Formal Proof for GPU Resource Management

Jia Zhu, Chuanqin Yan, Nigel Wang
Advanced Micro Devices Co., Ltd

AMD
Smarter Choice
Agenda

• Design Overview & Motivation
• Formal Verification Challenges & Solutions
• Mutation coverage in Sign-off
• Conclusions
Design Overview & Motivation
Radeon™ RX 480 GPU at a glance

- 14nm FinFET process
- 1 Graphics Command Processor
- 4 ACE
- 2 HWS
- 36 Compute Units
- 4 Geometry Processors
- 32 Pixels Output/Clock
- 144 Texture Units
- 576 32b Load/Store Units
- 2 MB L2 Cache
- 256-bit GDDR5
- AMD CrossFire™ technology
- DisplayPort™ 1.4-HDR, HDMI® 2.0b
- Video Encode/Decode acceleration
- PCIe® 3.0

Reference from: [http://www.tuicool.com/articles/meeaYfq](http://www.tuicool.com/articles/meeaYfq)
GPU Resources

• Compute units are the key module in modern GPU
  – Graphics shading
  – General-purpose computing

• Typical GPU resources required by compute units
  – Scalar/vector general-purpose-registers
  – Local data share
  – Barrier resource
  – Computing slot
  – Scratch buffer

• Resource management is critical in GPU
The resource block
- 6 different types of resources
- Uses a big bitmask to track the status of resources
- Decides which shader request can be launched
Verification Challenges

- **Controllability**
  - Large resource pools, hard for coverage closure
  - Numbers of configurations to walk through

- **Observability**
  - Arbitration results are timing-dependent
  - No way to create an accurate reference model for concurrent allocation/de-allocation
Formal Verification

• Why do we choose Formal Verification?
  – Exhaustive coverage with high controllability
  – White box verification with high observability
  – Advantage in verifying control-intensive logic
  – Friendly debug support

• Use VC formal from Synopsys
Formal Verification
Challenges & Solutions
Formal Verification Framework

- **Legality checkers for all resource types**
- End-to-end checkers for data integrity, etc.
  - e.g. alloc_size
- Interface checkers for interface protocols
  - e.g. task_id must be consecutive in a workgroup for group requests.
- Internal checkers for arbitration mechanism
  - e.g. LRU algorithm
- Automatically Extracted Properties generated by VC formal
Avoid State Explosion

State explosion cause

- Constraint/checker complexity
- Design complexity
- Deep sequential depth

Formal convergence skill

- Constraint/checker abstraction
- Design abstraction
- Bounded proof

Resource A

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmask size per SIMD:</td>
<td>m bits</td>
</tr>
<tr>
<td>Number of SIMDs:</td>
<td>n</td>
</tr>
<tr>
<td>req_size:</td>
<td>w bits</td>
</tr>
<tr>
<td>Possible states:</td>
<td>$2^w \times 2^{m\times n}$</td>
</tr>
</tbody>
</table>
Legality Checkers

• Unavailable resource can’t be allocated.
• Available resource can’t be de-allocated.
• Allocation and de-allocation can’t happen to one resource at the same time.

```verilog
alloc_legality_check_p: assert property(@(posedge clk) disable iff(rst)
   (alloc && alloc_res_hit |-> !scb)
);
dealloc_legality_check_p: assert property(@(posedge clk) disable iff(rst)
   (dealloc && dealloc_res_hit |-> scb)
);
exclusive_alloc_dealloc_p: assert property (@(posedge clk) disable iff(rst)
   $onehot0({(alloc && alloc_res_hit), (dealloc && dealloc_res_hit)})
);
```
Checker Abstraction

• Base
  – The “bit mask” design is a symmetric implementation

• Solution
  – Symbolic random variables to check if any single bit in the scoreboard is allocated and de-allocated legally

```verilog
// Symbolic variables are randomized at reset and keep stable then.
symbolic_cu_id:     assume property (@(posedge clk)
                     (##1 $stable(watched_cu_id)));
symbolic_simd_id: assume property (@(posedge clk)
                                 (##1 $stable(watched_simd_id)));
symbolic_res_id:   assume property (@(posedge clk)
                                 (##1 $stable(watched_res_id)));
```
Design Abstraction

• Base
  – DUT can be scaled down to a smaller configuration.
  – Different types of resources are independent from each other.

• Methodology
  – Down-scale configuration to smaller one
  – Blackbox unconcerned resource types.
Reset Abstraction

• Base:
  – If the scoreboard starts from an arbitrary legal status, then doing one allocation/de-allocation is enough to cover all of the possible scenarios.

• Solution:
  – Exclude reset logic to customize initial states
Bounded Proof

• Bounded proof depth calculation
  – 5 cycles needed to do one operation
    • allocation for 4 cycles, and/or
    • de-allocation for 5 cycles
  – 1 extra cycle added for safety
• Run time with depth=6 for resource A

![Graph showing run time in minutes for different checks](image)
Formal verification sign-off

• Original sign-off list
  – Every output signal is covered with at least one checker.
  – We achieved bounded proof for all of the checkers with depth 6.
  – We can achieve virtually 100% code coverage and 100% functional coverage in 6 cycles.

Only guarantee the reachability of formal verification
Mutation Coverage In Sign-off
Mutation Coverage

• Measure the quality of verification environment
  – Error injection to find property holes
  – 4 types of errors inserted
    ▪ The DUT allocates more resources than expected.
    ▪ The DUT allocates less resources than expected.
    ▪ The DUT de-allocates more resources than expected.
    ▪ The DUT de-allocates less resources than expected.

All the 3 legality checkers are still proven
There are verification holes
Resource Leakage Issue

- Will break forward progress with enough depth
  - Reset abstraction/bounded proof reduce the depth
- No immediate observability for scoreboard itself
  - Should add property for internal scoreboard status.
Scoreboard Checkers

- check if each resource state transition works correctly

// After alloc, the scoreboard bit must be the same as reference bit, scb
alloc_legality_check_internal_p: assert property (@(posedge clk) disable iff(rst)
  (alloc && alloc_res_hit |-> scb == internal_bit_mask[symbolic_id])
);

// After dealloc, the scoreboard bit must be the same as reference bit, scb
dealloc_legality_check_internal_p: assert property (@(posedge clk) disable iff(rst)
  (dealloc && dealloc_res_hit |-> ##3 scb = internal_bit_mask[symbolic_id])
);
• Challenges for mutation injection
  – Large number of mutation for certain design
  – Time-consuming to check mutation coverage
• Two mutation coverages are defined for trade-off
  1. **Functional mutation coverage**
     ✓ Manually defined, explicit coverage based on SPEC
     ✓ Measurement for observability of interests
  2. **Structural mutation coverage**
     ✓ Auto-generated, implicit coverage based on RTL structure
     ✓ Can be numerous

Combine functional and structural mutation coverage
Use certitude from Synopsys
Mutation coverage result from certitude

• **33%** non-detected faults.
  – **9** errors are the resource leakage related issues.
  – **19** errors are related to logic redundancy.
  – **37** errors are covered by other types of checkers, like the “reserve” and “cu_locking” logics.
  – **172** errors are related to resource search logic as we expect.
    ✓ similar to issues in resource status logic.
    ✓ after reset abstraction, the reset logic of resource search is cut off from DUT.

• **67%** detected faults.
Certitude Run-Time

Platform: HP 8 core workstation, 64G mem

Manual injection: manually defined, manually injected
Structural mutation: fully auto-error injection by tool
Functional mutation: manually defined, but auto-injected by tool
Mutation Coverage Steps

- Manual mutation coverage - functional
  - For critical functions, abstraction related parts
  - alloc more/less, de-alloc more/less for this case.
    ✓ Guide tool to inject arithmetic error into target design file

- Auto mutation coverage - structural
  - auto-inject by EDA tool
  - Reset, connectivity, arithmetic, etc.
  - Restrict the number of errors
  - Low priority errors.

- Trade-off between performance and confidence
- Add mutation coverage into our formal sign-off list.
Conclusions

• Formal verification solves our challenges in simulation
• Manual interventions are required for FPV convergence
• Verification holes may be introduced
• Mutation coverage helps a lot for formal sign-off
• Trade-off between functional mutation coverage and structural mutation coverage for best ROI
Thank Vigyan Singhal and Chirag Agarwal from Oski

Thank Xiaolin Chen, Jimmy Sun, Huang Feng from Synopsys

Thank Matthew Znoj, Mark Anderson from AMD Shader Processor Input team

Thank Christeen Gray from AMD GFXIP methodology team