NVLift: Lifting NVIDIA GPU Assembly to LLVM IR for Downstream Security Applications

Junpeng Wan
Purdue University
wan155@purdue.edu

Louis Zheng-Hua Tan
Purdue University
louistan@purdue.edu

Dave (Jing) Tian
Purdue University
daveti@purdue.edu

***Abstract*—NVIDIA GPUs underpin the vast majority of modern AI workloads. These workloads are ultimately executed in the form of Streaming Assembly (SASS), the lowest-level assembly for NVIDIA hardware. However, SASS remains largely undocumented, let alone well studied, posing a significant barrier to downstream security applications, such as security auditing, vulnerability discovery, binary hardening, etc.**

In this paper, we address this challenge with `NVLift`, a systematic framework that lifts NVIDIA GPU SASS into LLVM IR to enable downstream GPU binary analysis. To lift SASS instructions, `NVLift` reconstructs instruction semantics by consolidating prior reverse-engineering efforts and validating execution behaviors at runtime using `cuda-gdb`. To verify the semantic correctness of the lifted IR, we design and implement a differential testing pipeline by compiling the lifted IR into SASS and comparing the GPU execution results against the SASS generated from the reference CUDA kernel compilation. In total, `NVLift` supports 47 commonly used SASS instructions on the Turing architecture (SM75), covering 88.39% of instruction occurrence count in popular CUDA libraries. Using `NVLift`, we lifted 11 CUDA kernels, including representative DNN operators, and verified the semantic correctness of 5 kernels. We further provide a PoC implementation of GPU binary decompilation by translating the lifted LLVM IR into pseudo C code using `RetDec`. In sum, `NVLift` is a critical step towards enabling GPU binary analysis and downstream security applications.

I. Introduction

NVIDIA GPUs serve as the fundamental computational backbone of the contemporary artificial intelligence era. Their dominance is largely driven by the Compute Unified Device Architecture (CUDA), which provides the essential programming interface for leveraging the GPU's massive parallel processing power for general-purpose tasks. The CUDA compiler and runtime abstract complex low-level hardware details, enabling developers to write highly parallel code without directly managing warp scheduling, register allocation, or other microarchitectural resources.

To support execution across a rapidly evolving landscape of GPU microarchitectures, the NVIDIA CUDA compiler `nvcc` generates host executables that embed a *fat binary*. As illustrated in Figure 1, the final host binary contains native CPU code alongside this fat binary, which serves as a multi-architecture container for device-side representations. This container bundles a hierarchy of code, including architecture-independent Parallel Thread Execution (PTX) virtual instructions for *forward* compatibility and microarchitecture-specific SASS binaries for native execution on targeted GPUs. This structure allows targeting multiple generations of GPU microarchitectures simultaneously.

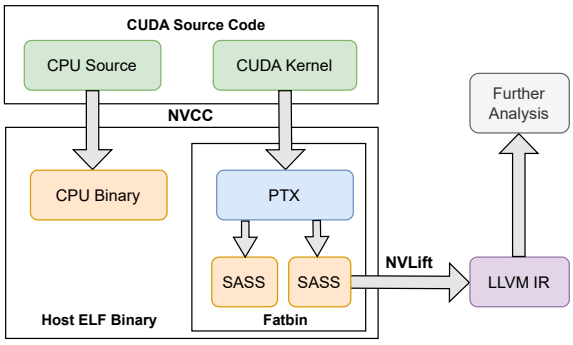

Fig. 1: CUDA binary and `NVLift` overview.

As the lowest-level executable representation, Streaming Assembly (SASS) is crucial for understanding the architectural behavior of NVIDIA GPUs, performance analysis, and security risk assessment of GPU binary programs. Unfortunately, NVIDIA provides minimal official documentation for SASS, typically exposing opcode names with minimal semantic detail [1]. As a result, while CPU binary analysis and its corresponding downstream security applications, such as decompilation, have been evolving and improving over the years, GPU binary analysis and downstream security applications have been stuck at the same starting point.

In this paper, we tackle this challenge and barrier with `NVLift`, a systematic framework that lifts NVIDIA GPU SASS into LLVM IR to enable GPU binary analysis and downstream security applications. Specifically, to lift SASS instructions, we systematically consolidate previously scattered reverse-engineering efforts and validate inferred instruction semantics and behavior at runtime through `cuda-gdb` inspection. To verify the semantic correctness of the lifted IRs, we design and implement a differential testing pipeline by compiling the lifted IRs into CUDA SASS and comparing the GPU execution results against the SASS generated from the reference CUDA kernel compilation using `clang`.

Workshop on Binary Analysis Research (BAR) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-08-4
https://dx.doi.org/10.14722/bar.2026.230028
www.ndss-symposium.org

In total, NVLift supports lifting 47 frequently used SASS instructions targeting the Turing architecture (SM75), covering 88.39% of all instruction occurrences observed in essential CUDA libraries such as cuBLAS and cuDNN. Using NVLift, we lift 11 CUDA kernel implementations of standard DNN operators, including ReLU, fully connected layers, convolution, and max pooling. Using our automated differential testing pipeline, we have verified the semantic correctness of 5 lifted CUDA kernels. To further demonstrate the applicability of the lifted IRs for downstream GPU binary analysis, we apply RetDec [2] to decompile the lifted IRs into C-like pseudocode that preserves the original functionality, realizing a proof-of-concept (PoC) implementation for GPU binary decompilation.

Our main contributions are as follows:

- **Recovered SASS semantics.** We consolidate prior reverse-engineering efforts and validate instruction behaviors through cuda-gdb inspection, resulting in a more complete and systematic characterization of NVIDIA GPU assembly instructions.
- **SASS-to-LLVM lifter.** We present NVLift, which lifts NVIDIA GPU assembly into LLVM IR. The current implementation supports 47 common SASS instructions in the Turing (SM75) architecture. Using NVLift, we manage to lift 11 representative CUDA kernels into LLVM IR.
- **Semantic correctness checking.** We develop an automated differential testing pipeline that compares the GPU execution behavior of lifted IRs against the reference IRs generated by compiling CUDA kernels using clang. 5 lifted CUDA kernels have been successfully verified.
- **PoC of GPU binary decompilation.** We demonstrate the first PoC of GPU binary decompilation via LLVM IR by applying RetDec to generate high-level C pseudocode representation.

NVLift is available at https://github.com/purseclab/Sass-LLVM-Lifter.

II. Background

A. GPU Architecture

GPUs are a cornerstone of parallel computation. Among them, NVIDIA GPUs are particularly popular due to their high performance and the Compute Unified Device Architecture (CUDA) programming framework. In NVIDIA GPUs, each Streaming Multiprocessor (SM) contains multiple cores that execute threads in a SIMT (Single-Instruction Multiple-Thread) fashion, organized into units called warps. All threads within a warp execute in lockstep, following the same program counter (PC) but operating on different registers and memory locations. Furthermore, each SM can execute multiple warps simultaneously, which are managed and scheduled by warp schedulers.

Typically, GPUs come in two forms: integrated and discrete. Integrated GPUs are built into the same chip as the CPU and share the same physical memory (e.g., Apple M-series SoCs). Other GPUs on the market are discrete, communicate with the system via a PCIe link, and have their own dedicated memory.

B. CUDA Binary

In a typical CUDA compilation flow, the programmer first writes CUDA C/C++ source code. The nvcc compiler (provided with CUDA) separates the host (CPU) part from the device (GPU) part and then compiles them. The host part is passed to a conventional host compiler like gcc and compiled to CPU machine code. The device part is first translated to NVVM IR [3], which is a subset of LLVM IR. Then, PTX is generated from the NVVM IR. PTX is NVIDIA's virtual GPU instruction set architecture (ISA) and is designed to provide forward compatibility, allowing PTX code to be just-in-time compiled by the CUDA driver into native code for newer GPU architectures. At build time, nvcc can further assemble PTX into SASS (the low-level, architecture-specific assembly language), which corresponds to the actual binary instructions executed by the GPU. The binary file, therefore, contains the CPU machine code together with the GPU part (often called a fat binary), which may include PTX, several SASS images for different GPU generations, or a combination of both. Our lifter translates SASS to LLVM IR for further security and program analysis, as shown in Figure 1.

C. LLVM IR and NVVM IR

LLVM IR is a low-level, strongly typed intermediate representation at the heart of the LLVM compiler framework [4], commonly used for program analysis, optimization, and cross-architecture translation.

NVVM IR is NVIDIA's GPU-oriented derivative of LLVM IR [3]. It preserves the LLVM IR structure while constraining the instruction set and introducing CUDA-specific intrinsics (e.g., accessing thread and block indices) to model the CUDA execution and memory hierarchy. NVVM IR serves as the immediate representation of CUDA device code before it is lowered to PTX.

III. Overview

A. Motivation.

The motivation for this work stems from two observations. First, SASS is the instruction set ultimately executed on NVIDIA GPUs. Thus, analyzing SASS is essential for accurately understanding the concrete execution behavior of CUDA programs, including their performance characteristics and security-relevant semantics. Moreover, we translate SASS rather than PTX for the following reasons. First, SASS is sometimes the sole GPU representation found in deployed CUDA binaries: while PTX can be included to enable forward compatibility, it may be omitted to protect intellectual property or reduce binary size. Second, even in the scenario where only the PTX is present, PTX can be compiled into SASS using NVIDIA's toolchain. Thus, regardless of whether a binary initially contains SASS or PTX, SASS can be reliably obtained and lifted into LLVM IR.

Second, lifting SASS to LLVM IR enables the reuse of a mature and extensive ecosystem of LLVM-based analysis and optimization tools. LLVM IR provides a structured, SSA-based intermediate representation that is well suited for tasks

like static analysis, equivalence checking, symbolic execution (e.g., KLEE [5]), and decompilation (e.g., RetDec [2]). Therefore, the lifting efforts offer the binary analysis researcher a practical foundation for applying established program analysis techniques to CUDA programs.

B. Challenges

The successful implementation of `NVLift` requires overcoming several challenges, which we address in the following sections.

Lack of Official SASS Documentation. The description of SASS ISA from NVIDIA's documentation is high-level and vague [1]: it only briefly summarizes the functionality of each instruction opcode, without specifying instruction formats, operand usage, or potential side effects. Although the document mentions different register types and the existence of constant memory, it lacks detailed architectural specifications. For instance, it does not specify how many GPRs (General-Purpose Registers) are supported at the SASS level. To recover the precise semantics of many instructions, we either manually reverse-engineer the SASS instruction behaviors or rely on previously scattered reverse-engineering efforts.

Type Analysis. Certain opcodes expose operand types directly (e.g., `IMAD R4, R4, R5, R3` or `FADD R7, R2, R7`), making type inference straightforward. However, many instructions, such as `MOV R6, R2`, do not reveal operand types syntactically. In these cases, type information must be inferred from the context.

Parsing SASS Instructions. Parsing and reconstructing SASS semantics is challenging due to its heterogeneous memory spaces (global, local, shared and constant), numerous architecture-specific special registers (e.g., `SR_TID`, `SR_CTAID`), and multiple register types.

IV. Recovering and Understanding NVIDIA SASS Instructions

To construct the lifter, we first need to study the functionality, syntax, encoding format, and usage of each instruction. Our current investigation targets the Turing GPU architecture (SM75). However, the methodology is readily extensible to other NVIDIA architectures, such as Volta and Ampere. This assumption is based on the fact that across GPU generations, most SASS instructions preserve their core semantics, with architectural changes being largely incremental rather than disruptive [6].

A. NVIDIA Official Documents

Although under-documented, the NVIDIA official documentation provides and confirms several essential facts. Specifically, Section 4 of the *CUDA Binary Utilities* [1] clarifies the semantic intent of individual opcodes. For example, `FADD` denotes the addition of two FP32 floating-point values, while `LDG` represents loading data from global memory. Furthermore, Table 6 of the *CUDA Binary Utilities* [1] lists the valid instructions for Turing GPUs, which serves as the foundation for our work.

B. PTX Documents and Existing Reverse Engineering

Due to the absence of official documentation, we rely on information gathered from other reverse-engineering sources, including blogs, repositories, prior research papers, and PTX documentation [6]–[15].

Compare Instruction Set from PTX. The PTX ISA documentation [9] provides a detailed specification of PTX, the lowest-level intermediate representation publicly exposed by NVIDIA. Although there is a semantic gap between PTX and SASS, the PTX documentation remains a valuable reference for inferring SASS behavior. In particular, by comparing the extracted SASS assembly with the documented semantics of corresponding PTX opcodes, we can approximate the functional behavior of SASS instructions. For example, given the SASS instruction `IMAD R0, R7, R0, R4`, the related PTX ISA document specifies the instruction `mad.mode.type d, a, b, c`, whose semantics are defined as `t = a * b` followed by `d = t + c`. Based on this, we can infer that the SASS instruction implements the computation `R0 = R7 * R0 + R4`, consistent with its definition in official documents as an `Integer Multiply And Add` operation. However, even when an opcode shares the same name, its SASS semantics are not guaranteed to be equivalent to those of PTX. Therefore, we need further information to accurately reconstruct and validate the SASS semantics.

Existing Reverse Engineering Work. Several key resources that are helpful for us to understand the SASS ISA are listed below.

First, Cloudcore's article [6] provides an extensive overview of NVIDIA's SASS instruction set from a microarchitectural perspective. The work summarizes instruction formats, operand types, execution pipelines, and special registers, offering detailed explanations of how different SASS instructions behave on NVIDIA GPUs. Although unofficial, it serves as a valuable reference for understanding SASS semantics in the absence of complete public documentation.

Second, TuringAs [13] is an open-source SASS assembler for NVIDIA Volta, Turing, and Ampere GPUs. It takes human-readable SASS assembly as input and produces `cubin` binaries for specific architectures (e.g., SM70, SM75, SM80), enabling developers to handcraft and assemble low-level GPU kernels beyond what the standard CUDA toolchain directly exposes.

Third, the Nintendo Switch SDK documentation set [14] includes an index of SASS opcodes for NVIDIA's Maxwell microarchitecture (e.g., SM52). The collection provides comprehensive per-instruction entries detailing operand layouts, control modifiers, and functional behavior. This material offers concrete architecture-specific insights that complement community reverse-engineering efforts, enabling more precise reasoning of instruction semantics.

In addition to SASS reverse engineering, Jia et al. [7] present an in-depth microbenchmarking study of the NVIDIA Turing T4 GPU. By designing targeted instruction- and

memory-level microbenchmarks, they reverse engineer key microarchitectural characteristics, including instruction throughput and latency, tensor core performance, instruction encoding, and so on. Their analysis quantitatively compares Turing against prior generations and provides practical guidelines for developers and researchers seeking to optimize performance on Turing T4 GPUs.

C. Instruction Tester by `cuda-gdb`

Based on the aforementioned documentation, we obtain a preliminary knowledge of the functionality, usage, and operand formats of individual SASS instructions. To validate the semantic information, we perform an additional confirmation step using `cuda-gdb` [16], a debugging tool provided by NVIDIA that extends GDB to support the debugging of CPU and GPU code.

To examine the behavior of a specific instruction, such as `IMAD R0, R7, R0, R4`, we utilize `cuda-gdb` to pause program execution immediately before the instruction is executed. We then inject controlled values into the source registers, `R7`, `R0`, and `R4` in this instance, perform a single-step execution (`si`), and verify whether the resulting value in `R0` matches our predicted outcome. This methodology is equally effective for resolving ambiguities in register roles; by observing which register's contents changed, we can definitively distinguish between source and destination operands in previously undocumented instructions.

The instruction tester grounds our intuition regarding unknown instructions and modifiers in concrete execution data. When necessary, we also perform in-place binary patching (e.g., manipulating the instruction's hexadecimal representation to replace a hardwired `RZ` register with a general-purpose `R4` register) to bypass hardware limitations, such as the immutability of the zero register.

D. Summary of Turing SASS ISA

Based on the above-mentioned documentation and our instruction tester, we provide an overall overview of the Turing SASS ISA.

Registers. The official document [1] identifies several types of registers, including RX for general-purpose registers, URX for uniform registers, SRX for special system-controlled registers, and PX for predicate registers, but omits further details. We therefore summarize these additional details based on previous reverse-engineering work and cuda-gdb.

General-purpose registers (GPRs) `R0-R255` are 256 32-bit registers. Among them, `R255` (or `RZ`) is the architectural zero register and always returns a zero value by any read operation. Although GPRs are 32-bit, a 64-bit value can still be represented by a pair of consecutive 32-bit registers, assembled as `R`*`hi`*`||R`*`lo`*. For example, in `IMAD.WIDE R40, R41, R18, c[0x0][0x168]`, the destination `R40` denotes a 64-bit register pair `R41||R40`, where `R41` holds the high 32 bits. Similarly, when a 128-bit operand is expected, `R40` implicitly refers to the concatenation `R43||R42||R41||R40`.

Predicate registers are denoted as `P0`, `P1`, ..., `P7` [6]. Among them, `P7` is a special predicate register also referred to as `PT`, which is hardwired to logical `true`. Predicate registers are used to support *predicated execution*, where a predicate condition guards the execution of an instruction. A predicate is specified by prefixing an instruction with `@P`*`i`* or its negation `@!P`*`i`*. For example, `@P0 IMAD.WIDE R40, R41, R18, c[0x0][0x168]` indicates that the `IMAD` instruction is executed only if predicate register `P0` evaluates to `true`; otherwise, the instruction is suppressed and has no architectural effect. Similarly, the prefix `@!P0` causes the instruction to execute only when `P0` is `false`. In short, predicated execution enables control flow without explicit branch instructions.

Starting from the Turing, NVIDIA GPUs introduce *uniform registers* and *uniform predicate registers* as operands [6]. Uniform general-purpose registers are denoted as `UR`*`i`*, where `UR0-UR63` are writable uniform registers. Similarly, `URZ` is hardwired to zero. Besides, uniform predicate registers are denoted as `UP`*`i`*, where `UP0-UP6` are general uniform predicates and `UP7`, also referred to as `UPT`, is hardwired to logical `true`. These registers are accessed by instructions executed on the *uniform datapath*. Unlike normal general-purpose registers and predicates, which are private to each thread, uniform registers and uniform predicates are shared at the warp level: all threads within the same warp observe the same value. As a result, uniform datapath instructions are executed once per warp rather than once per thread.

Additionally, SASS special registers provide access to CUDA execution and indexing state defined by the GPU architecture. Common examples include `SR_TID.X`, `SR_TID.Y`, and `SR_TID.Z`, which denote the thread index within a thread block. Additionally, `SR_CTAID.X`, `SR_CTAID.Y`, and `SR_CTAID.Z`, denotes the block index.

Instruction Format. A SASS instruction consists of an opcode, a list of operands, and optional modifiers. The opcode specifies the function of operation, while modifiers refine its behavior (e.g., data type, rounding mode, cache policy). Operands follow a consistent convention: the first operand is typically the destination (`dst`), and subsequent operands are sources (`src`), drawn from registers, immediates, or constant memory. For example, in `IMAD.WIDE R40, R41, R18, c[0x0][0x168]`, the opcode is `IMAD`, the modifier is `.WIDE`, `dst` is `R40`, and `src` are `R41`, `R18`, and `c[0x0][0x168]`. In addition to instruction-level modifiers, SASS also supports operand-level modifiers. Some modifiers, such as `reuse` (e.g., `R41.reuse`), affect microarchitectural behavior without altering the functional semantics of the instruction. Others directly change operand semantics, including negation (`-R0`), absolute value (`|R0|`), and bitwise inversion (`~R0`).

Instruction Encoding. In Turing, each SASS instruction is encoded in 16 bytes, which include both the instruction itself and a control code. According to previous research [15], the encoding format is as follows:

The 4-bit predicate field (`P`) controls per-instruction predication. The lower three bits select one of eight predicate registers (`P0`–`P7`), with the special all-ones encoding corresponding to PT, the always-true predicate. The most significant bit acts as a negation flag, inverting the selected predicate when set (e.g., `!P1` and `!PT`). This design enables conditional execution without introducing explicit control-flow branches. The 12-bit Opcode field specifies the instruction, and the 80-bit Operand field encodes registers, immediates, addressing modes, and instruction-specific modifiers. The 24-bit control code provides explicit warp-scheduling directives, including dependency barrier indices, stall cycles, and yield hints.

Operand Types. According to previous work [12], common operands can be categorized into several types, including registers (e.g., `R10`), immediate values (e.g., `0x100`, `3.2`), global memory (e.g., `[R10]`), and constant memory (e.g., `C[0x0][0x50]`). For the operand data types, in the SASS code we lift, they are 32-/64-bit integer and floating-point values, along with predicate registers used for predication and conditional control flow. The underlying register file is untyped: the same 32- or 64-bit register contents may be interpreted as signed or unsigned integers, floating-point values, or raw bits depending on the opcode and its modifiers.

Memory Accessing. NVIDIA GPUs expose several distinct memory spaces, each characterized by different visibility scopes, access latencies, and dedicated instruction support. Global memory is visible to all threads across all thread blocks and persists for the lifetime of a kernel; it is accessed using global load and store instructions such as `LDG` and `STG`. Shared memory is an on-chip, low-latency memory region shared among threads within the same thread block and explicitly managed by the programmer, accessed via instructions such as `LDS` and `STS`. Local memory is logically private to each thread; despite its name, it is typically backed by global memory and is used primarily for register spills or thread-private arrays, with accesses issued through `LDL` and `STL`. Constant memory provides a read-only memory space visible to all threads and blocks, optimized for broadcast-style access when multiple threads read the same address; it is accessed via `LDC` instructions and the `c[...]` addressing form (e.g., `c[0x0][0x300]`).

Core Instruction Set. The NVIDIA official documentation [1] briefly categorizes Turing instructions into ten classes: 22 floating-point instructions, 21 integer instructions, 4 conversion instructions, 4 movement instructions, 6 predicate/CC instructions, 7 texture instructions, 16 Load/Store instructions, 4 surface memory instructions, 17 control instructions, and 7 miscellaneous instructions.

Among them, our work uncovers the formats and functionality of 47 instructions, which we also refer to as the core instruction set in this paper. Their opcodes and associated modifiers are summarized in Table II in the appendix. Most of the SASS instructions within the core instruction set are floating-point and integer instructions, such as `IMAD`, `IABS`, and `FMUL`. We also include data movement and Load/Store instructions, including `MOV`/`UMOV` for register or immediate transfers and `LDG` for loading data from global memory. Control flow is handled by `BRA` as well as `CALL` and `RET`. Besides, the `@` prefix denotes predicated execution: a predicated `BRA` or a predicated instruction such as `@P1 IMAD` is executed conditionally, while instructions like `ISETP` evaluate conditions and write the results to predicate registers (`P0`-`P7`). Finally, reconvergence and scheduling-related instructions such as `BSSY` and `BSYNC` are used to manage warp divergence and reconvergence.

V. NVLift Design and Implementation

In this section, we describe the detailed design and implementation of `NVLift`. Figure 2 illustrates the overall workflow: we first extract the cubin and disassemble embedded SASS instructions; we then split SASS into tokens, construct a control-flow graph (CFG), and organize them into our internal representation consisting of functions, basic blocks, instructions, and opcode/operands; after that, we perform type analysis to recover operand types; finally, each SASS instruction is lifted to LLVM IR. `NVLift` produces a complete LLVM IR module that can serve as the base for further analysis.

A. Extract and Disassemble

To extract SASS assembly from a CUDA program, we first use `cuobjdump` to inspect the compiled binary and extract the relevant `.cubin` files, and then apply `nvdisasm` to disassemble the extracted `.cubin` and generate the corresponding SASS code for subsequent analysis.

B. Parse SASS

We first tokenize the SASS code. For example, we decompose `IADD3.WIDE R7, R15, 0x2, RZ` into opcode (`IADD3`), modifier (`WIDE`) and operands (`R7, R15, 0x2, RZ`). After tokenization, we identify all branch instructions to construct a CFG that represents the program's execution paths. In addition to explicit branch instructions, such as `BRA`, conditional execution also serves as a branching indicator. In detail, we treat predication denoted by @ as a conditional branch. For example, in `@P2 IADD3 R7, R15, 0x2, RZ`, the instruction `IADD3 R7, R15, 0x2, RZ` is put in a separate basic block, with an explicit branch to the next instruction.

After the parsing step, our internal representation organizes the SASS assembly hierarchically into functions, basic blocks, and instructions. Each instruction is further decomposed into its opcode and operands, as Figure 2 shows.

C. Type Analysis

The goal of this step is to recover the data type of all operands, which is essential for LLVM IR lifting.

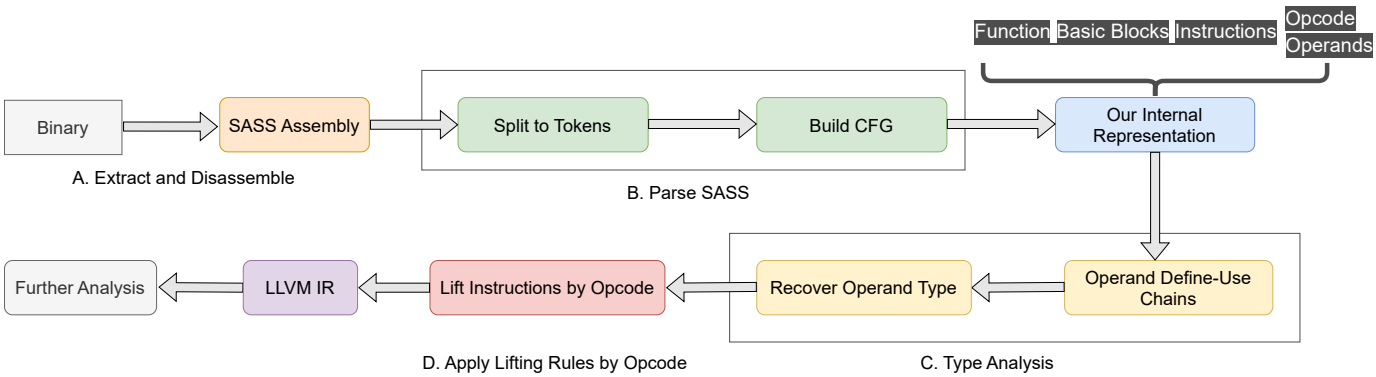

Fig. 2: The workflow of `NVLift`.

1) Infer Types from Opcode: Firstly, we try to infer operand types using the opcode semantics, as many opcodes directly imply their operands' types. For example, arithmetic instructions such as `IMAD` (integer multiply-add) and `FADD` (floating-point addition) explicitly operate on integer and floating-point operands, respectively. Consider the following instructions: `IMAD R4, R4, R5, R3` and `FADD R7, R2, R7`. In the first case, the `IMAD` opcode indicates that all source and destination operands (`R4`, `R5`, and `R3`) stores 32-bit integers. Similarly, the `FADD` opcode implies that `R7` and `R2` contain 32-bit floating-point values. Consequently, opcode semantics alone are sufficient to infer operand types in the above cases.

2) Infer Types from Context: Although operand types can often be inferred from opcodes, this approach does not apply to all SASS instructions. For instance, in `MOV R6, R2`, the value in `R2` is copied verbatim to `R6` without any type interpretation by the hardware, making the operand type unobservable from the instruction alone. A similar limitation applies to the destination register `R17` in `LDG.E.SYS R17, [R2+0x10]`, where the load instruction specifies the address space but does not encode the semantic type of the loaded value.

To solve the issue of the above instructions, we infer operand types from the assembly context. Taking `MOV R6, R2` as an example, if `R2` is previously defined or later used as an integer, we assign `R6` an integer type. Conversely, if `R6` is later used by a floating-point instruction before being redefined, we infer that `R2` also contains a floating-point value. This bidirectional and context-sensitive inference allows us to recover type information for type-agnostic instructions.

Operand Definition Reaching Analysis. To infer operand types from context, we first perform a reaching definition analysis [17], which allows us to construct both use–define (Use-Def) and define–use chains (Def-Use) [18], as shown in Figure 2. Built on the CFG, Use-Def associates each operand use with its most recent definition, while Def-Use tracks all subsequent uses following a definition. Since an operand's type must be consistent across its definitions and uses, we leverage these chains to infer operand types transitively.

The Reaching Definition Analysis process requires a predefined mapping of operands for each opcode to determine whether each operand constitutes a definition or a use. The detailed design and implementation of this component are described in Appendix A.

Type Analysis. After constructing the Use-Def and Def-Use chain, we analyze operand types with the following steps. The first step is opcode-based type assignment, where we traverse all SASS instructions whose opcodes provide type information, e.g., `IMAD` and `FFMA`, and assign types to each Definition and Use. The second step is type propagation, in which operand types inferred in the first step are propagated along the Use-Def and Def-Use chains. After several rounds of recursive propagation, the types of most operands can be recovered. The specific propagation rules are detailed in Section V-E. In rare cases, conflicting type information may arise during propagation; in such situations, the latest inferred type is applied, unless the operand type was previously marked as confident. Operands whose types cannot be determined from opcodes, modifiers, or surrounding context are assigned a default 32-bit integer type.

3) Recover CUDA Kernel Parameters and Types: We recover CUDA kernel parameters by analyzing constant memory that stores argument values. In our experiments with Turing (SM75), kernel arguments are typically stored in constant memory starting at `c[0x0][0x160]`, with subsequent arguments laid out sequentially. Integer arguments occupy 4 bytes, while pointer arguments occupy 8 bytes. We also optionally cross-validate argument types by demangling kernel symbols using `llvm-cxxfilt`; for example, `_Z4reluPfS_i` demangles to `relu(float*, float*, int)`. This heuristic relies on the presence of unstripped symbols and, when available, provides high-confidence type information for subsequent inference.

D. Apply Lifting Rules by Opcode

After recovering type information for each operand, the lifting process is continued using the `llvmlite` Python framework [19]. Specifically, `NVLift` employs `llvmlite.ir` to construct an `llvmir.Module` and lift each analyzed function into LLVM IR. `NVLift` also defines pseudo-functions and intrinsics, i.e., the NVVM thread index, to map GPU/SASS semantics into composable LLVM IR and

ultimately outputs standard NVVM IR text. In the lifting, we preserve the full program semantics while discarding low-level microarchitectural details that are irrelevant to semantics, such as warp scheduling behavior or hardware-specific optimizations.

For each instruction, we define a deterministic mapping rule to lift it into semantically equivalent LLVM IR. Before instruction translation, `NVLift` performs custom analyses and transformations, including CFG construction and type analysis, to determine control flow and operand types. Additionally, each SASS register is represented as a LLVM variable, with its contents loaded from or stored to the variable during instruction emission. In the following text, we briefly introduce the details of lifting each instruction type.

Arithmetic Instructions. A typical SASS instruction is an arithmetic instruction (e.g., multiplication or addition) on several operands and stores the result in a destination register. One example is the following instruction `IMAD R3, R3, c[0x0][0x4], R4`, which computes `R3 = R3 × c[0x0][0x4] + R4`.

Listing 1: LLVM IR for `IMAD R3, R3, c[0x0][0x4], R4`.

```
%"_.35" = load i32, ptr %"R3"
%"nvvm_blockdim_y" =
  call i32 @"llvm.nvvm.read.ptx.sreg.ntid.y"()
%"_.36" = load i32, ptr %"R4"
%"mul"  = mul i32 %"_.35", %"nvvm_blockdim_y"
%"add"  = add i32 %"mul", %"_.36"
store i32 %"add", ptr %"R3"
```

The lifted LLVM IR corresponding to this `IMAD` instruction is shown in Listing 1. The translation proceeds in three steps: first, the source operands `R3`, `c[0x0][0x4]`, and `R4` are loaded into temporary LLVM variables; next, the multiply-and-add computation is expressed using standard LLVM IR arithmetic instructions, namely `mul` and `add`; finally, the computed result is stored back into the variable for destination register `R3`.

To understand the semantics of this instruction, it is important to note that `c[0x0][0x4]` stores `blockDim.y` and is lifted via the corresponding NVVM special-register intrinsic. Besides, `R4` holds `threadIdx.y`, and `R3` contains `blockIdx.y`. Consequently, the updated value of `R3` is `blockIdx.y × blockDim.y + threadIdx.y`, which corresponds to the global thread index in the y-dimension of a two-dimensional CUDA grid.

In addition, we handle 64-bit operands during arithmetic instruction lifting. For example, in the instruction `IMAD.WIDE R12, R25, R24, c[0x0][0x170]`, the `WIDE` modifier denotes a 64-bit operation, indicating that both the destination and the final source operand are treated as 64-bit values [8], [11]. Consequently, `R12` represents a 64-bit register pair (`R13||R12`), and `c[0x0][0x170]` is accessed as a 64-bit constant.

Memory Access Instructions. We support memory accesses to global memory (`LDG`/`STG`) and shared memory (`LDS`/`STS`). In `NVLift`, effective addresses are lifted by mapping each memory region to the corresponding NVVM/LLVM address space: global (1), shared (3), constant (4), and local (5). For example, when lifting `LDG.E.SYS R15, [R2+0xc]`, `NVLift` materializes an address-space–qualified pointer, e.g., `%"inttoptr_bytes.6" = inttoptr i64 %".191" to ptr addrspace(1)`, which converts a 64-bit integer address into a pointer in address space 1. In the above example, the global memory load instruction `LDG` accesses the 64-bit global address space, implying that the base address register `R2` represents a 64-bit pointer. Accordingly, the effective address is formed using the register pair `R3||R2`.

Constant memory accesses appear in SASS as `c[bank][offset]`, e.g., `c[0x0][0x168]`. These locations are read-only from the kernel and are therefore only referenced (but not defined or updated) by SASS instructions. We further parse the constant-memory layout metadata to recover kernel-level information such as parameter locations and launch configuration values (e.g., dimensions) when available.

Branch Instructions. In `NVLift`, we handle two branch instruction types. The first type is an unconditional branch, such as `BRA`, which performs direct control-flow transfers. For example, `BRA `(.L_x_13)` unconditionally jumps to the label `(.L_x_13)`. And it is lifted as: `br label %".L_x_13"`. The second type is a conditional branch, implemented via *predication*. As mentioned in Section IV-D, a conditional branch is indicated by the predicate modifier `@P`$i$. For instance, `@!P1 BRA `(.L_x_12)` jumps to `(.L_x_12)` only if predicate `P1` evaluates to false. This instruction is lifted into LLVM IR as follows:

```
%".463" = icmp eq i1 %"P1.1", 1
br i1 %".463", label %".L_x_12", label %".
  L_x_8"
```

Function Calls. Internal functions could be invoked within the SASS code of a CUDA kernel. Such an internal function runs in the same thread context as the kernel and therefore uses the same register file and per-thread local memory address space. A `CALL` instruction transfers control to the Callee function, and `RET` returns control back to the Caller function. Take Listing 2 as an example, `R24` carries the input argument and `R22` carries the return value. Moreover, `R11||R10` are used to hold the return address (i.e., the callback target) across the call. In the lifting process, `NVLift` resolves internal function calls by expanding the `CALL` instruction into the CFG of the callee, ensuring that all instructions and operands are integrated into the caller's control flow.

Listing 2: An example of SASS function call

```
Caller:
/*2170*/                   MOV R24, R20 ;
/*2180*/                   MOV R10, 0x21a0 ;
/*2190*/                   CALL.REL.NOINC
`($__internal_0_$__cuda_sm20_rcp_rn_f32_slowpath) ;
/*21a0*/                   MOV R8, R22 ;


Callee:
```

```
$__internal_0_$__cuda_sm20_rcp_rn_f32_slowpath:
        /*4900*/                   SHF.L.U32 R11, R24, 0x1,
     RZ ;
        .......
        /*4c30*/                   MOV R22, R11 ;
        /*4c40*/                   MOV R11, 0x0 ;
        /*4c50*/                   RET.REL.NODEC R10
        `(_Z11gru_forwardPfS_S_iii) ;
```

Special Registers. CUDA exposes several SASS special registers that describe the execution configuration and the calling thread's coordinates. In particular, `SR_TID` encodes the thread index within a thread block (i.e., `threadIdx`) and `SR_CTAID` encodes the thread-block index within the grid (i.e., `blockIdx`); both are three-dimensional and thus provide `.x`, `.y`, and `.z` components. Here, *CTA* stands for Cooperative Thread Array (CUDA thread block), and TID stands for Thread ID. When lifting SASS that reads these special registers (e.g., via `S2R`), we map them directly to the corresponding NVVM intrinsics in NVVM IR [3], as shown in Listing 5 in the Appendix.

Special Instructions. One type of special instruction is the warp divergence instruction, such as `BSSY`, `BSYNC`, and `BMOV`: `BSSY` initiates a divergent region and records the reconvergence target for the warp; `BSYNC` denotes the corresponding reconvergence point where previously diverged paths are synchronized; `BMOV` resets the hardware bookkeeping state (e.g., the `b0`/`b1` branch-tracking registers) used by the SIMT control stack to manage per-thread execution. Our lifter ignores these instructions for semantics-preserving translation of functional behavior. However, when an explicit representation is desired, warp-level synchronization can also be modeled via NVVM intrinsics such as `@llvm.nvvm.bar.warp.sync(i32 %membermask)`. The `BAR` barrier is another class of special instructions. It typically originates from the high-level CUDA primitive `__syncthreads()`, which enforces synchronization among all threads within a thread block. In our lifter, such block-wide barriers are represented using the NVVM intrinsic `@llvm.nvvm.barrier0()`.

E. Other Details

Lifter Implementation. Before applying our SASS instruction recovery and lifting pipelines, we first implement 11 representative CUDA kernels, including standard DNN operators such as ReLU, fully connected layers, convolution, max pooling, and GRU. We then compile these kernels into CUDA binaries using `nvcc` with the `-arch=sm_75` option, producing Turing-specific `.cubin` files. We then use `cuobjdump` to extract the `.cubin` binaries and `nvdisasm` to disassemble them into SASS. Based on the obtained SASS, we reconstruct the semantics and usage of each encountered instruction using the strategies described in Section IV, and implement corresponding lifting rules for each instruction based on Section V-D. In total, the SASS of the 11 representative CUDA kernels contains roughly 40 instructions, all of which are covered by our supported core SASS instruction set, as listed in Table II in the appendix.

`NVLift` is implemented in 3,482 lines of Python and leverages `llvmlite` for programmatic LLVM IR generation. We containerized the toolchain using Docker to enforce a consistent input schema for the lifter, effectively pinning the SASS artifacts generated by `nvdisasm` to a known, compatible format. This prevents version-specific deviations in disassembly syntax from breaking the lifter's parsing logic, thereby ensuring stable IR generation and reproducibility. Furthermore, all 11 representative CUDA kernels can be lifted to LLVM IR, and the lifted IR can then be compiled back into PTX using `llc`. The generated PTX can be successfully compiled and executed on the GPU, confirming that the lifted IR is syntactically valid and executable.

Type Analysis Implementation. In detail, we perform type analysis through a multi-stage iterative process. After the build of Def-Use and Use-Def chains, type resolution proceeds by identifying *directly solvable* types. Instructions with explicit type semantic expectations, such as floating-point operations (e.g., `FADD`, `FMUL`) or integer arithmetic opcodes (e.g., `IADD3`, `IMAD`), allow us to assign types to their operands immediately. One caveat here is that we assume the compiler generating the SASS does not employ unconventional optimizations, such as performing integer arithmetic on floating-point data.

For the remaining operands, we apply a fixpoint iteration algorithm that propagates type constraints across the data-flow graph until convergence. The engine alternates between local resolution and global propagation. In the local resolution phase, types are inferred from instruction-specific semantics such as enforcing consistency for data movement (`MOV`, `UMOV`), applying type-coercion heuristics for bitwise logic (`LOP3`), and deriving value types from pointer pointees during memory accesses (`LDG`, `STG`). In the propagation phase, the engine traverses Use-Def and Def-Use chains bi-directionally to spread inferred types throughout the graph. This iterative process continues until the global type map stabilizes. To prevent conflicting data-flow type designations from introducing instability, we mark certain resolved operands as *confident types*. Once an operand is designated as confident, its type becomes immutable, acting as a fixed point that prevents upstream or downstream propagation from overwriting high-confidence assignments.

VI. Evaluation and Analysis

In this section, we first show that the core instruction set explored and supported by `NVLift` covers the most common instruction occurrences encountered in practice. Second, we check the functional correctness of the lifted LLVM IR. Finally, we present a decompiler pipeline as a downstream application to demonstrate the practical utility of `NVLift`.

A. Instruction Coverage

To evaluate instruction coverage by our supported instruction set, we analyzed opcode occurrences across 18 NVIDIA dynamic libraries (`.so`), including `libcublas` and `libcudnn`. Specifically, we extracted cubins targeting

TABLE I: Top-10 opcodes by occurrence (SM75).

Opcode	Occurrences	Share (%)	Opcode	Occurrences	Share (%)
IMAD	14,966,550	16.60%	LDG	4,837,123	5.37%
ISETP	8,323,456	9.23%	LEA	3,515,048	3.90%
MOV	6,968,693	7.73%	BRA	3,449,718	3.83%
IADD3	6,714,123	7.45%	LOP3	2,526,697	2.80%
FFMA	5,849,742	6.49%	SHF	2,286,332	2.54%

SM75 and disassembled them into SASS, yielding a total of 90,149,916 instructions. Among them, the core instruction set covers 79,683,855 occurrences, corresponding to 88.39% of all observed instructions. These results indicate that `NVLift` covers the majority of instructions commonly encountered in real-world GPU software.

Moreover, we observe that instruction usage is highly skewed: the top-10 opcodes alone account for 65.93% of all occurrences, as shown in Table I, and all of them are already supported by our implementation. Among these, `IMAD` is the most frequently executed instruction, accounting for 16.6% of the total instruction mix.

B. Testing for LLVM IR Lifted by `NVLift`

We check the functional correctness of the lifted IR through a differential testing framework that evaluates `NVLift`'s output against a compiler-generated ground truth. The end-to-end differential testing pipeline is illustrated in Figure 3.

Starting from a CUDA kernel, we use `nvcc` to compile the source into a binary, and then use `nvdisasm` to disassemble it into SASS. Then, given only the SASS, `NVLift` produces the lifted LLVM IR. Concurrently, we generate a reference LLVM IR by compiling the same CUDA source with `clang` using the `-emit-llvm` flag. Both the lifted and reference IRs are then lowered to PTX via the `llc` tool and then compiled into SASS.

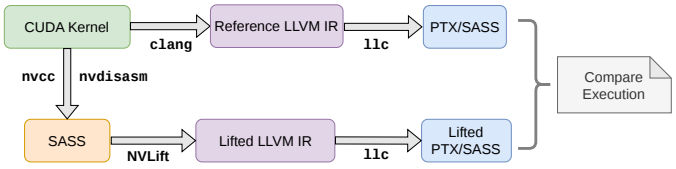

Fig. 3: The pipeline to test the lifted LLVM IR.

In our approach, we first specify a lightweight input schema that defines the execution configuration (e.g., block dimensions), argument types, and input distributions. For example, a ReLU kernel can be configured to use a uniform distribution over the range [-20.0, 20.0] with a fixed PRNG seed. Then, a Python harness consumes the above schema, uses the CUDA Driver API to load and execute both the lifted and reference assembly, and compares their outputs element-wise under configurable floating-point tolerances.

In our experiments, 5 of the 11 lifted CUDA kernels pass differential testing, including ReLU, fully connected layers, and kernels that exercise local, shared, and global memory address computations. The remaining six lifted LLVM IR kernels fail under the current implementation, with `Illegal Memory Access` error reported. This error indicates that the lifting process still misinterprets specific GPU addresses; addressing these limitations is deferred to future work and remains ongoing, as detailed in Section VII.

C. Case Study: Decompiler

In this case study, we demonstrate that `NVLift` can benefit downstream binary analysis tasks such as decompilation. Specifically, we build the SASS decompiler based on RetDec [2], an LLVM-based decompiler. Starting from the original `ReLU` CUDA kernel in Listing 3, we compile it with `nvcc` to a NVIDIA Turing cubin, obtain the corresponding SASS, and then use `NVLift` to lift the SASS into LLVM IR.

Listing 3: Original CUDA ReLU kernel

```
__global__ void relu(float *input, float *output
    , int size) {
    int idx = blockIdx.x * blockDim.x +
    threadIdx.x;
    if (idx < size)
        output[idx] = fmaxf(0.0f, input[idx]);
}
```

Listing 4: ReLU kernel pseudo-C code decompiled from Lifted LLVM IR

```
void _Z4reluPfS_i(float32_t * a1, float32_t * a2
    , int32_t a3) {
    uint32_t v1 =
    llvm_nvvm_read_ptx_sreg_ctaid_x() *
    llvm_nvvm_read_ptx_sreg_ntid_x() +
    llvm_nvvm_read_ptx_sreg_tid_x();

    if (v1 < a3 || false) {
        int64_t v2 = (int64_t)v1 * (int64_t)4 +
    (int64_t)a1;
        int32_t v3 = v2;
        float32_t v4 = *(float32_t *)(char *)(0
    x100000000 * (int64_t)(int32_t)((v2 & -0
    x100000000) >> 32) | (int64_t)v3);

        *(float32_t *)&v3 = v4;

        int64_t v5 = (int64_t)v1 * (int64_t)4 +
    (int64_t)a2;
        float32_t v6 = *(float32_t *)&v3;
        int32_t v7;

        *(float32_t *)&v7 =
            0 ? 0.0f < v6 ? 0.0f : v6
              : 0.0f > v6 ? 0.0f : v6;

        float32_t v8 = *(float32_t *)&v7;

        *(float32_t *)(char *)(0x100000000 * (
    int64_t)(int32_t)((v5 & -0x100000000) >> 32)
     | (int64_t)(int32_t)v5) = v8;
    }
}
```

We subsequently apply RetDec to the lifted LLVM IR to recover C-like pseudocode, as shown in Listing 4. While the recovered code is more verbose than the original due to lifting

and IR-level artifacts, it preserves the kernel's semantics. Improving the readability of the decompiled output via additional simplification passes is left to future work.

In addition to ReLU, we successfully applied RetDec to the lifted LLVM IR of 7 out of the remaining 10 CUDA kernels mentioned in Section V-E, recovering C-like pseudocode for each of them. However, RetDec reported segmentation faults when attempting to decompile the remaining three kernels, and we defer addressing these issues to future work.

VII. Discussion

Addressing Testing Failures. As noted in Section VI-B, 6 kernels we lifted failed differential execution with error `Illegal Memory Access`. We hypothesize that these errors stem from load or store operations on nonsensical addresses, likely due to underlying lifting bugs and faulty IR assumptions that lead to invalid pointer arithmetic, incorrect address loading logic, or type confusion in address-generating data.

Since `cuda-gdb` only reports the SASS instruction at which execution halts, we will trace the true source of the error back to the LLVM IR. We already mapped the executed SASS instruction to PTX using NVIDIA Nsight Compute. Specifically, we load the `cubin` file compiled with debugging flags, which allows Nsight Compute to correlate SASS instructions with PTX lines. Next, because `llc` is an open-source backend in the LLVM framework, we plan to modify it to emit LLVM IR that generates specific PTX instructions as comments. This will enable us to establish a correspondence between the relevant SASS region and the lifted IR, thereby narrowing the error scope. We leave the rest as future work.

Porting to Other SM Architectures. The current implementation of `NVLift` targets only the Turing architecture (SM75). Extending the lifter to other SM generations is left for future work. In large part, such a port can reuse the existing SASS recovery and lifting pipeline. This is further aided by the largely incremental nature of NVIDIA GPU ISA and microarchitectural evolution across generations, which often allows substantial portions of instruction semantics and lifting rules to be carried forward with limited adaptation.

Support More Instructions. The current implementation of `NVLift` supports 47 of the 163 Turing SASS instructions, focusing on the high-frequency ones. The implementation of the remaining instructions is also deferred to future work.

Indirect Branches. In the current implementation, `NVLift` supports lifting only SASS *direct* control-flow transfers, while indirect branches are not yet handled due to the static nature of our analysis pipeline. Support for indirect branch instructions, such as `BRX`, is deferred to future work.

Lifting to Other IRs for Decompilation. As an alternative direction, `NVLift` can be extended to lift SASS into other intermediate representations, such as Ghidra's p-code or the microcode used by Hex-Rays. Once lifted, existing decompilation infrastructures in Ghidra or Hex-Rays can be leveraged to recover high-level program representations from the translated code.

VIII. Related Work

SLifter [20] represents an early effort to lift NVIDIA GPU SASS to LLVM IR. However, it is limited to a narrow subset of instructions and lacks a proper type inference engine. Consequently, its capabilities are limited to trivial kernels and do not support complex neural network primitives, such as non-linear activation functions (e.g., ReLU). Furthermore, the framework lacks rigorous validation or testing methodologies for the generated IR. The project repository has remained incomplete and unmaintained for more than 3 years, rendering it insufficient for practical binary analysis tasks.

PLANG [21] translates NVIDIA PTX kernels into LLVM IR and further maps the PTX execution model onto multicore CPUs. However, to the best of our knowledge, only the presentation slides are publicly available; the codebase and full paper are not accessible. In contrast, our work lifts *SASS*, the architecture-specific binary ISA, enabling analysis even when PTX is absent and providing a lower-level reconstruction of GPU kernel semantics.

Recently, JEB also released an experimental decompiler capable of converting CUDA SASS into pseudo-C code [22]. In contrast with `NVLift`, their work does not involve LLVM IR, and the resulting software is a proprietary commercial product.

CuFuzz [23] translates CUDA programs into CPU-compilable code, enabling the use of existing CPU fuzzing tools to test GPU programs. Although CuFuzz shares a similar high-level idea with `NVLift`, it targets CUDA source code rather than GPU binaries, which fundamentally differs from our approach.

IX. Conclusion

In this work, we investigated the semantics of NVIDIA SASS assembly and presented `NVLift`, a framework that lifts SASS into LLVM IR to enable GPU binary analysis. Our implementation supports 47 commonly used SASS instructions on Turing (SM75), covering 88.39% of instruction occurrences in popular CUDA libraries. We lifted 11 representative CUDA kernels and further demonstrated practicality by decompiling the lifted IR into C-like pseudocode using RetDec. Overall, `NVLift` takes an important step toward analysis for deployed NVIDIA GPU binaries.

Acknowledgment

We thank the anonymous reviewers for their insightful feedback. This work was supported in part by the National Science Foundation (NSF) under Award Number CNS-2145744, the Office of Naval Research (ONR) under grant N00014-23-1-2157, and Lockheed Martin. Any opinions, findings, recommendations, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] NVIDIA Corporation, *CUDA Binary Utilities*, NVIDIA, 2025, accessed 2025-12-19. [Online]. Available: https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html

[2] Avast Software, "RetDec: A retargetable machine-code decompiler," https://retdec.com/.

[3] NVIDIA Corporation, *NVVM IR Specification*, https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html, 2025, accessed: 2025-12-19.

[4] LLVM Project, *LLVM Language Reference Manual*, https://llvm.org/docs/LangRef.html, accessed: 2025-12-19.

[5] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[6] Cloudcore. (2020) CUDA Microarchitecture and Instruction Set (3) - Classification of SASS Instructions (in Chinese). Accessed: 2025-06-17. [Online]. Available: https://zhuanlan.zhihu.com/p/163865260

[7] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[8] NVIDIA Corporation, *Parallel Thread Execution ISA Version 8.8*, 2025, accessed: 2025-06-30. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/

[9] ——, *Parallel Thread Execution (PTX) ISA*, NVIDIA, 2025, accessed 2025-12-19. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/#instruction-set

[10] 0xD0GF00D, "Documentsass: Unofficial documentation of nvidia sass instruction sets," https://github.com/0xD0GF00D/DocumentSASS, 2020, accessed: 2025-12-19.

[11] Cloudcore, "Cuda microarchitecture and instruction set (2) - overview of the sass instruction set," https://zhuanlan.zhihu.com/p/161624982, 2020, accessed: 2025-06-30.

[12] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal performance tuning," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 31–43.

[13] D. Yan, "Turingas: Assembler for nvidia volta, turing, and ampere gpus," https://github.com/daadaada/turingas, 2019, accessed: 2025-12-19.

[14] "Maxwell instruction set architecture," https://nintyconservation9619.github.io/Switch%20SDK/Docs-JAP/Documents/Package/contents/SASS/opcodes.htm, NintendoSDK Documentation, 2018, accessed: 2026-01-10.

[15] J. Roels, A. Jacobs, and S. Volckaert, "Cuda, woulda, shoulda: Returning exploits in a sass-y world," in *Proceedings of the 18th European Workshop on Systems Security*, 2025, pp. 40–48.

[16] NVIDIA Corporation, *CUDA-GDB Documentation*, 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-gdb/index.html

[17] W. contributors, "Reaching definition," https://en.wikipedia.org/wiki/Reaching_definition, accessed: 2025-06-30.

[18] ——, "Use-define chain," https://en.wikipedia.org/wiki/Use-define_chain, accessed: 2025-06-30.

[19] llvmlite Project, *llvmlite documentation*, Read the Docs, accessed: 2025-12-28. [Online]. Available: https://llvmlite.readthedocs.io/en/latest/

[20] Cupbop, "Slifter," https://github.com/cupbop/SLifter, 2025, accessed: 2025-12-19.

[21] S. L. Vinod Grover, Andrew Kerr, "Plang: Ptx frontend for llvm," https://llvm.org/devmtg/2009-10/Grover_PLANG.pdf, 2009, accessed: 2025-12-19.

[22] N. Falliere. (2025, August) Reversing nvidia gpu's sass code. Accessed: 2026-01-08. [Online]. Available: https://www.pnfsoftware.com/blog/2025/08/

[23] S. Singh, R. Han, J. Lee, S. Na, Y. Kim, T. Kim, and H. Kim, "Cufuzz: Hardening cuda programs through transformation and fuzzing," *arXiv preprint arXiv:2601.01048*, 2026.

APPENDIX

A. The Construction of Use-Def and Def-Use Chains

The construction of precise Use-Def and Def-Use proceeds through a sequence of steps, denoted RD0 through RD2.

(RD0) Instruction-level Use-Def Marking. For each instruction, we parse and record its *Definition* and *Use*. For example, in the instruction `IADD R1, R2, R3`, `R1` is marked as a definition (Def), while `R2` and `R3` are marked as uses (Use).

(RD1) Dataflow-based Reaching Definition Analysis. To track the source of register and memory values, we implement a forward dataflow analysis. For each basic block B in the Control Flow Graph (CFG), we define the following sets:

- GEN[B]: Definitions generated within B that reach the end of the block.
- KILL[B]: Definitions killed by re-definitions in B.
- IN[B]: Definitions reaching the entry of B, defined as

$$\text{IN}[B] = \bigcup_{P \in \text{pred}(B)} \text{OUT}[P]$$

- OUT[B]: Definitions reaching the exit of B, defined as

$$\text{OUT}[B] = \text{GEN}[B] \cup (\text{IN}[B] - \text{KILL}[B])$$

After pre-computing the GEN and KILL sets for each basic block, we iteratively solve the dataflow equations for IN and OUT over the CFG until convergence. The above idea is similar to that in [17].

(RD2) Chain Construction. We construct instruction-level Use-Def (UD) and Def-Use (DU) chains by refining block-level results to instruction-level granularity through a top-down intra-block traversal. For each instruction $i \in B$, we propagate the block's entry state IN[B] and update it incrementally according to the GEN and KILL sets of all preceding instructions. This allows us to resolve the UD chain for i by mapping each source operand to the set of reaching definitions that target that specific register or memory location. We subsequently derive the DU chain by inverting the UD mapping, associating each definition with the set of its corresponding uses.

B. Supported Instructions

Table II lists the instruction opcodes and modifiers supported by `NVLift`.

C. NVVM IR Intrinsics Functions

Listing 5 showcases example NVVM IR intrinsics used to access GPU special registers.

Listing 5: NVVM IR Intrinsics for Special Registers

```
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.tid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.tid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.ntid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.ctaid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.x()
declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.y()
declare i32 @llvm.nvvm.read.ptx.sreg.nctaid.z()
declare i32 @llvm.nvvm.read.ptx.sreg.warpsize()
```

TABLE II: Opcodes and modifiers for the core instruction set

Opcode	Modifiers
EXIT	–
NOP	–
BRA	–
S2R	–
MOV	–
UMOV	–
LDG	–
LDS	–
LDL	–
ULDC	–
STG	.E
STS	.E
STL	.E
IMAD	.WIDE, .MOV, .IADD, .SHL, .HI, .X, .U32, .S32
IABS	–
ISETP	.AND, .OR, .EQ, .NE, .LT, .LE, .GT, .GE, .FTZ
UISETP	.AND, .OR, .EQ, .NE, .LT, .LE, .GT, .GE, .FTZ
FSETP	.AND, .OR, .EQ, .NE, .LT, .LE, .GT, .GE, .FTZ
IADD	.FTZ
FADD	.FTZ
IADD3	.X
UIADD3	.X
IMNMX	–
FMNMX	–
FFMA	–
LEA	–
LOP3	–
ULOP3	–
PLOP3	–
I2I	–
I2F	–
F2F	–
F2I	.FTZ, .NTZ, .U32, .S32, .TRUNC, .FLOOR, .CEIL
MUFU	–
BMOV	–
BSSY	–
BSYNC	–
SEL	–
FSEL	–
USEL	–
SHF	.L, .R, .W, .C, .U32, .S32, .U64, .S64, .HI
USHF	.L, .R, .W, .C, .U32, .S32, .U64, .S64, .HI
CALL	.REL, .NOINC
RET	–
IMUL	–
FMUL	–
BAR	.SYNC