Lie Low & Stitch: Addressing the Context Window for Low-Level GPU-to-GPU Transpilation

Anonymous ACL submission

Abstract

Cross-architecture GPU transpilation is a pressing portability challenge: NVIDIA's dominance has left the GPU ecosystem heavily vendor-locked at the assembly level, where programs are tightly coupled to proprietary instruction sets. Prior work treats this as end-to-end sequence translation over full programs, an approach that breaks down as program length grows, suffers from weak instruction-level alignment, and fails to generalize across fine-grained ISA patterns.

We introduce a function-level formulation of GPU assembly transpilation. Rather than translating entire programs in a single pass, we decompose source and target assembly into aligned kernels and functions, translate each independently, and reconstruct the full program via a post-translation stitching procedure. This directly addresses the context length limitation, enables tighter cross-ISA alignment, and makes large-scale data construction tractable.

A key challenge is that GPU assembly programs contain compiler-generated metadata that does not localize cleanly to individual functions. We address this with a metadata-aware preprocessing pipeline that supports reliable decomposition and faithful reconstruction. Our approach enables far more realistic benchmark programs to be translated by fitting within the context window, and improves execution-level accuracy by 31.89% over prior work. These results demonstrate that function-level decomposition with structured reconstruction is a more principled and scalable paradigm for GPU assembly transpilation.

1 Introduction

Graphics Processing Units (GPUs) are central to modern high-performance computing, yet GPU software ecosystems remain fragmented across vendors. NVIDIA's CUDA stack dominates GPU programming, while AMD's ROCm stack provides the corresponding ecosystem for AMD hardware (NVIDIA Corporation, 2026b; Advanced Micro Devices, Inc., 2026a). Existing portability tools (Section 2) reduce portability barriers at the source or runtime level, but do not translate optimized low-level GPU assembly across vendor-specific instruction set architectures (ISAs).

Assembly-level translation is critical for preserving performance and portability when source code is unavailable or insufficient. GPU Ocelot explored dynamic CUDA translation, and recent work has shown that language models can perform assembly-to-assembly translation across CPU instruction sets (Diamos et al., 2009; Heakl et al., 2025). GPU assembly translation remains underexplored and more challenging, as it must preserve both program semantics and architecture-specific execution behavior. CASS introduces the first large-scale effort toward cross-architecture GPU transpilation (translating between assembly languages), targeting NVIDIA SASS, NVIDIA's native GPU assembly language, to AMD GPU assembly (Heakl et al., 2026). Performance, however, remains limited by its full-program formulation: treating each assembly file as one sequence-to-sequence example produces long inputs, weak source–target alignment, and incomplete coverage of instruction patterns.

We argue that assembly transpilation should instead be viewed as a structured composition problem over smaller semantic units. We propose function-level splitting, where source and target assembly programs are decomposed into aligned kernels and functions. This exposes the model to shorter, better-aligned examples while preserving local instruction patterns.

One Rodinia file illustrates this bottleneck. The `match-cuda` benchmark progressively optimizes a kernel for matching two sets of 16K 128D points (Ivanov, 2016). Its compiled source assembly contains roughly 126K SASS tokens, far beyond the 32K-token context limit of models such as

Qwen2.5-Coder, while the corresponding ground-truth RDNA3 assembly contains roughly 51K tokens. Full-file translation therefore fails at the input stage: the source assembly cannot be represented in context. After function-level decomposition and metadata-aware stitching, the same benchmark file executes successfully.

Building on this formulation, we develop an expanded data pipeline, incorporate additional sources such as CodeRosetta (TehraniJamsaz et al., 2024), and quantify instruction coverage through mnemonic-level analysis of NVIDIA's ISA. We also address the main reconstruction challenge introduced by function-level splitting: GPU assembly contains compiler-generated metadata that is not cleanly localized to individual functions.

We evaluate this formulation by training **DAFT** (**D**ecomposed **A**ssembly **F**unction-level **T**ranslation) models and testing on CASS-Bench-v2 and Rodinia (Che et al., 2009). Compared with the strongest CASS baseline, DAFT improves execution by **31.89%**, showing that decomposed function-level translation with metadata-aware reconstruction remains effective where full-file training degrades.

We also evaluate our methodology across distinct hardware tiers. Our pipeline targets two specific NVIDIA-to-AMD architecture pairs. We translate from NVIDIA's `sm_89` (Ada Lovelace, consumer-class) to AMD's RDNA3, and from NVIDIA's `sm_90` (Hopper, datacenter-class) to AMD's CDNA3. Crucially, our models operate directly on executable machine assembly rather than PTX (Parallel Thread Execution)—NVIDIA's intermediate, driver-translated virtual instruction set.

2 Related Work

Why assembly-level GPU translation. GPU programs pass through a vendor-specific compilation pipeline before reaching the hardware, producing assembly in which instruction selection, register allocation, predication, memory-access patterns, and control-flow layout are already committed. Translating at this layer is necessary when source code is unavailable (third-party kernels, proprietary libraries, optimized binaries), when compiler-emitted optimizations must be preserved across vendors, and in live-migration scenarios where a running GPU workload must be retargeted to different hardware without re-entering the source-to-assembly compilation pipeline. The dominant vendor stacks, NVIDIA CUDA and AMD ROCm, use distinct compilation pipelines and ISAs (NVIDIA Corporation, 2026b; Advanced Micro Devices, Inc., 2026a), making such retargeting nontrivial. Also, existing portability tools operate above the assembly layer: HIPIFY rewrites CUDA into HIP, ZLUDA provides a PTX-level runtime path for executing CUDA on non-NVIDIA GPUs, and Scale offers a CUDA-compatible compiler stack targeting both vendors (Advanced Micro Devices, Inc., 2026b; Janik, 2026; Spectral Compute, 2026). Earlier systems such as GPU Ocelot explored dynamic compilation for CUDA/PTX programs rather than direct vendor-specific assembly translation (Diamos et al., 2009). These approaches reduce portability barriers at higher abstraction layers but leave compiled, vendor-specific assembly outside their scope.

Assembly translation with language models. Language models can be effective for low-level code translation, including assembly migration across CPU instruction sets. Guaranteed Guess, for example, studies CISC-to-RISC assembly translation on CPUs, combining neural generation with validation to improve functional correctness (Heakl et al., 2025). GPU assembly translation is less explored and is complicated by sparse, difficult-to-align public data produced through vendor-specific compilation pipelines.

Dataset-driven GPU transpilation. A natural step toward cross-vendor GPU assembly translation is constructing aligned datasets pairing source-side and target-side assembly. CASS is the first large-scale effort in this direction, covering CUDA-to-HIP source translation and NVIDIA-to-AMD assembly translation (Heakl et al., 2026). DAFT differs from CASS in three ways. First, CASS treats each assembly file as a single sequence-to-sequence example, producing long inputs that mix functions with global metadata and lack function-level source-target pairing. DAFT instead operates at function granularity and reconstructs full programs through a metadata-aware stitching stage. Second, DAFT contributes a base-mnemonic ISA coverage analysis: we ask which SASS base mnemonics in NVIDIA's published ISA the corpus actually contains, and which are missing. This gives a direct measure of ISA coverage. Third, where CASS fixes a single NVIDIA–AMD architecture pair, DAFT extends the setting to previ-

ously unstudied pairs, including datacenter-class hardware.

3 ISA-Level Coverage Analysis

A model that translates from SASS to AMD assembly can only learn instruction patterns it has seen. If the training corpus misses entire families of SASS instructions, the model has no way to translate them at inference time, regardless of how well the function-level formulation handles the patterns it has seen. Coverage of the source ISA is therefore a precondition for any downstream evaluation: when DAFT succeeds, we want to know that success reflects translation capability over a representative slice of the SASS instruction set; when it fails, we want to know whether the failure is due to translation difficulty or to missing training data.

In this section, we measure source-ISA coverage of our training corpus, instantiated for `sm_89` as a concrete source variant. We organize the analysis around SASS base mnemonics, which correspond directly to instruction families in NVIDIA's published ISA reference and are coarse enough to aggregate across modifiers and operand encodings.

3.1 Anatomy of a SASS Instruction

SASS (**S**treaming **Ass**embly) is the native machine-level assembly representation executed by NVIDIA GPUs. A disassembled SASS instruction typically contains an offset, a symbolic instruction, operands, and hexadecimal encoding fields.

```
/*0160*/ LDGSTS.E.LTC128B.128
         [R7], desc[UR8][R2.64] ;
         /* 0x0000000002077fae */
         /* 0x008fe6000b9a1a08 */
```

The offset field, `/*0160*/`, identifies the instruction location within the kernel. Branch instructions may use these offsets as control-flow targets. The symbolic instruction, `LDGSTS.E.LTC128B.128`, consists of a base mnemonic followed by modifiers. In this example, `LDGSTS` is the base mnemonic, while `E`, `LTC128B`, and `128` refine the operation. The operand field specifies the registers, descriptors, memory locations, predicates, or immediates used by the instruction. The hexadecimal fields correspond to the encoded instruction and compiler-emitted control information.

For coverage analysis, we use the *base mnemonic* as the unit of comparison. This gives us an instruction-family view of SASS: fine-grained enough to distinguish different hardware operations, but coarse enough to aggregate across modifiers, operands, and scheduling-specific encodings. The next subsection describes how we extract these base mnemonics from the training corpus and compare them against the vendor ISA taxonomy.

3.2 Base-Mnemonic Coverage in the Training Corpus

We compile CUDA programs with the standard CUDA toolchain and disassemble the resulting binaries with `nvdisasm`, targeting `sm_89` to match the source-ISA variant used during training. Because disassembler output also contains section headers, labels, register-allocation summaries, source-line comments, and encoding fields, we apply a line-level filter that retains only canonical instruction records (offset, instruction token, operands, terminating semicolon). From each retained instruction we extract both the *raw mnemonic* (the full instruction token) and the *base mnemonic* (the opcode name before the first modifier). We keep both: raw mnemonics expose modifier-level detail useful for inspecting compiler-emitted variants, while base mnemonics provide the corpus-level coverage signal and align with NVIDIA's public instruction-set reference, which organizes SASS at the base-mnemonic level (NVIDIA Corporation, 2026a).

In total, the corpus contains 60,594 disassembled files and 59,153,960 retained instruction records, covering 766 unique raw mnemonics and 125 unique base mnemonics. We map each observed base mnemonic into NVIDIA's Ampere/Ada instruction categories (`floating_point`, `integer`, `conversion`, `movement`, `predicate`, `load_store`, `uniform_datapath`, `texture`, `surface`, `control`, `misc`) (NVIDIA Corporation, 2026a), using this vendor-defined taxonomy so the analysis reflects the architectural organization of the ISA and is easy to compare with future work. An auxiliary `OBS_UNDOC` bucket captures base mnemonics that appear in compiled SASS but are absent from NVIDIA's public listing: we observe two, `ARRIVES` and `F2FP`, which also appear in independently maintained reverse-engineered SASS instruction tables (Herrmann, 2024).

Figure 1 visualizes the `sm_89` SASS ISA tree: the bucket-level overview (left) and drill-downs into the two largest categories, integer (center) and load/store (right). Table 4 in Appendix A.1 reports the exact per-bucket counts. The distribution is consistent with a compute-oriented CUDA corpus:

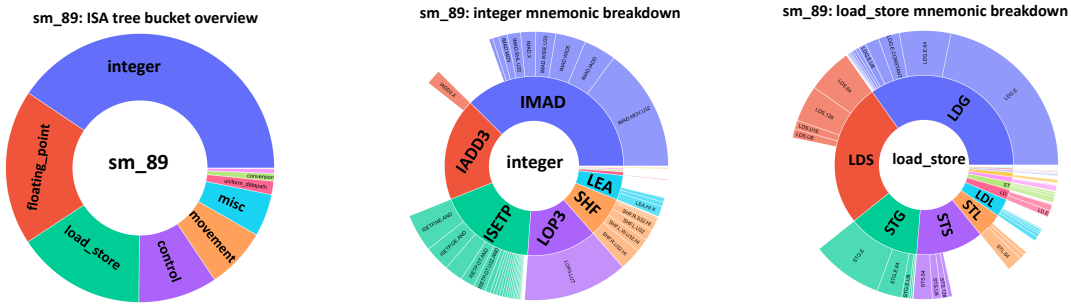

Figure 1: Sunburst visualization of the `sm_89` SASS ISA tree. Left: bucket-level overview. Center: integer bucket drill-down into base and raw mnemonics. Right: load/store drill-down. Sector width reflects observed frequency of the base mnemonic in the training corpus.

arithmetic and memory movement dominate, while texture and surface operations form a small tail.

Of the 169 documented base mnemonics in NVIDIA's Ampere/Ada reference, 123 appear in our corpus and 46 do not. These gaps are not evidence of weak ISA coverage: they correspond to alternate encodings of operations already covered (e.g., wide-immediate `*32I` forms and 2-input arithmetic superseded by fused 3-input variants), texture and graphics-control instructions outside compute workloads, and rare prologue or specialty mnemonics tied to specific intrinsics and data types. We interpret the 123 observed documented mnemonics as strong coverage of the realistic `ptxas` emission space for `sm_89`. Appendix A extends this analysis with an in-depth look at why each of the 46 missing mnemonics is absent.

4 Function-Level Cross-Vendor Assembly Translation

Prior NVIDIA-to-AMD assembly translation processes full files, which mixes executable kernels with metadata and provides weak supervision for long programs. We propose a function-level approach that scales across architectures (Figure 2). By splitting programs into independent executable units and reconstructing the final binary via metadata-aware stitching, we generate shorter, cleaner training examples. This method improves source-target alignment and allows long programs to bypass context window limitations.

We use *function* to denote any callable unit that appears as a separate code symbol in compiled GPU assembly, including both launchable kernels and non-kernel device routines. We use *kernel* in the standard CUDA/HIP sense: a GPU entry point launched by the host or runtime. On AMDGPU/HSA targets, kernels are distinguished from ordinary device functions by descriptor metadata required for dispatch. Thus, kernels form the launchable subclass of functions, whereas non-kernel functions are callable only from kernels or other device functions.

4.1 Function-Level Splitting

We replace full-file training with kernel-level examples to address context length bottlenecks. By removing non-executable metadata and file-level scaffolding from target AMD files, we retain longer programs and focus the model strictly on executable assembly.

This function-level formulation significantly reduces the token footprint of training examples. The median DAFT function unit is roughly $2.1\times$ smaller than a CASS full-file pair, with the average size dropping from 75.0K to 13.2K tokens (Figure 4). This reduction maximizes corpus utility, allowing context-constrained LLMs to dedicate their capacity entirely to instruction translation rather than boilerplate.

During decomposition, we split each target AMD program into executable per-kernel units and a single shared metadata file (Figure 2), aligning source and target kernels by their mangled symbol names (Figure 3). Each kernel unit retains only local executable structures (e.g., instruction streams and local directives), while the metadata file securely stores global material needed for eventual reconstruction. This separation enables the model to learn localized translation without the noise of repeated global metadata.

4.2 Metadata Handling

A key difficulty in function-level splitting is that AMD GPU assembly contains structured, compiler-emitted metadata required for successful compila-

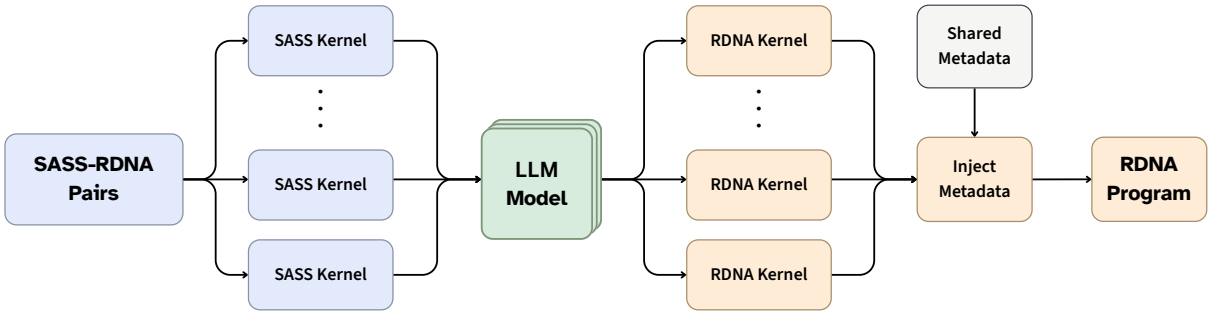

Figure 2: Overview of function-level cross-vendor assembly translation. Full programs are split into kernel-level SASS units, translated independently by the model, and stitched back into a complete AMD assembly program.

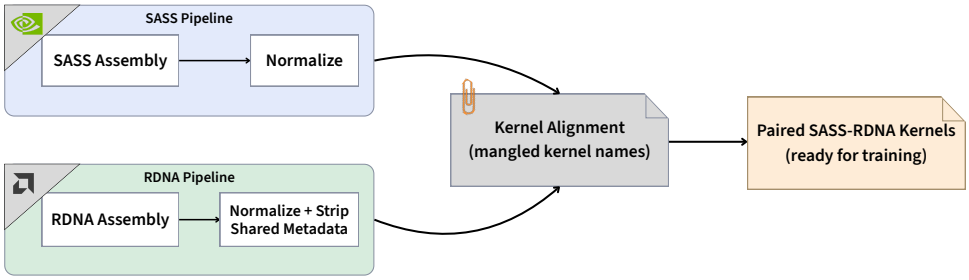

Figure 3: Kernel-level alignment pipeline. SASS and AMD assembly are normalized separately; shared AMD metadata is stripped into a metadata file; and kernels are aligned by mangled names to produce paired training examples.

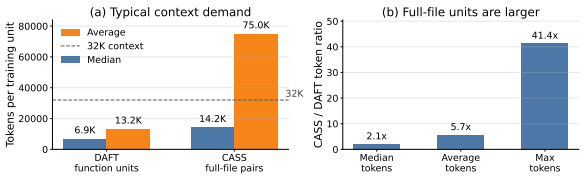

Figure 4: Comparison of typical context demand between DAFT's function units and CASS's full-file pairs. Function-level splitting dramatically reduces the median and average token footprint, maximizing the utility of standard LLM context windows.

Kept in Kernel Body (Model Target)	Moved to Metadata File / Stripped
Instruction stream, `.LBB*` labels	Full `.amdhsa_kernel` descriptor
`.size`, `.Lfunc_end*`	`.amdgpu_metadata` YAML (`metadata_yaml`)
Local `.rodata` / `.p2align`	File `start/end`, `.AMDGPU.csdata`, `.ident`
Symbol directives for the kernel	Compiler comments (`; Kernel info:`, etc.)

Table 1: Metadata split for RDNA3 function-level training (implemented in `AMD-gfx1100`).

tion and execution. This contrasts with most CPU assembly targets, which can be treated primarily as text. Naively including this metadata in every training target bloats sequence length and encourages the generation of brittle boilerplate; removing it entirely makes reconstruction impossible.

To resolve this, we implement a three-stage separation pipeline: **Cleaning Files** → **Extracting kernels** → **Stitching back into full files**.

Kernel Extraction and Metadata Files We split the assembly into the executable kernel body (which serves as the model's target) and a metadata file (which holds file-level metadata). During the cleaning and extraction stages, all file-level prologues, epilogues, and descriptor metadata are stripped from the kernel body and safely stored in the metadata file. Table 1 summarizes what is kept in the kernel body and what is moved to the metadata file or stripped.

Reconstruction Stage At stitch time, the model generates only executable code, bypassing descriptors or YAML metadata. Our automated pipeline rebuilds the full program as follows:

1. **Metadata Injection:** Reference descriptors are inserted into the corresponding translated kernel bodies.

2. **File Concatenation:** The original file header,

translated kernels (with metadata), and file footer are stitched together.

3. **Toolchain Delegation:** We avoid manual branch patching or label fixing by delegating to the LLVM AMDGPU toolchain, which safely resolves cross-function calls and local labels.

This automated reconstruction enables efficient function-level training while preserving end-to-end executability.

5 Methodology and Experiments

In this section, we describe our experimental methodology and report main results. Section 5.1 details the training and inference setup, including models, hardware, data construction, and the optimization recipe. Section 5.2 introduces the evaluation pipeline, benchmarks, and metrics used to assess cross-vendor assembly translation. Section 6 presents results comparing our function-level approach against the strongest existing baseline.

5.1 Experimental Setup

Models. We fine-tune a family of open-source code models under our function-level formulation, collectively referred to as DAFT models. We train Qwen3 (1.7B, 4B, 8B) and Qwen2.5-Coder (1.5B, 3B, 7B) variants to study how capacity interacts with our function-level granularity. Our strongest configuration is built on Qwen2.5-Coder-1.5B with the full optimization-diversification pipeline described below. We compare against CASS (Heakl et al., 2026) using its publicly available checkpoints (1.5B, 3B, 7B) for both the `sm_89` and `sm_80` source-side configurations.

Training data and architecture pairs. SASS can be extracted from compiled binaries with `cuobjdump` and `nvdisasm` but cannot be reassembled or executed (NVIDIA Corporation, 2026a), fixing the translation direction as NVIDIA → AMD. Our function-level dataset spans two NVIDIA–AMD pairs covering consumer-class and datacenter-class hardware: `sm_89` (RTX 4090) → RDNA3 (RX 7900 XTX), and `sm_90` (H100) → CDNA3 (MI300X).

Optimization diversification. A central design choice in our data pipeline is to compile each program under multiple optimization levels rather than a single setting. For each source program, we compile the CUDA and HIP versions under `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, and `-Oz`, keeping every successful compilation. The same source file therefore appears in multiple aligned SASS–target pairs, each reflecting a different optimization profile. Different levels produce substantially different instruction selection, register allocation, and control-flow shapes, exposing the model to a broader space of realistic assembly patterns than any single level. We quantify the effect of this design choice in Section 6.

Training recipe. All models are fully fine-tuned on 8× NVIDIA H100 GPUs using DeepSpeed ZeRO-2 with bfloat16 precision and FlashAttention-2. We use AdamW with a learning rate of 5×10^{-5}, cosine schedule with 5% warmup, weight decay of 0.1, per-device batch size of 1 with sequence packing, and a context length of 32k tokens. Single-optimization runs train for 5 epochs; the optimization-diversified Qwen2.5-Coder-1.5B run trains for 2 epochs, because the expanded optimization grid (six levels rather than four) increases the number of aligned SASS–target pairs per source program, yielding a comparable total token budget at lower wall-clock cost.

5.2 Kernel Alignment, Reconstruction, and Evaluation Pipeline

Function-wise reconstruction relies on the metadata-aware stitching pipeline from Section 4, injecting target-side metadata from the corresponding reference AMD assembly to account for program-level scaffolding. Conversely, full-file translation is evaluated without metadata injection. Appendix B details the alignment, extraction, and reconstruction procedures.

The completed target assembly is submitted to an automated execution server for the target architecture. We report compilation rate, the fraction of benchmark files that compile under the AMD toolchain, and execution rate, the fraction that compile and complete within a 120-second timeout without a runtime failure. Execution is therefore stricter than compilation, but does not require output matching.

5.3 Benchmarks

We evaluate DAFT on two benchmark suites: Cass-Bench-v2 and Rodinia. Cass-Bench-v2, introduced with CASS for NVIDIA-to-AMD GPU transpilation, serves as our baseline evaluation setting

		Function-level inference				Full-file inference			
		Cass-Bench-v2		Rodinia		Cass-Bench-v2		Rodinia	
	Model	Comp. %	Exec. %	Comp. %	Exec. %	Comp. %	Exec. %	Comp. %	Exec. %
Closed	Gemini 3.1 Flash Lite	14.84	14.84	6.67	0.56	1.17	1.17	0.69	0.69
	Gemini 3.1 Pro	21.48	19.92	23.33	3.33	6.64	6.64	21.38	**21.38**
	GPT-5.5	19.14	17.58	5.00	1.11	2.34	1.95	8.28	7.59
	Claude Opus 4.7	33.98	29.30	27.78	23.89	59.38	47.27	**44.83**	**44.83**
Open	Qwen2.5-Coder-3B	45.70	33.20	28.89	3.89	10.16	8.20	4.64	0.00
	Qwen3-1.7B	64.45	**49.22**	32.22	6.11	1.56	1.56	0.69	0.69
	Qwen3.5-4B	12.50	9.77	4.44	0.56	7.03	6.25	5.67	0.00
	Qwen3.6-27B	6.25	6.25	3.33	2.22	6.25	4.69	2.58	0.00
CASS	CASS-sm4090-1.5B	0.00	0.00	0.56	0.56	67.75	57.18	14.49	7.97
	CASS-sm4090-3B	1.95	1.17	0.00	0.00	99.19	69.38	30.43	5.80
	CASS-smA100-1.5B	0.00	0.00	0.00	0.00	**99.73**	69.65	0.00	0.00
	CASS-smA100-3B	0.39	0.39	1.11	0.56	**99.73**	**69.92**	0.00	0.00
	CASS-smA100-7B	0.00	0.00	0.00	0.00	**99.73**	69.65	0.72	0.00
DAFT (ours)	DAFT-Qwen3-1.7B	68.36	33.20	33.33	33.33	—	—	—	—
	DAFT-Qwen3-4B	70.70	35.16	**39.86**	**39.86**	—	—	—	—
	DAFT-Qwen3-8B	69.92	34.38	**39.86**	**39.86**	—	—	—	—
	DAFT-Qwen2.5-Coder-1.5B	70.31	33.98	25.36	25.36	—	—	—	—
	DAFT-Qwen2.5-Coder-1.5B†	**73.05**	40.23	27.54	27.54	—	—	—	—
	DAFT-Qwen2.5-Coder-3B	68.36	33.20	33.43	33.43	—	—	—	—
	DAFT-Qwen2.5-Coder-7B	71.48	35.55	38.41	38.41	—	—	—	—

Table 2: Compilation and execution rates (%) on Cass-Bench-v2 (340 functions from 256 files) and Rodinia (137 files), under function-level and file-level inference for SASS–>RDNA3. Closed-source and open-weight LLMs are evaluated zero-shot; CASS and DAFT are fine-tuned. Bold marks the per-column maximum. "—" indicates the model was not run in that regime (zero-shot LLMs with bounded context were not evaluated for full-file when input exceeded their window; DAFT models are designed for function-level inference and were not evaluated on full-file). † denotes the optimization-diversified variant (six `-O` levels; cf. Section 5.1).

(Heakl et al., 2026; Che et al., 2009). However, we extend our evaluation beyond Cass-Bench-v2 because transpilation-specific benchmarks may over-represent short or regular kernels and fail to capture the diversity of real GPU programs. We present more details regarding these benchmark kernel sizes in Appendix B.

Rodinia is a standard heterogeneous GPU benchmark suite. It covers a broad range of computational patterns inspired by the Berkeley dwarf taxonomy, including implementations across models like CUDA, OpenCL, and OpenMP. In our setting, Rodinia is valuable because it stresses the models on larger, more structurally diverse GPU programs with varied arithmetic, memory-access, and control-flow behavior.

For both suites, we evaluate only programs that exercise GPU kernels, compile successfully in the source toolchain, and admit a usable AMD-side reference when required. After filtering, the evaluation set contains 256 Cass-Bench-v2 programs and 137 Rodinia programs. This filtering matters because treating non-kernel programs or failed source builds as translation targets would distort the measured performance of an assembly-level transpiler. Together, the two benchmarks provide a graded evaluation: Cass-Bench-v2 measures performance in the closest setting to prior NVIDIA-to-AMD transpilation work, while Rodinia tests whether the translator generalizes to broader GPU workloads.

6 Results

6.1 SASS-SM89 to RDNA3

Table 2 reports the SASS-sm89 → RDNA3 results for SOTA closed- and open-source models, CASS models, and our fine-tuned DAFT models (Google, 2026a,b; OpenAI, 2026; Anthropic, 2026; Hui et al., 2024; Qwen Team, 2025, 2026a,b; Heakl et al., 2026). Across both benchmarks, the function-level regime dominates for any model that was actually trained or competent enough to operate on individual functions: fine-tuned DAFT models reach 70–73% compilation and 33–40% execution on Cass-Bench-v2, and 33–38% on Rodinia, where every CASS checkpoint collapses to below 1%. The optimization-diversified DAFT-Qwen2.5-

Coder-1.5B configuration achieves the strongest Cass-Bench-v2 compilation rate (73.05%) and execution rate (40.23%) at the 1.5B scale, supporting the optimization-diversification argument from Section 5.1: a 6.25-point absolute execution gain over the single-optimization variant of the same model. The opposite picture emerges in the file-level regime: CASS checkpoints near-saturate Cass-Bench-v2 (up to 99.73% compilation, 69.92% execution), but collapse to 0–5.80% execution on Rodinia, whose programs are longer and structurally further from the CASS training distribution. The Cass-Bench-v2 numbers are unsurprising: CASS was co-optimized for this benchmark and trained on similarly short, regular GPU kernels, so file-level performance there partly reflects distributional alignment rather than generalization. DAFT outperforms CASS on Rodinia, the first realistic classic GPU benchmark in this comparison. This asymmetry is the main empirical motivation for function-level training: full-file translation can memorize a curated distribution but fails to generalize, while function-level translation maintains performance across both benchmarks. Zero-shot baselines, whether closed-source or open-weight, lag behind fine-tuned approaches on Cass-Bench-v2 but remain competitive on Rodinia file-level, where long full-program translation is the binding constraint (See Appendix C), and Opus 4.7 in particular is a much larger and more expensive model than the smaller models we fine-tune.

6.2 SASS-SM90 to CDNA3

We also evaluate SASS-to-CDNA3 translation to test whether the function-level formulation transfers beyond RDNA3. Table 3 reports function-level results for three representative models. DAFT-Qwen2.5-Coder-1.5B trained on `sm90`–CDNA3 pairs reaches 67.06% compilation and 9.41% correct output on Cass-Bench-v2, and 46.38% compilation with 1.45% correct output on Rodinia, while zero-shot Qwen baselines are near zero on both. CDNA3 is harder than RDNA3 in our current setup, but function-level DAFT training transfers substantially better than general-purpose open models.

Model	Benchmark	Comp. %	Exec. %
Qwen2.5-Coder-3B	Cass-Bench-v2	0.00	0.00
Qwen3-1.7B	Cass-Bench-v2	1.18	0.39
DAFT-1.5B-sm90	Cass-Bench-v2	**67.06**	**9.41**
Qwen2.5-Coder-3B	Rodinia	0.00	0.00
Qwen3-1.7B	Rodinia	1.45	0.00
DAFT-1.5B-sm90	Rodinia	**46.38**	**1.45**

Table 3: Function-level SASS-SM90 to CDNA3 results. Comp. is the fraction of generated programs that compile after reconstruction; Exec. is the fraction that both run and produce output. DAFT is compared against zero-shot open-code baselines on Cass-Bench-v2 and Rodinia.

7 Conclusion

We introduced a function-level formulation for cross-vendor GPU assembly transpilation that replaces brittle full-file sequence-to-sequence translation with a strip-split-stitch paradigm. Rather than treating entire assembly files as single training examples, we decompose NVIDIA SASS and AMD assembly into aligned executable kernel units, separate compiler-generated metadata into a dedicated reconstruction pipeline. This reduces the average training token footprint while exposing models to a broader and better-aligned space of instruction patterns. We train a set of DAFT models under this formulation, which improve execution accuracy by 31.89% over the strongest CASS baseline and, critically, achieve 39.86% execution on Rodinia where as CASS checkpoints collapse below 8%, demonstrating that prior strong numbers on Cass-Bench-v2 reflect distributional memorization rather than generalization. The formulation also transfers to the harder sm90 to CDNA3 datacenter setting, where zero-shot open models remain near-ineffective. These results show that progress in GPU assembly transpilation depends not primarily on model scale, but on training granularity, kernel alignment, and metadata-aware reconstruction, providing a practical foundation for vendor-agnostic binary portability.

8 Limitations

Our function-wise evaluation uses target-side metadata injection during reconstruction: each generated kernel body is stitched back into a complete AMD assembly file using metadata extracted from the corresponding reference AMD assembly. We make this choice because AMD GPU metadata is partly program-level rather than function-local descriptors, resource annotations, and file-level scaffolding depend on the complete compiled module and are not always attributable to a single translated function. Function-wise results should therefore be interpreted as measuring kernel-body translation

under reference metadata reconstruction, not as an end-to-end solution for generating all target-side scaffolding. A natural future direction is a separate composition model that predicts or reconstructs the required program-level metadata from the set of translated functions.

Output validation on Rodinia is limited. Rodinia outputs in our setting are not reliably deterministic across runs, so we do not use exact output matching as an automatic correctness criterion; manual inspection shows that programs that execute successfully produce outputs consistent with expected behavior. We therefore report compilation and execution rates, where execution means the translated program compiles and completes without a runtime failure. These metrics capture functional viability but are weaker than exhaustive semantic validation.

Our alignment procedure favors high-confidence function and kernel matches, which avoids noisy supervision from mismatched or one-sided symbols but also excludes programs whose source and target assembly cannot be aligned reliably. Such exclusions may bias the training data toward cleaner compilation patterns and away from cases involving aggressive optimization, helper-function elimination, or complex cross-file structure. More robust alignment methods could recover additional training examples.

Finally, DAFT remains data-dependent. Instructions and idioms rare or absent in the training corpus are harder to translate reliably. Our ISA coverage analysis shows broad coverage of common compute-oriented SASS patterns, but rare instruction classes, unusual intrinsics, and specialized datapath behavior remain likely failure modes.

Use of Language Models

Large language models were used in a limited capacity to assist with minor editing and polishing of the manuscript, including improvements to clarity and grammar. All technical content, experimental design, results, and conclusions were produced, verified, and finalized by the authors.

References

Advanced Micro Devices, Inc. 2026a. HIP Documentation. https://rocm.docs.amd.com/projects/HIP/en/latest/.

Advanced Micro Devices, Inc. 2026b. HIPIFY documentation. https://rocm.docs.amd.com/projects/HIPIFY/en/latest/.

Anthropic. 2026. Introducing Claude Opus 4.7. https://www.anthropic.com/news/claude-opus-4-7.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*.

Gregory Diamos, Andrew Kerr, and Sudhakar Yalamanchili. 2009. GPU Ocelot: A dynamic compilation framework for PTX. https://github.com/gtcasl/gpuocelot.

GNU Project. GNU Binutils: c++filt. https://sourceware.org/binutils/docs/binutils/c_002b_002bfilt.html.

Google. 2026a. Gemini 3.1 Flash-Lite: Model card. https://deepmind.google/models/model-cards/gemini-3-1-flash-lite/.

Google. 2026b. Gemini 3.1 Pro: Model card. https://deepmind.google/models/model-cards/gemini-3-1-pro/.

Ahmed Heakl, Sarim Hashmi, Chaimaa Abi, Celine Lee, and Abdulrahman Mahmoud. 2025. Guaranteed guess: A language modeling approach for CISC-to-RISC transpilation with testing guarantees. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 24474–24488, Suzhou, China. Association for Computational Linguistics.

Ahmed Heakl, Gustavo Bertolo Stahl, Sarim Hashmi, Seung Hun Eddie Han, Mukul Ranjan, Arina Kharlamova, Salman Khan, and Abdulrahman Mahmoud. 2026. Cass: Nvidia to amd transpilation with data, models, and benchmark. *Preprint*, arXiv:2505.16968.

Fabian Herrmann. 2024. NVIDIA SASS instruction history. https://gist.github.com/herrmann/f721da109e0c5c7c34c847ff2cf3da1e.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*.

Itanium C++ ABI Working Group. 2012. Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/.

Ivan R. Ivanov. 2016. rodinia. https://github.com/ivanradanov/rodinia.

Andrzej Janik. 2026. ZLUDA: CUDA on non-NVIDIA GPUs. https://github.com/vosen/ZLUDA.

NVIDIA Corporation. 2026a. CUDA Binary Utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

NVIDIA Corporation. 2026b. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

OpenAI. 2026. Introducing GPT-5.5. https://openai.com/index/introducing-gpt-5-5/.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Qwen Team. 2026a. Qwen3.5: Towards native multimodal agents.

Qwen Team. 2026b. Qwen3.6-27B: Flagship-level coding in a 27B dense model.

Spectral Compute. 2026. SCALE: Cuda-compatible gpu programming toolkit. https://docs.scale-lang.com/stable/.

Ali TehraniJamsaz, Arijit Bhattacharjee, Le Chen, Nesreen K Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. 2024. Coderosetta: Pushing the boundaries of unsupervised code translation for parallel programming. In *Proceedings of the 38th International Conference on Neural Information Processing Systems*.

A Extended ISA Coverage Analysis

This appendix extends the ISA-level coverage analysis of Section 3. Appendix A.1 reports the exact per-bucket instruction counts that underlie the distribution visualized in Figure 1. Appendix A.2 then accounts for the 46 documented `sm_89` base mnemonics that are absent from the corpus, grouping them by their likely mechanism of absence.

A.1 Per-Bucket Instruction Counts

Table 4 reports the exact instruction counts per bucket, summarizing the frequency distribution visualized in Figure 1 (left panel). Buckets follow NVIDIA's Ampere/Ada instruction categories; `OBS_UNDOC` contains observed but undocumented base mnemonics.

Bucket	Instruction count
integer	23,999,966
floating_point	11,062,834
load_store	9,216,054
control	5,632,869
movement	4,331,695
misc	3,012,029
uniform_datapath	964,559
conversion	543,636
predicate	332,119
OBS_UNDOC	49,780
texture	3,375
surface	266

Table 4: Bucket-level distribution of SASS instructions in the training corpus.

A.2 Accounting for the Unobserved ISA

The 46 documented base mnemonics absent from our corpus are not missing for a single reason. They are also not a deprecation artifact: NVIDIA's Ampere/Ada listing already reflects the post-Volta instruction set, and the only mnemonics removed in the Ampere-to-Ada transition are `R2B` and `RTT`, neither of which appears in our missing set (Hermann, 2024). Thus, the unobserved mnemonics are part of the documented `sm_89` ISA, but were not emitted by `ptxas` for our CUDA corpus.

We group the 46 missing mnemonics by their likely mechanism of absence. The groups are disjoint and sum to 46, as shown in Figure 5. This partition helps distinguish true gaps in operation coverage from cases where the same underlying computation appears under a different encoding.

Alternate 32-bit immediate encoding (12). This group contains `FADD32I`, `FFMA32I`, `FMUL32I`,

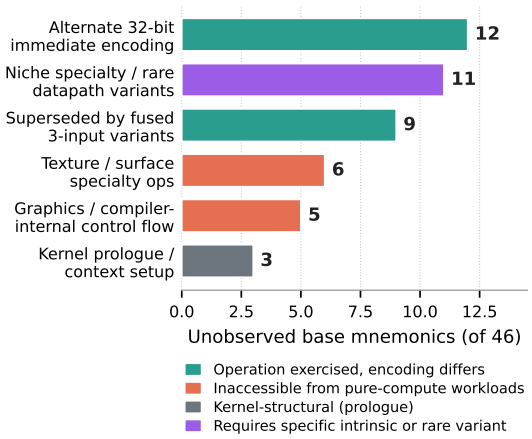

Figure 5: The 46 documented Ampere/Ada base mnemonics absent from our `sm_89` corpus, grouped by mechanism of absence. The two largest groups correspond to operations exercised in the corpus under different encodings.

`HADD2_32I`, `HFMA2_32I`, `HMUL2_32I`, `IADD32I`, `IMUL32I`, `ISCADD32I`, `LOP32I`, `MOV32I`, and `ULOP32I`. These are wide-immediate forms of operations already present in the corpus. For the small constants common in compute kernels, such as loop bounds, offsets, and masks, `ptxas` typically uses compact encodings instead. The operation is therefore covered, but the `*32I` encoding is not.

Superseded by fused 3-input variants (9). This group contains `IADD`, `IMUL`, `ISCADD`, `LOP`, `SHL`, `SHR`, `ULOP`, `USHL`, and `USHR`. Since Volta, `ptxas` commonly emits fused 3-input variants such as `IADD3`, `IMAD`, `LEA`, `LOP3`, and `SHF`, all of which appear in our corpus. The older 2-input forms remain in the documented ISA (NVIDIA Corporation, 2026a), but are rarely selected for CUDA code generation.

Texture and surface specialty operations (6). This group contains `TLD4`, `TMML`, `TXD`, `TXQ`, `SUATOM`, and `SURED`. These instructions are tied to specialized texture and surface paths, such as gather-style texture fetches, mipmap queries, and surface atomics. They are not expected to appear frequently in a compute-oriented CUDA corpus. The more general texture instructions `TEX` and `TLD` are observed.

Graphics or compiler-internal control flow (5). This group contains `JMP`, `JMX`, `JMXU`, `KILL`, and `RPCMOV`. For CUDA control flow, including switch-like patterns, `ptxas` generally emits relative branches such as `BRA` or indirect branches such as `BRX`. By contrast, `KILL` is associated with graphics-shader termination, and `RPCMOV` is tied to specialized call or return-control patterns not normally exposed by CUDA C++.

Kernel prologue and rare specialty instructions (14). The remaining 14 missing mnemonics fall into two thin tails. Three, `GETLMEMBASE`, `SETLMEMBASE`, and `SETCTAID`, are compiler-generated prologue instructions that appear only under specific setup conditions such as local-memory allocation or explicit CTA-ID handling. The other eleven are specialty mnemonics tied to particular intrinsics, data types, or uniform-datapath patterns, including `DMMA` for FP64 tensor-core code, `IDP4A` for integer dot-product intrinsics, and several `U*` variants for uncommon uniform-datapath usage. Both groups are absent because they require code patterns the corpus does not exercise at scale.

B Evaluation and Reconstruction Details

For function-wise inference, the model translates each SASS kernel/function independently. The generated AMD kernel bodies are then stitched into a complete target assembly file before compilation and execution. Since AMD GPU metadata is partly program-level rather than function-local, this reconstruction step copies target-side metadata from the corresponding reference AMD assembly. Full-file inference does not use metadata injection: the model directly generates the complete AMD assembly file that is passed to the execution pipeline. The reconstruction pipeline first aligns source and target kernels using high-confidence name matches. We demangle kernel names with `c++filt` before alignment (GNU Project), strip shared target-side metadata into a metadata file, translate the executable bodies, and then reinsert the reference descriptors and file-level scaffolding during stitching. As a sanity check, the same stitching procedure succeeds on the ground-truth target assembly files, so failures on model outputs primarily reflect malformed or incompatible translated bodies rather than the stitching procedure itself.

The completed target assembly is submitted to an automated execution server. The RDNA3 pipeline runs on an AMD Radeon RX 7900 XTX system using ROCm/toolchain version 6.3.2. The CDNA3 pipeline, when used, runs on a separate AMD Instinct MI300X system using ROCm/toolchain version 7.2.0. In both cases, the server compiles the translated assembly, executes the resulting program, and records whether execution completes within the 120-second timeout.

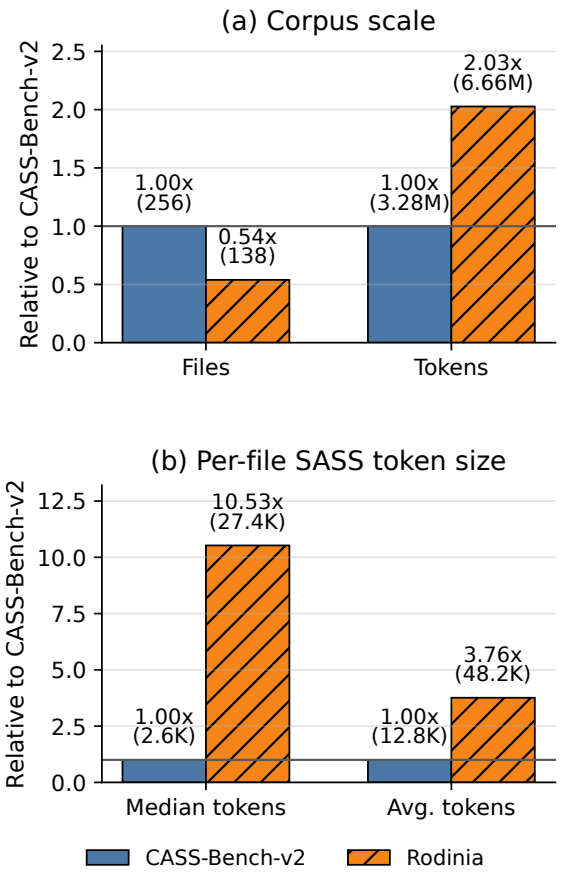

Figure 6: Corpus scale and per-file token size comparison. While Rodinia contains fewer total files than Cass-Bench-v2, its median file size is approximately 10.5× larger, presenting a stricter context and generalization challenge.

C Benchmark Size Comparison

Token statistics (Fig 6) reveal that Rodinia stresses substantially larger translation instances than Cass-Bench-v2. While the filtered Rodinia set contains fewer files, its median file size is roughly 10.5× larger than that of Cass-Bench-v2. This disparity reinforces Rodinia's utility as a generalization benchmark: Cass-Bench-v2 enables direct comparison with prior NVIDIA-to-AMD transpilation work, while Rodinia better captures the scale and structural diversity of standard heterogeneous workloads.

D Kernel Alignment Mismatch Patterns

A non-trivial alignment challenge is that source and target symbol names are often mangled C++ identifiers. In the main evaluation pipeline, we demangle all kernel names with `c++filt` before attempting alignment (GNU Project). This converts raw symbol-table entries into human-readable signatures that can be compared across vendors. Even after demangling, however, names do not always match cleanly. We identify three recurring mismatch patterns.

Type-name divergence from imperfect HIPIFY translation. CUDA and HIP maintain parallel but distinct random-number-state type hierarchies. A kernel whose CUDA source passes a `curandStateXORWOW*` argument is compiled by `nvcc` with the type name encoded in the mangled symbol. The HIPIFY-translated HIP version may substitute the generic `hiprandState*` instead of the architecturally corresponding `hiprandStateXORWOW*`, producing a different demangled signature even when the function name is identical. For example:

Listing 1: Type-name divergence: `curandStateXORWOW*` vs. `hiprandState*`.

```
SASS (mangled): _ZN6curfil3gpu12setup_
    ↪ kernelEiP17curandStateXORWOW
SASS (demangled): curfil::gpu::setup_kernel(int,
    ↪  curandStateXORWOW*)

RDNA (mangled): _ZN6curfil3gpu12setup_
    ↪ kernelEiP12hiprandState
RDNA (demangled): curfil::gpu::setup_kernel(int,
    ↪  hiprandState*)
```

The tokens `curandStateXORWOW` and `hiprandState` are encoded directly in the Itanium ABI mangled name as length-prefixed identifiers (Itanium C++ ABI Working Group, 2012). A structural mismatch in HIPIFY's type mapping therefore propagates into the symbol table and breaks naive name-based pairing.

Linkage divergence between `nvcc` and `amdclang++`. Under the Itanium C++ ABI (Itanium C++ ABI Working Group, 2012), symbols with internal linkage, such as symbols declared `static` at file scope or placed in an anonymous namespace, receive an `L` infix directly after the `_Z` prefix, yielding `_ZL...`. Externally visible symbols instead use the plain `_Z...` prefix. We observe cases where `nvcc` emits a kernel with external linkage while `amdclang++` emits the semantically identical kernel with internal linkage:

Listing 2: Linkage divergence: external `_Z` vs. internal `_ZL`.

```
SASS (mangled): _Z18prefix_scan_devicePfS_i
SASS (demangled): prefix_scan_device(float*,
    ↪ float*, int)
```

```
RDNA (mangled): _ZL18prefix_scan_devicePfS_i
RDNA (demangled): prefix_scan_device(float*,
    ↪ float*, int)
```

The demangled signatures are identical, but the mangled names differ by a single character. This appears to be a toolchain artifact. In particular, `nvcc`, especially when relocatable device code is enabled, may expose device functions across translation units and emit them as globally visible symbols. By contrast, the HIP toolchain under `amdclang++`, when compiled without `-fgpu-rdc`, may compile device code in a translation unit as a single module, allowing locally referenced helper functions to be assigned internal linkage by the LLVM backend.

Asymmetric elimination. Some kernels present in SASS have no counterpart in the RDNA3/CDNA3 binary. The most common cause is aggressive dead-code elimination by the LLVM optimizer invoked through `amdclang++`: a `__device__` helper, or even a `__global__` kernel that is not reachable from any callable entry point in the HIP translation, can be removed at link time. By contrast, `nvcc` tends to be more conservative and may retain such symbols in the CUBIN. These one-sided entries cannot be paired reliably and are excluded rather than forcing noisy supervision.

The downside of kernel-level splitting is that some files do not admit a perfect one-to-one kernel mapping, for example when one source-side kernel corresponds to multiple target-side fragments. The alignment pipeline therefore favors exact, high-confidence kernel alignments and excludes problematic cases rather than forcing noisy supervision.

E Licenses

The CASS dataset and model checkpoints are released under the Apache 2.0 license. Rodinia is a publicly available academic benchmark suite. CUDA toolchains is proprietary but freely available for research use whereas the ROCm software stack is open source available under the MIT license. Our dataset and code will be released under Apache 2.0.