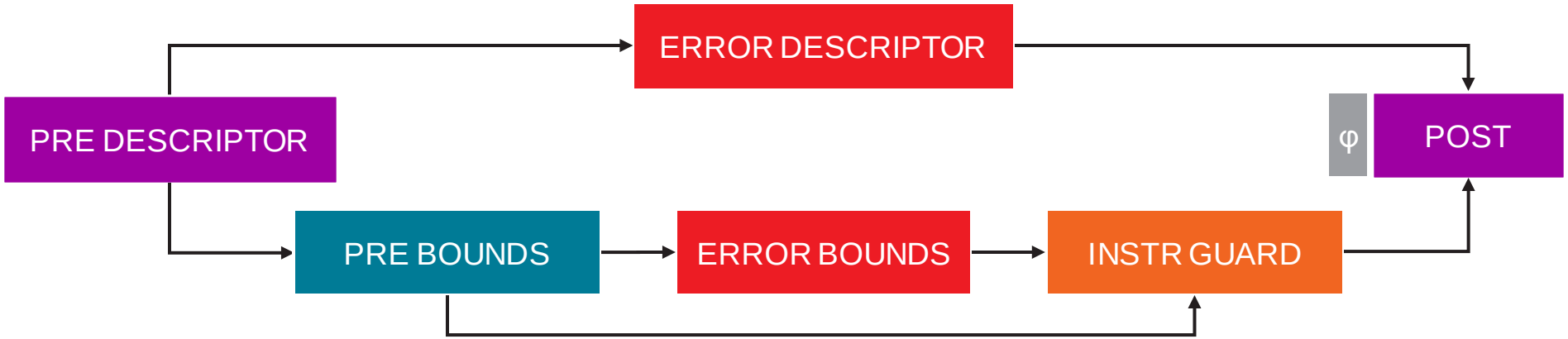
  - If signaled for termination, escape the loop




  - Unstructured programs reconstruct loop tree
  - Branching to loop exits requires resolving $\varphi$ merges
    - $\varphi(B_0 \dots, B_n) \rightarrow \varphi(B_0 \dots, B_n,\ B_{SIGNAL})$
- Unsolved problem is getting data to a running shader
  - **Makes architectural assumptions** as of today

# FEATURES

- Features are not infallible
  - Validation must never produce issues
  - Resource Bounds validation expects valid descriptors
    - Size queried on buffer/texture descriptors
    - **Invalid descriptors will fault the GPU**
- Add feature dependencies
  - Hierarchical instrumentation
  - Resource Bounds / Initialization / Etc. → Descriptors (Safe Guarded)



# FEATURES

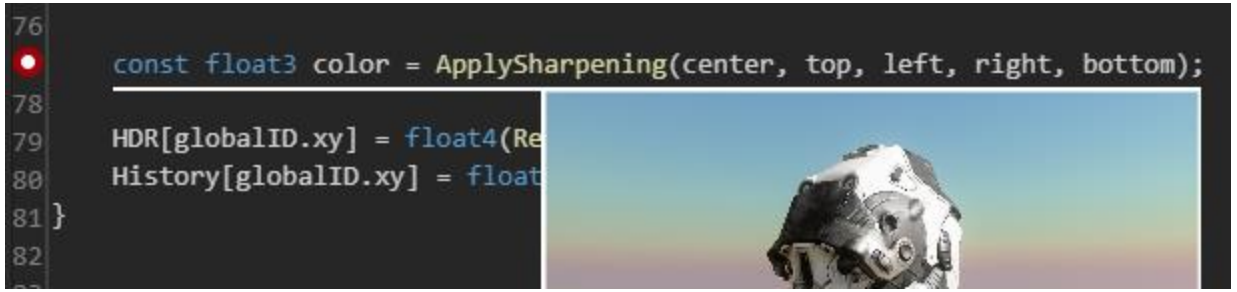
- Not a monolithic framework
  - All features exposed as plugins
  - Backends are entirely decoupled from features
- Backends are kept as minimal as possible
  - Heavy lifting in the abstracted layer
  - Keeps things clean
- Current feature scope constrained to validation
  - Let’s get one thing right before the next
  - Exciting things in the works!

# WHAT ABOUT TOMORROW?

- Instrumentation is here to stay
- Road map for future features
  - **Debugging**
  - **Profiling**

# WHAT ABOUT TOMORROW?

- Full fledged **in-shader debugging**
- See exactly what shaders see with “live” instruction breakpoints




(Not a real screenshot)

  - Realtime, as it is happening
  - Visualize values however you please (e.g., 2D texture for post processing debugging)
- Make shader assertions common place

```
assert(roughness > kGGXMinRoughness, "Invalid roughness encoded");
```

  - Staple of the CPU world
  - Requires source integration/annotation

# WHAT ABOUT TOMORROW?

- **In-shader profiling**
- Inspect branch coherence and coverage in real-time
  - Turn the camera, another branch lit up!
  - Diagnose highly divergent paths

```
88    if (bAreWeInMagnifierOnScreenBorderRegion || bAreWeInMagnifiedAreaBorderRegion)
89    {
90        outColor.r = fBorderColorRGB[0];
91        outColor.g = fBorderColorRGB[1];
92        outColor.b = fBorderColorRGB[2];
93    }
94    if (bAreWeInMagnifierOnScreenRegion)
95    {
96        float2 sampleUVOffsetFromCenter = uv - uvMagnifierOnScreen;
97        sampleUVOffsetFromCenter /= fMagnificationAmount;
98        const float2 magnifierUV = uvMagnifiedArea + sampleUVOffsetFromCenter;
99        outColor = srcColor.SampleLevel(samPoint, magnifierUV, 0);
100       return outColor;
101   }
```

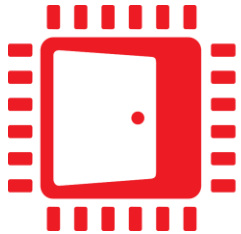
(Not a real screenshot)

- Inspect branch timings in real-time, where is the shader spending its time?
  - Some challenges with (driver) pipeline reordering

# WHAT ABOUT TOMORROW?

- I don’t see instrumentation as something niche
  - Has serious potential to become part of everyday development
  - Offers a unique way to unbox the GPU
- A long road ahead
  - Numerous features planned
  - Ongoing stabilization efforts
- A fully open-source collaboration
  - For issues, proposals, and general discussion, please reach out!
  - https://github.com/GPUOpen-Tools/GPU-Reshape
- **Genuine thanks**
  - Avalanche Studios Group
  - Advanced Micro Devices
  - Striking Distance Studios

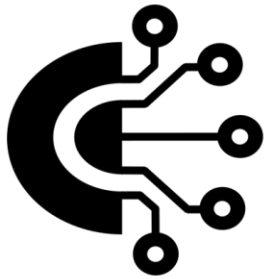
# ANY QUESTIONS?



AMD GPUOpen

AMD RADEON Developer Tool Suite

https://gpuopen.com/tools/



GPU Reshape

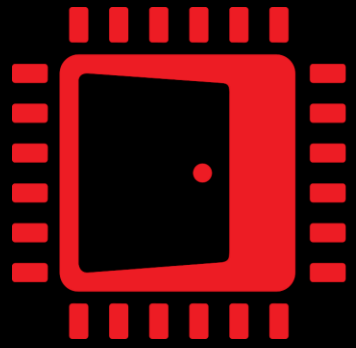
Avalanche Studios Group

https://gpuopen.com/gpu-reshape/

# DISCLAIMER

GENERAL DISCLAIMER

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18



AMD
GPUOpen

AMD
together we advance_

AMD
EPYC

AMD
RYZEN

AMD
RADEON