Towards Truly Dynamic Instrumentation on AMD GPUs With Luthier

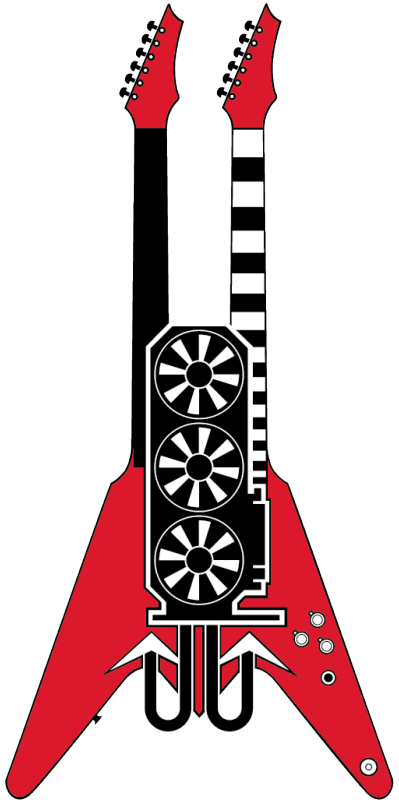
Matin Raayai-Ardakani

Northeastern University Computer Architecture Research Lab (NUCAR)

Boston MA, USA

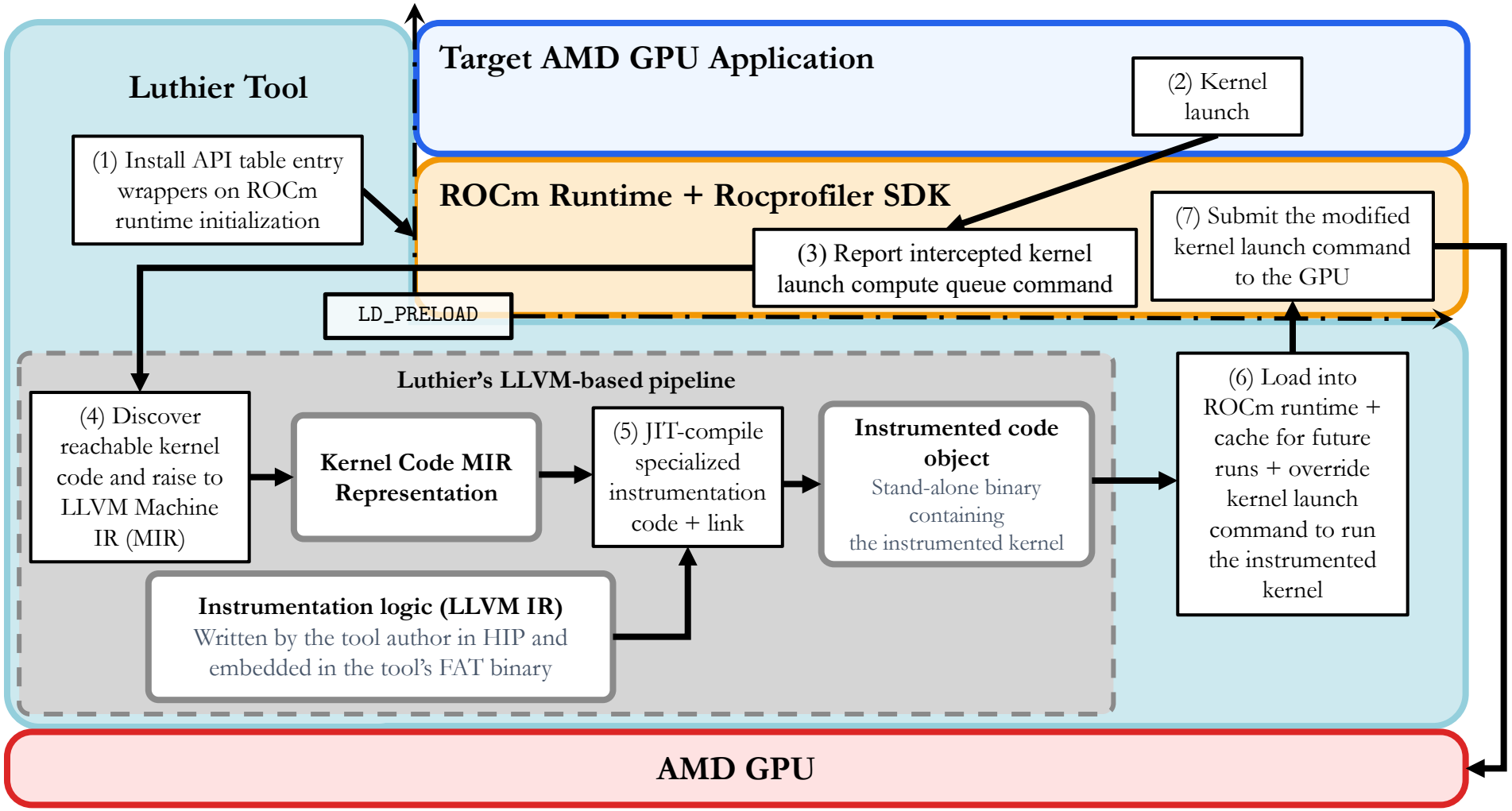
What is Luthier?

- An open-source dynamic binary instrumentation (DBI) framework targeting AMD GPU compute applications
- Supports:
  - Instrumentation at the device ISA level
  - Injection of high-level device functions and low-level assembly instructions
  - Inspection and modification of ISA visible state
  - Selective instrumentation of kernels
  - All ROCm-compatible GPU runtimes (e.g., OpenMP, OpenCL, HIP, Kokkos, SYCL)
- Built using AMD ROCm components
  - Can be used with other ROCm tooling libraries (e.g., rocprofiler-sdk)
  - Provides a fully transparent and debuggable instrumentation process



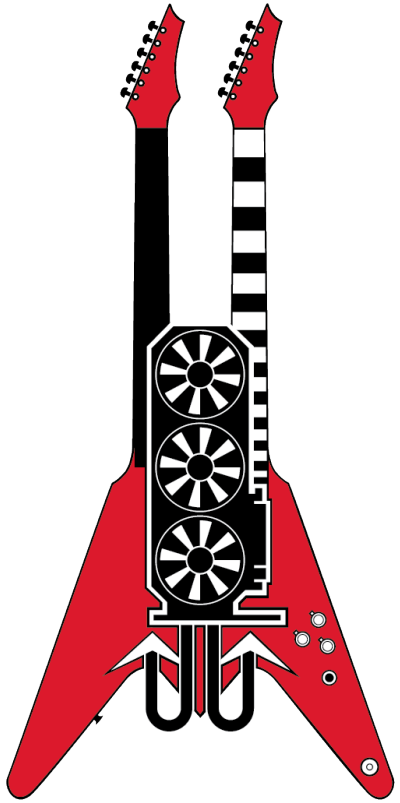
matinraayai/Luthier

How Luthier Tools Work



On-going Luthier Enhancements

- Support for instrumenting more complex applications
  - e.g., device code with stripped symbols
- Support for isolated testing of internal components
  - Adding LLVM Integrated Tester (LIT) testing infrastructure to tame the growing complexity of Luthier LLVM passes and the supported GPU targets
- Overall improvements to the programming model
  - Raise the level of abstraction for instrumentation
  - Increase reusability and independence of components
  - Increase user control over the instrumentation process



matinraayai/Luthier

Dealing with (AMD) GPU Application Complexities

Observations

- Unlike host DBI, most GPU DBI frameworks are **automated static**, not **truly dynamic**
  - No code discovery or course-correction is happening during the kernel's execution
- Most DBI GPU frameworks **assume the inspected code is "well-behaved"**:
  - All possibly executed device code is discoverable in a single code object via a linear top-to-bottom scan and queries from the underlying runtime driver
  - The host portion of the application always go through "sanctioned" channels to load its code
- Luthier is positioned to support these complex scenarios

Stripped Symbol Tables

- An application may decide to strip its device code object `symtab` section
- The GPU DBI framework would not be able to easily locate all statically reachable code before the first launch
  - No padding is required between device functions in AMD GPUs

```
Symbol table '.dynsym' contains 4 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000001a00   172 FUNC    GLOBAL PROTECTED    7 _Z3fooPii
     2: 0000000000000800    64 OBJECT  GLOBAL PROTECTED    6 _Z3fooPii.kd
     3: 0000000000003cf0     1 OBJECT  GLOBAL DEFAULT   10 __hip_cuid_3f75d[...]

Symbol table '.symtab' contains 18 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000001900    28 FUNC    LOCAL  DEFAULT    7 _Z3bari
     2: 0000000000000004     0 NOTYPE  LOCAL  DEFAULT  ABS _Z3fooPii.num_vgpr
     3: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  ABS _Z3fooPii.num_agpr
     4: 0000000000000021     0 NOTYPE  LOCAL  DEFAULT  ABS _Z3fooPii.number[...]
     5: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  ABS _Z3fooPii.privat[...]
     6: 0000000000000001     0 NOTYPE  LOCAL  DEFAULT  ABS _Z3fooPii.uses_vcc
     7: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  ABS _Z3fooPii.uses_f[...]
     8: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  ABS _Z3fooPii.has_dy[...]
     9: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  ABS _Z3fooPii.has_re[...]
    10: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  ABS _Z3fooPii.has_in[...]
    11: 0000000000000001     0 NOTYPE  LOCAL  DEFAULT  ABS amdgpu.max_num_vgpr
    12: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  ABS amdgpu.max_num_agpr
    13: 0000000000000020     0 NOTYPE  LOCAL  DEFAULT  ABS amdgpu.max_num_sgpr
    14: 0000000000002c80     0 NOTYPE  LOCAL  HIDDEN     8 _DYNAMIC
    15: 0000000000001a00   172 FUNC    GLOBAL PROTECTED    7 _Z3fooPii
    16: 0000000000000800    64 OBJECT  GLOBAL PROTECTED    6 _Z3fooPii.kd
    17: 0000000000003cf0     1 OBJECT  GLOBAL DEFAULT   10 __hip_cuid_3f75d[...]
```

Calling Externally-Sourced Device Functions

```
__global__ void myKernel(int* array, unsigned long long devFunc, int n) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid < n) {
        auto *myFunc = reinterpret_cast<int (*)(int)>(devFunc);
        array[tid] = myFunc(array[tid]);
    }
}
```

- No way to determine all reachable device code for this kernel statically
  - `devFunc` could be any device function inside or outside `myKernel`'s code object
- Cannot cache `myKernel`'s instrumented version
  - Subsequent invocations might pass a device function loaded after `myKernel` was instrumented
- Easy for the instrumented code to escape into original code's address space

Overlapping Functions and Overlapping Instructions

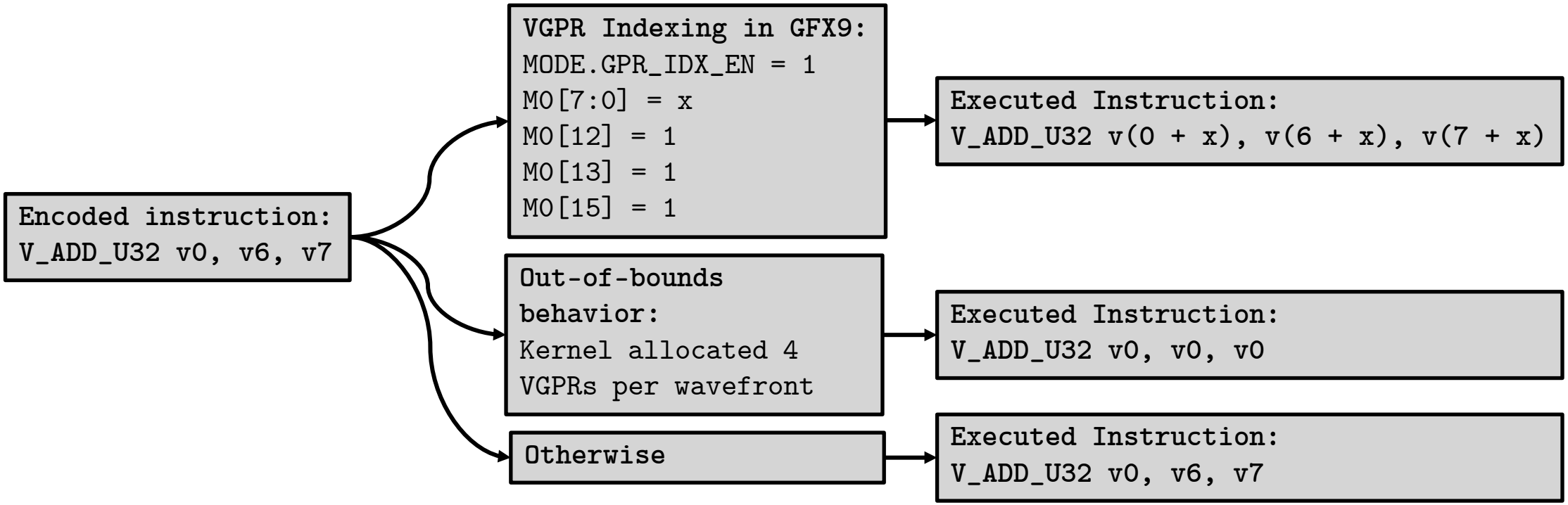
- Functions with overlapping logic (upper) and instructions with overlapping bytes (lower) can occur in AMD GPUs
  - AMD GPUs have variable-length instructions
  - Same size padding requirement as the minimum instruction size
  - These issues were also reported for x86 binaries by X. Meng and B. Miller in "Binary Code Is Not Easy", ISSTA '16
- In-place instrumentation is not feasible in this scenario

```
0x1700 <upper_func>:
  v_add_nc_u32_e32 v0, 1, v0 // 4A000081
0x1704 <lower_func>:
  v_add_nc_u32_e32 v0, 2, v0 // 4A000082
  s_setpc_b64 s[30:31]       // BE80201E
```

```
overlap_kernel:
  ...
  s_cmp_lt_u32 s0, 5
  s_cbranch_scc0 main_path
  s_getpc_b64 s[10:11]
.Lpc_pos:
  s_add_u32 s10, s10, (end_p - .Lpc_pos)
  s_addc_u32 s11, s11, (end_p - .Lpc_pos) >> 32
  s_setpc_b64 s[30:31], s[10:11]
  ...
main_path:
  s_mov_b32 s3, 0xbf810000 // imm op encodes s_endpgm
  ...
overlap_kernel_end:
  .size   overlap_kernel, overlap_kernel_end - overlap_kernel
  .set    end_p, main_path + 4 // Points to s_endpgm
```

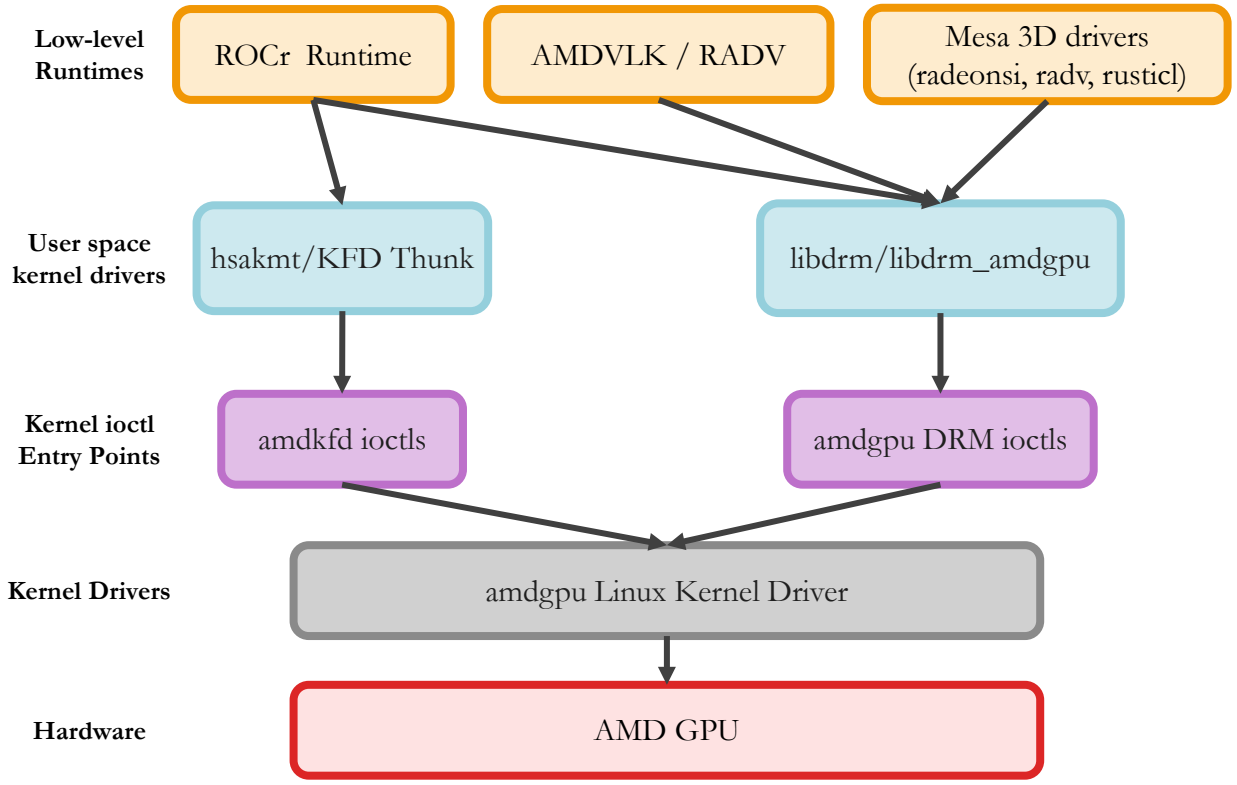
The Devil is in the Wave State

- Depending on a wavefront's state, a simple instruction can have statically undetermined behavior
- Detailed instruction semantics and a symbolic evaluation system is required to reason about uses and definitions of an instruction
  - Register scavenging would not work otherwise



Device Code Side-Loading And Third-Party GPU Runtimes

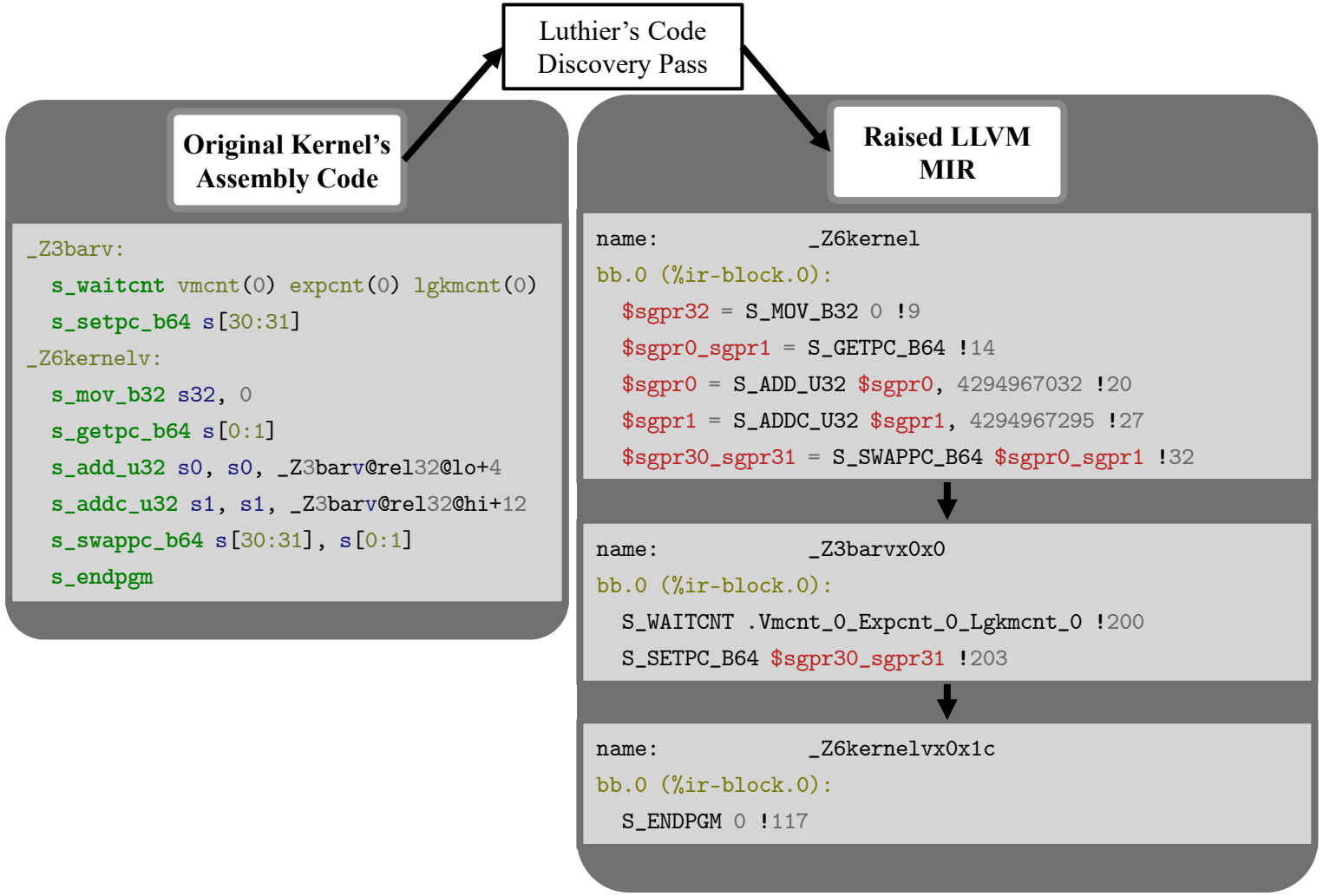
- Recently ROCr started to allow allocating memory pools with executable privileges
  - An application can write its own code loader on top of the pool allocator
- An application can even write its own runtime directly on top of the GPU driver!
  - For example, Spectral Compute's SCALE runtime, TinyCorp's TinyGrad
- A GPU DBI framework should not assume anything about the origin of the loaded code



Work-in-Progress Solutions

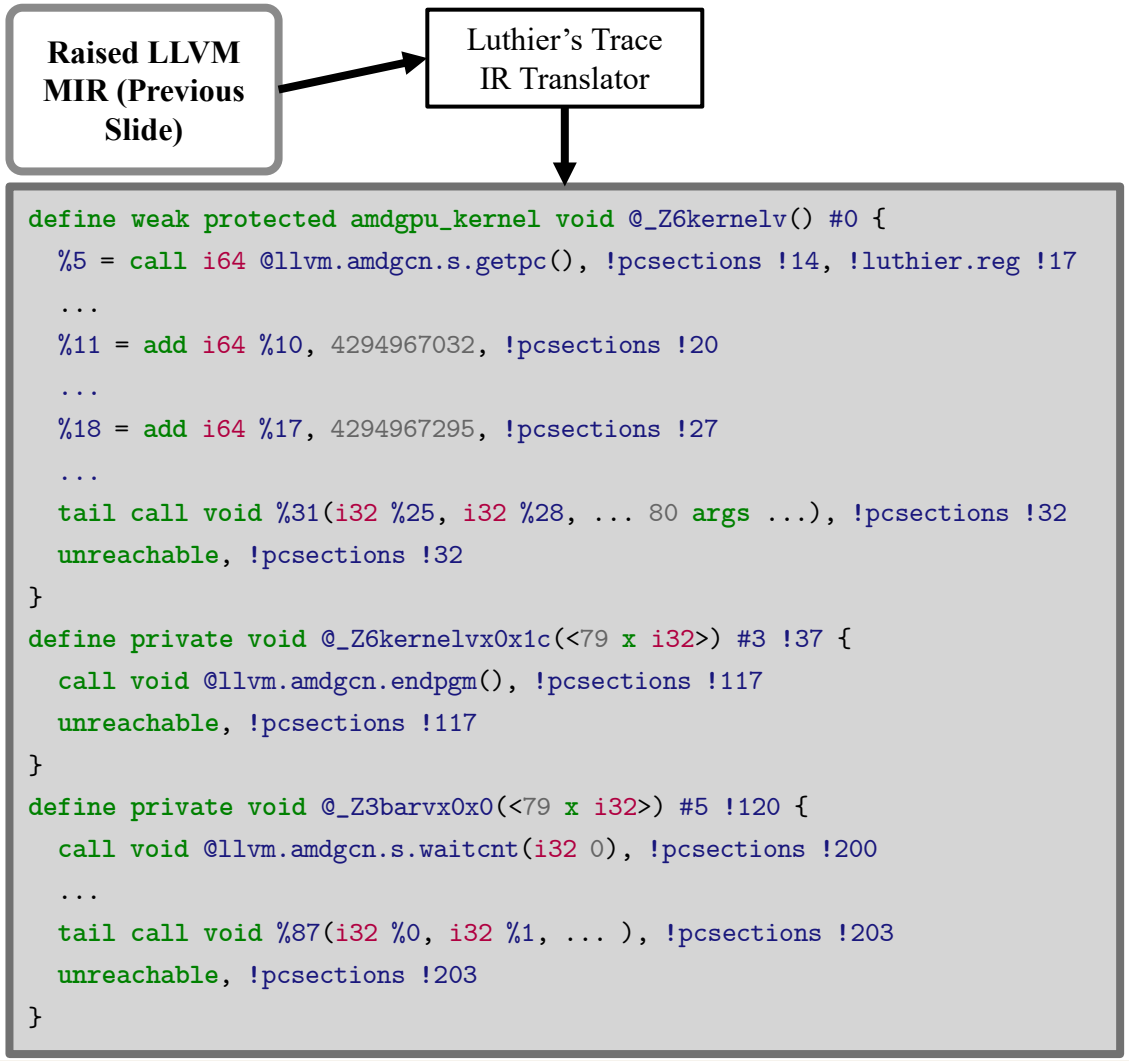
Adopt Host DBI Code Discovery Strategies

- Luthier now discovers *Trace Groups* of reachable device code starting from an initial *entry point*
  - An entry point can be a kernel descriptor or a plain device address
  - Device code's memory allocation is linearly scanned and disassembled from top to bottom
  - Direct branches are followed and added to the trace group
  - Reaching an indirect branch, terminator, or any call instructions ends the trace
- Trace groups are then converted to individual LLVM machine functions
- Code object information is used as a hint when available:
  - E.g., for naming discovered functions or adding debug symbols
- Supports overlapping instructions and functions



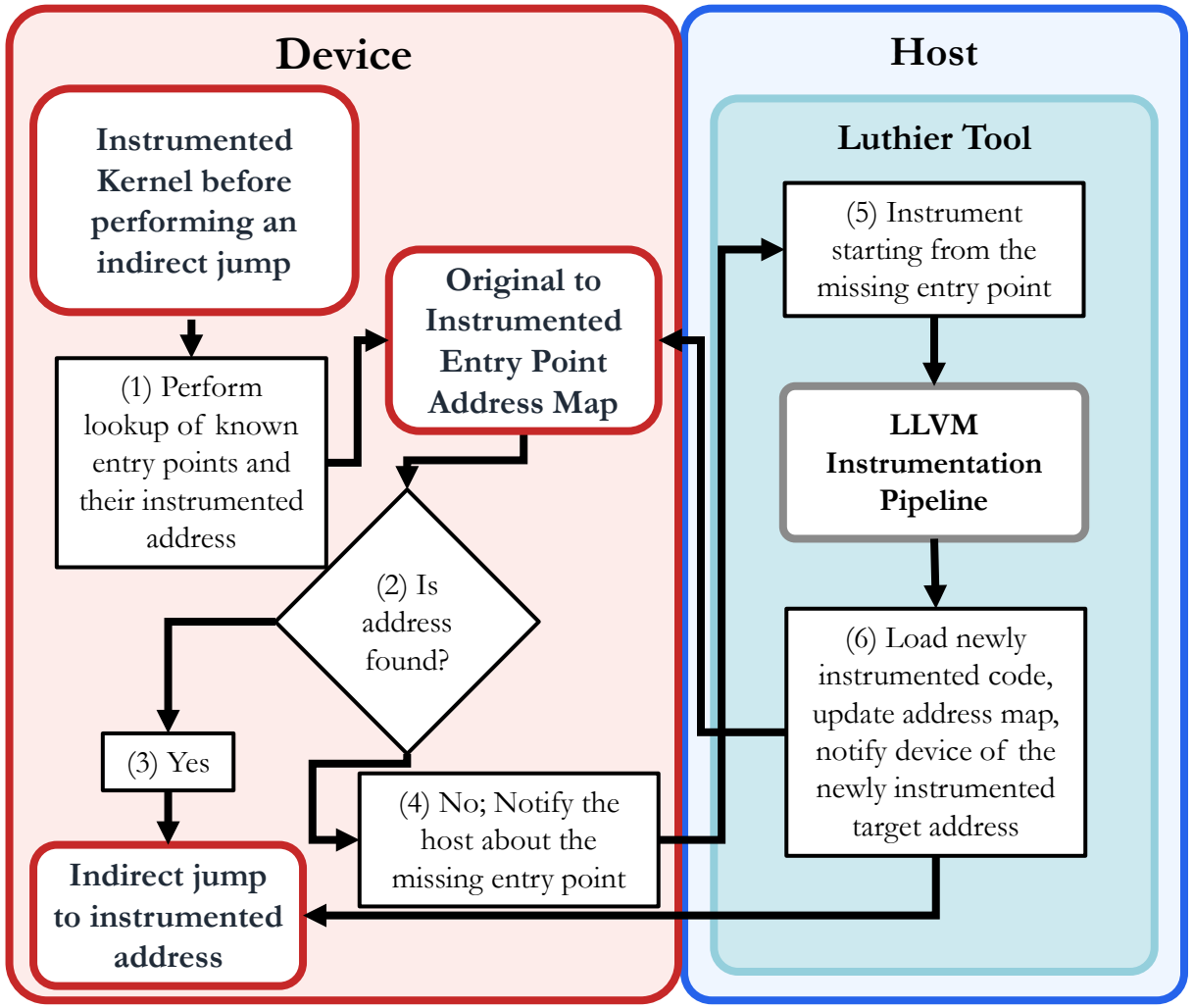
AMD GPU Instruction Semantics

- Written in LLVM TableGen
- Translates individual AMDGPU instructions to a sequence of low-level LLVM IR operations
- Custom TableGen backend generates C++ code used to translate raised machine functions
- Can handle indirect and out-of-bounds register accesses
- Translated IR instructions is then used to:
  - Discovering statically reachable code
  - Reasoning about register liveness and values
- Can also provide a high-level abstraction over instrumentation
  - LLVM IR is more approachable to beginners
- Needs to be verified on real hardware using a fuzzer
  - Most of the semantics were generated using AI by reading AMD's public documentation and LLVM TableGen records



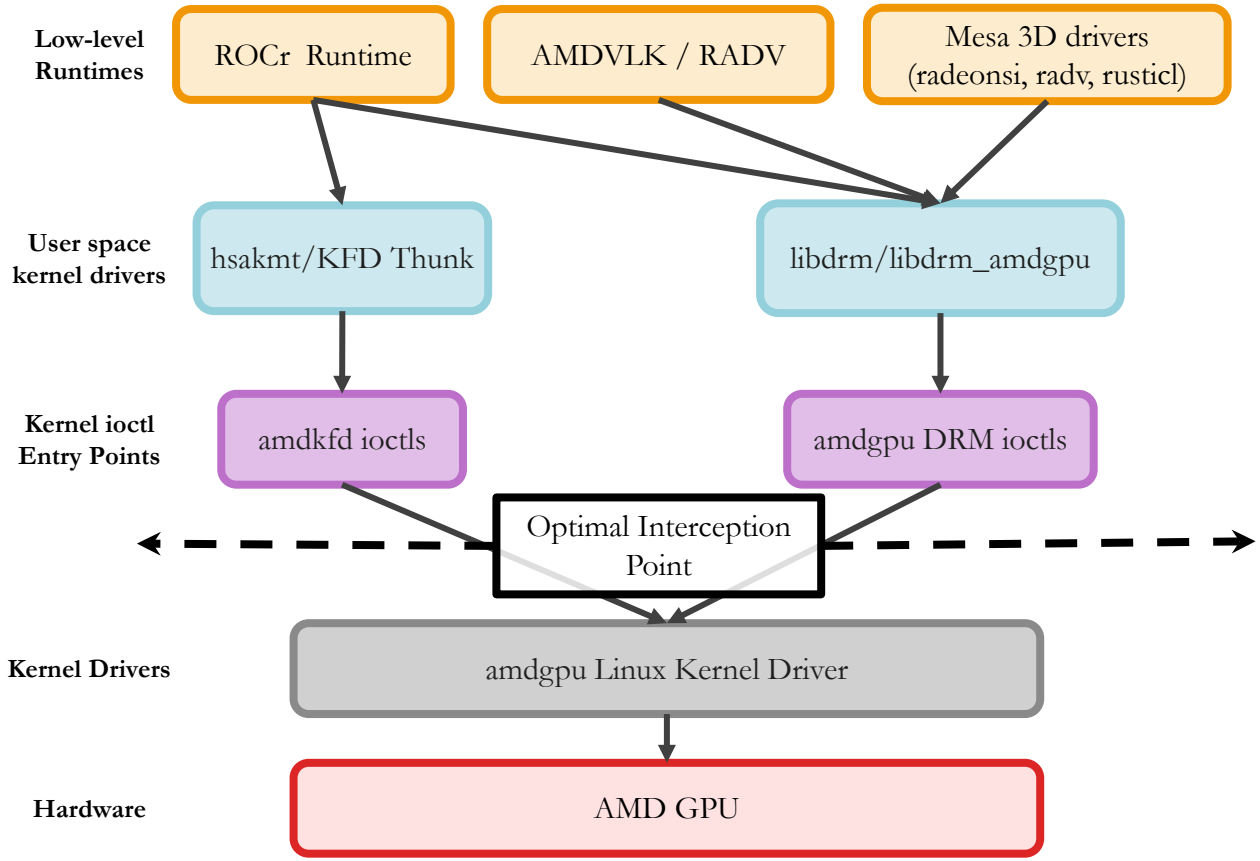
Host Calls to JIT Instrument Unseen Code

- Luthier generates a "relocatable" copy of the original code
  - Any reads of the program counter is patched to point to its un-instrumented address
  - The program only sees the original address space
- Indirect jumps and calls must be redirected to their instrumented versions
  - Redirection for statically discovered entry points happens on the device
  - If the reached entry point was not seen before, the host is notified
- Host can then generate and load newly instrumented code and notify the device
  - Exactly what the hypervisor does in host DBI



Perform Interception at the IOCL Level

- Intercepting AMDGPU IOCTL commands is the only way to support virtually all AMD GPU applications
- Low-level runtime interception is still necessary
  - Provides extra information about the code objects used to load the device code
- Currently investigating feasibility and tradeoffs of IOCTL interception compared to low-level runtime
  - E.g., how can compute queues be software wrapped for intercepting kernel launches similar to ROCr



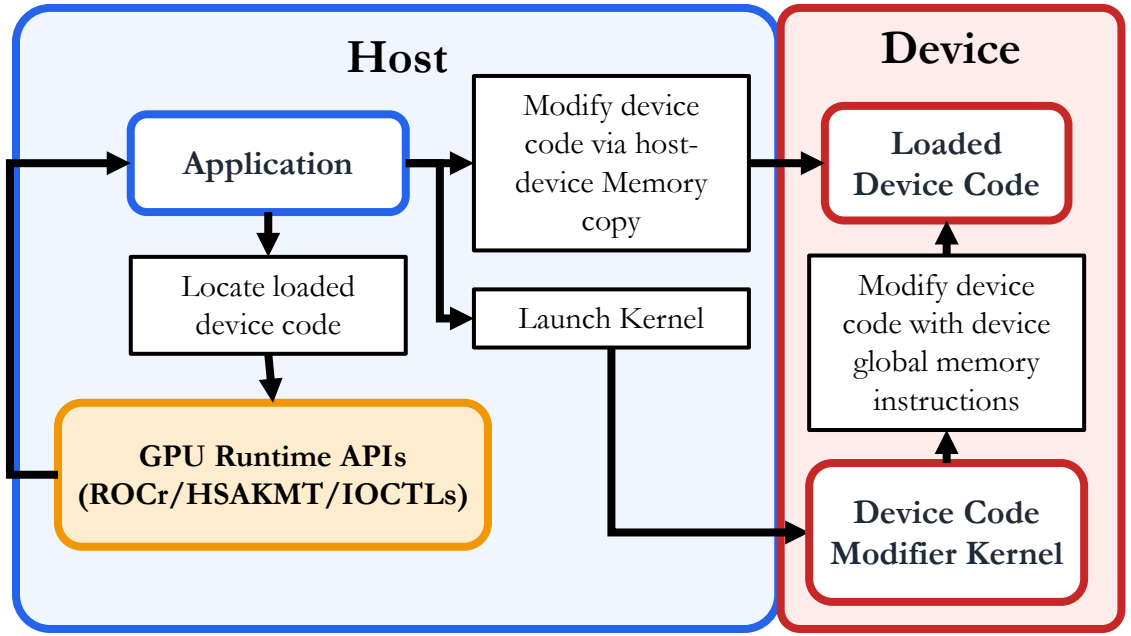
Conclusion And Final Thoughts

- GPU DBI frameworks can:
  - Only rely on what the hardware and the kernel driver sees
    - Especially those with an open software ecosystem
  - Treat information from low-level runtimes as hints
- Detailed GPU ISA semantic information is necessary for correct instrumentation of device code
  - For calculating register liveness and reachable code

Backup Slides

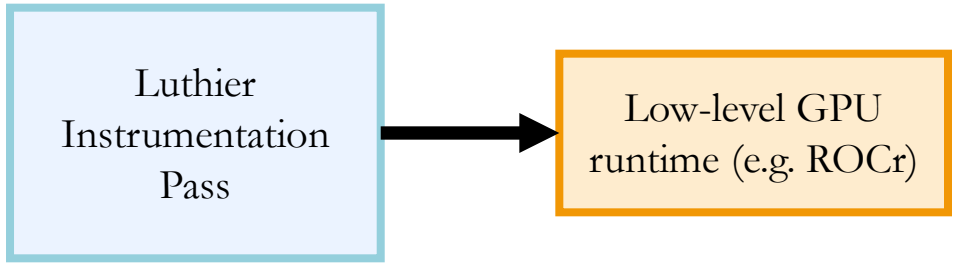
Self-Modifying Code

- Applications can use underlying GPU runtime's APIs to locate their loaded device code address
  - Same APIs used by the GPU DBI frameworks
- There is nothing stopping the underlying application from modifying its device code after it is loaded
  - Can be done from both host and device
  - Example use-case: implement a special ELF relocation type not supported by the underlying runtime
- The GPU instrumentation framework must monitor all memory copies to detect code modification and invalidate their code cache accordingly



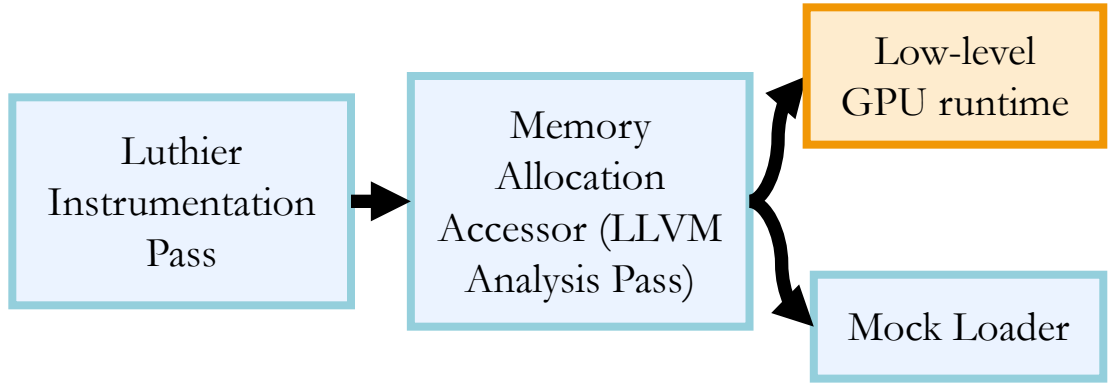
AMDGPU Mock Loader

- Testing and developing the code generation portion of a GPU DBI framework is non-trivial
  - Instrumentation passes need to inspect GPU memory directly
- Testing the naïve way:
  - Have access to a representative sample of GPUs
  - Load the code objects onto the GPUs
  - Have the passes inspect the GPU memory allocations directly via GPU runtime APIs
- Drawbacks:
  - Development and testing the passes requires access to multiple generations of GPUs
  - Harder to checkpoint code generation pass pipeline
    - Must always start from the beginning



AMDGPU Mock Loader

- A level of indirection is introduced to abstract away the details of accessing target device memory
- For testing, instead of loading device code onto device memory, we instead load it onto host memory
- This is done via Luthier's AMDGPU Mock Loader
- The mock loader supports loading and dynamically linking HSA and Mesa (graphics) code objects
- LLVM LIT can be used to test every stage of Luthier's code generation in a CI pipeline without access to an AMD GPU



The Kernel Argument Expansion Challenge

- Instrumentation logic usually needs its own set of arguments
- They need to be passed alongside the original kernel argument
- Kernel argument buffer size might be unspecified
  - Code object's metadata might not be present nor correct
- Must pass extra arguments without relying on re-using or extending the kernel's argument buffer

```
amdhsa.kernels:
  - .args:
      - .address_space:  global
        .name:           array.coerce
        .offset:         0
        .size:           8
        .value_kind:     global_buffer
      - .name:           n
        .offset:         8
        .size:           4
        .value_kind:     by_value
      - .offset:         16
        .size:           4
        .value_kind:     hidden_block_count_x
      - .offset:         20
        .size:           4
    .group_segment_fixed_size: 0
    .kernarg_segment_align: 8
    .kernarg_segment_size: 24
...
```

Passing Kernel Arguments to Instrumentation Logic

- Instead of extending the original kernel's argument buffer, create a new one solely meant for instrumentation logic
- Pass the instrumented arguments to the kernel instead
- Instrumented kernel's prologue logic will store back the original arguments after it's done processing the instrumented arguments