



# EoSS: Exploring SASS— A Systematic Framework to Explore NVIDIA’s ISA

Cesar Avalos Baddouh  
Purdue University  
West Lafayette, IN, USA  
[cavalosb@purdue.edu](mailto:cavalosb@purdue.edu)

Mathias Payer  
EPFL  
Lausanne, Switzerland

Timothy G. Rogers  
Purdue University  
West Lafayette, IN, USA

## Abstract

NVIDIA GPUs are ubiquitous in AI, HPC, and graphics, yet their native SASS instruction set remains undocumented. No public infrastructure exists to answer fundamental questions: which instructions exist on a given architecture, what encodings the hardware accepts, and how instructions behave at the microarchitectural level. Each research effort must rediscover encoding formats, build ad hoc tooling, and reverse-engineer instruction semantics from scratch.

We present EoSS, an end-to-end framework for systematic SASS exploration. EoSS extracts machine-readable instruction specifications from data embedded in NVIDIA tooling, generates valid instruction encodings, and executes them on real hardware through binary patching. The framework produces complete toolchains, kernel templates, operand setup, and measurement harnesses that turn opaque SASS into a workable experimental interface. We release validated instruction sets for seven architectures spanning Volta through Blackwell, with 539–1,338 runtime-valid instruction classes per target.

Our infrastructure supports diverse research tasks: reverse-engineering and executing exotic ray-tracing instructions, performing extensive timing characterization across execution pipelines, measuring contention-based timing effects on resources unreachable through PTX, and systematically probing undocumented special registers. EoSS provides the missing foundation for GPU ISA research, making native SASS exploration routine and reproducible for compiler optimization, performance modeling, and security analysis on modern GPUs.

**CCS Concepts:** • **Computing methodologies** → **Graphics processors**; • **Security and privacy** → *Hardware reverse engineering*; • **General and reference** → *Measurement*.

**Keywords:** GPU architecture, NVIDIA SASS, instruction set architecture, reverse engineering, binary rewriting, microbenchmarking, instruction fuzzing, RT cores, raytracing



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

ASPLOS ’27, Heraklion, Greece

© 2027 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-3012-2/2027/04

<https://doi.org/10.1145/3845814.3850082>

## ACM Reference Format:

Cesar Avalos Baddouh, Mathias Payer, and Timothy G. Rogers. 2027. EoSS: Exploring SASS— A Systematic Framework to Explore NVIDIA’s ISA . In *Proceedings of the 32nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS ’27)*, April 11–15, 2027, Heraklion, Greece. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3845814.3850082>

## 1 Introduction

NVIDIA GPUs are fundamental to AI training and inference [34] and HPC workloads [51], and widely adopted in autonomous systems, yet their native SASS instruction sets remain proprietary black boxes. Much of the execution interface—opcodes, operand encodings, control bits, and special-function pipelines—is undocumented or only partially described by public tooling.

This opacity makes basic ISA questions difficult for researchers and engineers to answer: which instruction families exist, which encodings the hardware accepts, what operand constraints they impose, and how they interact with the execution context. The lowest-level interface generally available to developers is the textual Parallel Thread eXecution (PTX) virtual ISA [39], which hides specialized instructions (e.g., ray-tracing and graphics operations) and microarchitectural control. Security researchers and independent compiler writers lack ground truth for instruction behavior beyond PTX.

Undocumented native behaviors have already been implicated in GPU exploitation and cross-tenant leakage [15, 29, 43, 57]. Undocumented SASS-level knowledge also has practical value, as evidenced by DeepSeek [58], which leveraged the undocumented yield scheduling flag to improve FFMA interleaving for measurable throughput gains.

NVIDIA’s disassembly tools still leave parts of the ISA opaque: certain opcodes, fields, and modes are omitted from `nvdiasm` output [32] (Figure 4), and differential analysis offers only a partial view. There have been several third-party attempts to reverse-engineer the ISA and produce assemblers [1, 6, 14, 56], but none offers full coverage.

Recently, the community [8] uncovered a domain-specific script embedded in `nvdiasm` that encodes a comprehensive specification of SASS, from bit-field definitions to the conditions that render an instruction statically valid. We refer to this artifact as the Rosetta String. While prior work [26] uses

the Rosetta String for latency measurement on a single architecture, it does not programmatically evaluate the constraint system, validate encodings at runtime, or explore instruction classes beyond those reachable through PTX. We use the full specification, including constraint ASTs, shader-type bit-masks, and exotic instruction families, to enable systematic exploration across the entire ISA.

We present EoSS, a framework that turns the Rosetta String into runnable experiments for exploring the GPU’s native instruction set. It integrates specification extraction, constraint-aware encoding generation, binary injection, runtime validation, and fine-grained performance instrumentation into an extensible research platform.

EoSS generates valid SASS kernels combining instruction classes never jointly emitted by existing binary-generation tools. For instance, it can assemble and execute programs mixing ray-tracing instructions with tensor-core and tensor-memory instructions used in ML workloads. These programs let us study undocumented resources and instruction-level interactions between units that do not normally interact.

EoSS supports every NVIDIA architecture released since 2017, including Volta [38], Ampere [33], Ada, Hopper [37], and Blackwell [36] (SM70–SM120). It generates constraint-satisfying encodings and patches them into test kernels and harness variants with appropriate operand setup, control flow, and measurement templates. The resulting programs are validated and characterized on real hardware.

**Contributions.** We make the following contributions:

1. **Systematic native-ISA exploration across GPU generations:** Every prior effort to study NVIDIA’s native ISA has required building ad hoc, architecture-specific tooling<sup>1</sup> from scratch. We provide reusable infrastructure that extracts ISA specifications, generates constraint-satisfying encodings, validates them on real hardware, and measures their behavior, spanning seven architectures (SM70–SM120) with 539–1,338 runtime-valid instruction classes per target. We release instruction datasets and tooling so that future work in architectural characterization and security analysis can build on a common foundation rather than re-deriving encodings.
2. **Undocumented instruction discovery:** Using our framework, we are the first to identify, characterize, and execute eight TTU (ray-tracing) instructions at the ISA level, extracting complete BVH (Bounding Volume Hierarchy) traversal sequences from JIT-compiled NVIDIA OptiX [41] kernels. We demonstrate that the full traversal sequence not only executes from a standard CUDA compute kernel but also supports applications, enabling direct experimental access to the hardware and opening new research opportunities.

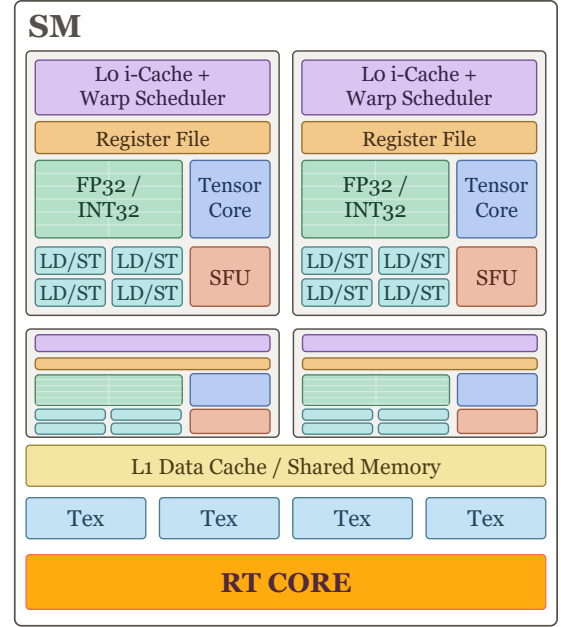


**Figure 1.** Streaming multiprocessor organization (shown for Blackwell SM120). Each SM contains four sub-partitions (SMSPs), each with its own warp scheduler, register file, and functional units. L1/shared memory and the RT Core are shared across all SMSPs.

3. **Infrastructure-enabled case studies:** Using EoSS, we conduct studies that demonstrate its versatility: timing-differential OSIRIS-style fuzzing [53] that combines compute and graphics operations unreachable by any other public toolchain, contention analysis of PTX-unreachable hardware resources, and modification of driver-based kernels. These studies reveal hidden microarchitectural behaviors.

**Responsible disclosure:** We disclosed to NVIDIA that the Rosetta String could be extracted from its tooling; it was subsequently encrypted in CUDA Toolkit 13.0.

## 2 Background

### 2.1 CUDA Programming Model

NVIDIA GPUs execute kernels in Single-Instruction Multiple-Thread (SIMT) fashion: threads are grouped into *warps* of 32 that issue the same instruction in lockstep, with divergent control flow handled by predicated execution [15]. A warp is the fundamental scheduling and measurement unit; our timing harnesses instrument individual warps to isolate instruction-level behavior.

Warps execute on Streaming Multiprocessors (SMs, Figure 1). Each SM is divided into four sub-partitions (SMSPs), each containing a warp scheduler, register file, and functional units; the scheduler interleaves resident warps to hide memory and functional-unit latency.

<sup>1</sup><https://github.com/eoss-framework/eoss-framework>

Consumer and data-center GPUs differ in their functional units. Consumer GPUs include RT Cores, introduced with Turing (SM75), for hardware-accelerated ray tracing. Despite appearing in public architecture diagrams for four generations, the RT Core’s instruction interface remains largely undocumented and is ordinarily accessed through closed NVIDIA code [41].

## 2.2 GPU Software Stack

Between user-space CUDA calls and GPU hardware sits a multi-layered, largely opaque software stack. The CUDA runtime and driver translate API calls into Resource Manager (RM) commands, which are forwarded to the GPU System Processor (GSP), a dedicated RISC-V microcontroller running proprietary firmware [35]. Work is dispatched through engine-specific *classes* (compute, DMA, video, 3D graphics), each with its own object allocation and state configuration.

This architecture has a critical implication for ISA exploration: certain hardware units may only be accessible through specific engine classes. Graphics-pipeline instructions (e.g., attribute loads, tessellation output), for example, require a 3D graphics context with object allocation that the compute API does not provide. An instruction may be encodable and even present in the ISA specification, yet fail at runtime because the necessary hardware context was never established. This distinction between instructions that are *encodable* and those that are *executable* in a given context shapes our two-phase validation methodology (Section 3).

Notably, we show in Section 6 that RT Core traversal does *not* require a graphics context: the full TTU instruction protocol executes from a standard compute kernel, contrary to the assumption that RT Core access is engine-gated.

## 2.3 PTX: The Virtual ISA

PTX (Parallel Thread Execution) is NVIDIA’s documented virtual ISA [39]: a portable intermediate representation lowered to native SASS by `ptxas` at compile time or JIT-compiled by the driver at load time.

PTX deliberately abstracts away microarchitectural detail. Specialized instructions for ray tracing (RT Core operations) and graphics (e.g., OUT, attribute interpolation) are absent from PTX or accessible only through opaque intrinsics. Similarly, hardware registers are selectively exposed: `%clock` and `%smid` are available, but architecture-specific registers such as `SM_SHADER_TYPE` and `HW_TASK_ID` are hidden.

Most importantly, PTX provides no mechanism to emit or pin exact native instruction sequences; instruction selection, scheduling, and register allocation are all determined by the `ptxas` backend. Blocks marked as `asm volatile` in CUDA C++ constrain only host-compiler motion *around* the block; the `ptxas` backend remains free to reorder, fuse, or eliminate the emitted native instructions. There is no user-visible notion of a native opcode. For microarchitectural experiments that require cycle-level control over the instruction stream,

such as OSIRIS-style timing measurements (Section 8), operating directly on native SASS is therefore necessary.

## 2.4 SASS: The Native ISA

Streaming Assembler (SASS) is NVIDIA’s native ISA: the binary instruction set executed directly by GPU hardware. Since Volta (SM70), every SASS instruction occupies a fixed 128-bit slot [21], densely packing opcode, operand fields, immediates, predicates, and scheduling metadata into 16 bytes (Figure 2). NVIDIA publishes no specification; the only official reference is a partial instruction listing in the binary utilities documentation [32], which omits entire instruction families (Table 1).

Across the seven architectures we study, the Rosetta String (Section 4.2) contains 2.3–3.8× as many unique base opcodes as there are instruction entries in the corresponding official listing. Hopper (SM90) defines the largest instruction set, with 1,584 instruction classes (unique opcode–modifier combinations) and 727 unique base opcodes, compared with 204 instruction entries in NVIDIA’s Hopper listing.

**Table 1.** SASS instruction vocabulary across GPU generations. *Rosetta Classes* counts distinct instruction classes (unique opcode-modifier combinations, each corresponding to one CLASS entry in the Rosetta String); *Rosetta Opcodes* counts unique base opcodes. *Official Listing* counts instruction rows, excluding category headings, in the CUDA Toolkit 12.8 Binary Utilities reference [32]; Ampere and Ada share one listing, as do both Blackwell targets. Entries marked with a dagger (†) are server architectures.

| Name        | SM    | Rosetta Classes | Rosetta Opcodes | Official Listing |
|-------------|-------|-----------------|-----------------|------------------|
| Volta †     | SM70  | 646             | 305             | 130              |
| Ampere †    | SM80  | 1,215           | 616             | 169              |
| Ampere      | SM86  | 1,269           | 642             | 169              |
| Ada         | SM89  | 1,337           | 644             | 169              |
| Hopper †    | SM90  | 1,584           | 727             | 204              |
| Blackwell † | SM100 | 1,377           | 564             | 244              |
| Blackwell   | SM120 | 1,411           | 578             | 244              |

Post-Volta SASS uses explicit pipeline controls [1, 14, 18, 19, 21]: *stall counters* (4 bits) block warp issue for a fixed cycle count, while *scoreboard barriers* (6 per warp) track variable-latency operations via producer/consumer wait-masks [19]. Both fields occupy fixed positions in bytes 13–15 of the 128-bit encoding. Correctly setting these fields is a prerequisite for executing instructions outside the official toolchain: an incorrect stall count or missing scoreboard wait produces silent data corruption or hangs.

**Compilation flow.** `nvcc` lowers device code through PTX to SASS in a *fatbinary*; the driver loads matching SASS or JIT-compiles PTX. OptiX [41] uses JIT compilation to emit ray-tracing instructions unavailable through offline `ptxas`

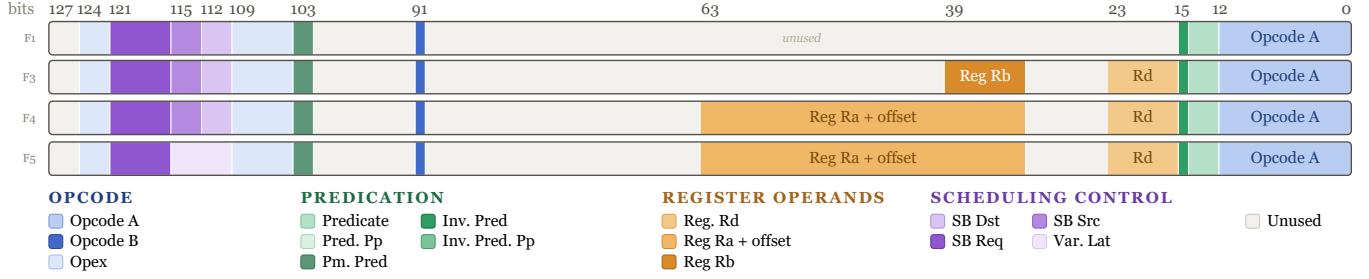


**Figure 2.** Four of the five most common 128-bit SASS instruction formats on consumer Blackwell (SM120), extracted from the Rosetta String. Color-coded fields show opcode, predicate, register operand, and scheduling control regions.

(Section 4.8). Third-party assemblers [6, 14, 56] and Nouveau NAK [1] support architecture-specific SASS subsets. EoSS instead patches official-toolchain binaries, avoiding a separate assembler implementation for each target.

## 3 Methodology

### 3.1 Approach

Our methodology rests on one empirical property: a valid SASS encoding, given identical register inputs and execution context, produces deterministic behavior. This makes instruction behavior amenable to black-box characterization through repeated measurement. Our harness observes whether the instruction faults, its issue and completion latency, register modifications, and scoreboard behavior.

The central difficulty is generating valid encodings. Each 128-bit SASS instruction densely packs opcodes, operands, immediates, and scheduling fields, yielding a raw encoding space far too large for brute-force exploration. We exploit the Rosetta String (Section 4.2), a machine-readable ISA specification embedded in NVIDIA’s tooling, to perform constraint-aware generation. We parse constraint expressions from each instruction class definition and generate one representative encoding per class that satisfies all constraints. For SM120, this yields 1,411 candidate encodings across 578 unique opcodes.

Static validity does not guarantee runtime executability: instructions may require unspecified hardware state, such as memory descriptors, texture objects, or graphics-pipeline initialization. An illegal instruction triggers a sticky error that invalidates the entire CUDA context [7]. We execute each candidate in isolation, discarding context-corrupting encodings before batching survivors into the timing harness so one invalid encoding cannot compromise a campaign.

### 3.2 Scope

We study seven architecture variants spanning five GPU generations (SM70–SM120, Table 1). Experiments use standard CUDA APIs in unprivileged user space, without driver modifications, firmware access, or privileged interfaces. This scope enables third-party performance characterization

and security analysis using capabilities available to ordinary CUDA applications.

### 3.3 Measurement Design

Following established GPU micro-benchmarking practice [21, 28, 31, 54], our harness brackets each target instruction with clock reads via CS2R. In its default configuration, this captures *issue latency*: the number of cycles between the dispatch of the measured instruction and the dispatch of the trailing clock read, with no intervening scoreboard dependency.

Issue latency can understate completion time for variable-latency operations: the trailing clock read may issue before the measured instruction’s result is available. For instructions supporting writeback dependencies, we capture *completion latency* by associating the measured instruction with a scoreboard dependency counter, incremented after issue and decremented at writeback [19]. A corresponding wait-mask prevents the trailing CS2R from issuing until the counter reaches zero. This artificial dependency, a technique previously used in [21], orders the clock read after the tracked completion event; it does not timestamp the exact writeback cycle. Completion here refers to the event tracked by the scoreboard dependency, not necessarily completion of separately launched asynchronous work.

Completion-latency results exclude cases that crash when dependency controls are applied. Incompatible control-flow instructions can instead return small timing values; control-flow instructions are excluded from these measurements, and those values are not interpreted as completion latencies.

## 4 EoSS Architecture

### 4.1 Design Overview

EoSS combines offline specification processing with online validation and measurement (Figure 3). Its browser-based SASS editor supports instruction manipulation, reordering, and ELF metadata editing (e.g., `.nv.info` attributes), with static analysis that flags constraint violations in real time.

Once per target, the offline stage converts `nvdiasm`’s Rosetta instruction grammars into a queryable JSON schema

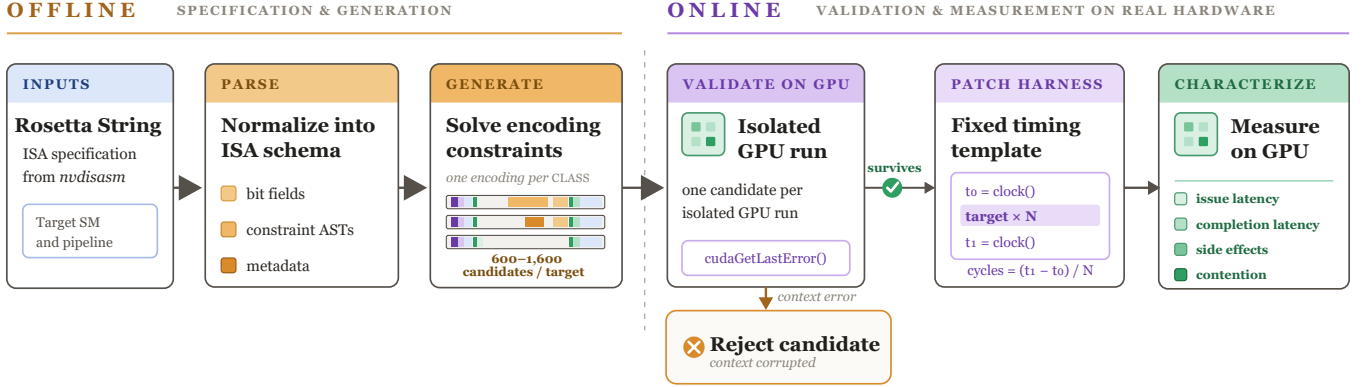


**Figure 3.** EoSS pipeline. The offline stage extracts an ISA grammar from the Rosetta String and generates one candidate encoding per instruction class. A runtime validation phase filters encodings that crash the GPU context; surviving instructions are patched into a timing harness and executed on real hardware.

and generates constraint-satisfying encodings for 646–1,584 candidate instruction classes.

The online stage operates in two phases. A *validation phase* executes each candidate in isolation, discarding any encoding that corrupts the GPU context. A *measurement phase* patches surviving instructions into a timing harness for cycle-accurate characterization. Binary patching avoids recompilation, so thousands of variants share a kernel template with fixed control flow and register allocation.

## 4.2 Rosetta String

The Rosetta String is a domain-specific SASS specification embedded in `nvdiasm`. First documented by the community [8], it has also been used for static performance modeling [26]. It is **descriptive** rather than **semantic**: it encodes binary validity, not computation or side effects.

```

1 CLASS "instruction_class_name"
2 OPCODES
3     InstructionName = 0b1010110110;
4 ENCODING
5     BITS_13_91_0_12_opcode = opcode;
6     BITS_8_31_24_Ra = Ra;
7 CONDITIONS
8     OOR_REG_ERROR (Ra <= %MAX_REG_COUNT-1) :
9         "Register Ra is out of range"
10 PROPERTIES
11     VALID_IN_SHADERS = (1<<ISHADER_TRAP)+(1<<
12         ISHADER_CS) ;

```

**Listing 1.** Sample definition format from the Rosetta String.

For each instruction variant (a CLASS), we extract the opcode, operand bit-field mappings, validation conditions as an AST, scoreboard behavior, and a shader-validity bitmask that records the permitted execution contexts.

Listing 1 illustrates the format. ENCODING maps named fields to bit ranges, while CONDITIONS expresses register

alignment, field dependencies, mutual exclusions, and other constraints that we parse into ASTs. PROPERTIES reveals operand sizes (up to 256 bits for tensor-memory instructions and six 32-bit registers for one argument) and a VALID\_IN\_SHADERS bitmask that restricts execution contexts.

Beyond instruction definitions, the Rosetta String supplies typed modifier enums (cache policies, memory ordering, atomic operations), scheduling controls (REQ wait-mask, RD/WR scoreboard indices, USCHED\_INFO stall count), and per-architecture pipeline latency tables with dependency matrices [26]. Incorrect scoreboard defaults can deadlock the GPU (Section 5).

## 4.3 Instruction Generation

Generation follows four steps: (1) extract a Rosetta schema for the target, (2) parse constraint expressions into an AST, (3) attach instruction metadata (opcode class, operand kinds, and defaults), and (4) enumerate valid encodings by evaluating the constraints.

Constraint expressions are transformed into evaluable ASTs via a recursive-descent parser. Nodes represent binary operations (comparison, logical, arithmetic, bitwise), unary operations, identifiers (operand references, architecture parameters such as %MAX\_REG\_COUNT), and literals. Symbol resolution binds identifiers to concrete values from the instruction’s arguments, architecture constants, or enumeration tables. A practical benefit of parsing these trees is precise error diagnosis: whereas `nvdiasm` reports only a generic “Illegal Instruction Error” for invalid encodings, our ASTs pinpoint the exact violated constraint.

EoSS can emit all valid modifier and suffix variants of an opcode. Within each class, callers can use a representative default encoding, pin operand values, or enumerate all or the first  $k$  satisfying assignments using the AST validator. Enumeration is disabled by default because of its cost.

#### 4.4 Harness Design

The timing harness is a minimal CUDA kernel with fixed control flow and register lifetimes. Patching designated instruction slots without recompilation isolates instruction timing from compiler artifacts.

A patched instruction executes against whatever register state the harness has established, so the generator must emit operands that name the exact registers holding each resource: the global-memory pointer, the texture and surface handles, texture coordinates, and the shared-memory base, with predicate registers defaulting to true and shared memory zero-initialized. Rather than hardcode a convention, we recover these registers from the harness’s own compiled SASS—the harness performs a real access on each resource, and the registers those accesses use become the map the generator targets.

Register assignment is generation-specific: the texture handle, for example, occupies a uniform register on Hopper and Blackwell but a general-purpose register on earlier architectures. Features whose setup would not compile on a given target (e.g., the Tensor Memory Accelerator, available from Hopper onward) are gated by the SM version.

The harness unrolls up to 4,096 measurement slots to characterize thousands of instructions per launch. Each slot has its own lexical scope, ending temporary lifetimes at the slot boundary so the compiler can reuse a small register set; otherwise, a few dozen slots would exhaust the 256-register-per-thread budget. We patch each slot with the instruction under test to isolate its timing without recompilation.

#### 4.5 Extending the Framework

EoSS ships as a Python package whose campaigns can target instruction subsets, pipelines, modifiers, and required harness features. A companion TUI enables field editing.

With an available Rosetta String, porting is largely automated: EoSS extracts the JSON specification, generates encodings, and recovers resource-register assignments from the compiled harness. Across five generations, only a handful of extractor branches were needed for new forms, such as pre-Hopper GPR texture handles. With the encrypted CUDA Toolkit 13.0 string, recovered field mappings still enable manual differential analysis of official-toolchain binaries to infer changed layouts, although this fallback cannot match a Rosetta-backed port.

#### 4.6 Driver Device Functions

The CUDA driver library (`libcuda.so`) embeds precompiled GPU ELF binaries inside an internal container. These binaries implement device-side system functions, including the trap handler (`.traphandler`); CILP routines for saving and restoring context (e.g., `.cilp_traphandler_save`); device-side `malloc` and `printf`; and syscall trampolines. The raw

SASS for each function is stored in a per-architecture ELF and duplicated across supported SM versions.

We locate these functions by scanning `libcuda.so` for the container magic value, parsing the ELF directory, and identifying sections by name.

Because official disassemblers neither extract nor display the code inside these driver files, our framework handles both disassembly and modification. Once located, the SASS bytes can be modified directly in a copy of the library. Loading the patched copy via `LD_LIBRARY_PATH` (no root privileges required) causes the GPU to execute our modified system functions. We validated this mechanism by replacing the first instruction of `.text.vprintf` with `EXIT`: subsequent `printf()` calls from device code produce no output and silently terminate the calling warp, confirming that the modification takes effect at runtime.

We use this approach for two reverse-engineering experiments. First, we patch the trap handler with arbitrary SASS payloads and trigger execution via `BPT.TRAP`, allowing us to test instructions in the trap handler’s execution context. Second, by analyzing the CILP save-context routine, we discover how the driver manages RT Core session state across preemption boundaries (Section 6).

#### 4.7 Graphics Instruction Pipeline

To execute graphics-only instructions, we create a graphics context entirely from user space through `tinygrad`’s [50] NVKIFace and the stock proprietary driver, without OpenGL, Vulkan, Nouveau, or a display switch. We configure a 3D channel and its fixed-function pipeline using command streams, then supply hand-assembled vertex and fragment shaders, each comprising a header followed by raw SASS. A single-primitive draw executes the shaders, which write results to a buffer read by the host. This lets us test arbitrary graphics-only instructions in real shader stages, overcoming the compute context’s shader-type restriction.

#### 4.8 Reverse-Engineering Ray-Tracing Instructions

RT Cores accelerate BVH traversal and ray-triangle intersection [40]. Introduced with Turing (SM75), they appear on consumer architectures but not server variants. The Rosetta String exposes eight TTU (Tree Traversal Unit, the RT Core codename [44]) instruction classes and variants: `TTUOPEN`, `TTUMACROFUSE`, `TTUST`, `TTUGO`, `TTULD` (with and without close), `TTUCLOSE`, and `TTUCCTL` (cache control). Official disassemblers omit their mnemonics (Figure 4).

**Extraction.** Static encodings from the Rosetta String provide bit layouts but lack operand semantics and instruction ordering. To recover complete sequences, we extract *in vivo* TTU code from OptiX [41], NVIDIA’s ray-tracing framework. Examining `libnvoptix.so` reveals no TTU instructions in offline SASS, but LLVM bytecode within the library references TTU intrinsics (e.g., `nv.ttu.open`), and OptiX depends

```

/*0090*/          CS2R R4, SR_CLOCKLO /* 0x00000000000047805 */
/*00a0*/          LDC.64 R2, c[0x0][0x3a0] /* 0x000f0000000015000 */
/*00b0*/          LDC.64 R8, c[0x0][0x3a8] /* 0x0000e800ff027b82 */
/*00c0*/          /* 0x001e300000000a00 */
/*00d0*/          /* 0x0000e800ff027b82 */
/*00e0*/          /* 0x0000e620000000a00 */
/*00f0*/          /* 0x000000000000073d0 */
/*0100*/          /* 0x000fc000000000000 */
/*0110*/          /* 0x000005000000073d4 */
/*0120*/          /* 0x000fc000000000000 */
/*0130*/          /* 0x000000000000073d1 */
/*0140*/          /* 0x000fe200000000000 */
/*0150*/          LDC.64 R4, c[0x0][0x390] /* 0x0000e400ff047b82 */
/*0160*/          /* 0x000e280000000a00 */
/*0170*/          STG.E desc[UR4][R2.64], R6 /* 0x0000000000020798c */
/*0180*/          /* 0x000fe2000c101904 */
/*0190*/          IMAD.WIDE.U32 R4, R11, 0x4, R4 /* 0x00000000040b04825 */
/*01a0*/          /* 0x001fc600078e0004 */
/*01b0*/          BAR.SYNC.DEFER_BLOCKING 0x0 /* 0x0000000000007b1d */

```

**Figure 4.** cuobjdump disassembly of a kernel containing TTU instructions. TTU encodings appear as raw hex and blank spaces. The tool accepts the binary as valid but rejects corrupted binaries.

on the driver’s JIT compiler<sup>2</sup>. This suggests that TTU instructions are assembled at runtime rather than offline.

We ran `optixTriangle`, dumped process memory, and searched for TTU opcode signatures across 26 JIT-compiled cubins. We recovered complete TTU sequences from two kernel families: `traversal_ttu_21h` (328 instructions, 18 TTU) and `triangle_fetch_ttu_21h` (72 instructions, 6 TTU). Section 6 presents the protocol ordering and characterizes the execution of these instructions in depth.

#### 4.9 Further Hidden Instructions

As Figure 4 shows, TTU instructions are not the only instructions whose mnemonics are replaced with blank spaces and raw bytes in disassembly. On SM120, we identified fifteen such instructions and classified the most interesting ones into several categories. The first category comprises trap and interrupt utilities (`BPTONLYDRAIN`, `IDE`, `NANOTRAP`, and `RTT`), which may trigger traps, return from traps, or manage breakpoints. Within the CUDA driver device functions (Section 4.6), we find one use of `IDE`. It appears to protect a critical section: the sequence `IDE`  $\rightarrow$  `ATOMG`  $\rightarrow$  `IDE`. `EN` suggests that the mnemonic stands for Interrupt Disable and that the `.EN` option re-enables interrupts.

The second category comprises conversion instructions, including several F2FP variants that convert floating-point values to nonstandard formats such as microscaling 8 (`MX8`). A third, more intriguing category is the indexedRF group, comprising the `MOV`, `HMMASPARSE`, and `HMMAX8` operations with the indexedRF qualifier. This group appears related to tensor-core accumulators. These instructions occur in the CILP context-switch driver function (analyzed further in Section 6). Before a context switch, the GPU separately stores general-purpose, uniform, and indexedRF registers.

The final category contains the mathematical instruction `CLMAD`, which we hypothesize stands for carry-less multiply-add, an operation often associated with cryptographic methods. Different architectures may have different sets of

**Table 2.** Validation by architecture. *Static Valid*: encodings satisfying Rosetta String constraints; *Encoding Verified*: accepted by `cuobjdump` after patching; *Runtime Valid*: executing without context corruption. *Executable %*: Runtime Valid / Static Valid. Some instructions require additional descriptors or context setup.

| Arch  | Static Valid | Encoding Verified | Runtime Valid | Executable % |
|-------|--------------|-------------------|---------------|--------------|
| SM70  | 646          | 601               | 539           | 83.4%        |
| SM80  | 1,215        | 1,171             | 1,066         | 87.7%        |
| SM86  | 1,269        | 1,225             | 1,123         | 88.5%        |
| SM89  | 1,337        | 1,292             | 1,159         | 86.7%        |
| SM90  | 1,584        | 1,549             | 1,338         | 84.5%        |
| SM100 | 1,377        | 1,355             | 1,140         | 82.8%        |
| SM120 | 1,411        | 1,361             | 1,221         | 86.5%        |

hidden instructions; server GPUs have instructions related to the handling and encoding of sparse matrices [24], such as `GENMETADATA`, `SCATTER`, and `GATHER`. Some of these instructions may be vestigial; others are commonly used in the device functions.

## 5 Evaluation

We evaluate EoSS across seven architecture targets spanning five GPU generations (Volta through Blackwell) under CUDA Toolkit versions 12.8–13.0. We validate each target in three phases: static constraint checking, `cuobjdump` encoding verification, and runtime execution testing.

### 5.1 Runtime Validation

Table 2 summarizes the validation results across all architectures. Hopper (SM90) achieves the highest runtime-valid count at 1,338 of 1,584 instruction classes (84.5%), benefiting from its large ISA, which includes tensor and cooperative instructions. Consumer Blackwell (SM120) validates 1,221 of 1,411 (86.5%); the remaining instructions include graphics-pipeline opcodes (attribute loads, pixel shading, tessellation output) and specialized cluster and barrier variants that require descriptor or synchronization state not provided by the compute harness.

**Campaign Runtime.** Generation is negligible: SM70, SM90, and SM120 take 0.14, 0.42, and 0.68 s at  $k = 1$  variant, and 0.77, 2.13, and 2.84 s at  $k = 4$  variants per instruction class. On SM120,  $k = 1$  breaks down as 0.30 s for parsing, 0.37 s for generation, and 0.02 s for output. Full-ISA validation instead takes 14–52 min. On an RTX 5090, a multi-sequence survey of 846 campaign-safe instructions takes 2.3 s (five iterations, ten hot/cold launches). Single-sequence mode records 4,095 samples per triple; 1,221 classes take 31 min at three trials (1.53 s/triple) or 10 min at one. Startup (about 0.25 s/launch) dominates, so this mode is reserved for

<sup>2</sup>libnvidia-ptxjitcompiler.so

isolated instructions. A30 times are similar; without persistence, cold data-center launches take up to 2.4 s each. Issue and completion modes use the same launch count.

**Hardware Requirements.** The parsed ISA uses 70 MB (SM70) to 284 MB (SM120) of host RAM. Disk use is 100 MB per ISA and ~2 MB per run. The harness uses ~80 KB of device memory, so any GPU of the target architecture running CUDA Toolkit 12.8–13.0 suffices; no special privileges are required. Output is ~400 KB in multi-sequence mode but up to 300 MB for a full single-sequence log.

## 5.2 Runtime-Invalid Instructions

The Rosetta String defines a static validity check called an “Illegal Instruction Encoding Error,” which validates the program’s shader type against known enums. The shader type is specified as a binary parameter, similar to the maximum register count per thread, and defaults to 0 (ST UNKNOWN). There are nine shader types: six for graphics (one per rendering pipeline stage, e.g., vertex, geometry, tessellation), one for compute (ST CS), one for trap handlers (ST TRAP), and one unknown (ST UNKNOWN). Most instructions are compatible with either compute or graphics, and all instructions are compatible with ST TRAP and ST UNKNOWN.

A special register called SM\_SHADER\_TYPE (not reachable from PTX) reports the active shader type at the hardware level. Inside a regular compute kernel, this register returns 6, which in the Rosetta String’s enum ordering corresponds to ST VSA (Vertex Shader A), indicating that the hardware’s internal enum ordering differs from the specification’s. Using our framework to patch a special-register read into the trap handler (Section 4.6) and then triggering it, we still observe a value of 6, meaning that the trap handler does not run in ST\_TRAP mode. Consistent with this, patching any graphics-only instruction into the trap handler causes immediate warp termination, confirming its compute-mode execution context.

Binary parameters are stored in the .nv.info ELF section as typed attributes (EIATTRs). For example, EIATTR\_MAXREG\_COUNT controls the maximum register count per warp. One such attribute is EIATTR\_SHADER\_TYPE. However, setting this attribute to any value, including graphics shader type enums, produces no change in the hardware-reported SM\_SHADER\_TYPE register and does not enable execution of graphics-only instructions. The shader type is enforced at the engine or context level, not through binary metadata.

Using the user-space graphics pipeline from Section 4.7, we bind a shader whose program header declares a chosen pipeline stage and confirm that the hardware then reports the matching SM\_SHADER\_TYPE. We do not write the register directly (it is read-only); instead, we select the stage through the header field and the pipeline binding slot. The hardware enforces consistency between these values. Crucially, SM\_SHADER\_TYPE is not a passive hint but the enforced

**Table 3.** Runtime SM\_SHADER\_TYPE (SR20) per pipeline stage, read by an S2R inside an executing shader. Vertex, pixel, and compute values are measured; geometry and tessellation values are fixed by the SPH stage enum. “Inst exec.” counts SM120 graphics-only instructions executed in that stage; compute rejects them all.

| Stage            | SR20 Value | Inst exec. |
|------------------|------------|------------|
| Vertex           | 1          | 19         |
| Pixel            | 5          | 9          |
| Compute          | 6          | 0          |
| Geometry         | 4          | 0          |
| Tess. Control    | 2          | 0          |
| Tess. Evaluation | 3          | 0          |

runtime legality gate. In a differential test, identical instruction bytes execute in one stage and fault in another purely as a function of SM\_SHADER\_TYPE: IPA (attribute interpolation) executes only in the pixel stage, while AST (attribute store) executes only in the vertex/geometry stages. Section 5.3 reports the full runtime shader-type mapping and per-stage execution coverage.

## 5.3 Graphics Instruction Execution

The user-space graphics pipeline (Section 4.7) lets us execute the graphics-only opcodes that the compute harness cannot reach. On consumer Blackwell (SM120), 37 instruction variants are graphics-restricted (valid in a graphics stage but not in compute). Of these, 28 are reachable from the vertex or pixel stage, and *all 28 execute successfully* in our context: 19 in the vertex stage (attribute load/store ALD/AST, AL2P, CSMTEST, ISBERD, ISBEWR, VOTE\_VTG) and 9 in the pixel stage (IPA×6, KILL, LDTRAM, PIXLD). The remaining nine are exclusive to stages that require additional setup and are left for future work: six are geometry-only (OUT emit/cut/final), and three are tessellation-control-only (patch-domain AST). No instruction is exclusive to the tessellation-evaluation stage.

We identify each stage by reading SM\_SHADER\_TYPE (SR20) from within an executing shader via an S2R whose result is stored to host-readable memory. Table 3 reports the measured values. For every stage we can launch (vertex, pixel, and compute), the runtime value agrees with the shader-program-header (SPH) pipeline-stage enum.

## 6 Case Study 1: TTU Instructions

The eight TTU instructions recovered in Section 4.8 are absent from server GPUs, hidden by `nvdiasm` and `cuobjdump` (Figure 4), unreachable from PTX, and previously observed only in JIT-compiled OptiX kernels. We test whether they execute independently of OptiX and characterize their protocol and scheduling constraints.

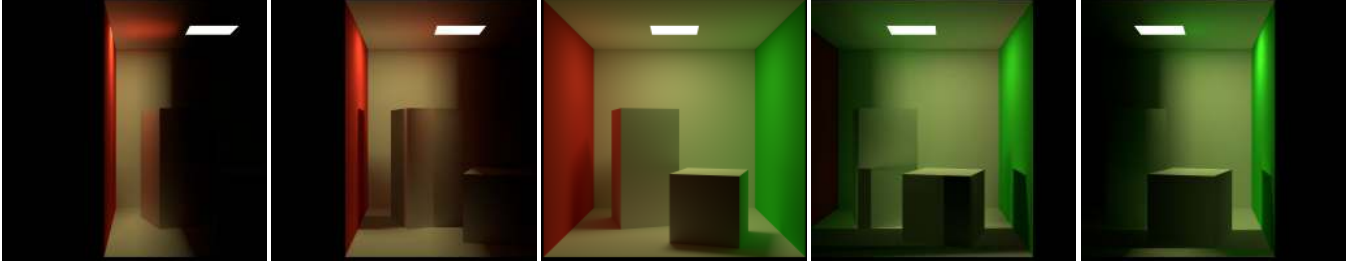


**Figure 5.** Effect of TTU instruction modification on the OptiX optixPathTracer sample (Cornell box scene). We intercept JIT-compiled traversal cubins in GPU memory before pipeline creation and apply targeted binary patches. We sweep one of the input registers of the TTUST[0xC0] store instruction across a range of values. This produces lateral camera movement, indicating that either the ray origin or direction changes. Every ray, including each recast ray, is modified; hence, lighting artifacts occur.

**Table 4.** TTU instruction set on SM120. *Standalone* indicates execution outside a fused group; *Latency* is per-instance overhead above the NOP baseline. The fuse count must match the number of fused instructions. The 62-cycle TTULD.CLOSE entry likely reflects the cost of closing the TTU session; asynchronous traversal latency is reported below.

| Instruction  | Standalone | Latency |
|--------------|------------|---------|
| TTUOPEN      | ✓          | 6 cy    |
| TTUMACROFUSE | ✓*         | 6 cy    |
| TTUST (×8)   | fused only | 9 cy    |
| TTUGO        | fused only | 5 cy    |
| TTULD (×6)   | fused only | 5 cy    |
| TTULD.CLOSE  | fused only | 62 cy   |
| TTUCCTL      | ✓          | 10 cy   |

\*Standalone only with a fuse count of 0.

**Protocol Constraints.** We patch each of the eight TTU instruction classes into our timing harness with constraint-satisfying encodings, varying the context: single vs. repeated invocation, with and without scoreboard barriers, and distances from preceding TTU instructions.

Three instruction classes execute from a standalone compute kernel. TTUOPEN completes without error but faults upon a second invocation, indicating per-warp session state that rejects reopening an active context. TTUMACROFUSE executes when its operand field is set to zero; nonzero values cause a ~6-second GPU watchdog timeout. An NVIDIA patent [22] reveals that its operand specifies an *instruction count*: the macro-fuse executes that many subsequent instructions atomically with interrupts disabled. In our harness, a nonzero count attempts to fuse non-TTU instructions and deadlocks. TTUCCTL, with the IVALL (invalidate all) cache-control modifier, similarly executes in its default configuration but faults when patched with an active scheduling scoreboard write.

When tested *individually* in the timing harness, the three traversal-core instructions—TTUST, TTUGO, and TTULD—all fault because they must appear within a correctly fused instruction group initiated by TTUMACROFUSE. Although dumping process memory helped us identify the instructions, fully understanding the required protocol ordering involved intercepting live OptiX optixPathTracer kernels, modifying TTU instructions in flight, and observing the effects on rendered output.

We obtain the target virtual address (VA) by dereferencing OptiX API structure handles (e.g., OptixPipeline). Because cudaMemcpy faults on the host when directed to this VA, we take advantage of a property of the GPU memory system: a kernel can write to instruction pages. Within the same context, we launch a small kernel that loads instructions from or stores them to the previously obtained VA. We scan the region around the base address for TTU instructions. After finding them, we can modify their operands or replace the instructions entirely.

We confirmed the complete protocol sequence: TTUOPEN → TTUMACROFUSE → TTUST (×8) → TTUGO → TTULD (×6) → TTULD.CLOSE. Figure 5 shows the effects of our *in vivo* TTU modifications. Varying the register feeding TTUST[0xC0] moves the camera from left to right. Other, more dramatic modifications include omitting TTULD.CLOSE, which causes a permanent GPU hang, and zeroing individual TTUST writes, which reveals the offsets that carry ray-origin, ray-direction, and handshake data. These identical sequences across SM generations make the protocol applicable to every architecture with RT Core instructions.

**Full Traversal from Compute.** Using a fuse count of 8, we construct a complete traversal. We open the session with TTUOPEN, then issue TTUMACROFUSE with count 8. The fused group contains four TTUSTs, one TTUGO, two TTULDs, and TTULD.CLOSE. This sequence executes successfully from a standalone compute kernel, without OptiX, without driver modifications, and without any privileged interfaces. We verified the result using NVIDIA’s hardware performance

counter `rtcore__rays.sum`, which reports exactly one ray processed by the RT Core. Incorrect fuse counts (e.g., 15 instead of 8) produce the previously observed watchdog timeout, confirming that the fuse count is the critical parameter.

**Traversal Latency.** Traversal is asynchronous, so we measure it from TTUGO to the first result-consuming instruction. A single-level tree takes  $\sim 1,600$  cycles. Adding leaves without increasing tree depth leaves traversal time nearly unchanged, whereas each additional level produces a step increase:  $\sim 2,100$  cycles at two levels,  $\sim 2,900$  at three, and  $\sim 4,400$  at four. Latency is nearly flat within each depth band despite 16–64 triangles at two levels and 180–1,500 at three. Dependent node fetches and sibling tests at greater depths, rather than triangle count alone, dominate traversal cost.

Together, these experiments provide the first public ISA-level characterization of all eight TTU classes.

**Writing a Simple Ray Tracer.** After reverse-engineering the ray-tracing unit, mapping its output from OptiX and Vulkan, and validating that it works in compute mode, we bypass all frameworks and drive the hardware directly. To render a scene, we encode it as a BVH: each triangle is enclosed in a box, and boxes are grouped into progressively larger boxes until one box encloses the entire model. A ray tests a few large boxes and descends only where it might hit geometry, allowing it to skip most of the mesh.

We wrote a builder that takes an arbitrary 3D object and encodes its triangles into this tree. We allocate GPU memory through the low-level CUDA virtual-memory API (`cuMemCreate/cuMemMap`), requesting a base address in the low 4 GB of the address space because the TTU dereferences some references as 32-bit words. We copy the BVH into this region and relocate it. Because child pointers are absolute virtual addresses, we update them accordingly. After relocation, we pass the structures to the ray-tracing kernel.

The simple ray tracer assigns one pixel to each thread and converts the pixel coordinates into a camera ray. We use an orthographic camera, so all rays are parallel. We configure the TTU using `TTUST[0x40]` for the ray origin ( $X$  and  $Y$ , while  $Z$  remains constant) and `TTUST[0x50]` for the ray direction (only  $Z$  is set to the nonzero value 1.0).

We store the BVH root node in `TTUST[0x100]` and copy the rest of the configuration directly from Vulkan, including the cull mask, ray flags, and identity. We use TTUGO to initiate traversal. A `TTULD[0x300]` instruction reads back the traversal result, including a hit/miss bit and, for a hit, the triangle’s primitive ID and distance  $t$  (in FP32). We store the hit bit and distance for each pixel.

We assemble an image by coloring each pixel according to its distance; darker colors indicate points closer to the camera. Figure 6 shows the result.

**Session Lifecycle and Preemption.** The requirement that `TTULD.CLOSE` must be issued exactly once raises a practical question: what happens when the GPU preempts a warp mid-traversal? Analysis of the CILP context-save routine



**Figure 6. Ecce Mono.** Our simple ray tracer, driven entirely by TTU instructions in compute mode, produces a monkey model [2]. Each pixel is colored according to the depth returned by the TTU.

in `libcuda.so` (Section 4.6) using our framework reveals the mechanism. The save handler reads a special register, `SR_TTU_TICKET_INFO` (SR133), defined in the Rosetta String but absent from any public documentation. If the value is nonzero, indicating an active RT Core session, the handler branches to execute `TTULD.CLOSE RZ, RZ, TTU[0x300]` to close the session before saving warp state. If SR133 is zero, the handler skips the `TTULD` entirely via an unconditional branch. This confirms that TTU session state is tracked per warp by a dedicated hardware register and that the driver must explicitly clean up active sessions during preemption to prevent the permanent GPU hangs we observed when `TTULD.CLOSE` is omitted.

The value of this special register is stored with the context metadata in global memory. When the context is restored, the value is checked again, and `TTUOPEN` is issued if the session was previously active.

## 7 Case Study 2: Contention-Based Effects

Contention on shared CPU resources can leak information across threads [13, 49]. We investigate analogous effects in GPU special registers and hardware state, using EoSS to reach resources not fully exposed through PTX.

**Read-side contention.** We sweep all 256 CS2R selector values while saturating all four SM sub-partitions (128 threads / block), measuring per-warp latency variance across block counts from 1 to the full SM complement. On both the H100 (SM90) and RTX 5090 (SM120), read latency remains flat across all 256 selectors and all concurrency levels; 2,048 configurations on SM120 produce identical latency with zero variance. The CS2R broadcast path delivers special-register values without serialization.

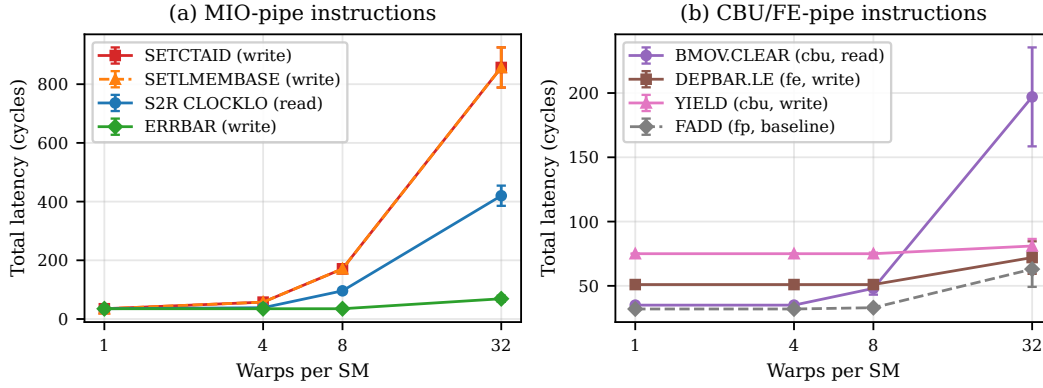


**Figure 7.** Instruction latency vs. SM occupancy on SM120. MIO-pipe operations (SETCTAID, S2R) show linear contention scaling with warp count; writes serialize  $2\times$  more than reads. FP-pipe operations (FADD) show a modest latency increase, much smaller than that of the MIO-pipe operations. Contention is per-SM only.

**Write-side contention.** To test whether *writing* to hardware state creates contention, we patch 128 consecutive slots with state-modifying instructions (SETCTAID, SETLMEMBASE) or MIO-pipe reads (S2R) and sweep occupancy from 1 to 32 warps per SM on SM120.

Figure 7 reveals a clear asymmetry: MIO-pipe writes serialize at 3.96 cycles per operation per competing warp, while MIO reads serialize at 1.91 cy/op/warp, a  $2.0\times$  penalty. FP-pipe instructions (FADD) show a modest latency increase, much smaller than that of the MIO-pipe operations. At 32 warps per SM, SETCTAID latency grows from 35 to 857 cycles ( $24.5\times$ ), making warp count inferable from instruction timing and creating a measurable intra-SM timing signal substantially larger than the FADD baseline. One possible explanation for the jump at 32 warps is that the available MIO resources cannot serve all requests concurrently, causing additional waiting; the much smaller increase in FADD latency suggests that the stronger contention effect is specific to the MIO pipeline.

Contention is strictly intra-SM. To test cross-SM independence with guaranteed temporal overlap, we execute 1.28 million operations per warp (128 unrolled slots  $\times$  10,000 iterations;  $\sim 1.8$  ms) while scaling from 1 to 170 CTAs and from 4 to 32 warps per SM. The CTA sweep adds no overhead, confirming that MIO-pipe resources are per-SM and share no interconnect contention.

## 8 Case Study 3: OSIRIS-Style Fuzzing

OSIRIS [53] discovers microarchitectural side channels on CPUs by executing instruction triples (reset, trigger, measure) and comparing execution time with and without the trigger. We adapt this methodology to GPU SASS, testing over 1,100 of SM120’s 1,221 runtime-valid instruction classes (RTX 5090) to test whether differential timing exposes microarchitectural state changes across GPU pipelines.

**Measurement design.** Each measurement slot follows the sequence: RESET  $\rightarrow$  TRIGGER  $\rightarrow$  CS2R (start clock)  $\rightarrow$  MEASURE  $\rightarrow$  CS2R (end clock). The clock window brackets only the MEASURE instruction. In the *hot path*, TRIGGER contains a real instruction; in the *cold path*, TRIGGER is replaced with a NOP via binary patching, ensuring the instruction is physically absent rather than merely predicated off. The hot/cold differential isolates whether the presence of a preceding instruction affects MEASURE’s observed latency—the signature of a timing side channel.

In *issue mode*, no scoreboard dependencies link the instructions: the differential reflects whether TRIGGER’s microarchitectural side effects alter MEASURE’s dispatch timing. In *completion mode*, scoreboard barriers enforce ordering for eligible instructions: the tracked completion event for TRIGGER must precede CS2R\_START, and that for MEASURE must precede CS2R\_END. The measured window therefore includes MEASURE’s scoreboard-tracked completion latency and the timestamp overhead.

**Campaign configuration.** The high-confidence campaign uses a *single-sequence* harness (4,096 identical slots, 4,095 samples per launch). The faster *multi-sequence* harness assigns different triples per slot with randomized positions; its results match those of the single-sequence harness within the noise, confirming no positional bias.

**Per-pipeline examination.** Figure 8 groups completion latency across architectures by pipeline.

**Compute.** Compute latency declines slightly across generations; SFU latency, for example, falls by approximately 20%. The high-latency peaks on consumer architectures correspond to FP64 operations. The panel combines eligible ALU, FP, and SFU instruction classes, so its modes characterize the tested instruction mix rather than a single representative compute operation. As in the other panels, the measured intervals include the clock-read overhead.

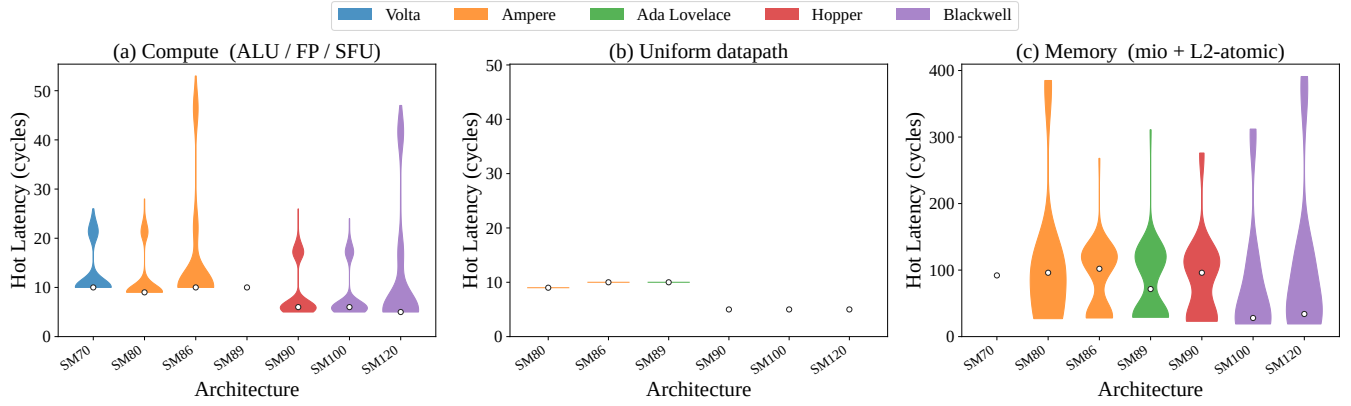


**Figure 8.** Distribution of completion latency for eligible instructions across seven GPU architectures for the scoreboard-enforced hot path, including the latency of the special-register capture instructions. Panels group instructions by pipeline: (a) compute, (b) uniform, and (c) memory.

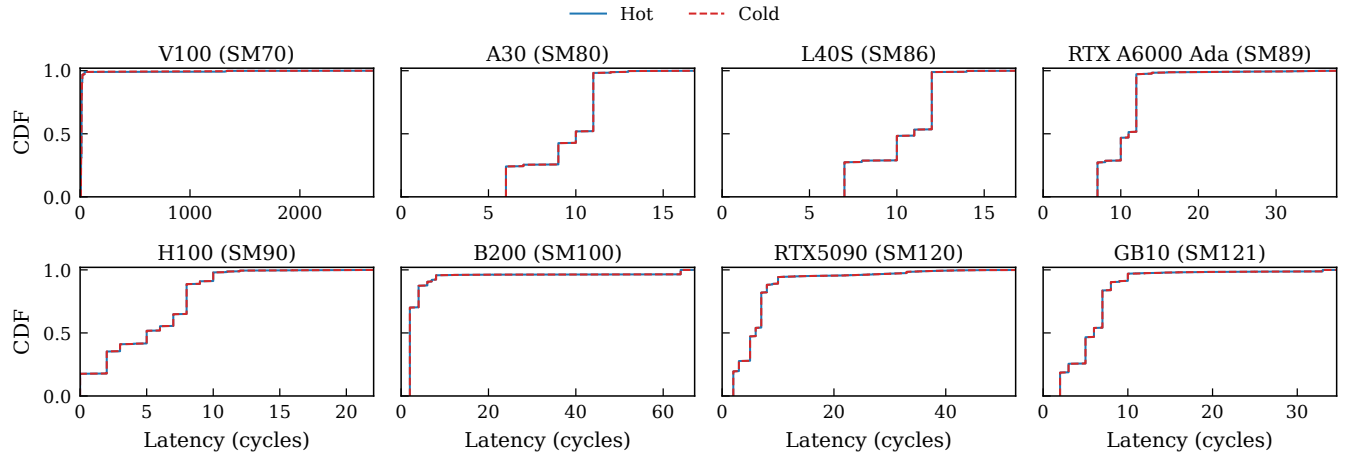


**Figure 9.** Cumulative issue-latency distribution for hot and cold paths across seven architectures. Hot and cold CDFs overlap nearly perfectly on all architectures, confirming that no tested instruction produces a measurable issue-mode side channel.

**Memory.** Memory behaves differently: L1-hit latency is stable across generations; texture instructions form a second mode, while L2-backed atomics form a third, highest-latency mode. L2 latency rises across successive consumer and server generations, except on server Ampere, whose sixfold L2 expansion introduced a new subsystem organization retained by later microarchitectures as capacity grew more modestly.

**Uniform.** Uniform instructions, introduced with Turing, execute on NVIDIA’s scalar unit and issue from one thread per warp. Their count nearly triples from 70 to about 200 on Hopper, and they increasingly drive complex asynchronous operations: all TMA instructions use the uniform pipeline, as do Blackwell tensor-core instructions. Their latency rises with this shift from common warp-wide scalar work to a single-threaded interface for complex operations.

**Issue-mode results: a negative finding.** On SM120, the vast majority of instructions produce zero hot/cold differential. Figure 9 confirms this across all seven architectures: hot and cold CDFs overlap nearly perfectly on every generation. OSIRIS finds dozens of interacting pairs per CPU pipeline; our tested encodings and trigger/measure configurations showed little or no issue-mode hot/cold differential.

## 9 Related Work

**Binary Tools.** NVBit [52] dynamically inspects and removes SASS instructions, but it does not expose field-level encoding control, construct arbitrary native encodings, or persistently rewrite cubins. EoSS instead recovers mappings from bits to opcodes, operands, qualifiers, validity constraints, and scheduling fields, enabling constraint-satisfying construction and persistent modification. NVBit also inherits NVIDIA

disassembler visibility limits; EoSS works on raw encodings and therefore reaches suppressed instructions and fields.

**SASS Reverse Engineering.** Manatschal [26] parses the Rosetta String and generates latency benchmarks for one GPU (SM86). EoSS additionally evaluates constraint ASTs across the entire ISA, validates runtime executability, targets seven architectures (SM70–SM120) with 539–1,338 runtime-valid classes each, and explores the `VALID_IN_SHADERS` bitmask and TTU family. Other efforts enumerate visible instructions by fuzzing `nvdisasm` output [9]; Nouveau NAK [1] is an open-source compiler for consumer GPUs.

SASS King [27] is a complementary reverse-engineering effort for consumer Blackwell (SM120) that focuses on compiler choices and patterns; as of this writing, it supports tensor operations and some asynchronous instructions. In contrast, EoSS systematically recovers field-level ISA specifications across the full instruction set on SM70–SM120 and uses this coverage to study exotic instruction families hidden by official tools, including TTU and graphics-only operations. For older GPU generations (prior to the 128-bit encoding adopted in Volta), the work in [17] sets a precedent for binary-level GPU ISA analysis.

**ISA Fuzzing and Side-Channel Discovery.** ISA-level fuzzers [10, 11, 47, 48] and side-channel synthesizers [5, 13, 42, 53] explore CPU decoder surfaces and microarchitectural leakage. We adapt OSIRIS [53], a black-box CPU fuzzer using instruction triples and timing oracles, to GPU SASS. RTL-level fuzzers [20, 23] and program-generation approaches [16, 45, 46, 55] target pre-silicon bugs or software proxies; GPU compiler fuzzing [12] targets shading languages. Prior work on GPU memory safety [15, 43], uncore leakage [29], TLB reuse across MIG partitions [57], and transient-execution attacks [3, 4, 25, 30] motivates ISA-level exploration. EoSS differs by targeting native ISA behavior on production GPU hardware.

## 10 Limitations, Future Work, and Conclusion

**Limitations.** By default, EoSS generates one representative encoding per instruction class; although callers can request multiple satisfying encodings, exhaustive exploration of operand-dependent behaviors (e.g., cache-line-crossing loads and denormalized FP inputs) is not performed. EoSS’s default harness operates in a compute context; to reach instructions gated behind graphics-engine state, we additionally build a user-space graphics context (Section 4.7), which executes all 28 vertex- and pixel-stage graphics-only instructions on SM120 (Section 5.3). The nine instruction types exclusive to the geometry and tessellation stages remain unexecuted pending bring-up of those stages, and the `EIATTR_SHADER_TYPE` attribute alone has no observable effect (Section 5). EoSS does not infer computational semantics:

it observes timing and register side effects but cannot determine what an instruction computes.

**Future Work.** Further analysis of the driver device functions, particularly the CILP save/restore protocol and device-side memory allocator, may reveal additional hardware state and scheduling mechanisms. Bringing up the geometry and tessellation stages would enable execution of the remaining graphics-gated instruction variants. More broadly, extracting and executing undocumented JIT-compiled sequences may generalize to other accelerators reached through opaque driver paths.

**Conclusion.** EoSS enables systematic native-ISA exploration on seven NVIDIA GPU architectures, from SM70 through SM120. Its case studies provide the first public execution and characterization of all eight RT Core TTU instructions from compute, demonstrate measurable intra-SM contention on PTX-unreachable resources, and find little or no issue-mode hot/cold timing differential in the tested configurations across five GPU generations. We release the framework, instruction datasets, fuzzing scripts, campaigns, and measurement harnesses.

## 11 Acknowledgments

Claude was used to refine portions of this work, including the prose, plotting scripts, and general-purpose scripts. This work was supported, in part, by NSF CCF #1943379 and NSF CCF #2450447, and by SNSF grant number #200021-236559.

## References

- [1] freedesktop.org [n. d.]. Nouveau: Accelerated Open Source driver for NVIDIA GPUs. freedesktop.org. <https://nouveau.freedesktop.org/>
- [2] Blender Online Community. 2026. Blender – a 3D modelling and rendering package. Blender Foundation, Amsterdam. <https://www.blender.org> Version 4.x.
- [3] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022.  $\mathcal{A}$ PIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture, In USENIX Security Symposium. *USENIX Security Symposium*.
- [4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association.
- [5] Anirban Chakraborty, Nimish Mishra, and Debdeep Mukhopadhyay. 2024. Shesha: multi-head microarchitectural leakage discovery in new-generation intel processors. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) (SEC ’24). USENIX Association, USA, Article 34, 18 pages.
- [6] cloudcores. [n. d.]. CuAssembler: An unofficial CUDA assembler. <https://github.com/cloudcores/CuAssembler>
- [7] Robert Crovella. 2023. Nvidia Developer Forums. (2023). <https://forums.developer.nvidia.com/t/cuda-errors-determine-sticky-ness/271625/9>
- [8] D0gf00d. 2019. NVIDIA SASS ISA Documentation (Rosetta String). Technical note. <https://github.com/0xD0GF00D/DocumentSASS>
- [9] Kuter Dinel. [n. d.]. Nvidia SM90a Instruction Set Architecture. [https://kuterdinel.com/nv\\_isa/](https://kuterdinel.com/nv_isa/)

- [10] R. Dofferhoff, M. Göbel, K. F. D. Rietveld, and E. van der Kouwe. 2020. iScanU: A Portable Scanner for Undocumented Instructions on RISC Processors. In *Proceedings of the 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
- [11] Christopher Domas. 2017. Breaking the x86 ISA. *Black Hat 1* (2017), 1–6.
- [12] Alastair F. Donaldson, Ben Clayton, Ryan Harrison, Hasan Mohsin, David Neto, Vasyi Teliman, and Hana Watson. 2022. Industrial Deployment of Compiler Fuzzing Techniques for Two GPU Shading Languages. Technical report.
- [13] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *Network and Distributed System Security Symposium (NDSS)*.
- [14] Scott Gray. [n. d.]. Maxas: Assembler for nvidia maxwell architecture. <https://github.com/NervanaSystems/maxas>
- [15] Yanan Guo, Zhenkai Zhang, and Jun Yang. 2024. GPU Memory Exploitation for Fun and Profit. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association.
- [16] Zihui Guo, Miaomiao Yuan, Yanqi Yang, Liwei Chen, Gang Shi, and Dan Meng. 2025. DiveFuzz: Enhancing CPU Fuzzing via Diverse Instruction Construction. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
- [17] Ari B. Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z. Zhang. 2019. Decoding CUDA Binary. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 229–241.
- [18] Guoliang He and Eiko Yoneki. 2025. CuAsmRL: Optimizing GPU SASS Schedules via Deep Reinforcement Learning. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (Las Vegas, NV, USA) (CGO '25)*. Association for Computing Machinery, New York, NY, USA, 493–506.
- [19] Rodrigo Huerta, Mojtaba Abaie Shoushtary, José-Lorenzo Cruz, and Antonio Gonzalez. 2025. Dissecting and Modeling the Architecture of Modern GPU Cores. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 369–384.
- [20] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1286–1303.
- [21] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. arXiv:1804.06826 [cs.DC] <https://arxiv.org/abs/1804.06826>
- [22] Ronald Charles Babich Jr., John Burgess, Jack Choquette, Tero Karras, Samuli Laine, Ignacio Llamas, Gregory Muthler, William Parsons Newhall Jr., and Nvidia Corp. October 5, 2021. Robust, efficient multiprocessor-coprocessor interface. US Patent US11138009B2.
- [23] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3219–3236.
- [24] Jorge Albericio Latorre, Ming Y. Siu, and Nvidia Corp. November 1, 2022. Compression techniques for data structures suitable for artificial neural networks US Patent US11489541B2.
- [25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.
- [26] Dumeni Manatschal. 2025. Towards Calculating HPC CUDA Kernel Performance on Nvidia GPUs.
- [27] Florian Mattana. [n. d.]. SASS King. <https://github.com/florianmattana/sass-king>
- [28] Xinxin Mei and Xiaowen Chu. 2017. Dissecting GPU Memory Hierarchy Through Microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems* 28, 1 (2017), 72–86.
- [29] Yuanqing Miao, Yingtian Zhang, Dinghao Wu, Danfeng Zhang, Gang Tan, Rui Zhang, and Mahmut Taylan Kandemir. 2024. Veiled Pathways: Investigating Covert and Side Channels within GPU Uncore. Technical report.
- [30] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. 2020. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *IEEE Symposium on Security and Privacy*.
- [31] Cedric Nugteren, Gert-Jan van den Braak, Henk Corporaal, and Henri Bal. 2014. A detailed GPU cache model based on reuse distance theory. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 37–48.
- [32] NVIDIA. [n. d.]. CUDA Binary Utilities, CUDA Toolkit 12.8. <https://docs.nvidia.com/cuda/archive/12.8.0/cuda-binary-utilities/index.html>
- [33] NVIDIA. [n. d.]. NVIDIA Ampere A100 GPU ARCHITECTURE. <https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf>
- [34] NVIDIA. [n. d.]. NVIDIA Blackwell Architecture Sweeps MLPerf Training v5.1 Benchmarks. <https://developer.nvidia.com/blog/nvidia-blackwell-architecture-sweeps-mlperf-training-v5-1-benchmarks/>
- [35] Nvidia. [n. d.]. NVIDIA Linux Open GPU Kernel Module Source. <https://github.com/NVIDIA/open-gpu-kernel-modules/>
- [36] NVIDIA. [n. d.]. NVIDIA TESLA B200 GPU ARCHITECTURE.
- [37] NVIDIA. [n. d.]. NVIDIA TESLA H100 GPU ARCHITECTURE. <https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c>
- [38] NVIDIA. [n. d.]. NVIDIA TESLA V100 GPU ARCHITECTURE. <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>
- [39] NVIDIA. [n. d.]. Parallel Thread Execution - ISA. <https://docs.nvidia.com/cuda/parallel-thread-execution/index.html>
- [40] NVIDIA. 2018. NVIDIA Turing Architecture Whitepaper. <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>
- [41] NVIDIA. 2023. NVIDIA OptiX Ray Tracing Engine. <https://developer.nvidia.com/optix>.
- [42] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing Black-Box CPUs against Speculation Contracts. Preprint.
- [43] Jonas Roels, Adriaan Jacobs, and Stijn Volckaert. 2025. CUDA, Woulda, Shoulda: Returning Exploits in a SASS-y World. In *Proceedings of the 18th European Workshop on Systems Security (Rotterdam, Netherlands) (EuroSec'25)*. Association for Computing Machinery, New York, NY, USA, 40–48.
- [44] John Russell. [n. d.]. GTC 2019: Chief Scientist Bill Dally Provides Glimpse into Nvidia Research Engine. ([n. d.]).
- [45] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. 2021. SiliFuzz: Fuzzing CPUs by proxy. Technical report.
- [46] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2023. Cascade: CPU Fuzzing via Intricate Program Generation. In *USENIX Security 2024*.
- [47] Fredrik Strupe and Rakesh Kumar. 2021. Uncovering Hidden Instructions in Armv8-A Implementations. In *Proceedings of the 9th International Workshop on Hardware and Architectural Support for Security and Privacy (Virtual, Greece) (HASP '20)*. Association for Computing Machinery, New York, NY, USA, Article 3, 9 pages.

- [48] Fabian Thomas, Eric García Arribas, Lorenz Hetterich, Daniel Weber, Lukas Gerlach, Ruiyi Zhang, and Michael Schwarz. 2025. RISCover: Automatic Discovery of User-exploitable Architectural Security Vulnerabilities in Closed-Source RISC-V CPUs. (2025).
- [49] Fabian Thomas, Michael Torres, Daniel Moghimi, and Michael Schwarz. 2025. ExfilState: Automated Discovery of Timer-Free Cache Side Channels on ARM CPUs. *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.
- [50] tiny corp. [n. d.]. tinygrad. <https://github.com/tinygrad/tinygrad/>
- [51] TOP500. [n. d.]. TOP500 - November 2025. [https://top500.org/lists/top500/2025/11/download/TOP500\\_202511\\_all.xml](https://top500.org/lists/top500/2025/11/download/TOP500_202511_all.xml)
- [52] Oreste Villa, Mark Stephenson, David Nellans, and Stephen W. Keckler. 2019. NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (Columbus, OH, USA) (MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 372–383.
- [53] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1415–1432.
- [54] Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. 235–246.
- [55] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. 2023. MorFuzz: Fuzzing Processor via Runtime Instruction Morphing Enhanced Synchronizable Co-simulation. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association.
- [56] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Optimizing Batched Winograd Convolution on GPUs. In *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP ’20)*. ACM, San Diego, CA, USA.
- [57] Zhenkai Zhang, Tyler Allen, Fan Yao, Xing Gao, and Rong Ge. 2023. TunnelS for Bootlegging: Fully Reverse-Engineering GPU TLBs for Challenging Isolation Guarantees of NVIDIA MIG. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (Copenhagen, Denmark) (CCS ’23)*. Association for Computing Machinery, New York, NY, USA, 960–974.
- [58] Chenggang Zhao, Zhean Xu, Liang Zhao, Jiashi Li, Chenhao Xu, Anyi Xu, Shengyu Liu, Kexing Zhou, and Kuai Yu. 2025. DeepGEMM: clean and efficient BLAS kernel library on GPU. <https://github.com/deepseek-ai/DeepGEMM>.

## A Artifact Appendix

### A.1 Description

This artifact enables the generation of arbitrary GPU SASS instructions and is hosted on Figshare at <https://doi.org/10.6084/m9.figshare.33709948>. It consists of several components:

- **JSON/**: SASS JSON schemas derived from the Rosetta strings.
- **SASS/**: A placeholder folder for per-architecture Rosetta extracts (expected filenames: `sm_xx_instructions.txt` and `sm_xx_latency.txt`). The Rosetta strings themselves are *not* redistributed as part of this artifact. We explain how to obtain the Rosetta strings in Section A.3.1.
- **OSIRIS/**: OSIRIS-style harnesses, build scripts, and campaign runners.

- **Contention/**: Code and configuration for the contention campaign.
- **TTU/**: A simple demonstration of TTU sequence patching and execution.
- **VisualModifier/**: An HTML-based tool for modifying the SASS instructions in CUDA applications.

#### A.1.1 Dependencies.

- An NVIDIA GPU. We tested the artifact on the B200, RTX 5090, GB10, H100, L40S, RTX 4060, A100, A30, RTX A6000, and V100.
- The CUDA Toolkit. We tested the artifact with version 12.8.
  - A system can have multiple versions of the CUDA Toolkit installed.
  - Rosetta extraction requires a CUDA Toolkit version earlier than 13.0.
  - For all other experiments, we recommend CUDA Toolkit 12.8, as it offers the broadest compatibility, spanning Volta through Blackwell.
- Python 3.12 or later.
  - Package requirements are listed in the `requirements.txt` file.

### A.2 Installation

Clone the artifact, install the required Python packages, and ensure that the binaries for the intended CUDA Toolkit version are on PATH.

### A.3 Experiment workflow

**A.3.1 Extracting Rosetta Strings.** We use DocumentSASS (commit 5d9acfa) to extract the Rosetta strings.

- Clone the repository:
  - <https://github.com/0xD0GF00D/DocumentSASS.git>
- Update the paths to NVCC and NVDISASM in the makefile (this extraction method only works with CUDA versions earlier than 13.0):
  - NVCC=/usr/local/cuda-12.8/bin/nvcc
  - NVDISASM=/usr/local/cuda-12.8/bin/nvdisasm
- Set the architectures in the makefile to those for which you want to extract Rosetta strings, for example: `sm_80 sm_86 sm_90 sm_120`.
- Run `make` to generate the Rosetta strings.
- Move the extracted strings to the artifact’s SASS/ folder.

#### A.3.2 OSIRIS Campaign.

**Run from scratch.** From the repository root, run:

```
1 cd OSIRIS
2 make full
```

The script detects the appropriate SM number for the main GPU on the system. If no Rosetta strings are detected, it uses the included post-processed JSON ISA files. Depending on the machine, building the harness may take some time.

After the harness has been built, candidate encodings are generated, checked with `cuobjdump`, and tested in patched harness binaries. These patched binaries are stored in the results folder and can be inspected afterward. You can disassemble them with `cuobjdump`, execute them, profile them, or use `compute-sanitizer` to validate that only valid binaries are generated.

Each instruction is then tested at runtime, with progress updates indicating how many instructions have been processed. Runtime varies across systems; we have observed durations ranging from 5 to 30 minutes.

The makefile reports statistics, including how many instructions were statically valid, which were not and why, and how many were valid at runtime. After this phase, run the following command to launch the issue and completion latency campaigns:

```
1 make campaign
```

To launch a full OSIRIS campaign, specify `PATTERN`:

```
1 make campaign PATTERN=osiris
```

See the README and the makefile for instructions on customizing `TRIALS`, `ITERATIONS`, and other parameters.

The OSIRIS campaign takes significantly longer than the diagonal campaign. To speed up execution, we use the multi-slot harness, with a different instruction triple written to each slot.

After the campaign finishes, generate plots for individual results by running:

```
1 make plot-violin CSV=results/smtx/..completion...
  csv
2 make plot-hotcold CSV=results/smtx/..issue...csv
```

#### A.4 Contention Analysis (Tested Only on SM120)

From the repository root, run:

```
1 cd Contention
2 make demo
```

This compiles all the code, runs the analysis, collects data, and generates plots.

#### A.5 TTU (SM120 only)

To run the TTU demo, navigate to the `TTU/` folder and run:

```
1 make demo
```

The TTU artifact generates a harness with dummy instructions, which are then replaced by the canonical TTU sequence.

Inspecting the binary (`ttu_demo_patched`) with `cuobjdump` shows blank instructions, as shown in Figure 4. The demo runs `compute-sanitizer` on the patched binary to show that the instructions execute without execution or memory errors. Finally, it uses `ncu` with the `rtcore__rays.sum` metric to show that the unpatched demo produces no rays, whereas the patched demo produces 32 rays.

You can also use the `VisualModifier` tool (described in the next subsection) to inspect the binaries and confirm that TTU instructions were introduced.

Another experiment is to run `NVBit` on the binary containing these invisible instructions and observe that they are ignored.

#### A.6 VisualModifier

The `VisualModifier` tool is an HTML application that can open any executable or standalone cubin and allows users to manipulate its SASS instructions. You can reorder instructions by dragging them, change instructions and their scheduling, modify CUBIN headers, request more registers, and perform other modifications. We provide a detailed usage guide in the README file.