CASS: Nvidia to AMD Transpilation with Data, Models, and Benchmark

Anonymous ACL submission

Abstract

Cross-architecture GPU code transpilation is essential for unlocking low-level hardware portability, yet no scalable solution exists. We introduce CASS, the first dataset and model suite for source- and assembly-level GPU translation (CUDA ↔ HIP, SASS ↔ RDNA3). CASS contains 60k verified host-device code pairs, enabling learning-based translation across both ISA and runtime boundaries. We generate each sample using our automated pipeline that scrapes, translates, compiles, and aligns GPU programs across vendor stacks. Leveraging CASS, we train a suite of domain-specific translation models that achieve 88.2% accuracy on CUDA→HIP and 69.1% on SASS→RDNA3, outperforming commercial baselines including GPT-5.1, Claude-4.5, and Hipify by wide margins. Generated code matches native performance in 95% of cases, preserving both *runtime and memory behavior*. To support rigorous evaluation, we introduce CASS-Bench, a curated benchmark spanning *18 GPU domains* with ground-truth execution. All data, models, and evaluation tools will be released as open source to support progress in GPU compiler tooling, binary compatibility, and LLM-guided code translation.

1 Introduction

Graphics Processing Units (GPUs) are foundational to modern machine learning and scientific computing workloads due to their high-throughput parallelism. Nvidia's Compute Unified Device Architecture (CUDA) (Harris, 2024) has become the dominant programming model for GPU acceleration, but its tight coupling to proprietary hardware introduces severe vendor lock-in: CUDA code cannot run on non-Nvidia GPUs due to incompatible instruction set architectures (ISAs) (NVIDIA Corporation, 2021). As a result, organizations with large CUDA-based codebases face steep engineering costs when migrating to alternative platforms. Meanwhile, AMD GPUs, offering potential favorable performance-per-dollar (AMD, 2024a; Tom's Hardware, 2025), are increasingly adopted across both data centers and consumer devices (Financial Times, 2024), creating a growing need to execute legacy CUDA programs on AMD hardware without full rewrites in software (Janik, 2024).

In response, AMD introduced the Heterogeneous-computing Interface for Portability (HIP) (AMD, 2024b), a C++ GPU API built into the Radeon Open Compute platforM (ROCm) stack (Advanced Micro Devices (AMD), 2024), designed to mirror CUDA's functionality while supporting cross-platform development. HIP enables a unified codebase for both Nvidia and AMD GPUs. Tools like HIPIFY (Advanced Micro Devices, Inc., 2025), a static translator, assist migration by converting CUDA-specific constructs into their HIP equivalents, streamlining adoption of the ROCm stack. However, HIPIFY only operates at the source level and cannot execute precompiled CUDA binaries. Furthermore, it exhibits a high failure rate when converting CUDA programs, highlighting the need for more reliable and lower-level transpilation approaches (Zahid et al., 2024).

Translating GPU assembly across vendors is hindered by divergent ISAs and compilation pipelines. Nvidia employs a proprietary toolchain centered on `nvcc`, producing PTX and low-level SASS (Harris, 2024), while AMD uses GCN/RDNA architectures compiled via the open-source ROCm stack using `hipcc` (Advanced Micro Devices (AMD), 2024) (Figure 2 provides a detailed breakdown of both stacks). Bridging this gap at the assembly level is critical for democratizing the hardware computing landscape, transfer of hardware-specific optimizations across vendors, and enabling automation beyond source-level rewrites, especially for legacy CUDA codebases rich in low-level tuning. Our work introduces the first foundation for Nvidia-to-AMD assembly and source translation, focusing on

correctness and alignment. While not optimization-aware yet, it paves the way for future systems that preserve and adapt performance-critical patterns across GPU backends.

To address the lack of cross-architecture GPU translation datasets, we introduce CASS (CUDA-AMD ASsembly and Source Mapping), a large-scale corpus of 60k semantically aligned CUDA-HIP source pairs and their corresponding host (CPU-x86 ISA) and device (GPU) assemblies for Nvidia (SASS) and AMD (RDNA3) platforms. Each sample comprises functionally equivalent low-level code across vendors, verified through successful compilation and execution, enabling instruction-level analysis across execution boundaries. Unlike generic code corpora like The Stack (Lozhkov et al., 2024a), which lack GPU-aligned and compilable content, CASS provides complete source and binary representations across both GPU compute stacks. To construct CASS, we developed a fully open-source pipeline that scrapes, synthesizes, translates (via HIPIFY (Advanced Micro Devices, Inc., 2025)), compiles, and aligns GPU code. We evaluate CASS along two dimensions: (1) instruction coverage, capturing diverse SASS and RDNA3 opcodes; and (2) domain coverage, spanning real-world compute kernels from ML, graphics, and HPC. CASS is the first dataset to enable source- and assembly-level translation research for GPU architectures.

To validate the utility of our dataset, we introduce CASS-Bench, the first benchmark tailored to cross-architecture GPU transpilation. It spans 18 diverse GPU domains with execution-verified source and assembly pairs, providing a standardized testbed for future work in low-level translation and performance-aware code generation. Building on this benchmark, we present the CASS model family, a suite of domain-specific large language models fine-tuned for both source- and assembly-level GPU code translation. These models are trained on our curated corpus and demonstrate significant improvements over SoTA proprietary systems such as GPT-5.1 (Hurst et al., 2024), Claude-4.5 (Anthropic, 2025a), and traditional tools like HIPIFY (Advanced Micro Devices, Inc., 2025), achieving 88.2% source and 69.1% assembly translation accuracy. Our contributions are as follows:

- **CASS Dataset Pipeline.** We designed a scalable pipeline for scraping, synthesizing, translating, and compiling CUDA/HIP code into aligned host and device assemblies across Nvidia and AMD GPUs.
- **CASS Dataset.** We introduce CASS, the first large-scale dataset for GPU transpilation, containing 60k semantically aligned Nvidia $\leftrightarrow$ AMD pairs at both the source (CUDA $\leftrightarrow$ HIP) and assembly levels (SASS $\leftrightarrow$ RDNA3), covering 18 real-world GPU domains.
- **CASS-Bench.** We contribute the first evaluation benchmark for cross-architecture GPU translation, with 369 curated tasks across 18 domains, including functionally verified outputs and aligned CUDA/HIP source and SASS/RDNA3 assembly.
- **CASS Models.** We release domain-specialized CASS LLMs trained for cross-architecture code translation. Our 7B model achieves 88% source and 69% assembly accuracy, outperforming GPT-5.1 and Claude by 22.2% on CASS-Bench. Crucially, 95% of translated assemblies preserve execution runtime and memory compared to native compilation, confirming semantic and performance fidelity.

2 Related Works

Translating from Nvidia to AMD. The fragmentation of GPU software ecosystems has driven the need for robust CUDA-to-HIP translation tools. HIPIFY (AMD ROCm Documentation, 2024) statically converts CUDA source code into HIP source code, enabling ROCm compatibility via direct syntax substitution. Operating at a lower abstraction, CuPBoP-AMD (Chen et al., 2023) translates NVVM IR to HIP-compatible LLVM IR using the LLVM toolchain (Lattner and Adve, 2004; The Clang Team, 2025), offering more flexible intermediate-level interoperability. Earlier, GPU Ocelot (Diamos et al., 2009) explored dynamic binary translation, recompiling CUDA to AMD/x86 ISAs at runtime. Although innovative, it was limited by poor scalability and high overhead, making it impractical for modern GPU workloads. All these tools have lacked consistent updates to keep up with CUDA advances, suffer from usability issues, and operate only at the source level.

More recently, ZLUDA (Janik, 2024) introduced a runtime for executing unmodified CUDA binaries on AMD GPUs by intercepting CUDA APIs and translating PTX/SASS into AMD-compatible code via LLVM IR. Operating at the IR rather than hardware-assembly level, it cannot exploit Nvidia's

Domain/ Characteristics	ComputeEval NVIDIA	Rodinia Bench	SHOC	Poly Bench	Babel Stream	Ours
CUDA (src)	✓	✓	✓	✓	✓	✓
OpenCL (src)	✗	✓	✓	✓	✓	✓
SASS (asm)	✗	✗	✗	✗	✗	✓
RDNA3 (asm)	✗	✗	✗	✗	✗	✓

Table 1: Comparison of Domain/Characteristics across Different Datasets. src: source, asm: assembly.

backend optimizations below PTX. By contrast, we target direct assembly-to-assembly translation to transfer hardware-specific optimizations not reflected in intermediate representations.

Assembly-to-Assembly Translation. Translating assembly across ISAs is challenging due to divergent instruction sets and execution models. Recent work employs LLMs for this task, including GG (Heakl et al., 2025), a lightweight x86-to-ARM transpiler, and Guess & Sketch (Lee et al., 2023), which integrates symbolic reasoning for ARMv8-to-RISC-V translation. A key factor in their success is large CPU-centric datasets enabling cross-ISA training. Given the lack of such datasets for GPUs, a primary goal of this work is to enable transpilation across GPU vendors.

Datasets for CUDA and HIP. As shown in Table 1, existing benchmarks in the GPU space generally focus on runtime performance, do not target the assembly level, and lack paired/aligned data across Nvidia/AMD codebases. ComputeEval (NVIDIA, 2024) includes only CUDA code for hardware evaluation. Rodinia (Che et al., 2009) and SHOC (Danalis et al., 2010) provide heterogeneous benchmarks using CUDA/OpenCL/OpenMP but omit AMD code and assembly. PolyBench (Grauer-Gray et al., 2012) evaluates compilers with CUDA/OpenCL kernels, yet lacks assembly-level or AMD support. BabelStream (Deakin et al., 2016) benchmarks HIP/CUDA/OpenCL memory bandwidth but excludes assembly and domain diversity. Hetero-Mark (Sun et al., 2016) targets CPU-GPU workloads where GPU code is minimal. The Stack (Lozhkov et al., 2024a,b) contains nearly 200k CUDA files but no AMD coverage or aligned assembly. In contrast, `CASS` offers 60k semantically aligned CUDA-HIP source and SASS-RDNA3 assembly pairs across host and device, forming the first dataset for cross-vendor GPU assembly translation.

To the best of our knowledge, no existing dataset provides *paired* source- and assembly-level Nvidia-AMD code, hindering effective training and benchmarking.

3 Methods

This section outlines the end-to-end methodology behind `CASS`, including data collection and code compilation for Nvidia and AMD GPUs.

3.1 CUDA Code Scraping

We leveraged the Stackv2 dataset (Lozhkov et al., 2024b) to extract CUDA source files. This dataset, curated from a vast array of public code repositories, offers deduplicated and license-compliant samples, facilitating the assembly of a diverse corpus of GPU-oriented code. To maximize the number of compiled files in the later stage, we used the dataset's metadata to identify and download the top 200 repositories with the largest number of CUDA files. This repository-level download preserved the original directory structure and relative imports, as shown in Figure 1, and improved compilation success by 23.7% compared to isolated file scraping. After extraction, we applied additional filtering to remove overly long files ($>$ 7k lines), trivially short files ($<$10 lines), naive boilerplate samples (e.g., "Hello World"), and files lacking CUDA kernel definitions. This process resulted in a final set of 24k usable samples.

3.2 Synthetic Data Generation

To circumvent the issue of low architectural and semantic diversity in underrepresented GPU kernels from "real-world" code pairs, we employed a coding-oriented large language model (`Qwen2.5-Coder32B`) to synthesize CUDA kernel code using our variable-augmented persona strategy. We found this important because the amount of GPU code online in general is limited, both in quantity and diversity, and hence one of the goals of `CASS` is to address this problem for ourselves and others in the space.

The process begins by defining a set of natural language prompt templates with variable placeholders. For example, a template might read:

> *Generate a CUDA kernel for cloth simulation with a {size}X{size} grid. Optimize for {optimization}.*

To fill these templates, we prepared predefined lists of variable values. For instance, `{size}` was instantiated with values such as 32, 64, and 128, while `{optimization}` was sampled from options like "memory bandwidth", "register usage", and "multi-GPU scaling". This allowed

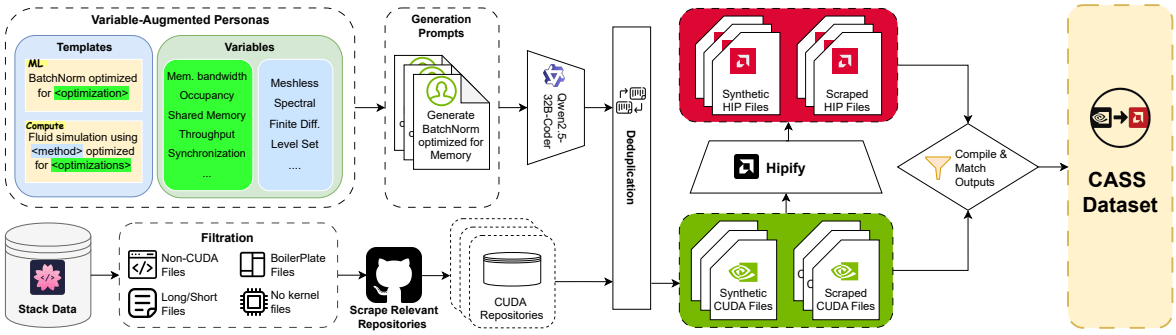

Figure 1: **CASS Data Collection Pipeline.** We collect CUDA code from public repositories and synthesize additional samples via prompt-based LLM generation. After filtering and deduplication, all CUDA files are translated to HIP using HIPIFY, then compiled to extract host and device assembly. Matched outputs form the CASS dataset with aligned source and assembly pairs across Nvidia and AMD stacks.

us to systematically generate a broad range of prompts, each specifying different values for the placeholders in the templates. Appendix A.8 includes full details on prompts, variables, and value ranges used for synthetic data generation.

These prompts were then passed to the LLM, which generated CUDA source files accordingly. While this method introduced some functional inconsistencies that required significant post-generation filtering (syntactic errors, missing definitions, or invalid memory operations), it enabled the creation of rich and diverse CUDA samples. In total, we generated 85k CUDA samples, of which 49.1% compiled successfully, yielding a final set of 20.6k valid files of synthetic data (complementing the 34.1k "real-world" date described in §3.1).

3.3 Transpilation and Compilation

After collecting CUDA files from the previous stages, we performed a deduplication sanity check to ensure all samples are unique in our dataset. We then used AMD's Hipify tool (Advanced Micro Devices, Inc., 2025) to convert CUDA source files by replacing CUDA-specific API calls with HIP equivalents. Files that failed conversion (approx. 43.9%) were discarded. Once CUDA-HIP pairs were available, we compiled them to host and device assemblies using `-O3` compilation flag to reduce code size, achieving a 9.3% average token reduction compared to `O3`. Given the architectural divergence of the two stacks (see Figure 2), their compilation pipelines differed substantially, requiring significant effort to engineer and standardize our described workflow.

In Figure 2, a key distinction between the CUDA and HIP compilation pipelines lies in how they manage host and device assembly separation. In ROCm, the device binary is typically embedded into the host binary during the BitCode-to-assembly transition. We modified this behavior by deferring insertion until after host assembly was converted to object code, enabling: **(1)** independent extraction of pure host (CPU) and device (GPU) assemblies, and **(2)** selective recombination for controlled translation and evaluation.

Conversely, Nvidia provides no access to its binary injection process, device and host assemblies remain intertwined, with no official method for extraction or reintegration (NVIDIA Corporation, 2025). Since our goal was to support host-to-host and device-to-device transpilation, recombination on the CUDA side was unnecessary. Instead, we developed a regex-based filtering pipeline to disentangle host and device assembly sections during CUDA compilation.

After compiling both stacks to SASS and RDNA3, we retained only samples that compiled successfully on both Nvidia and AMD pipelines, accounting for asymmetric failures. The final dataset includes matched CUDA-HIP source pairs, SASS-RDNA3 device assemblies, and host assemblies. In total, 64k samples were collected after this stage.

3.4 OpenCL Pipeline

OpenCL stands as an independent pipeline in generating Nvidia to AMD mapping datasets outside of the CUDA/HIP framework. In other words, it compiles down to the assembly level

Table 2: Dataset composition by source and size

Dataset	Collected	Final
Synthetic	85.5k	20.6k
Stack	124.1k	34.1k
OpenCL	6.6k	5.9k
Total		**60.7k**

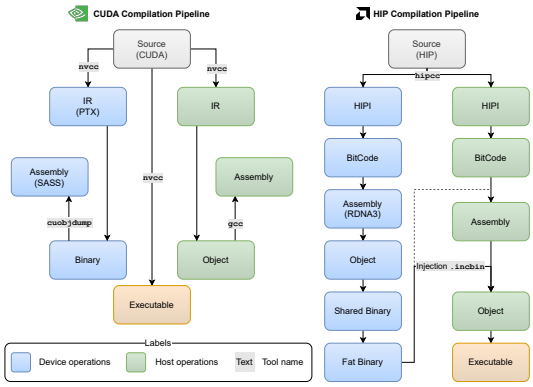

Figure 2: The Nvidia (left) and AMD (right) stacks illustrate the compilation process for CUDA and HIP. Blue denotes device-side steps; green denotes host-side steps. Nvidia's stack is opaque; accessing device assembly (SASS) requires first compiling to binary, then using `cuobjdump`. In contrast, AMD's process is transparent, allowing direct inspection and modification of device assembly (RDNA3) before host integration.

without going through the aforementioned stacks, operating as a single "source" for GPU code deveolpment (Group, 2025). Approximately 6k OpenCL code snippets were collected from the Stack dataset and compiled down to the device assemblies. On the Nvidia stack, a wrapper C++ function was used to encapsulate the clBuildProgram library provided by OpenCL (Group, 2020) and convert them into PTX, after which the CUDA stack was used to compile them down to assemblies. On the AMD stack, clang was used to directly transpile the OpenCL files into device assemblies whilst forcing it to emit intermediate LLVM during this process (The Clang Team, 2025).

The final instruction training dataset (`CASS`) comprises 60,694 aligned samples spanning a broad range of domains, with a primary focus on GPU compute and GPU-related data structures (Figure 3, Table 2). Each sample includes both CUDA and HIP source code alongside its compiled assembly representation, with pairwise source/assembly alignments verified to compile successfully. All compilations were performed on an Nvidia A100 PCIe machine for the CUDA stack (SASS sm85 ISA) and on AMD Radeon RX 7900 XT GPUs (RDNA3 ISA) for the AMD stack.

4 CASS and CASS-Bench Datasets

This section presents `CASS`, a large-scale dataset of aligned CUDA/HIP source and SASS/RDNA3 assembly code, and `CASS-Bench`, a curated benchmark for cross-architecture GPU translation.

4.1 Dataset Analysis

`CASS` reveals pronounced structural divergence between CUDA and HIP at both source and assembly levels, underscoring the inherent complexity of cross-architecture GPU transpilation. We analyze this by looking at the length of the assembly files, their syntactic similarity, and opcode diversity.

Length of Assembly Files. Figure 3 (right) shows that AMD device assembly is, on average, twice as long as Nvidia's in both synthetic and Stack subsets, while Nvidia's device assembly exceeds HIP device assembly by $\sim$50% in the OpenCL set. We found an exponential relationship between source complexity and assembly size, with CUDA producing more verbose outputs than HIP for equivalent code. This highlights the growing difficulty of asm-level translation as code complexity scales. See appendix A.5.1 for full details.

Opcode Diversity. Tensor operations dominate both CUDA and HIP assembly, especially in device code, with memory-related instructions such as `mov` and `call` appearing most frequently (refer to appendix A.5). HIP opcodes like `s_mov_b32` and `v_add_co_u32` are used extensively, reflecting low-level vector and memory operations unique to AMD's ISA, while Nvidia is dominated by instructions such as `movq`, `call`, and `jmp`, with greater host-side integration. Both stacks share common control and memory ops (e.g., `mov`, `test`), but HIP provides finer-grained access to GPU internals, revealing deeper visibility into parallelism. The synthetic subset emphasizes memory-oriented instructions, aligning with LLM-driven template optimizations. We further conduct a t-SNE visualization of opcode embeddings generated by CodeBERT (Feng, 2020) to examine semantic relationships between Nvidia and AMD instructions. The resulting clusters indicate that, despite backend differences, the two vendors exhibit semantically aligned opcode distributions across device and host levels (Figure 9).

4.2 CASS-Bench

`CASS-Bench` is a 369-sample evaluation benchmark for cross-architecture GPU transpilation, covering both source-level (CUDA $\rightarrow$ HIP) and assembly-level (SASS $\rightarrow$ RDNA3) translation. We constructed the benchmark from the same public repositories used for training data collection (§3), ap-

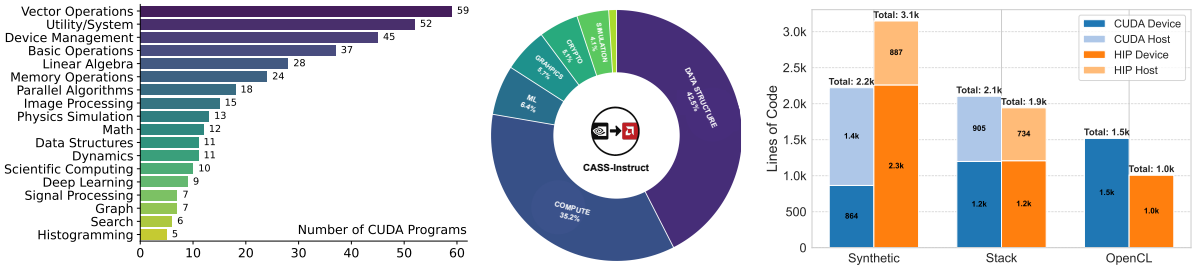

Figure 3: **CASS coverage across dataset and benchmark.** (left) category distribution in CASS-Bench (center) domain distribution of training samples (right) verbosity across subsets and backends.

plying strict deduplication to ensure zero overlap between benchmark and training sets. Each sample was selected to compile and execute successfully on both Nvidia and AMD hardware, with output equivalence verified through automated differential testing. The benchmark spans 18 GPU-centric domains with distribution shown in figure 3 (left).

5 Experiments

We evaluate the CASS dataset by instruction-supervised fine-tuning the Qwen2.5-Coder (Hui et al., 2024a) models at various parameter scales. Two variants are developed: one for assembly translation (SASS → RDNA3) and another for source translation (CUDA → HIP). We benchmark these models against both proprietary and open-source baselines, including larger-scale systems.

Instruction Supervised Finetuning. To ensure that input samples fit within the 16K-token context window of the LLM, we normalized CUDA assembly code by removing redundant whitespace and comments, which reduced token count by roughly 15%. No preprocessing was applied to HIP assembly code due to its sensitivity to whitespace changes. We fine-tuned the Qwen2.5-Coder models at 1.5B, 3B and 7B parameter scales on 4xA100 GPUs, using a batch size of 4, gradient accumulation of 32 (effective batch size of 512) and a learning rate of 1×10^{-5}. The relatively aggressive learning rate was selected due to the dataset's distributional divergence from the models' pretraining corpus. Training employed DeepSpeed (Rasley et al., 2020) with optimizer state sharding to maximize hardware efficiency, achieving 98% GPU utilization. Additionally, we incorporated Liger Kernel (Hsu et al., 2024) and Paged Adam optimizer (Loshchilov and Hutter, 2017) to accelerate training and manage memory more effectively. We utilized LLaMA-Factory (Zheng et al., 2024) to implement all of these optimizations. All models were trained with a 16K-token context window. At inference time, we applied RoPE (Su et al., 2024) extrapolation to support up to 32.7K tokens. Inference was efficient, requiring approximately 56 seconds per a 16K-token sample.

6 Results

Assembly-to-Assembly Performance. Table 3 reports CASS-Bench results across LLMs and tools, revealing a striking performance gap. Even frontier models struggle: GPT-5.1 achieves only 22.22% accuracy, while coding-specialized models like Qwen2.5-Coder-32B reach just 5.42%. This failure stems from fundamental exposure gaps, SASS is a *proprietary, undocumented* ISA whose opcode-to-binary mappings have only been partially reverse-engineered (Hayes et al., 2019), meaning these models encountered minimal GPU assembly during pretraining. ZLUDA's poor performance (7.86%) despite operating on compiled binaries reflects its architectural constraints: it translates at the LLVM IR level rather than native assembly, missing backend-specific optimizations like NVIDIA's instruction scheduling and register allocation that exist only in SASS (Janik, 2024). The ISA divergence compounds the challenge, SASS uses fixed 16-byte instructions with implicit stall counts for latency hiding, while RDNA3 employs explicit s_waitcnt synchronization and separates scalar/vector pipelines (Advanced Micro Devices, Inc., 2023). Our CASS-sm80-7B achieves 69.11% accuracy by learning these vendor-specific patterns from aligned training pairs, demonstrating that domain-specific fine-tuning can bridge ISA gaps that general-purpose models cannot. Analysis of compilation failures (Figure 4) reveals that invalid instructions and operand constraint violations account for the majority of errors, reflecting the model's difficulty in learning vendor-specific encoding rules. Figure 4 further shows that these failures are most prevalent in computation-intensive

	Model	Assembly			Source	
		Test Acc. ↑	Compile Rate ↑	chrF ↑	Test Acc. ↑	Compile Rate ↑
Tools	ZLUDA (Larabel, 2024)	7.86 ±0.00	8.40 ±0.00	7.84	19.24 ±0.00	27.59 ±0.00
	Hipify (Advanced Micro Devices, Inc., 2025)	–	–	–	87.46 ±0.00	92.95 ±0.00
General	GPT-5-mini (OpenAI, 2025a)	14.63 ±1.68	15.72 ±3.03	14.16	66.67 ±1.93	93.91 ±0.98
	Claude-Haiku-4.5 (Anthropic, 2025b)	11.24 ±1.52	13.87 ±2.90	15.31	69.42 ±1.93	95.27 ±0.85
	Gemini-2.5-Flash (Comanici et al., 2025)	8.94 ±1.39	10.30 ±2.53	14.75	65.95 ±2.02	94.98 ±0.88
	LLaMA-3.1-8B (Dubey et al., 2024)	2.10 ±0.67	2.89 ±1.39	8.67	28.63 ±1.86	42.15 ±2.02
	Gemma-3-27B (Team et al., 2025)	2.44 ±0.72	2.71 ±1.26	8.54	35.48 ±1.96	50.18 ±2.05
	Qwen3-30B-A3B (Yang et al., 2025)	16.53 ±1.77	18.43 ±3.16	6.79	51.25 ±2.08	78.14 ±1.74
Reasoning	GPT-5.1 (OpenAI, 2025b)	22.22 ±1.77	25.20 ±3.66	16.62	84.95 ±1.45	98.92 ±0.41
	Gemini-2.5-Pro (Comanici et al., 2025)	6.50 ±1.14	8.94 ±2.40	14.52	66.67 ±1.93	93.19 ±1.04
	Olmo-3.1-32B (Olmo et al., 2025)	8.60 ±1.35	11.11 ±2.53	9.43	27.91 ±1.83	31.17 ±1.89
	DeepSeek-R1-Qwen-14B (Guo et al., 2025)	4.07 ±0.93	4.34 ±1.64	5.31	8.60 ±1.14	11.11 ±1.29
Coding	Qwen3-Coder-30B-A3B (Yang et al., 2025)	1.36 ±0.55	1.36 ±0.88	8.21	51.25 ±2.05	78.14 ±1.70
	Qwen2.5-Coder-32B (Hui et al., 2024b)	5.42 ±1.05	6.50 ±2.02	9.65	46.95 ±2.05	73.48 ±1.83
	DeepSeek-Coder-V2-Lite (Zhu et al., 2024)	2.99 ±0.80	4.08 ±1.64	6.21	24.17 ±1.77	35.82 ±1.99
Ours	CASS-sm89-1.5B	54.74 ±2.05	67.21 ±2.78	81.28	85.91 ±1.48	96.75 ±0.72
	CASS-sm89-3B	59.89 ±2.02	81.84 ±2.21	79.91	86.99 ±1.42	98.37 ±0.51
	CASS-sm80-1.5B	67.48 ±1.89	74.53 ±2.59	77.53	86.74 ±1.42	97.49 ±0.60
	CASS-sm80-3B	68.83 ±1.93	75.88 ±2.40	79.91	87.81 ±1.36	99.28 ±0.28
	CASS-sm80-7B	**69.11 ±1.89**	**90.24 ±1.70**	**81.28**	**88.17 ±1.29**	**98.92 ±0.41**

Table 3: **Performance of different models on `CASS-Bench`.** *Test Acc.* is strict end-to-end accuracy (compile + exact output match) over 369 tasks. *Compile Rate* measures number of samples that compiled successfully. *chrF* provides character-level n-gram similarity for assembly. `sm80` and `sm89` denote CUDA compute capabilities for A100 and RTX 4090 GPUs, respectively.

domains such as vector operations and linear algebra, indicating that complex arithmetic kernels pose the greatest challenge for assembly-level translation.

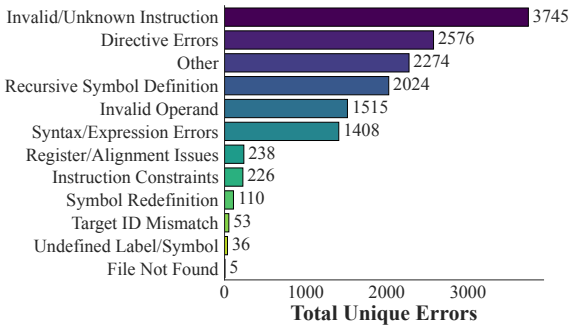

Figure 4: Distribution of compilation errors.

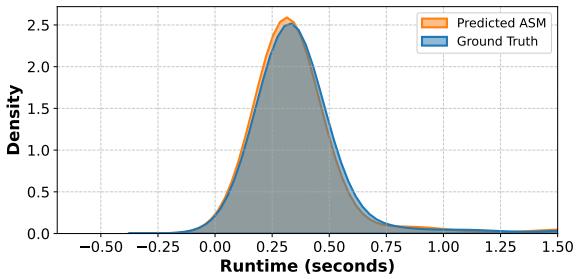

Figure 5: Runtime distribution comparison between ground truth HIP and CASS-translated assembly.

Code Efficiency. Figure 5 and Figure 10 show that the translated code closely matches the original in execution: predicted and ground-truth runtime distributions largely overlap, differing only slightly in the tail. The GPU memory usage deviates by less than ±0.3 MB, and runtime differences are bounded within ±1.5 s, with over 95% of samples falling within ±0.5 s, confirming that our model preserves both memory and runtime efficiency during assembly translation. Each test was executed 20 times, and the reported values reflect the average across runs to mitigate noise.

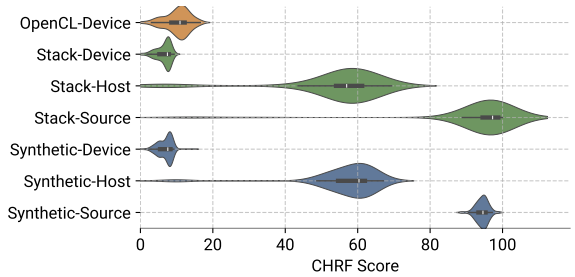

Figure 6: Syntactic similarity (CHRF) between CUDA and HIP across code levels.

Source-to-Source Performance. Source transpilation proves substantially easier than assembly translation, with our best model achieving 88.17% accuracy compared to 69.11% for assembly. This gap reflects a deliberate design choice: HIP was engineered as a "dialect" of CUDA where most API calls differ only by prefix (`cuda*`→`hip*`), function signatures remain identical, and both languages share the same C++ semantics including templates, lambdas, and classes (Advanced Micro Devices,

Inc., 2024). The high syntactic similarity (CHRF 90%, Figure 6) means Hipify can rely on regex-based string replacement for the majority of conversions. However, Hipify fails on constructs requiring semantic understanding: inline PTX assembly (which has no HIP equivalent), hardcoded warp sizes (`32` vs. `64` on AMD), CUDA-specific intrinsics like `__shfl_down_sync`, and architecture-dependent preprocessor guards (Advanced Micro Devices, Inc., 2025). Our model surpasses Hipify (88.17% vs. 87.46%) despite training exclusively on Hipify outputs, a form of "self-improvement" where the model learns the underlying semantic mapping rather than mimicking surface-level substitutions, enabling correct translation of edge cases that defeat pattern matching.

Experiment	Source Acc.	Assembly Acc.	Δ Impact
Stack subset	79.67%	48.24%	-
+Synthetic	84.28%	59.89%	+11.65%
+OpenCL	87.18%	65.23%	+5.34%
+RoPE Extrapolation	88.17%	69.11%	+3.88%

Table 4: **Ablation study on data sources and context extension.** Each row adds a component cumulatively; Δ Impact denotes absolute assembly accuracy gain over the previous row.

Ablation Study. Table 4 reveals that synthetic data contributes the largest improvement (+11.65%), confirming LLM-generated samples provide architectural diversity absent from real-world code (e.g., sparse tensor ops, N-body simulations). OpenCL adds +5.34% despite only 6k samples, as its distinct compilation pathway introduces different instruction patterns (e.g., explicit `__local` memory management absent in CUDA). RoPE extrapolation yields +3.88% by extending context length, beneficial for longer sequences where position encodings degrade (Figure 7).

Hardware Generalization. To assess cross-microarchitecture transfer, we evaluate our `CASS-sm80-7B` model (trained on A100 SASS) on RTX4090 (sm89, Ada Lovelace) ↔ RX7900 XT. Accuracy drops sharply from 69.11% to 32.5%, despite both being NVIDIA architectures. This degradation reflects ISA divergence between compute capabilities: `sm89` introduces FP8 tensor instructions absent in `sm80`, doubles FP32 throughput per SM (altering instruction scheduling), and employs a 16× larger L2 cache that changes memory access patterns (NVIDIA Corporation, 2024). The compiler also generates different stall counts and register allocation strategies across generations (PNF Software, 2025). That these variations occur *within* NVIDIA's ecosystem yet cause >50% relative accuracy loss underscores a key insight: GPU assembly translation is fundamentally a cross-microarchitecture problem, where even generational changes invalidate learned instruction mappings. This motivates CASS as a living benchmark the community can extend to new ISA pairings as hardware evolves.

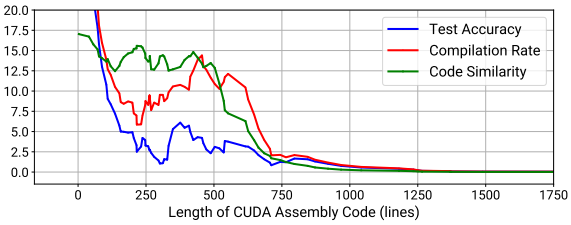

Figure 7: **Effect of input length on assembly translation performance.** All metrics degrade as sequence length increases.

Impact of Input Length. Figure 7 reveals a strong inverse relationship between input sequence length and translation quality. As assembly code length increases, both test accuracy and compilation rate degrade sharply, with accuracy approaching zero beyond 750 lines. This trend reflects the compounding difficulty of longer sequences: increased code complexity introduces more intricate control flow and memory patterns, while LLMs inherently suffer from long-context degradation where attention quality diminishes over extended inputs (Liu et al., 2024).

7 Conclusion

We presented CASS, a large-scale dataset and model suite for cross-architecture GPU transpilation, with 60k aligned source and assembly pairs for Nvidia and AMD. CASS bridges source-to-source (CUDA to HIP) and assembly-to-assembly (SASS to RDNA3) mappings, addressing a key gap in low-level portability. We train the CASS model family, achieving 88.2% accuracy in source translation and 69.1% in assembly translation, outperforming proprietary and open-source baselines. Our transpiled code preserves functional behavior: over 95% of samples match native execution in memory usage and runtime. We introduce CASS-Bench, an evaluation suite spanning 18 GPU-centric domains. All models and data are released open-source to support future work in compiler tooling and hardware interoperability.

8 Limitations and Future Work

Despite this progress, CASS does not yet fully capture certain advanced GPU instruction classes, most notably Tensor Core and matrix-oriented operations. This limitation primarily stems from incomplete support in HIPIFY, AMD's official CUDA-to-HIP source translator, which currently lacks mappings for WMMA, cuBLAS TensorOp functions, and other low-level CUDA primitives. As a result, kernels relying on Tensor Core instructions or warp-level matrix operations fail to translate or compile successfully. While we explored extending coverage by incorporating additional HIP libraries (e.g., hipBLAS, hipTensor), these efforts yielded only marginal gains due to compilation pipeline constraints. We view this as an opportunity for future expansion. As HIPIFY and the ROCm ecosystem evolve to support a broader range of CUDA functionality, CASS can be extended to include richer tensor and matrix workloads, enabling deeper study of performance-preserving translation across increasingly heterogeneous GPU backends.

9 Use of Language Models

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

References

Advanced Micro Devices (AMD). 2024. Amd rocm™ 6: Open software platform for gpu computing. Technical report, Advanced Micro Devices, Inc.

Advanced Micro Devices, Inc. 2023. AMD RDNA3 instruction set architecture reference guide. Technical report, AMD. Accessed: 2025-05-15.

Advanced Micro Devices, Inc. 2024. Porting NVIDIA CUDA code to HIP. ROCm Documentation. Accessed: 2025-05-15.

Advanced Micro Devices, Inc. 2025. *HIPIFY Documentation*. Accessed: 2025-04-28.

AMD. 2024a. Gaming gpu benchmarks. https://www.amd.com/en/products/graphics/gaming/gaming-benchmarks.html. Accessed: 2025-05-15.

AMD. 2024b. HIP: Heterogeneous-computing Interface for Portability. https://github.com/ROCm-Developer-Tools/HIP. Accessed: 2025-04-30.

AMD ROCm Documentation. 2024. *HIP Porting Guide*. Accessed: 2025-01-29.

Anthropic. 2025a. Claude 3.7 sonnet and claude code. Accessed: 2025-05-14.

Anthropic. 2025b. Claude haiku 4.5. https://www.anthropic.com. Large language model.

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 *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee.

Jun Chen, Xule Zhou, and Hyesoon Kim. 2023. Cupbop-amd: Extending cuda to amd platforms. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 1093–1104.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S Meredith, Philip C Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S Vetter. 2010. The scalable heterogeneous computing (shoc) benchmark suite. In *Proceedings of the 3rd workshop on general-purpose computation on graphics processing units*, pages 63–74.

Tom Deakin, James Price, Matt Martineau, and Simon McIntosh-Smith. 2016. Gpu-stream v2. 0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models. In *High Performance Computing: ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19–23, 2016, Revised Selected Papers 31*, pages 489–507. Springer.

Gregory Diamos, Andrew Kerr, and Sudhakar Yalamanchili. 2009. Gpuocelot: A dynamic compilation framework for ptx. https://github.com/gtcasl/gpuocelot. Accessed: 2025-04-28.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Z Feng. 2020. Codebert: A pre-trained model for program-ming and natural languages. *arXiv preprint arXiv:2002.08155*.

Financial Times. 2024. Nvidia's rivals take aim at its software dominance. Accessed: 2025-05-14.

Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Autotuning a high-level language targeted to gpu codes. in 2012 innovative parallel computing (inpar). *IEEE, Piscataway, NJ, USA*, pages 1–10.

Khronos Group. 2020. clbuildprogram - opencl 3.0 reference pages. https://registry.khronos.org/OpenCL/sdk/3.0/docs/man/html/clBuildProgram.html. Accessed: 2025-05-14.

The Khronos Group. 2025. Opencl guide. https://github.com/KhronosGroup/OpenCL-Guide. Accessed: 2025-05-14.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Mark Harris. 2024. An even easier introduction to cuda. *NVIDIA Developer Blog*.

Ari B Hayes, Fei Hua, Jin Huang, Yanhao Chen, and Eddy Z Zhang. 2019. Decoding cuda binary. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 229–241. IEEE.

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. *arXiv preprint arXiv:2506.14606*.

Pin-Lun Hsu, Yun Dai, Vignesh Kothapalli, Qingquan Song, Shao Tang, Siyu Zhu, Steven Shimizu, Shivam Sahni, Haowen Ning, and Yanning Chen. 2024. Liger kernel: Efficient triton kernels for llm training. *arXiv preprint arXiv:2410.10989*.

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

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

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Andrzej Janik. 2024. Zluda: Cuda on non-nvidia gpus. https://github.com/vosen/ZLUDA. Accessed: 2025-04-28.

Michael Larabel. 2024. Amd quietly funded a drop-in cuda implementation built on rocm: It's now open-source. *Phoronix*. Accessed: 2025-05-14.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, San Jose, CA, USA. IEEE Computer Society.

Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M Rush. 2023. Guess & sketch: Language model guided transpilation. *arXiv preprint arXiv:2309.14396*.

Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173.

Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024a. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024b. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

NVIDIA. 2024. Computeeval: Evaluating large language models for cuda code generation. https://github.com/NVIDIA/compute-eval. Accessed: May 2025.

NVIDIA Corporation. 2021. *Turing Compatibility Guide for CUDA Applications*. Version 11.4.2.

NVIDIA Corporation. 2024. NVIDIA Ada GPU architecture tuning guide. Technical report, NVIDIA. Accessed: 2025-05-15.

NVIDIA Corporation. 2025. *CUDA Binary Utilities*. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html.

Team Olmo, Allyson Ettinger, Amanda Bertsch, Bailey Kuehl, David Graham, David Heineman, Dirk Groeneveld, Faeze Brahman, Finbarr Timbers, Hamish Ivison, and 1 others. 2025. Olmo 3. *arXiv preprint arXiv:2512.13961*.

OpenAI. 2025a. GPT-5. https://openai.com/index/introducing-gpt-5/. Accessed: 2025-01-05.

OpenAI. 2025b. GPT-5.1. https://openai.com/index/gpt-5-1-for-developers/. Accessed: 2025-01-05.

PNF Software. 2025. Reversing Nvidia GPU's SASS code. JEB Decompiler Blog. Accessed: 2025-05-15.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3505–3506.

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063.

Yifan Sun, Xiang Gong, Amir Kavyan Ziabari, Leiming Yu, Xiangyu Li, Saoni Mukherjee, Carter McCardwell, Alejandro Villegas, and David Kaeli. 2016. Hetero-mark, a benchmark suite for cpu-gpu collaborative computing. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE.

Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, and 1 others. 2025. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*.

The Clang Team. 2025. Clang: a c language family frontend for llvm. Accessed: 2025-01-29.

Tom's Hardware. 2025. AMD Radeon RX 9070 XT and RX 9070 review. https://www.tomshardware.com/pc-components/gpus/amd-radeon-rx-9070-xt-review. Accessed: 2025-05-15.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Anwar Hossain Zahid, Ignacio Laguna, and Wei Le. 2024. Testing gpu numerics: Finding numerical differences between nvidia and amd gpus. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 547–557. IEEE.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Appendix

A.1 Computational Resources and Environmental Impact

Hardware used. All experiments were conducted on two distinct machines to generate architecture-specific outputs. For AMD-related compilation and execution, we used a workstation equipped with an Intel i7-14700KF CPU and an AMD Radeon RX 7900 XT GPU. For Nvidia-related outputs, we used a server with an AMD EPYC 9654 CPU and an Nvidia A100 (80GB) GPU. Furthermore, to ensure consistency and reproducibility across platforms, all file generation was performed within Docker containers tailored to each architecture.

Carbon footprint. Energy consumption was measured using the `CodeCarbon` tool during fine-tuning experiments conducted on 4×A100 (40GB) GPUs for 2 epochs across 60k samples. The results are as follows:

Model	Energy (kWh)	CO_2 (kg)
CASS-1.5B	20.13	12.11
CASS-3B	34.27	20.72

These measurements align with expected energy usage benchmarks for models of similar size and training duration.

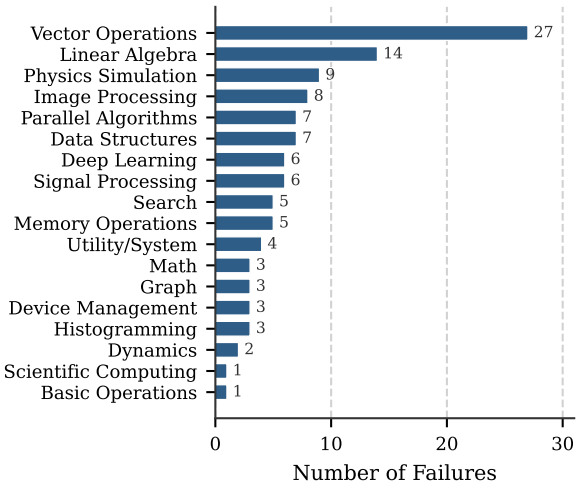

Figure 8: Assembly-level failures across categories.

A.2 Prompting Strategy for Closed-source Models

For the results reported in the paper, we evaluated the assembly-to-assembly translation capacity of

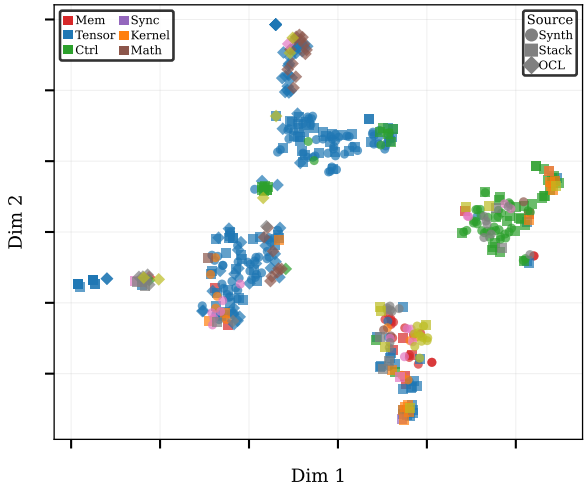

Figure 9: t-SNE projection of CUDA and HIP assembly embeddings.

closed-source models (GPT-5.1 (OpenAI, 2025b), Claude-Haiku-4.5 (Anthropic, 2025b), Gemini-2.5-Flash (Comanici et al., 2025), Gemini-2.5-Pro (Comanici et al., 2025)) and open-source models (LLaMA-3.1-8B (Dubey et al., 2024), Gemma-3-27B (Team et al., 2025), Qwen3-30B-A3B (Yang et al., 2025), Olmo-3.1-32B (Olmo et al., 2025), DeepSeek-R1-Qwen-14B (Guo et al., 2025), Qwen3-Coder-30B-A3B (Yang et al., 2025), Qwen2.5-Coder-32B (Hui et al., 2024b), DeepSeek-Coder-V2-Lite (Zhu et al., 2024)) using the prompt:

```
You are given a CUDA assembly (SASS) code and
you are required to convert it into HIP assembly
(RDNA3) code without changing the functionality.
The code output from CUDA and HIP should be
the same when executed.
Target architecture: {architecture}.
Return the output in the following format:
```amdasm
<HIP assembly code>
`` ``.
CUDA Assembly:
```cudaasm
{cuda_asm}
```
```

For the same models, we use the following prompt for evaluating source code:

```
You are given a CUDA source code and you are
required to convert it into HIP code without
changing the functionality. The code output from
CUDA and HIP should be the same when executed.
Target architecture: {architecture}.
Return the output in the following format:
```amd
<HIP code>
`` ``.
CUDA Code:
```cuda
{cuda_code}
```
```

We use the default inference hyperparameters (`temperature`, `top_p`, `max_tokens`) provided by each model.

We also experimented with more advanced prompting strategies, adding few-shot examples of (SASS, RDNA3) pairs and applying chain-of-thought (CoT) prompting, but observed no significant performance gains. This outcome likely stems from the limited prior exposure these models have low-level GPU assembly code during pretraining. Even with better prompting, the models lack the internal structure or inductive bias needed to reason over hardware-specific instruction patterns.

A.3 Evaluation on ZLUDA

To assess ZLUDA's ability to execute CUDA code on AMD GPUs, we designed a two-track evaluation strategy targeting both source-level and binary-level workflows (the latter being akin to assembly-level translation). In the source-to-source setting, we leveraged access to the original CUDA source files to manually compile them into PTX using `nvcc`. These PTX files were then ingested by ZLUDA, which translated them into AMD-compatible LLVM IR before lowering them into native executables targeting RDNA3 hardware. In the assembly-to-assembly setting, we instead compiled the CUDA source into a complete executable and invoked it directly. ZLUDA intercepted the CUDA runtime calls, dynamically translated the embedded PTX or SASS, and executed the resulting code on the AMD backend. This dual strategy allowed us to assess both ZLUDA's static translation capabilities and its runtime interoperability under realistic execution conditions.

A.4 CASS Domain Coverage

To obtain the domain-level breakdown shown in Figure 3, we developed a static analysis pipeline that categorizes each source file based on its content. The classification is performed by matching the file's text against curated sets of domain-specific keywords corresponding to seven high-level categories: *general compute, simulation, data structure, machine learning, graphics, cryptography,* and *scientific computing*. Each keyword set includes terms commonly associated with the respective domain; for example, the *machine learning* category includes terms such as `neural`, `gradient`,

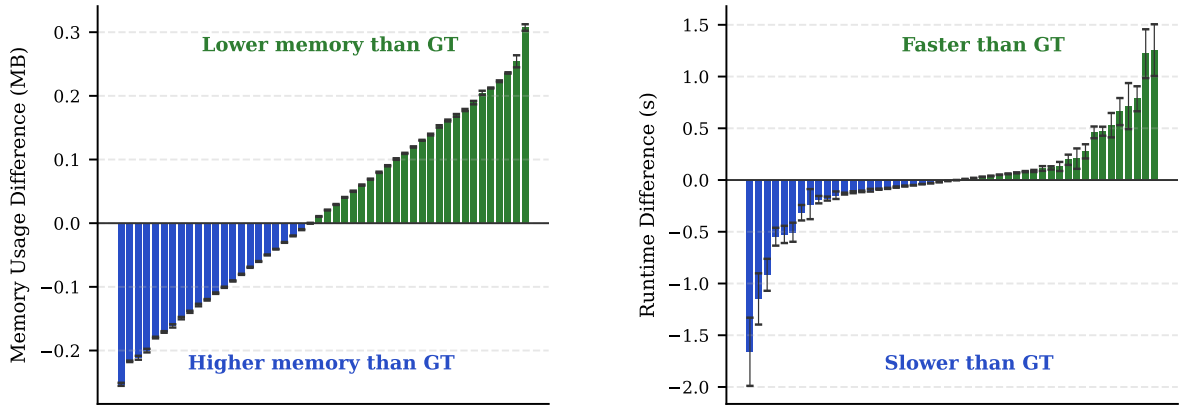

Figure 10: Comparison of memory usage (left) and execution time (right) between predicted and ground truth HIP programs, measured via compilation and runtime profiling.

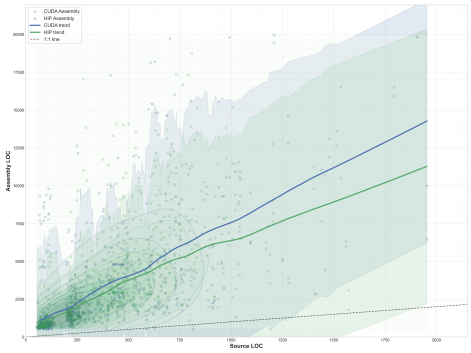

Figure 11: Relationship between source and assembly-level LoC in the CASS dataset. Scatter plot comparing source code lines of code (LoC) to the corresponding assembly LoC for both CUDA and HIP backends across the Stackv2 and Synthetic subsets. Trend lines and density contours illustrate that CUDA typically produces more verbose assembly output than HIP for equivalent source sizes.

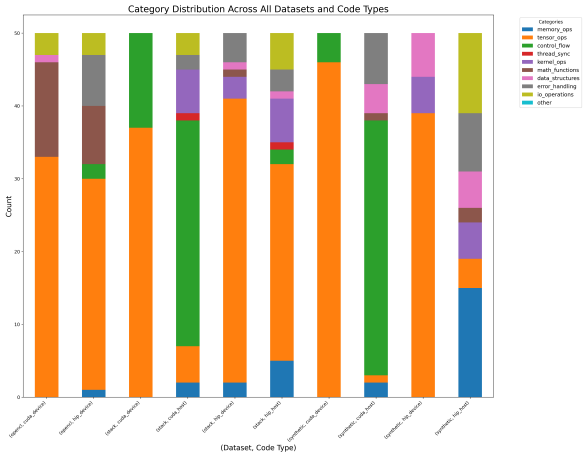

Figure 12: Opcode Category Distribution by Dataset and Code Type. Stacked bar chart showing the distribution of assembly instructions across 10 opcode categories for device and host code in the Synthetic, Stackv2, and OpenCL subsets. Each bar represents a (dataset, code type) pair, illustrating the functional composition of the code across memory, tensor, control flow, synchronization, and other operations.

and `activation`, while *cryptography* includes `hash`, `encrypt`, and `signature`. For a given file, the domain with the highest keyword match count is assigned. If no keywords are matched, a default label (e.g., *general compute*) is applied. After all files are processed, their assignments are aggregated to produce the final domain distribution. This process provides a simple yet straightforward and interpretable way of grouping source files by their functional domain.

A.5 Extra Data Analysis

A.5.1 Length of Assembly Files

As shown in the Figure 11 We found an exponential relationship between source complexity and assembly size, with CUDA producing more ver-

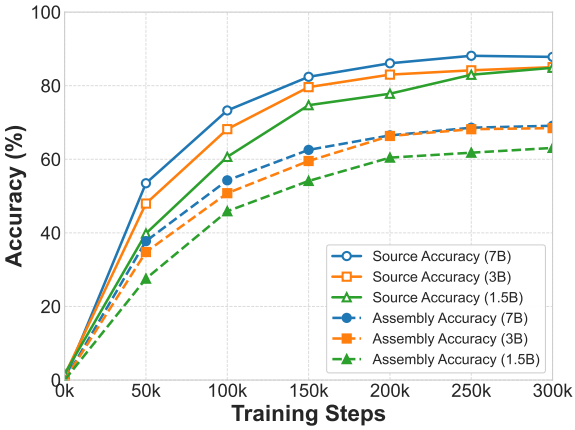

Figure 13: Accuracy vs. training steps for source/assembly across CASS model scales (1.5B, 3B, 7B).

bose outputs than HIP for equivalent code. This highlights the growing difficulty of assembly-level translation as code complexity scales.

A.5.2 Opcode Diversity

In the HIP case, many opcodes, such as `s_mov_b32`, `v_add_co_u32`, and `s_waitcnt`, come directly from AMD's GPU instruction set. These reflect fine-grained control over the hardware, including scalar and vector operations and synchronization. On the other hand, the CUDA assembly is mostly made up of x86-64 instructions like `movq`, `call`, `jmp`, and `pushq`, which are typically used on the CPU. This suggests that the CUDA output includes more host-side code or that GPU instructions are hidden behind a higher level of abstraction. Still, both stacks share common instructions like `mov` and `test`, showing that some basic control and memory operations are similar. In general, HIP provides more visibility into what the GPU is doing, while CUDA hides many of those low-level details behind a more unified host-device model.

A.6 Performance Degradation Analysis

We analyzed failed kernel translations and categorized them by key operation types. Among the failed cases, we found: 100% involved control flow (e.g., `for`, `while`), 75% accessed global memory, 62.07% used synchronization (e.g., `__syncthreads()`), 10.34% involved atomic operations (e.g., `atomicAdd`), 6.82% used shared memory, and 11.36% included local arrays. It's important to emphasize that these categories are not mutually exclusive, as most failed files involve multiple overlapping operation types. Moreover, as shown in Figure 8 in the paper, assembly-level failures are concentrated in Math, Data Structures, Graph, and parts of ML domains, indicating that control-heavy or abstract computation tasks remain the most challenging for the model. These trends suggest that failures are strongly correlated with increased kernel complexity, particularly in terms of global memory access and memory synchronization, which likely strain the model's limited context and structural understanding at the assembly level.

A.7 Stratified Benchmarking

We performed a stratified evaluation of the assembly benchmark by grouping samples based on input length measured in tokens: easy (<9k), medium (9k-12k), and hard (>12k). The model's accuracy decreases with input length, 35.0% (easy), 33.33% (medium), and 17.65% (hard), highlighting that longer sequences pose greater challenges, likely due to context compression.

A.8 Sythetic Generation

To generate large-scale, diverse CUDA programs, we design a multiprocessing Python pipeline that interacts with a locally hosted large language model via a chat-based API. The pipeline leverages a wide array of handcrafted prompt templates, each parameterized with variables such as problem size, optimization target, algorithm type, and architectural features (see Appendix A.8.1). At runtime, these templates are instantiated with randomly sampled values from curated sets covering domains like matrix operations, graph algorithms, scientific computing, machine learning, and sparse computation (see Table 5). Each worker process independently generated prompts, sends them to the model, extracts valid CUDA code from the response, and saves the output in a structured format. Robust fault-tolerance mechanisms, including retry logic, output validation, and file existence checks, ensure resilience to model failures and concurrent access.

Additionally, to avoid reproducing training data seen by the LLM, we apply both prompt-space and output-space deduplication: (1) prompt templates are checked for novelty against LLM training corpora, where verifiable, and (2) generated samples are structurally parsed and filtered using AST and opcode similarity to eliminate near-duplicates. The system supports parallel generation with controlled API concurrency and automatic resumption from previous checkpoints, enabling scalable and efficient generation of compilable CUDA code samples suitable for downstream benchmarking or training.

A.8.1 Prompt Templates for Synthetic CUDA Code Generation

The prompts used, listed below, were designed with variations in computation patterns (e.g., memory operations, thread sync, and control flow) and domains (e.g., ML, simulation, graphics), with the intent of diversifying the scope of the synthetic samples.

Basic Operations

```
1. Implement a CUDA kernel for {size}D FFT (Fast
     Fourier Transform). Optimize for {
    optimization}.
2. Generate a CUDA implementation for {size}D
    stencil computation with radius {radius}.
    Optimize for {optimization}.
```

Placeholder	Example Values
{size}	64, 1024, 16384
{dimension}	1, 3, 6
{optimization}	memory coalescing, shared memory usage, warp-level programming
{operation}	sum, histogram, L2 norm
{algorithm}	matrix multiplication, radix sort, BFS
{radius}	1, 5, 13
{graph_format}	adjacency matrix, CSR, edge list
{md_algorithm}	Verlet integration, leapfrog, Runge-Kutta
{linear_solver}	conjugate gradient, Jacobi, multigrid
{numerical_method}	finite difference, spectral, Crank-Nicolson
{factorization_method}	SVD, LU, eigenvalue decomposition
{conv_layer_count}	2, 6, 12
{neuron_count}	64, 512, 2048
{sparse_format}	CSR, ELL, HYB
{nbody_algorithm}	Barnes-Hut, brute force, particle mesh
{filter_type}	Gaussian, Sobel, Gabor
{filter_size}	3, 7, 15
{resolution}	720p, 1080p, 4K
{segmentation_algorithm}	watershed, region growing, U-Net
{signal_transform}	FFT, wavelet, Hilbert
{optimization_algorithm}	Adam, simulated annealing, particle swarm
{crypto_algorithm}	AES, RSA, Argon2
{cracking_method}	brute force, dictionary attack, rainbow table
{hash_algorithm}	SHA-256, BLAKE3, Bcrypt
{data_structure}	binary tree, hash table, bloom filter
{collision_strategy}	linear probing, cuckoo hashing, separate chaining

Table 5: Representative values for prompt placeholders used in the synthetic code generation.

```
3. Write a CUDA kernel for parallel reduction to
     compute the {operation} of an array of size
     {size}. Focus on {optimization}.
4. Create a CUDA implementation for convolution
    operation with a {size}x{size} filter. Focus
     on {optimization} optimization.
5. Generate a CUDA kernel for matrix
    multiplication of two matrices A and B of
    size {size}x{size}. Include error handling
    and optimize for {optimization}.
```

Graph Algorithms

```
1. Write a CUDA implementation for graph
    coloring of a graph with {size} nodes. Focus
     on {optimization}.
2. Implement a CUDA kernel for community
    detection in a graph with {size} nodes using
     the {community_algorithm} algorithm.
3. Implement a CUDA kernel for graph processing
    that computes {algorithm} on a graph with {
    size} nodes. Optimize for {optimization}.
4. Generate a CUDA kernel for finding strongly
    connected components in a directed graph
    with {size} nodes. Optimize for {
    optimization}.
5. Create a CUDA implementation for breadth-
    first traversal on a graph with {size} nodes
     stored in {graph_format}. Optimize for {
    optimization}.
```

Scientific Computing

```
1. Write a CUDA implementation for {size}D fluid
     simulation using {method}. Focus on {
    optimization}.
2. Create a CUDA kernel for Monte Carlo
    simulation of {size} paths for option
    pricing. Focus on {optimization}.
3. Implement a CUDA solver for {size}x{size}
    sparse linear system using {linear_solver}.
    Focus on {optimization}.
4. Generate a CUDA implementation for {size}D
    heat equation solver using {numerical_method
    }. Optimize for {optimization}.
5. Create a CUDA kernel for molecular dynamics
    simulation of {size} particles using {
    md_algorithm}. Optimize for {optimization}.
```

Machine Learning

```
1. Generate a CUDA kernel for k-means clustering
     of {size} data points in {dimension}D space
    . Optimize for {optimization}.
2. Implement a CUDA kernel for {size}x{size}
    matrix factorization using {
    factorization_method}. Optimize for {
    optimization}.
3. Create a CUDA implementation for computing
    attention mechanism in a transformer with {
    size} tokens. Focus on {optimization}.
4. Implement a CUDA kernel for backpropagation
    in a convolutional neural network with {
    conv_layer_count} conv layers. Optimize for
    {optimization}.
5. Write a CUDA implementation for training a
    neural network with {layer_count} layers and
     {neuron_count} neurons per layer. Focus on
    {optimization}.
```

Sparse Operations

```
1. Generate a CUDA kernel for sparse FFT
    computation. Optimize for {optimization}.
2. Implement a CUDA kernel for sparse tensor
    operations with {size} non-zero elements.
    Optimize for {optimization}.
3. Write a CUDA implementation for sparse
    convolution with {size}x{size} filter on
    sparse input. Focus on {optimization}.
4. Create a CUDA implementation for sparse
    matrix-matrix multiplication in {
    sparse_format} format. Focus on {
    optimization}.
5. Generate a CUDA kernel for sparse matrix-
    vector multiplication where the matrix has
    approximately {size} non-zero elements.
    Optimize for {optimization}.
```

Simulation

```
1. Generate a CUDA kernel for cloth simulation
    with {size}x{size} grid. Optimize for {
    optimization}.
2. Write a CUDA implementation for raytracing of
     a scene with {size} objects. Focus on {
    optimization}.
3. Create a CUDA implementation for {algorithm}
    of {size} particles in a {dimension}D space.
     Focus on {optimization}.
4. Create a CUDA implementation for fluid-
    structure interaction with {size} boundary
    elements. Focus on {optimization}.
5. Implement a CUDA kernel for N-body simulation
     of {size} particles using {nbody_algorithm
    }. Optimize for {optimization}.
```

Image and Signal Processing

```
1. Create a CUDA implementation for feature
    extraction from {size}x{size} images. Focus
    on {optimization}.
2. Generate a CUDA kernel for image segmentation
     using {segmentation_algorithm}. Optimize
    for {optimization}.
3. Write a CUDA implementation for real-time
    video processing of {resolution} frames.
    Focus on {optimization}.
4. Implement a CUDA kernel for signal processing
     with {size}-point {signal_transform}.
    Optimize for {optimization}.
5. Implement a CUDA kernel for image filtering
    using {filter_type} filter of size {
    filter_size}x{filter_size}. Optimize for {
    optimization}.
```

Optimization Algorithms

```
1. Implement a CUDA kernel for simulated
    annealing with {size} states. Optimize for {
    optimization}.
2. Generate a CUDA kernel for genetic algorithm
    with population size {size}. Optimize for {
    optimization}.
3. Write a CUDA implementation for {
    optimization_algorithm} with {size}
    variables. Focus on {optimization}.
4. Write a CUDA implementation for gradient
    descent optimization with {size} parameters.
     Focus on {optimization}.
5. Create a CUDA implementation for particle
    swarm optimization with {size} particles in
    {dimension}D space. Focus on {optimization}.
```

Cryptography and Security

```
1. Generate a CUDA kernel for homomorphic
    encryption operations. Optimize for {
    optimization}.
2. Write a CUDA implementation for secure
    hashing using {hash_algorithm}. Focus on {
    optimization}.
3. Generate a CUDA kernel for {crypto_algorithm}
     encryption/decryption. Optimize for {
    optimization}.
4. Create a CUDA implementation for blockchain
    mining with difficulty {size}. Focus on {
    optimization}.
5. Implement a CUDA kernel for password cracking
     using {cracking_method}. Optimize for {
    optimization}.
```

Data Structures

```
1. Create a CUDA implementation for priority
    queue with {size} elements. Focus on {
    optimization}.
2. Create a CUDA implementation for {
    data_structure} with {size} elements. Focus
    on {optimization}.
3. Implement a CUDA kernel for operations on a B-
    tree with {size} nodes. Optimize for {
    optimization}.
4. Generate a CUDA kernel for skip list
    operations with {size} elements. Optimize
    for {optimization}.
5. Write a CUDA implementation for hash table
    with {size} buckets using {
    collision_strategy}. Focus on {optimization
    }.
```

A.8.2 Qualitative Comparison with Other LLMs

We highlight several cases where CASS-7B outperforms existing LLMs such as Claude, Qwen-Coder, and GPT-4o in faithfully transpiling CUDA to HIP. For example, in one instance, CASS-7B correctly transpiled the CUDA code while preserving the exact string constants from the original program, including the label `CUDA` in the output format string. Maintaining these strings is essential for preserving the intended user-facing behavior, particularly in logging or debugging scenarios where clarity and consistency matter. In contrast, Claude, Qwen-Coder, and GPT4o unnecessarily altered the string to say `HIP`, despite the output still originating from a CUDA kernel. This substitution introduces a semantic error, as the original string refers to CUDA, not HIP, and should remain unchanged.

CASS-7B

```
printf("tanh(%f) = %f CUDA vs %f (CPU)\n",
       h_input[idx], h_output[idx], tanh(h_input
    [idx]));
```

Claude, Qwen-Coder, GPT4o

```
printf("tanh(%f) = %f (HIP) vs %f (CPU)\n",
       h_input[idx], h_output[idx], tanh(h_input
    [idx]));
```

In another example, CASS-7B retained the classical CUDA-style kernel launch syntax using triple angle brackets (`<<<...>>>`), while also ensuring that the generated code remained compilable by correctly including the required HIP header `<hip/hip_runtime.h>`. This demonstrates a high degree of structural fidelity to the source code, which is especially important for developers familiar with standard CUDA conventions. In contrast, other models such as Claude and Qwen-Coder replaced the launch expression with the HIP-specific macro `hipLaunchKernelGGL`, which, while functionally valid, deviates from the original representation. More critically, they failed to include the necessary HIP header, rendering the output uncompilable. This example highlights how CASS-7B goes beyond syntactic accuracy to produce code that is both faithful to the original structure and immediately usable in a real compilation setting.

CASS-7B

```
#include <hip/hip_runtime.h>
#include <iostream>
...
add<<<(N + 255) / 256, 256>>>(d_a, d_b, d_c, N);
```

Claude, Qwen-Coder

```
#include <iostream>
...
hipLaunchKernelGGL(add, (N + 255) / 256, 256,
    0, 0, d_a, d_b, d_c, N);
```

Lastly, when verifying numerical correctness, CASS-7B preserved the original logging behavior by correctly emitting output to `std::cout`, as in the source code. This choice maintains consistency with the original program's semantics, especially in distinguishing between standard output and error streams; important in contexts where output may be redirected or parsed. In contrast, GPT-4o unnecessarily altered the output stream to `std::cerr`, which, while syntactically valid, changes the runtime behavior of the program. Such a change could lead to unexpected side effects in downstream tools or logging pipelines. This example further demonstrates CASS-7B's attention to both structural and behavioral fidelity in its translations.

CASS-7B

```
std::cout << "Error at element " << i << ": " <<
      h_output[I]
          << " vs. expected " << h_reference[i]
    << std::endl;
```

GPT4o

```
std::cerr << "Error at element " << i << ": " <<
      h_output[i]
          << " vs expected " << h_reference[i]
    << std::endl;
```

A.9 Qualitative Analysis of the Failure Cases

We analyze common failure modes in assembly translation by examining representative cases where the model produces syntactically invalid RDNA3 code. These failures reveal critical challenges in learning GPU assembly constraints that are architectural rather than semantic in nature. The six examples below illustrate distinct categories of errors: register alignment violations (Example 1), bank constraint mismatches (Example 2), instruction hallucination (Example 3), literal operand restrictions (Example 4), control flow inconsistencies (Example 5), and operand type mismatches (Example 6). Each case demonstrates that while the model captures high-level translation patterns, it struggles with vendor-specific encoding rules that are not easily inferred from statistical patterns alone. These constraints, documented sparsely in AMD's ISA manuals and often only enforced at assembly time, represent a key bottleneck in achieving higher accuracy.

Example 1: Invalid Register Alignment

Predicted Assembly:

```
...
    v_ashrrev_i32_e32 v2, 31, v1
    s_ashr_i32 s5, s4, 31
    s_mov_b32 s3, s4
    s_delay_alu instid0(SALU_CYCLE_1) | instskip(NEXT) |
       instid1(VALU_DEP_1)
    s_lshl_b64 s[4:5], s[3:4], 2
    v_lshlrev_b64 v[0:1], 2, v[1:2]
    s_waitcnt lgkmcnt(0)
...
```

Error:

- `error: invalid register alignment`
- Register pair `s[3:4]` requires even/odd alignment

Ground Truth (Correct):

```
...
    s_load_b128 s[4:7], s[0:1], 0x0
    v_ashrrev_i32_e32 v2, 31, v1
    s_delay_alu instid0(VALU_DEP_1) | instskip(SKIP_1) |
       instid1(VALU_DEP_1)
    v_lshlrev_b64 v[0:1], 2, v[1:2]
    s_waitcnt lgkmcnt(0)
...
```

Example 2: VGPR Bank Constraint Violation

Predicted Assembly:

```
...
    v_mov_b32_e32 v4, 0x1b8
.LBB0_2:
    s_delay_alu instid0(VALU_DEP_1) | instskip(SKIP_1) |
       instid1(SALU_CYCLE_1)
    v_dual_mov_b32 v11, v2 :: v_dual_mov_b32 v2, v6
    s_add_i32 s5, s4, 4
    s_add_i32 s4, s4, 8
...
```

Error:

- `error: src0 operands must use different VGPR banks`
- Dual instructions require operands from different banks

Ground Truth (Correct):

```
...
    v_mov_b32_e32 v4, 0x1b8
.LBB0_2:
    s_delay_alu instid0(VALU_DEP_1)
    v_mov_b32_e32 v11, v2
    v_mov_b32_e32 v2, v6
    s_add_i32 s5, s4, 4
...
```

Example 3: Hallucinated Instruction

Predicted Assembly:

```
...
    v_add_nc_u32_e32 v0, 5, v0
    ds_store_b32 v3, v0
    s_waitcnt lgkmcnt(0)
    sumba_send_msg_b32 v0, v3
    s_endpgm
.LBB0_3:
...
```

Error:

- `error: invalid instruction`
- `sumba_send_msg_b32` does not exist in AMD ISA

Ground Truth (Correct):

```
...
    v_add_nc_u32_e32 v3, 5, v3
    ds_store_b32 v0, v3
.LBB0_4:
    s_or_b32 exec_lo, exec_lo, s0
    s_waitcnt lgkmcnt(0)
    s_barrier
...
```

Example 4: Multiple Literal Operand Constraint

Predicted Assembly:

```
...
    s_delay_alu instid0(SALU_CYCLE_1) | instskip(NEXT) |
       instid1(SALU_CYCLE_1)
    s_or_b32 s12, s8, s2
    s_cmp_eq_u32 s12, s9
    s_cselect_b32 s12, 0x88, 0x100
    s_cmp_eq_u32 s13, 0
    s_cselect_b32 s13, s9, 0
...
```

Error:

- `error: only one unique literal operand is allowed`
- Cannot use both `0x88` and `0x100` as literals

Ground Truth (Correct):

```
...
    s_or_b32 s12, s8, s2
    s_cmp_eq_u32 s12, s9
    s_mov_b32 s12, 0x88
    s_cselect_b32 s12, s12, 0x100
    s_cmp_eq_u32 s13, 0
...
```

Example 5: Missing Label Definition

Predicted Assembly:

```
...
    v_lshl_add_u32 v9, v0, 2, 0
    s_mov_b32 s3, exec_lo
    v_cmpx_gt_u32_e64 v0, v1
    s_cbranch_execz .LBB0_8
; %bb.4:
    v_lshl_add_u32 v2, s4, 3, v9
...
```

Error:

- `error: undefined label '.LBB0_8'`
- Branch target label was never defined in assembly

Ground Truth (Correct):

```
...
    v_lshl_add_u32 v9, v0, 2, 0
    s_mov_b32 s3, exec_lo
    v_cmpx_gt_u32_e64 v0, v1
    s_cbranch_execz .LBB0_6
; %bb.4:
    v_lshl_add_u32 v2, s4, 3, v9
...
```

Example 6: Invalid Operand Type

Predicted Assembly:

```
...
    s_mov_b32 s6, 0
    s_waitcnt lgkmcnt(0)
    s_load_b32 s5, s[2:3], 0x0
    v_cmp_gt_u64_e64 s2, 0x800001000, s[4:5]
    s_delay_alu instid0(VALU_DEP_1)
    s_and_b32 s2, s2, exec_lo
...
```

Error:

- `error: invalid operand for instruction`
- Immediate `0x800001000` exceeds 32-bit limit for this encoding

Ground Truth (Correct):

```
...
    s_mov_b32 s6, 0
    s_waitcnt lgkmcnt(0)
    s_load_b32 s5, s[2:3], 0x0
    v_cmp_gt_u64_e64 s2, s[4:5], 0x7fffffff
    s_delay_alu instid0(VALU_DEP_1)
    s_and_b32 s2, s2, exec_lo
...
```

A.10 Quantitative Analysis of the Failure Cases

To complement the qualitative analysis, we provide a quantitative breakdown of compilation errors across all models and failure cases. Figure 4 shows the aggregate distribution of error types, which reveal that invalid or unknown instructions (3745 occurrences) constitute the most common failure mode, followed by directive parsing errors (2576) and operand constraint violations (2024). These patterns confirm that the primary challenge in assembly translation lies not in high-level semantic understanding but in learning the precise encoding rules and instruction constraints specific to RDNA3. Also, register alignment and instruction constraint errors, while less frequent in absolute terms (238 and 226 occurrences respectively), represent categorical failures that prevent any execution, whereas invalid operands may sometimes be caught and corrected through iterative refinement. The relatively low occurrence of control flow errors, such as undefined labels (36 cases), suggests that the model successfully captures basic program structure even when it fails on instruction-level details. This distribution motivates our focus on architectural constraints in the qualitative examples below.

Figure 14 presents a model-wise breakdown of error distributions across 17 models, enabling direct comparison of failure modes across different architectures, scales, and training paradigms. We observe several patterns from this analysis. First, our CASS models trained on A100 data (sm80) consistently exhibit fewer invalid instruction errors compared to all baselines, including frontier models (GPT-5.1, Gemini-2.5-Pro, Gemini-2.5-Flash), coding-specialized models (Qwen2.5-Coder-32B, DeepSeek-Coder-V2), and general instruction-tuned models (Llama-3.1-8B, Gemma-3-27B). This suggests that domain-specific training on aligned CUDA-HIP pairs provides better coverage of the RDNA3 instruction space than general pretraining or code-focused curricula. Second, proprietary frontier models show disproportionately high rates of invalid instruction hallucination, with GPT-5.1 and Gemini-2.5-Flash generating over 800 invalid instructions each, nearly 4× higher than our best CASS model. This is consistent with their limited exposure to GPU assembly during pretraining, as these models were optimized for high-level code generation rather than low-level ISA translation. Interestingly, even coding-specialized open-source

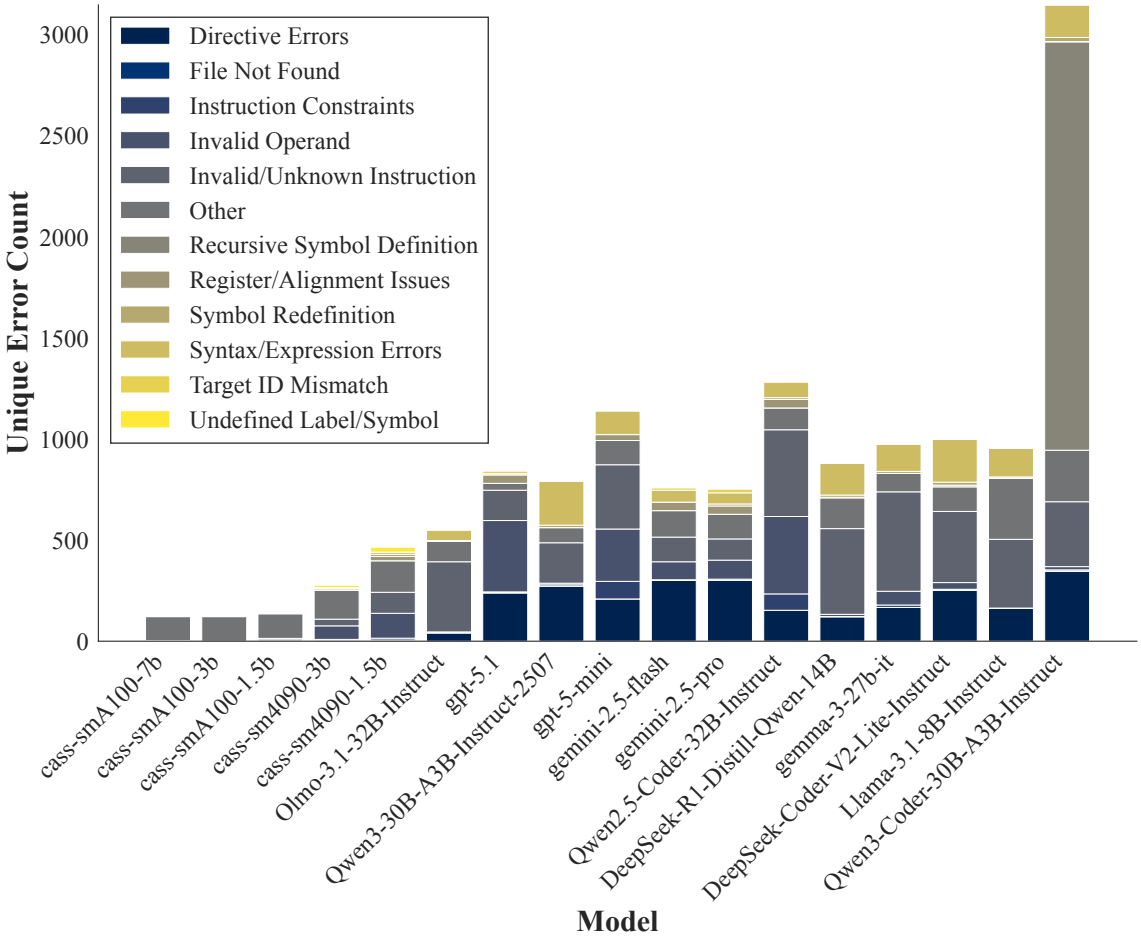

Figure 14: **Model-wise distribution of compilation errors across different models.** CASS models trained on A100 data exhibit significantly fewer invalid instruction and operand errors compared to all baselines, including frontier models (GPT-5.1, Gemini-2.5-Pro), coding-specialized models (Qwen2.5-Coder-32B, DeepSeek-Coder-V2), and general instruction-tuned models (Llama-3.1, Gemma-3). Scaling from 1.5B to 7B parameters reduces constraint violations by 35-40%. Models trained on sm89 (4090) show elevated error rates due to ISA divergence.

models like Qwen2.5-Coder-32B and DeepSeek-Coder-V2-Lite fail at similar rates, indicating that generic code pretraining does not transfer to GPU assembly domains. Third, scaling from 1.5B to 7B parameters within our CASS family reduces invalid operand errors by approximately 40% and instruction constraint violations by 35%, indicating that larger models better internalize complex operand encoding constraints and vendor-specific ISA rules. However, the gains diminish beyond 3B parameters, suggesting that architectural knowledge rather than pure capacity becomes the bottleneck. Fourth, models trained on sm89 (RTX 4090) show elevated rates of instruction constraint violations compared to their sm80 counterparts, with CASS-sm4090-3B exhibiting 28% more constraint errors than CASS-smA100-3B. This reflects the ISA divergence between Ada Lovelace and Ampere architectures discussed in Section 6, where sm89-specific optimizations fail to generalize to RDNA3. Finally, reasoning-focused models like DeepSeek-R1-Distill-Qwen-14B do not show significant advantages over standard instruction-tuned models, suggesting that assembly translation requires pattern matching over low-level encoding rules rather than high-level reasoning capabilities. The near-uniform performance of general instruction-tuned models (Llama, Gemma, Olmo) at the bottom of the ranking confirms that assembly translation is a specialized skill requiring domain-specific training data.