2026 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SANTA CLARA, CA, USA
MARCH 2 - 5, 2026

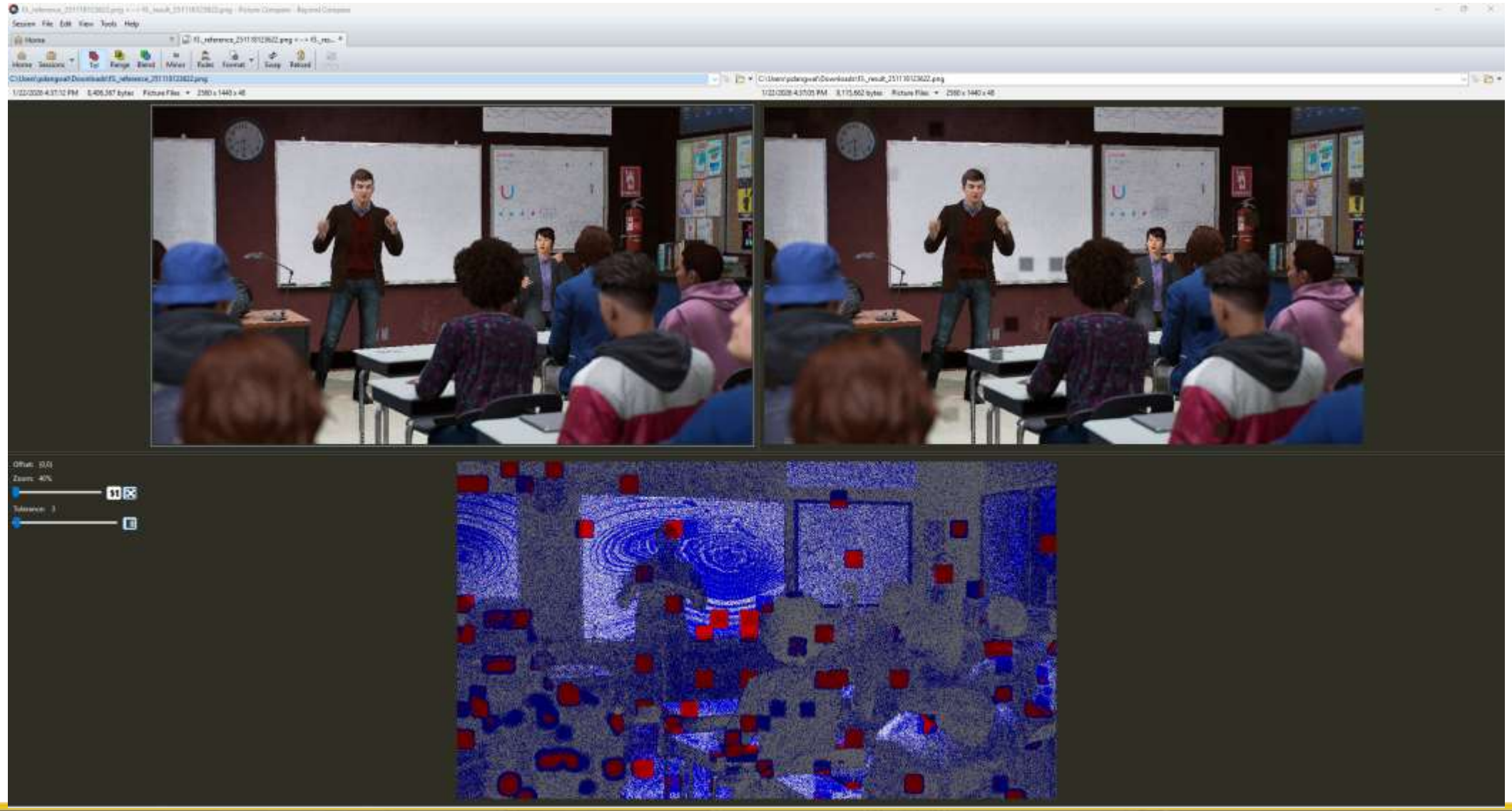
Automated Root-Cause Analysis of GPU Pipeline Corruptions in Graphics and Compute Workloads

Pravesh Dangwal – Intel Corporation
Himanshu Somaiya – Intel Corporation

Agenda

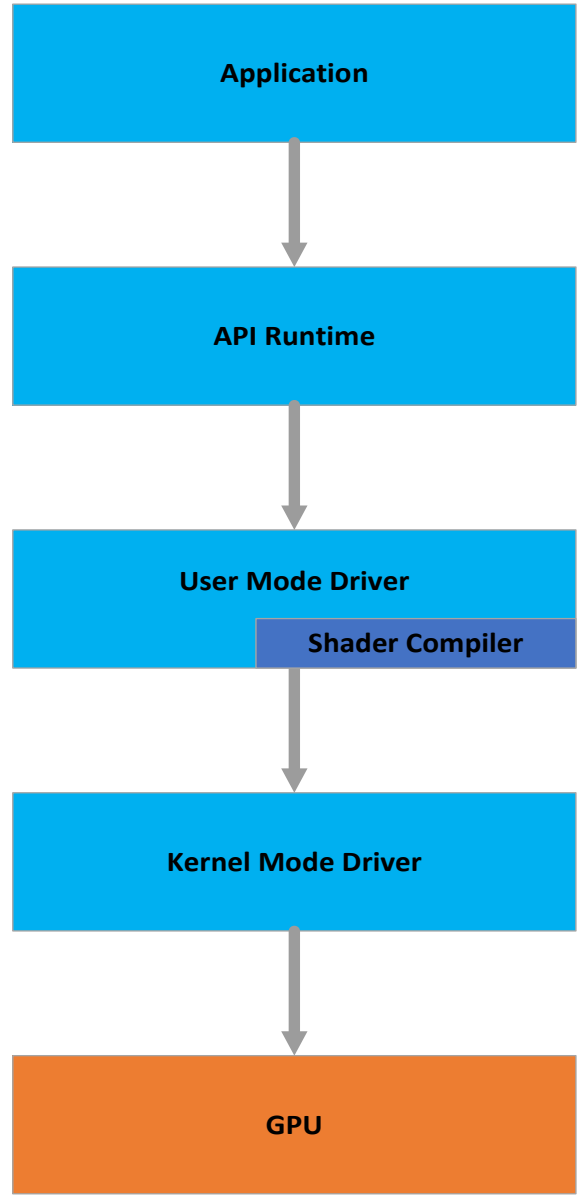
- What is corruption in Graphics
- Brief overview of Graphics System and Graphics Pipeline
- Complexity of a frame and GPU HW
- Challenges in manual debug
- ARGO tool features
- ARGO tool flow
- Results
- Q&A

Image Corruption example



Graphics System

- Application: The game or the AI/ML workload
- API Runtime: Intermediary between the application and hardware, example DirectX11, DirectX12, Vulkan, OpenGL, OpenCL, Metal etc.
- UMD: User Mode Driver converts API commands to low level hardware commands and states. Different UMD for different APIs, Runs at low privilege
- Shader Compiler: Converts high level programs written in languages like HLSL to low level GPU HW instruction set language.
- KMD: Kernel Mode Driver, interacts directly with HW, manages Memory allocations and hardware scheduling. Runs at highest privilege.
- GPU: This is the actual silicon which is running the shader programs and underlying fix function GPU algorithms.

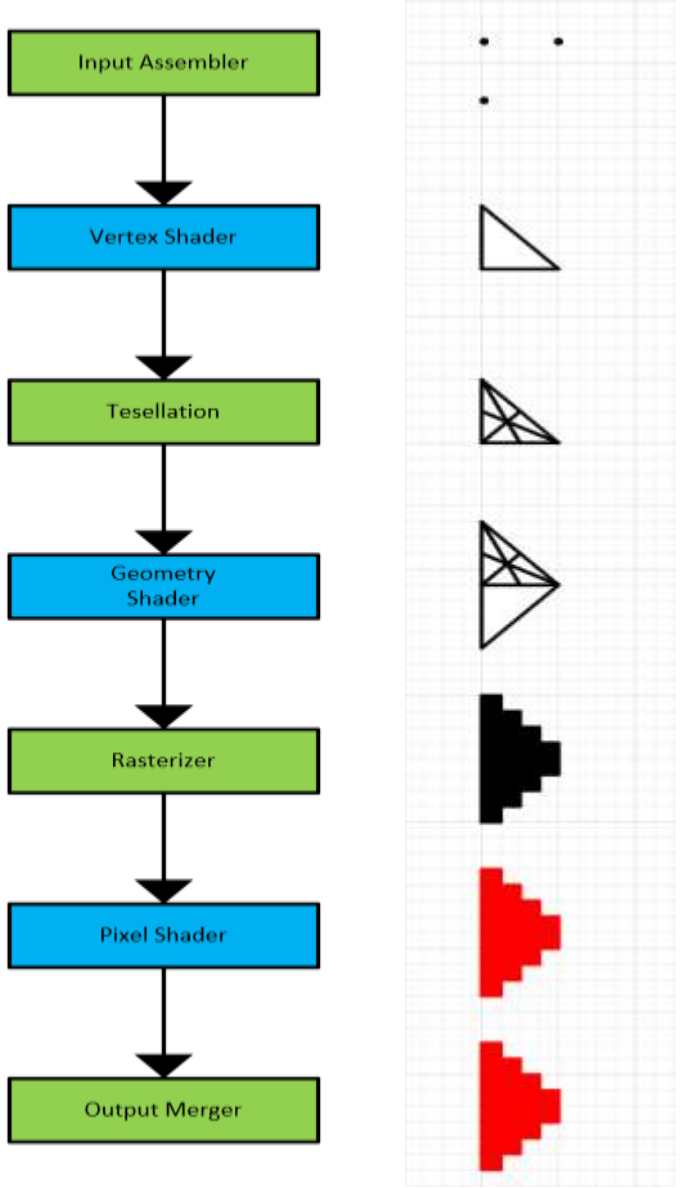


Bugs can be anywhere!

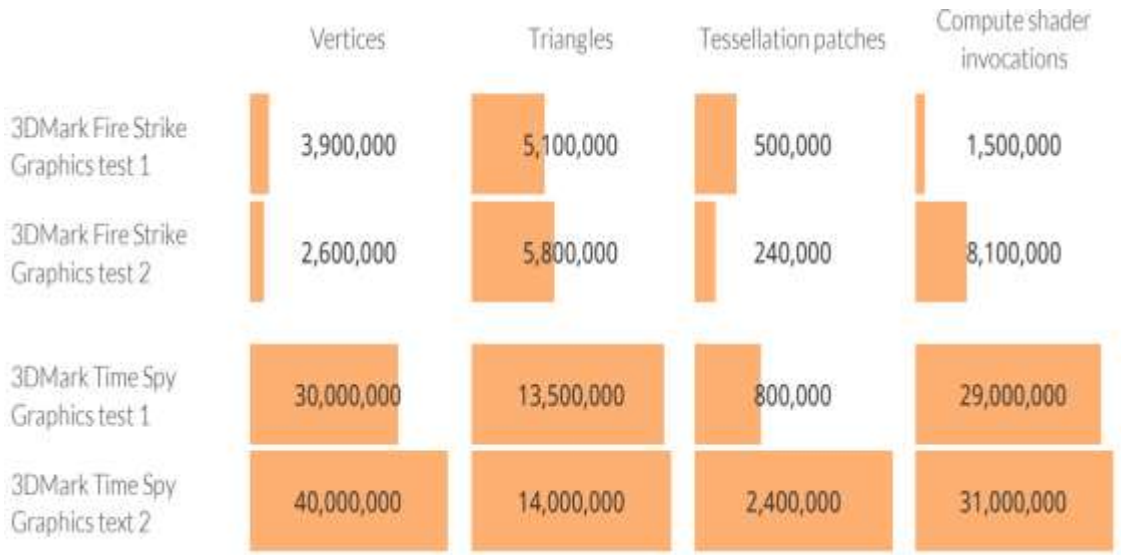
Graphics Pipeline Corruptions typically result in following category of issues

- Application Bugs
- SW Bugs
  - UMD
  - KMD
  - Compiler
- HW RTL Bugs

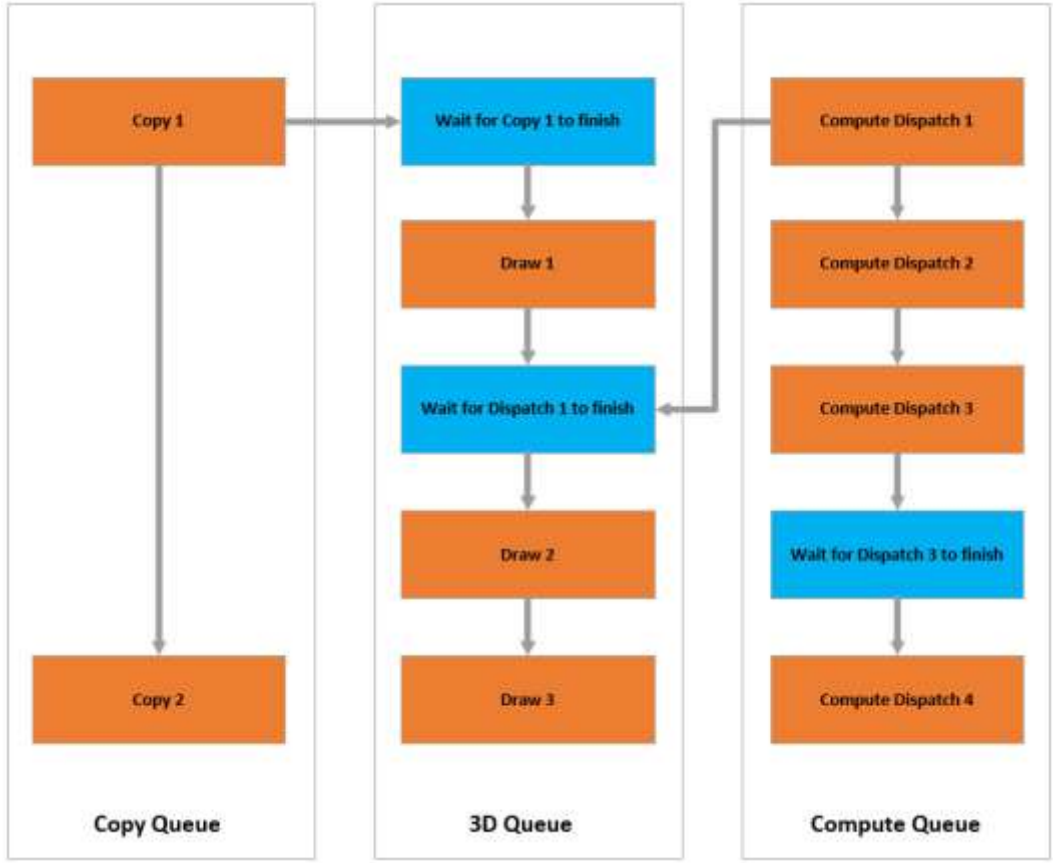
Graphics Pipeline



What happens in a single Frame



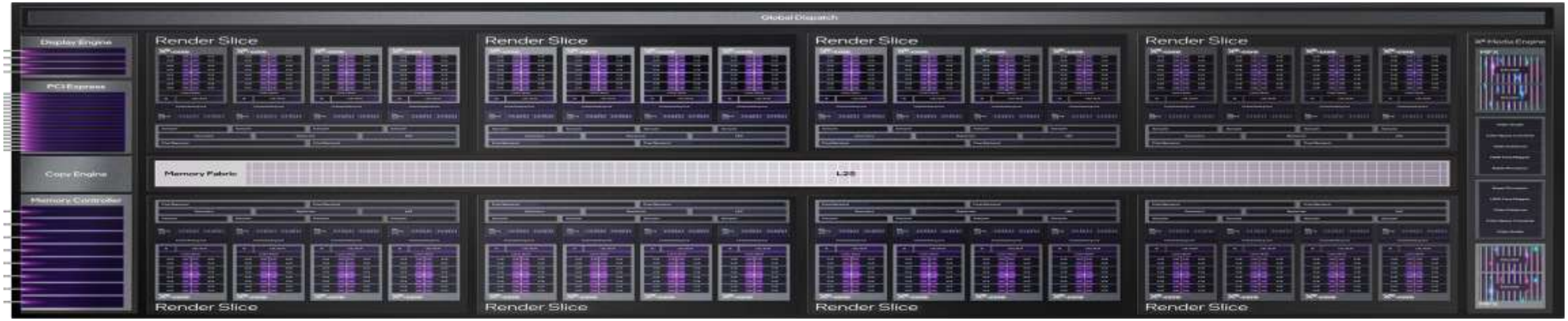
Source: https://s3.amazonaws.com/download-aws.futuremark.com/3dmark-technical-guide.pdf



Typical flow of commands in a game

Typical GPU HW

- Typical GPU HW consists of hundreds of individual IPs
- Debug process involves analyzing trackers at all these interfaces
- For a big workload like a game frame or AI/ML kernels this tracker data is huge typically around 1TB+



Source: https://cdrdv2-public.intel.com/758302/introduction-to-the-xe-hpg-architecture-white-paper.pdf

Challenges of manual Debug

- Pipeline complexity: Hundreds of interacting IP blocks with shared caches and global memory.
- Data volume: Massive amounts of interface and trace data make anomaly isolation time-consuming.
- Model mismatch: Behavioral differences between golden reference models (e.g., C++) and RTL implementations complicate direct comparison.
- Workload diversity: Debug methods must be effective for both graphic pipelines (requiring pixel-level accuracy) and compute pipelines (requiring precise numerical validation).
- Platform Diversity: Testbench difference across platform like RTL simulation, Driver Emulation, Post Silicon etc.

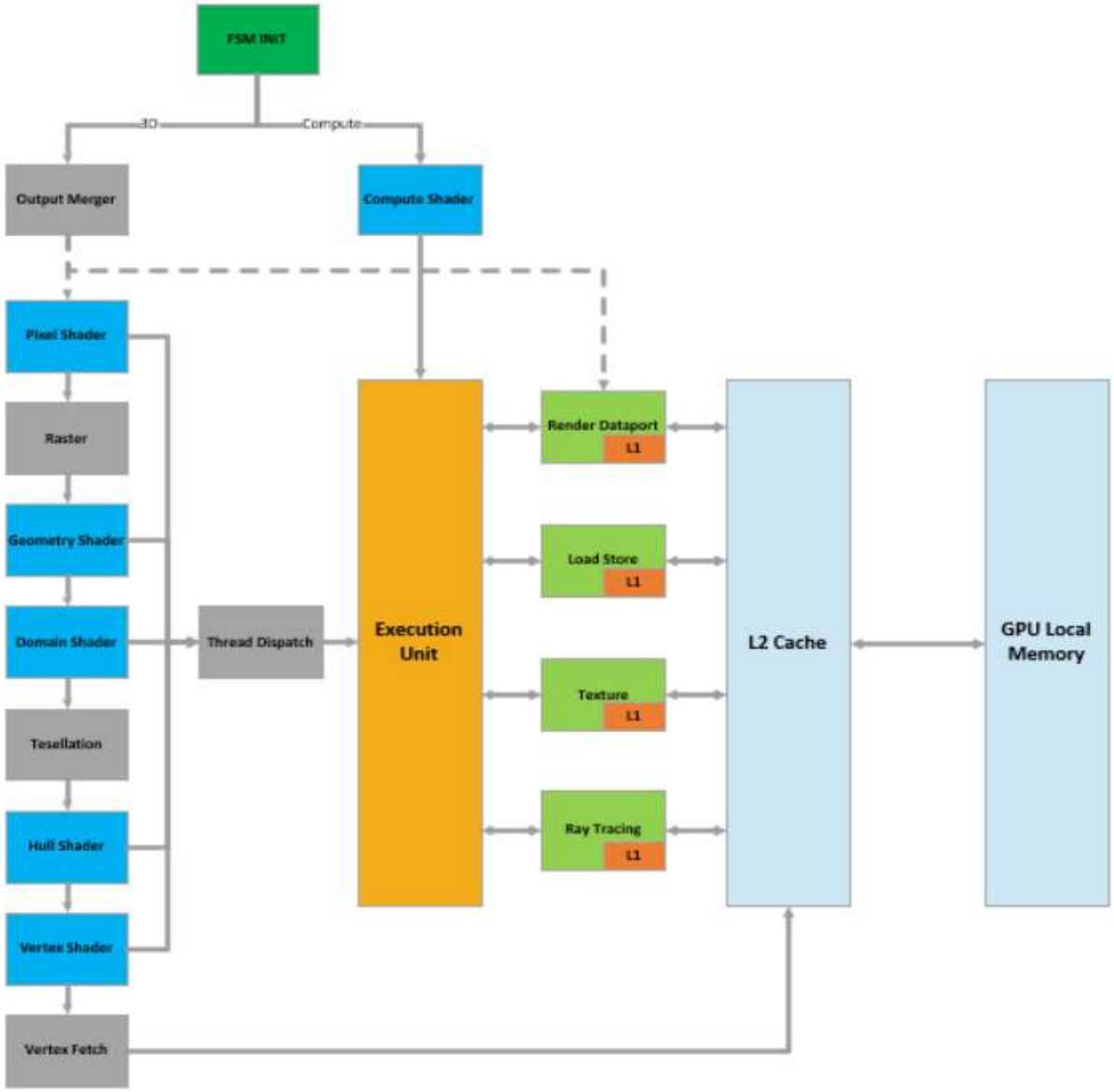
ARGO Tool

- ARGO (**A**utomated **R**oot-Cause for **G**PU pipeline c**O**rruptions), is an automated debug framework implemented in Python
- Automate corruption debug by identifying the exact RTL interface and transaction responsible for a failure.
- Supports multiple environments including pre-silicon simulation, pre-silicon driver-based Emulation, and post-silicon validation employing capture replay on emulator.
- Target both workload classes: graphics (pixel level tracking) and compute workloads
- Incorporates GPU microarchitectural knowledge to understand mismatch between golden reference vs. RTL datasets.
- Improve efficiency by restricting analysis only to transactions directly contributing to corrupted data, avoiding terabyte-scale overhead.

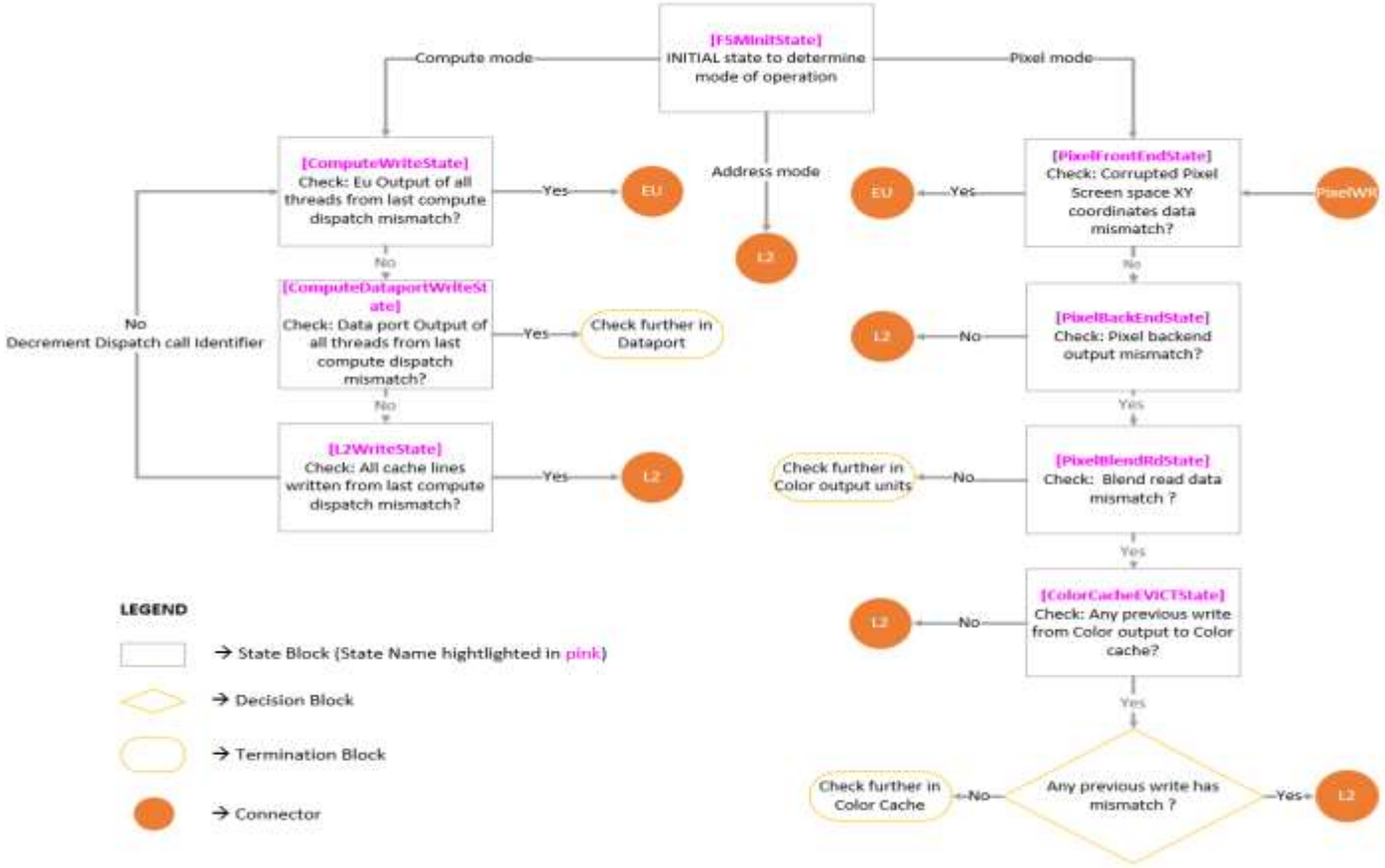
ARGO Methodology

- **Architecture**: ARGO utilizes a **Finite State Machine (FSM)** architecture, where state transitions are designed to mirror the microarchitectural datapath of the GPU pipeline
- **Data Collection:** ARGO ingests interface trackers from both passing and failing runs. Passing datasets may be derived from a C++ golden model or a verified RTL run, while failing datasets are obtained from the faulty RTL model.
- **Comparison Engine:** At each interface, ARGO compares inputs and outputs across passing and failing runs. Mismatches identify the corrupted interface.

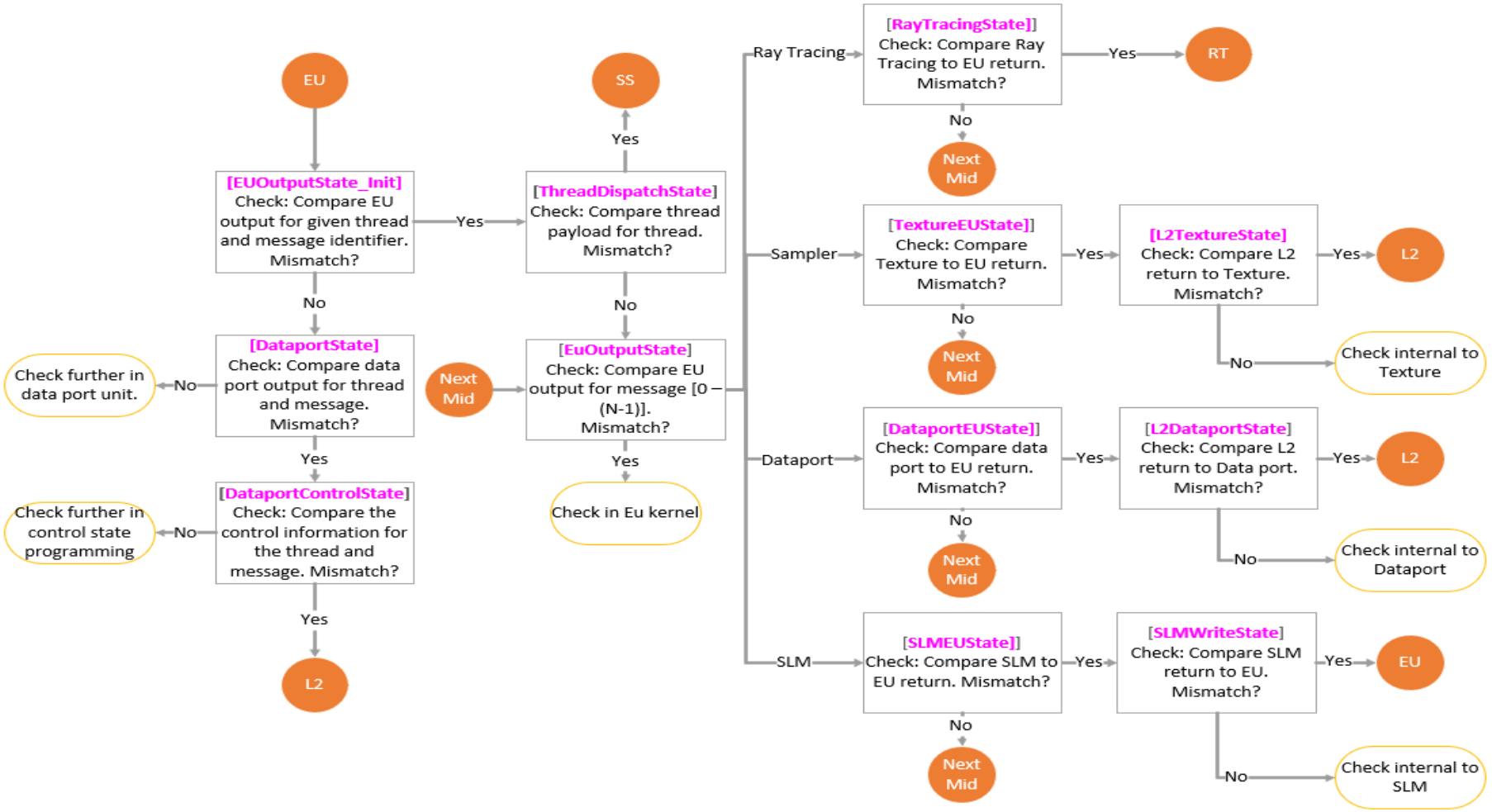
ARGO Flow Overview



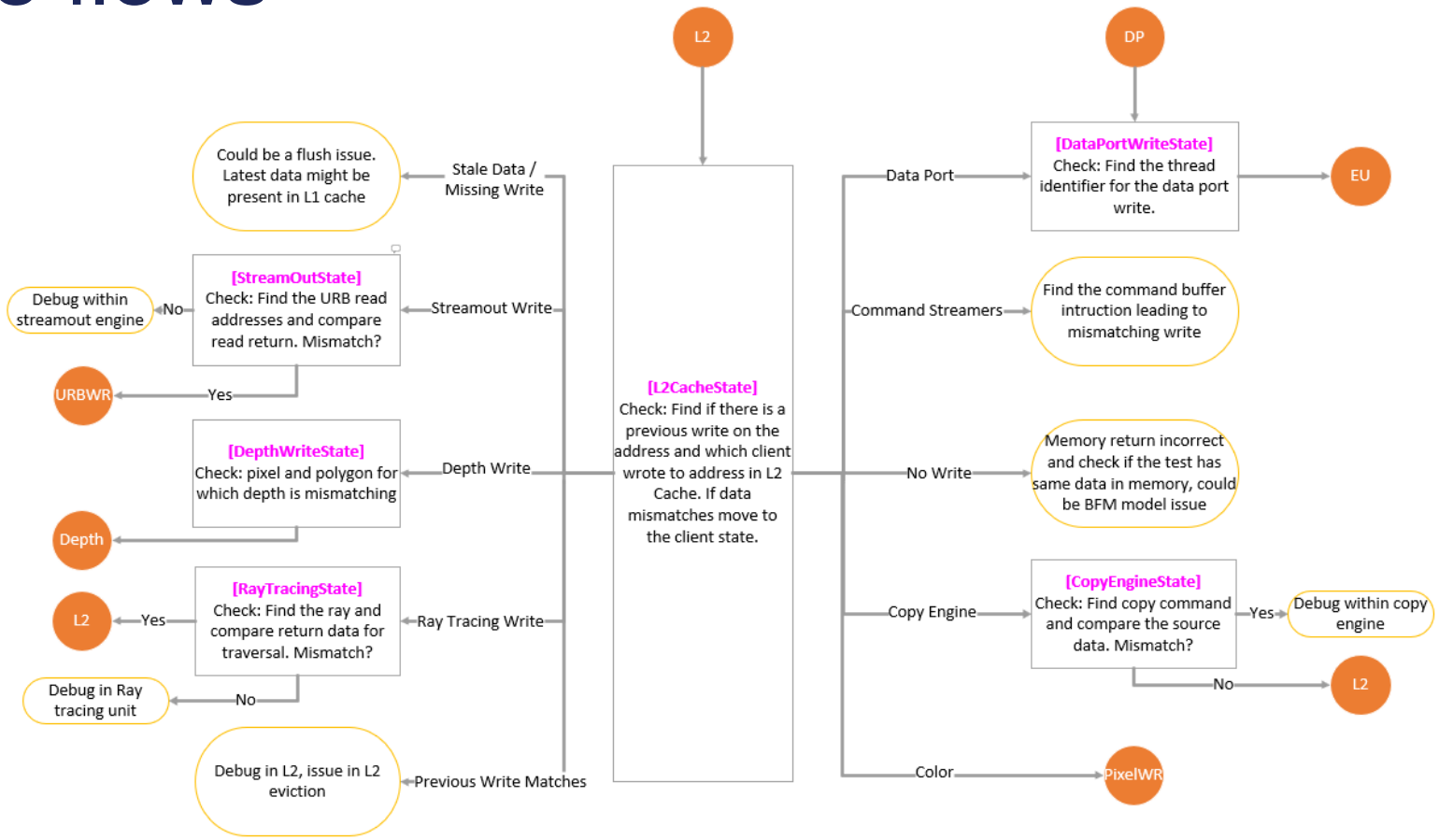
More detailed view



Shader Core (EU) and shared functions flow



L2 Cache flows



Results

Deployment of ARGO in production GPU debug environments has demonstrated:

- **100x – 1000x reduction in debug turnaround time in simulation and emulation compared to manual debug.**
  - Typical manual debug takes 1 day to a week to root cause. ARGO shows turnaround time of minutes to hours depending on size of test collaterals.
- **Significant accuracy improvements and emulator capacity saving**
  - Improves root-cause localization by identifying the exact issue in one step, avoiding inefficient manual debug hops.
  - Reduces unnecessary emulation captures, saving substantial engineering effort and emulator capacity.
- **Efficient bug categorization, preventing duplication of effort across teams.**
  - ARGO groups failures by common root cause, reducing duplicate debug effort and enabling faster prioritization to improve regression pass rates.