GUIDED REASONING ACROSS ISA DIALECTS: INFERENCE-TIME CONTROL FOR GPU ASSEMBLY GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

GPU software is increasingly shipped as compiled NVIDIA binaries, while the hardware that runs it is diversifying. Porting such software to another vendor usually requires source code that is often unavailable, or a hand-written binary translator for each instruction set. Large reasoning models can translate native GPU assembly without task-specific training, but not reliably: programs are long, reasoning and translation share one output budget, and the compiler can confirm that a translation is legal but not that it is correct. We treat translation as inference-time control of a stochastic model and present GRIT, which translates the native assembly in an NVIDIA executable into an AMD executable. GRIT splits programs into kernel translation units, grounds the model in context recovered from the binary, repairs rejected candidates with compiler diagnostics, and uses a learned Reasoning Controller to choose effort, context, repair budget, and attempts per program. On 375 programs from three benchmark suites, GRIT raises Gemini 3.8 Flash from 4.0% for whole-program translation to 77.3% in a single attempt, and the controller reaches 88.0% on a held-out test set with repeated attempts. We also find that more reasoning is not free: high effort exhausts the response limit six times as often as medium.

1 INTRODUCTION

GPU hardware is becoming more diverse, while the software written for it is not. AMD accelerators now power several of the largest supercomputers (TOP500, 2026; Atchley et al., 2023) and are offered alongside NVIDIA GPUs by major clouds, with many organizations now running mixed vendor fleets. Much of the GPU software ecosystem, however, was written for CUDA and distributed as NVIDIA-specific binaries: an application that runs well on one vendor's hardware often cannot run on another's. When source is available, AMD's HIPIFY (Advanced Micro Devices, 2026) can port it, though this still takes engineering effort and a full build. Often, parts of the stack exist only in compiled form: commercial applications, vendor libraries, and unmaintained research code. Such software can reach another accelerator only by translating its machine code. Binary translation has a long history on CPUs (Ebcioglu & Altman, 1997; Dehnert et al., 2003; Bellard, 2005; Apple Inc., 2020), but relies on hand-written rules for each pair of instruction set architectures (ISAs). GPU ISAs make this especially difficult: NVIDIA's assembly (SASS) is only partially documented (NVIDIA Corporation, 2026c); both vendors revise their ISAs each generation; and correct translation depends on execution-model details like register allocation and calling conventions.

Large language models (LLMs) offer an alternative to hand-written rules. Specialized models can *learn* instruction-set mappings from paired assembly corpora (Lee et al., 2024; Heakl et al., 2026). General-purpose reasoning models (LRMs) are not trained for assembly translation, but they have seen a great deal of low-level code and can *reason* their way through the details of an unfamiliar target, letting them translate across ISAs without fine-tuning (Gemini Team, 2024; Guo et al., 2025).

Having a capability is not the same as using it reliably: our first attempts to put LRM reasoning to work on this problem failed. Prompting Gemini 3.8 Flash on 375 kernels from established CUDA benchmarks produced only 4.0% correct AMD programs (successfully assembled, executed

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

on AMD hardware, and matching the original's output). Assembly leaves little room for error: one misnumbered register or instruction can make a program wrong or unable to run.

A closer look, however, suggests an LRM was often closer to success than the raw numbers imply. For example, the same kernel, translated twice under identical settings, could succeed once and fail once (Figure 7, Appendix G); raising the reasoning budget often exhausted the response limit before completion (§5.5); and many rejected translations were only a few instructions from a correct one: 81.5% of rejected units were resolved by a single repair turn (§5.3). These observations point to *how* the model is used at inference time, not what it knows, as a large part of the problem. The usual ways of spending inference-time compute, namely longer reasoning, repeated sampling, and iterative repair from execution or compiler feedback (Snell et al., 2025; Brown et al., 2024; Li et al., 2022; Chen et al., 2024), assume short programs whose candidates can be checked against unit tests or reference outputs (Chen et al., 2021). For realistic binary portability across GPU vendors, neither condition holds. A kernel and its helper functions can span tens to hundreds of thousands of tokens, so input, reasoning, and output compete for a fixed budget. And the only way to know that a translation is correct is to run it against a reference output, which does not exist at deployment; the compiler can tell us whether a translation is legal, but not whether it is right. This leaves three practical questions open: how much to reason before writing, what to show when the assembler rejects output, and how many candidates to generate when none can be checked?

We study these questions by treating low-level translation as *inference-time control of a stochastic generator*. Rather than asking whether a model can translate a kernel, we view it and its inference configuration as inducing a *distribution* over candidate programs, some correct, some not. Inference-time control then amounts to shaping this distribution and sampling from it using only deterministic signals. We examine three levers: reasoning allocation, a compiler-guided repair loop returning assembler diagnostics after a failed attempt, and bounded independent sampling. We implement this approach in GRIT (**G**uided **R**easoning for **I**SA **T**ranslation). At its core is a *Reasoning Controller* (RC), a lightweight learned policy setting these levers and the model's context from properties of the input binary (Figure 2). With GRIT, the same LRM translates 19× more programs to AMD than a single whole-program attempt (77.3% against 4.0%, both single-attempt).

In summary, this paper makes the following contributions:

- **More reasoning is not free.** Reasoning and output tokens are drawn from the same response limit, and an unfinished translation is unusable. More reasoning can leave too little room to finish: at high effort, Gemini 3.8 Flash exhausts its limit on 6× as many programs as at medium, gaining only 5.8 points of accuracy for 59% more reasoning tokens (§5.5).
- **The compiler is an inexpensive and effective critic.** Returning LLVM assembler diagnostics to the model and allowing corrections raises the fraction of correct translations from 32.3% to 77.3%. Because diagnostics identify the offending instruction, the model edits a nearly correct translation instead of regenerating it, at a fraction of the original cost (§5.3).
- **Single attempts understate capability.** With repairs, a single attempt yields a correct program for 68.3% of programs, whereas three independent attempts, evaluated by an oracle, reach 78.9%, 40 more programs (§5.4).
- **A learned Reasoning Controller.** The RC then selects the reasoning level, repair budget, context mode, and sample count for each program only based on the static properties of the binary, implemented as a decision tree and random forest. With the same LRM, it correctly translates 88% of held-out programs, 2.7× as many as a single attempt at medium effort without repair (32.3%), at 1.8× its median reasoning tokens (§5.5).

2 Background and Related Work

Source-Free GPU Kernel Translation. NVIDIA's toolchain lowers CUDA device code to PTX, a virtual ISA, and compiles PTX into architecture-specific SASS stored in ELF-format CUBIN files (NVIDIA Corporation, 2026a;b;c). A deployed executable embeds these images in a fatbinary that may hold CUBINs, PTX, or both (Figure 1), and what survives determines which porting strategies remain: prior source-free approaches require CUDA source, compiler IR, or retained PTX, while CuLifter (Zhao et al., 2026) and CASS (Heakl et al., 2026) work from SASS alone (Appendix A).

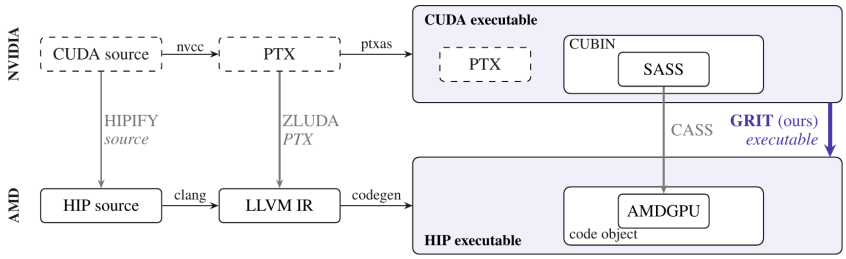


Figure 1: CUDA-to-AMD translation at the source, PTX, and native-assembly levels. GRIT translates native SASS into AMDGPU instructions without CUDA source or PTX.

We study this most constrained setting: the input is a deployed CUBIN, with no CUDA source or PTX. A runnable AMD program then needs two kinds of work. The *semantic* part rewrites each SASS function as AMDGPU instructions that compute the same result under a different register file, memory hierarchy, and calling convention. The *structural* part wraps the translated functions in a code object with the metadata and kernel descriptors the AMD runtime expects (LLVM Project, 2024). The model does only the semantic part; the structural part is recovered from the binary and assembled deterministically (§3), so failures come mainly from the model's translation.

Learned Assembly Translation. Language models have been applied to low-level code in several settings (Lee et al., 2024; Heakl et al., 2025; Armengol-Estapé et al., 2024; Tan et al., 2024; 2026; Jiang et al., 2025; TehraniJamsaz et al., 2024; Cummins et al., 2024) (Appendix A). For GPUs, CASS fine-tunes specialist models for SASS-to-AMD translation and reports that accuracy declines as input length grows (Heakl et al., 2026). These systems fine-tune a model for the task and evaluate what it produces, not how inference is carried out. We instead use general-purpose reasoning models without task-specific training and study inference itself.

2.1 Inference-Time Control of Reasoning Models

Reasoning and output budgets. Reasoning models deliberate before answering, a behavior elicited by prompting or developed through reinforcement learning (Wei et al., 2022; Guo et al., 2025). Extra inference-time computation does not help uniformly: allocating it by estimated problem difficulty outperforms a uniform allocation (Snell et al., 2025). Our Reasoning Controller (§3.6) follows this principle but predicts the allocation from static features of the binary before any model call. Providers expose this computation as a reasoning level that is not comparable across models; in the Gemini API, reasoning and answer tokens share one output limit (Google, 2026), so more reasoning leaves less room for the answer. A long context window also does not guarantee that the information in it is used reliably (Liu et al., 2024).

Sampling and selection. Drawing several independent samples and keeping the best raises the chance of a correct answer substantially (Chen et al., 2021; Li et al., 2022; Brown et al., 2024). Such gains are usually reported as pass@N, which assumes a correct sample can be recognized once generated, by unit tests, hidden tests, or a reference output. GRIT has no such signal during generation: the assembler and static checks establish that a translation is legal, not that it computes the right result. It therefore keeps every validated sample (§3.5) and defers selection to offline testing against the original executable. We report pass@1 and this oracle-checked pass@N (§5.4).

3 Approach

GRIT translates the native NVIDIA device code in a deployed CUBIN into executable AMD device code, without CUDA source or embedded PTX. Following Section 1, it places a general-purpose reasoning model, treated as a stochastic generator, inside an otherwise deterministic pipeline: binary facts, decomposition, legality checks, and packaging are handled outside the model, which only translates the semantics of each function from SASS into AMDGPU assembly (Figure 2). A planner splits the CUBIN into Kernel Translation Units (KTUs, §3.2) and attaches context recovered from

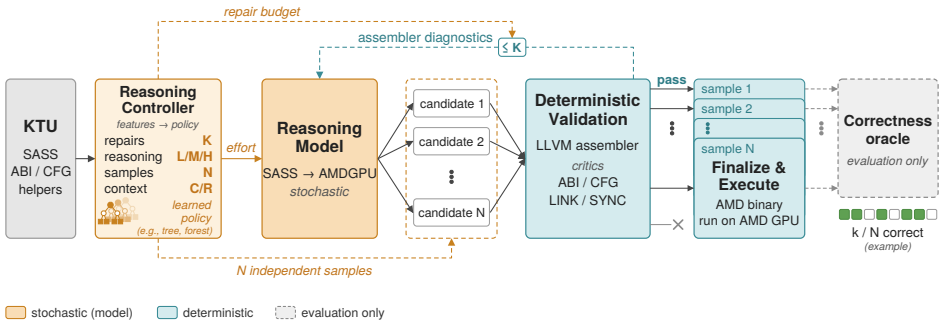


Figure 2: **Overview of GRIT.** The Reasoning Controller sets an inference policy for each program from static features. The model's candidates are validated deterministically, repaired from assembler diagnostics up to K times, and every accepted sample is finalized and executed. The oracle is used only for evaluation.

the binary (§3.3). The Reasoning Controller chooses one inference policy per program before any model call (§3.6). Candidates are checked by the LLVM AMDGPU assembler and structural critics; rejected ones return to the model with diagnostics for a bounded number of repair turns (§3.4), and N independent samples provide further candidates (§3.5). A deterministic finalizer packages each sample into an AMD executable (§3.7), which is evaluated against the oracle.

3.1 Problem Formulation

GRIT is controlled by an inference policy $\pi = (e, K, m, N)$, chosen once per program: reasoning effort e, up to K repair turns per KTU, context mode m (§3.3), and N independent whole-program samples. Each candidate translation is checked by a deterministic validator V, available at inference time, which establishes legality, not correctness; only an execution-based correctness oracle O, available only for evaluation, determines whether a sample is actually right. Inference-time control is therefore the problem of choosing π, using only the model's inputs and V, to maximize the probability that at least one of the N samples satisfies O (full formalization in Appendix B).

3.2 Kernel Translation Units (KTUs)

GRIT splits a program along its statically recoverable call structure, not by length or instruction count. The planner builds one KTU per selected kernel, containing the kernel and every function it can statically reach (if kernel A calls B and B calls C, the KTU holds all three), found from direct call targets, relocations, and symbol references in the binary. No request asks the model to translate a kernel without its helpers, and each is bounded by one kernel's call closure rather than the whole module, which shortens both the input and the output the model must emit within its shared budget (§2.1). A KTU is never split during repair; unresolved calls and module-wide data are handled later.

3.3 Executable-Derived Context

The model never sees source code. Its input is SASS disassembled from the CUBIN, together with facts recovered deterministically from the surrounding ELF structure (Table 1). Some facts cannot be inferred from SASS, which describes only the source convention: `threadIdx.x` is read from a special register, whereas RDNA3 packs work-item coordinates into `v0`, so the AMD launch ABI is supplied explicitly. **Compact** mode supplies what translation requires; **rich** mode is a strict superset that adds static analyses and module data the SASS leaves implicit. In both, the model receives only information recoverable from the binary and is never asked to infer ABI, resource, or symbol information it could get wrong.

Table 1: Context supplied to the model. Rich mode is a strict superset of compact mode; its column lists only the fields it adds. Resource metadata covers register count, static shared memory, local memory, and stack size; extended CUDA metadata covers parameter offsets, sizes, and alignments, raw `.nv.info` records, and relocation records.

Compact	**Rich (adds)**
Disassembled SASS	Control-flow graph
Resource metadata	Register liveness
Kernel arguments and AMD launch ABI	Symbol table and relocations
Call graph and unresolved symbols	Constant initializer bytes
Constant and global references	Extended CUDA metadata

3.4 Compiler-Guided Repair

The validator V (Figure 2) has two layers: the LLVM AMDGPU assembler, which rejects illegal mnemonics, wrong-target instructions, invalid register classes, unsupported modifiers, malformed operands, and illegal immediates; and six critics for truncation, linking, ABI, and synchronization errors the assembler misses. The assembler supplies about 94% of the 1,688 diagnostics across 451 repair calls. A rejected candidate becomes structured feedback: the diagnostic type, the affected function, and the compiler's message verbatim. In HeCBench's `affine` kernel, for example, the assembler rejected one of 85 instructions, `s_lshrrev_b32 s6, 16, s6`, with `invalid instruction, did you mean: v_lshrrev_b32?`, since the reversed-operand shift exists only in the vector ISA; the repair turn replaced it with `s_lshr_b32 s6, s6, 16` and left the other 84 unchanged. A repair turn receives the original context, the previous candidate, and this feedback, and returns a complete replacement without changing the ABI, target, kernel descriptor, metadata, or runtime behavior. Because the diagnostic locates the error, it needs far less reasoning than the initial translation: over the 363 units that needed repair, median reasoning falls from 36,983 to 8,443 tokens and latency from 134 s to 44 s, while the input grows by 31% to carry the previous candidate. Repair never sees program outputs, NVIDIA results, or the oracle, so an accepted candidate is known to be legal AMDGPU assembly, not known to be correct.

3.5 Independent Sampling

Because the model is stochastic, identical inputs and settings can yield different translations. GRIT therefore draws N independent samples of the whole program, each rerunning one initial turn and up to K repairs per KTU and sharing no candidates or diagnostics with the others. Sampling is per program rather than per KTU, since combining per-unit candidates grows multiplicatively with the number of units, and every sample passes through the same validation and finalization. Because V checks only legality, it cannot tell which sample is correct. We therefore report pass@1, the expected success of a single sample, and pass@N, the fraction of programs with at least one correct sample among N. Pass@N requires running the original executable, so it is an upper bound when the NVIDIA hardware is no longer available.

3.6 Reasoning Controller

No fixed setting of effort, repair budget, and context mode serves every program, and the right choice depends on properties visible before translation. The Reasoning Controller (RC) therefore chooses $\pi = (e, K, m, N)$ (§3.1) for each program before any model call, from static features of the program's size and resources computed from the translation request (Appendix D). Any deterministic function from these features to a policy can serve, provided it is fixed before evaluation. We implement two: a shallow decision tree (depth 4, 15 leaves), whose every estimate traces to a few input properties, and a random forest of 400 trees, which trades this transparency for capacity (§5.5). Both estimate the probability that a program passes under a candidate configuration and select the highest; N grows as the predicted pass probability of the chosen configuration falls. Both are trained on twelve fixed-configuration campaigns over 283 development programs and frozen before evaluation on a held-out test set of 92 programs, split by program so that no test kernel is seen in training. During evaluation, a program is retranslated up to N times and stops at the first attempt that passes the oracle, so the resulting accuracy is pass@N (Table 2).

3.7 FINALIZATION

For each sample in which every KTU has an accepted candidate, the finalizer links the translated functions, resolves symbols and relocations, emits the kernel descriptors and metadata the loader requires, and validates the resulting code object. The unmodified host program then runs without the model or the original NVIDIA binary (Appendix E), so translation is a one-time offline cost.

4 EVALUATION

We report end-to-end accuracy against source-, PTX-, and SASS-level baselines (§5.1), then isolate the effects of executable-derived context and KTUs, compiler-guided repair, and independent sampling, evaluate the Reasoning Controller, and analyze the reasoning budget.

4.1 BENCHMARKS AND CORRECTNESS ORACLE

We evaluate on three suites of paired CUDA and HIP programs, using each program's native HIP compile as a no-translation reference. **CASS-bench-v2** (Heakl et al., 2026) contributes 190 admitted programs; **HeCBench** (Jin & Vetter, 2023) contributes 170 hipified programs from `ivanradanov/rodinia` (Ivanov, 2016); **Rodinia** (Che et al., 2009) contributes 15 classic Rodinia 3.1 programs from the same repository (admission and filtering criteria in Appendix F).

Evaluating O (§3.1) requires a stable reference, so we run every reference under fixed inputs, a fixed environment, and a single GPU, normalizing away run-dependent output such as timing and device banners. *CASS-bench-v2 and HeCBench* admit a program, and pass a translation, only when native CUDA and HIP output matches after normalization; *Rodinia* instead compares translated device result arrays with the CUDA original's using its built-in verifiers and tolerances (full normalization and verifier details in Appendix F).

In all suites, O certifies equivalence to the original program, not algorithmic correctness: three admitted CASS-bench-v2 programs fail their own self-check identically on both GPUs, and a translation that reproduces this output passes.

4.2 EXPERIMENTAL SETUP

Binaries are compiled for SM89 with `-Xptxas -O0` (SM80 for the MI210 and CASS runs). Unless noted, Gemini 3.8 Flash runs at medium effort with a 65,536-token response limit and targets RDNA3 (gfx1100), or MI210 (gfx90a) for CDNA2; all runs share one prompt, with no worked example, and one deterministic backend. We compare two controlled baselines with the same model: whole-file translation in one request, and GRIT with one turn per KTU and no repair. We run CASS (Heakl et al., 2026), the closest learned baseline, locally from its released checkpoint (Heakl, 2025), on whole programs and inside GRIT's KTUs. Retargeting to CDNA2 changes only target-dependent parts (the launch ABI, kernel descriptor, device runtime, and target description in the prompt); no model is retrained. SCALE (Spectral Compute, 2026), which recompiles the full CUDA source with a proprietary toolchain, and ZLUDA (Janik, 2026), which JIT-compiles embedded PTX, show what source- and PTX-level approaches achieve under the same oracles. Each condition is run once, except pass@3 (three runs), the controller (up to N attempts), and the response-limit study (three runs on HeCBench); incomplete generations count as failures. Appendix C gives full details.

5 RESULTS

5.1 OVERALL ACCURACY

Table 2 reports end-to-end accuracy for every method and suite. With rich input and up to three repairs per unit, GRIT translates 77.3% of the 375 programs correctly in a single attempt, while the same model translating whole programs in one request reaches at most 4.0%. Among single-attempt methods that use only SASS, this configuration is the most accurate on every suite. Three independent runs with compact input cover 78.9% of programs (pass@3), and the Reasoning Controller reaches 88.0% on the held-out test set (pass@N). SCALE is the most accurate method overall

Table 2: End-to-end accuracy (%); *All* covers all 375 programs. SCALE takes CUDA source and uses a proprietary toolchain, ZLUDA takes PTX, and all other methods take only SASS. CASS and Gemini translate whole programs; GRIT translates KTUs, with Gemini unless noted. *Basic* is the SASS-to-RDNA prompt that CASS uses for inference. *Repairs* counts compiler-guided repair turns per unit. Bold: main results of GRIT, with a fixed configuration (three repairs) and with the RC. [a]Correct if any of three runs is correct. [b]Configuration and attempt budget chosen per program by the Reasoning Controller (pass@N); held-out test set only. [c]CDNA2 target (MI210); rich input with compact fallback.

Method	Input	Repairs	CASS-v2	HeCBench	Rodinia	All
SCALE	CUDA source	–	93.7	97.1	100.0	95.5
ZLUDA	PTX	–	66.3	94.1	93.3	80.0
CASS	Basic	0	54.7	2.4	0.0	28.8
	Compact	0	20.5	1.2	0.0	10.9
	Rich	0	16.3	1.2	0.0	8.8
Gemini	Basic	0	6.3	1.8	0.0	4.0
	Compact	0	3.7	1.2	0.0	2.4
	Rich	0	3.2	2.4	0.0	2.7
GRIT (CASS)	Compact	0	43.7	2.4	0.0	23.2
	Rich	0	39.0	2.9	0.0	21.1
	Compact	3	43.2	3.5	0.0	23.5
	Rich	3	38.4	2.9	0.0	20.8
GRIT (Gemini)	Compact	0	50.5	12.4	26.7	32.3
	Rich	0	50.0	12.9	26.7	32.3
	Compact	3	**80.0**	**54.7**	**73.3**	**68.3**
	Rich	3	**94.2**	**58.2**	**80.0**	**77.3**
GRIT, pass@3[a]	Compact	3	83.7	72.9	86.7	78.9
GRIT, controller[b]	Adaptive	Adaptive	**93.8**	**81.4**	**100.0**	**88.0**
GRIT, MI210[c]	Rich	3	92.6	54.7	53.3	73.9

(95.5%), but it reaches this by recompiling each program from its CUDA source with a proprietary, closed-source compiler and runtime, so it cannot be applied when only the executable is available. ZLUDA, which JIT-compiles the PTX embedded in the binary, reaches 80.0% and requires that PTX was retained; most of its CASS-bench-v2 failures come from two features it does not implement, managed memory and formatted device-side `printf`. Neither applies to executables that ship only SASS. Working from SASS alone, GRIT matches SCALE on CASS-bench-v2 and exceeds ZLUDA (94.2% against 93.7% and 66.3%). Targeting CDNA2 (MI210) with rich input and three repairs, GRIT reaches 73.9%: CASS-bench-v2 stays close to RDNA3 (92.6% against 94.2%), while HeCBench (54.7% against 58.2%) and Rodinia (53.3% against 80.0%) are lower.

5.2 Context and Kernel Units

Context mode. Without repair, rich context changes little: whole-program accuracy moves from 2.4% to 2.7% for Gemini and from 10.9% to 8.8% for CASS (Table 2), and inside KTUs with one turn both modes reach 32.3% with Gemini. On Rodinia each mode passes four programs, but only `nn` and `pathfinder` pass under both. With three repairs, rich context raises accuracy from 68.3% to 77.3%, mainly on CASS-bench-v2 (80.0% to 94.2%); the HeCBench gain (54.7% to 58.2%) is smaller than the spread between repeated rich runs (57.4% to 62.3%, §5.5), and on Rodinia it is a single program. For CASS, rich input also lowers accuracy inside units (23.2% to 21.1%).

KTUs. At one turn, KTUs raise Gemini from 2.4% to 32.3% with compact input and from 2.7% to 32.3% with rich input (Figure 3). Within our request format, they also raise CASS, from 10.9% to 23.2% with compact input and from 8.8% to 21.1% with rich input, almost entirely on CASS-bench-v2 (20.5% to 43.7% with compact input). CASS's own prompt on whole programs still does better,

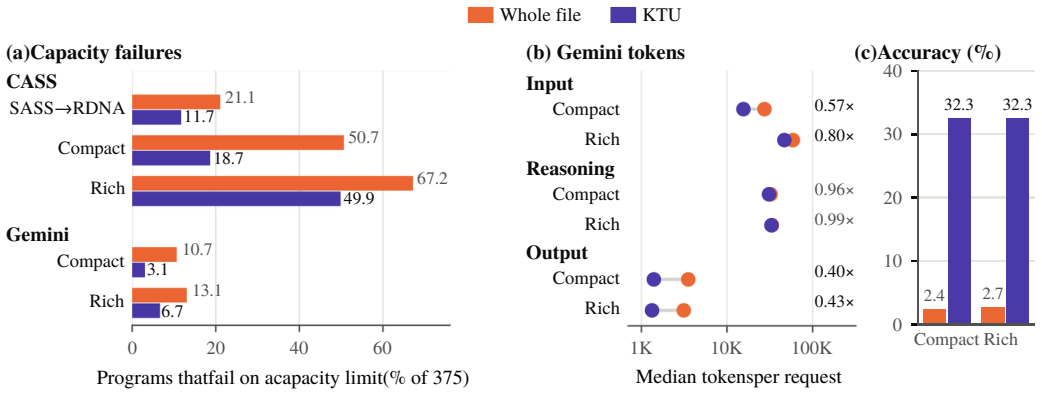


Figure 3: Whole-file translation versus KTUs, both with one model turn and no repair. (a) Programs that fail because a request exceeds the input window or a response reaches the response limit. (b) Median Gemini tokens per request, with the KTU median as a fraction of the whole-file median. (c) Gemini end-to-end accuracy.

at 28.8% overall and 54.7% on CASS-bench-v2, possibly because per-unit requests differ from the whole-file format CASS was fine-tuned on.

CASS input window. For CASS, the input window is the binding limit. Whole-program requests exceed it for 46.4% of programs with compact input and 63.7% with rich input, but only 6.4% with its own, much shorter prompt; KTUs lower these shares to 5.6%, 44.5%, and 3.5% (Figure 3a). These figures describe the programs we measured only.

5.3 Compiler Feedback

Up to three repairs per unit raise end-to-end accuracy from 32.3% to 68.3% with compact input and to 77.3% with rich input (Table 2). Most repairs are small: a unit uses 0.57 repair turns on average, 363 of the 786 units (46.2%) needed any, and 81.5% of those were fixed in one. Repair does not help CASS (23.2% to 23.5% with compact input, 21.1% to 20.8% with rich). Validation does not imply correctness: in the 65,536-token rich HeCBench run (§5.5), 153 of 170 programs produce a validated package but only 103 match the oracle, so 50 (33%) run yet compute the wrong result.

5.4 Sampling and Run-to-Run Variation

A single attempt understates what GRIT can produce. With compact input and three repairs, one run translates 68.3% of the 375 programs correctly, while three independent runs cover 78.9% (pass@3; an upper bound without NVIDIA hardware, §3.5), 40 programs more (Table 2). The gain is uneven: most programs pass in all three runs or in none, and the gap comes from a smaller set that pass in one or two (Figure 7, Appendix G). Variation also differs by suite: between identical runs, 36.5% of HeCBench programs change verdict, against 5.8% on CASS-bench-v2.

5.5 Reasoning Controller

The Reasoning Controller predicts effort, repair budget, context mode, and attempt budget N for each program before any model call (§3.6). On the 92 programs of the held-out test set, its chosen configurations pass 66.3% of the time, against 58.9% for the best fixed configuration and 46.2% for a random one, closing a third of the gap to a per-program oracle at 81.5% (Figure 6a). Its attempt budgets track difficulty: programs advised one attempt all passed, while those advised ten passed 16% of the time (Figure 6b). Together, these choices translate 88.0% of the held-out test set correctly (pass@N). The depth-4 decision tree nearly matches the random forest (balanced accuracy 0.789 against 0.798), so a few interpretable rules capture most of the policy (Figure 5).

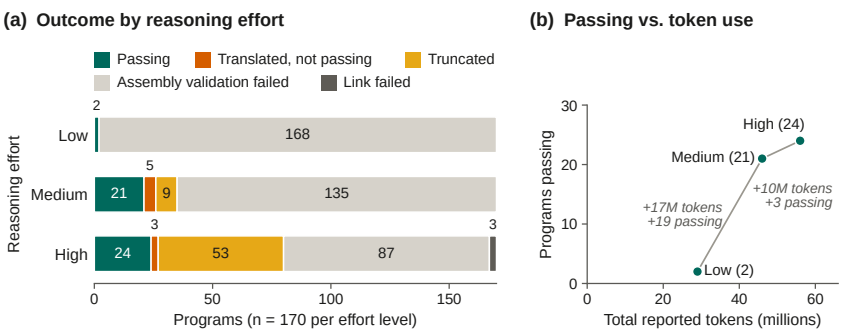


Figure 4: Outcomes on the 170 HeCBench programs at medium and high reasoning effort (compact input, one turn, no repair). (a) Passing, non-passing, truncated, and validation or link failures per effort level. (b) Tokens spent against programs passing.

Reasoning effort. With compact input, one turn, and no repair, accuracy is 8.9% at low effort, 32.3% at medium, and 38.1% at high. Because reasoning and translation share the response limit, high effort loses more programs to it (60 against 10), and costs 50% more median latency (173.3 s against 115.4 s) and 59% more median reasoning tokens (49,069 against 30,927). No single effort level suits every program, so the RC chooses effort per program (Figure 4 shows HeCBench).

Response limit. With rich input and three repairs on HeCBench, accuracy rises from 7.1% at a 16,384-token limit to 36.5% at 32,768 and 60.6% at 65,536. Programs with a truncated call fall from 156 of 170 to 136 and 31, and those calls had spent about 96% of the limit on reasoning. Three runs of 162 of these programs at 65,536 tokens pass 62.3%, 58.6%, and 57.4%.

6 Limitations

All GRIT results use a single reasoning model, Gemini 3.8 Flash, and the controller is trained only on its outcomes. Each translation also relies on paid calls to a commercial model API, and the number of calls grows with program size, repairs, and attempts; we report tokens but not monetary cost. An open-weight model or a cost-aware controller would make the approach more widely accessible. Checking a translation requires running the original executable, so when the NVIDIA hardware is no longer available, correctness cannot be confirmed and our pass@N numbers, which assume such a check, become an upper bound. Finally, we test only `-O0` SASS from SM80 and SM89 on two AMD targets, and we measure correctness rather than performance.

7 Conclusion

Native GPU assembly translation turns out to hinge less on how well a model reasons than on how that reasoning is spent. GRIT keeps each call within budget with executable-derived context and dependency-aware kernel units, turns rejections into targeted edits with compiler diagnostics, and lets a learned controller allocate the remaining effort, repairs, and attempts per program. Across 375 workloads, this raises end-to-end correctness from 4.0% for whole-program translation to 77.3% in a single attempt, and to 88.0% on held-out programs with the Reasoning Controller (pass@N). As reasoning models take on more systems-level work under fixed compute budgets, controlling how they reason may matter as much as how well they reason.

AI use statement

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.

Ethics Statement

This work translates existing GPU kernels between vendor instruction sets to improve hardware portability; it does not involve human subjects or personal data, and no foundation model is trained or fine-tuned beyond the lightweight Reasoning Controller described in §3.6. All benchmark programs are drawn from public research benchmark suites (CASS-bench-v2, HeCBench, Rodinia) under their original licenses (Appendix H). We are not aware of dual-use concerns specific to this method beyond those already inherent to the general-purpose reasoning models it uses off the shelf.

Reproducibility Statement

The evaluation hardware, benchmark suites, and correctness oracle are described in §4.1; the exact context fields, repair critics, and Reasoning Controller features and training procedure are given in §3.3, §3.4, and Appendix D. Code, benchmark manifests, and trained Reasoning Controller models will be released publicly upon publication (Appendix H).

References

Advanced Micro Devices. HIPIFY: Convert CUDA to portable C++ code. `https://github.com/ROCm/HIPIFY`, 2026. Accessed September 2026.

Apple Inc. About the Rosetta translation environment. Apple Developer Documentation, `https://developer.apple.com/documentation/apple-silicon/about-the-rosetta-translation-environment`, 2020. Accessed September 2026.

Jordi Armengol-Estapé, Rodrigo C. O. Rocha, Jackson Woodruff, Pasquale Minervini, and Michael F. P. O'Boyle. Forklift: An extensible neural lifter. In *First Conference on Language Modeling (COLM)*, 2024. URL `https://openreview.net/forum?id=LWfDcI6txJ`.

Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, et al. Frontier: Exploring exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '23)*, Denver, CO, USA, 2023. ACM. doi: 10.1145/3581784.3607089.

Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, Anaheim, CA, USA, 2005. USENIX Association.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, Austin, TX, USA, 2009. IEEE. doi: 10.1109/IISWC.2009.5306797.

Jun Chen, Xule Zhou, and Hyesoon Kim. 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*, pp. 1093–1104. ACM, 2023. doi: 10.1145/3624062.3624185.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schaerli, and Denny Zhou. Teaching large language models to self-debug. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (eds.), *International Conference on Learning Representations*, volume 2024, pp. 8746–8825, 2024. URL `https://proceedings.iclr.cc/paper_files/paper/2024/file/2460396f2d0d421885997dd1612ac56b-Paper-Conference.pdf`.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235 of *Proceedings of Machine Learning Research*, pp. 9688–9709, 2024. URL https://proceedings.mlr.press/v235/cummins24a.html.

James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pp. 15–24. IEEE, 2003. doi: 10.1109/CGO.2003.1191529.

Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 353–364, 2010. doi: 10.1145/1854273.1854318.

Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 1997. doi: 10.1145/264107.264126.

Gemini Team. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*, 2024.

Google. Gemini thinking. Google AI for Developers, https://ai.google.dev/gemini-api/docs/thinking, 2026. Accessed September 2026.

Daya Guo, Dejian Yang, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025. doi: 10.1038/s41586-025-09422-z.

Ahmed Heakl. cass-smA100-7b. https://huggingface.co/ahmedheakl/cass-smA100-7b, 2025. Model checkpoint. Accessed September 2026.

Ahmed Heakl, Sarim Hashmi, Chaimaa Abi, Celine Lee, and Abdulrahman Mahmoud. Guaranteed guess: A language modeling approach for CISC-to-RISC transpilation with testing guarantees. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 24474–24488, Suzhou, China, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-emnlp.1330.

Ahmed Heakl, Gustavo Bertolo Stahl, Sarim Hashmi, Seung Hun Eddie Han, Mukul Ranjan, Arina Kharlamova, Salman Khan, and Abdulrahman Mahmoud. CASS: Nvidia to AMD transpilation with data, models, and benchmark. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 34489–34508, San Diego, California, United States, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.1592. URL https://aclanthology.org/2026.acl-long.1592/.

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

Andrzej Janik. ZLUDA: CUDA on non-NVIDIA GPUs. https://github.com/vosen/ZLUDA, 2026. For the PTX-only module-loading path, see https://github.com/vosen/ZLUDA/blob/c9d4bbdf30a95d16421c1c36ef4f0ba55a894930/zluda/src/impl/module.rs.

Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. Nova: Generative language models for assembly code with hierarchical attention and contrastive learning. In *International Conference on Learning Representations (ICLR)*, 2025.

Zheming Jin and Jeffrey S. Vetter. A benchmark suite for improving performance portability of the SYCL programming model. In *Proceedings of the 2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 325–327, Raleigh, NC, USA, 2023. IEEE. doi: 10.1109/ISPASS57527.2023.00041.

Celine Lee, Abdulrahman Mahmoud, Michal Kurek, Simone Campanoni, David Brooks, Stephen Chong, Gu-Yeon Wei, and Alexander M. Rush. Guess & sketch: Language model guided transpilation. In *International Conference on Learning Representations (ICLR)*, 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/abs/10.1126/science.abq1158.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 12:157–173, 2024. doi: 10.1162/tacl_a_00638.

LLVM Project. User Guide for AMDGPU Backend. LLVM 18.1.7 Documentation, https://releases.llvm.org/18.1.7/docs/AMDGPUUsage.html, 2024.

NVIDIA Corporation. NVIDIA CUDA Compiler Driver NVCC Documentation. https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/, 2026a.

NVIDIA Corporation. Parallel thread execution ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/, 2026b.

NVIDIA Corporation. CUDA binary utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/, 2026c. Lists SASS opcodes per architecture without full semantics. Accessed September 2026.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. In *International Conference on Learning Representations (ICLR)*, 2025.

Spectral Compute. SCALE: CUDA-compatible GPU programming toolkit. https://docs.scale-lang.com/stable/, 2026.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4Decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 3473–3487, Miami, Florida, USA, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.203.

Hanzhuo Tan, Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, and Yuqun Zhang. SK2Decompile: LLM-based two-phase binary decompilation from skeleton to skin. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=jSQPqdoidy.

Ali TehraniJamsaz, Arijit Bhattacharjee, Le Chen, Nesreen K. Ahmed, Amir Yazdanbakhsh, and Ali Jannesari. CodeRosetta: Pushing the boundaries of unsupervised code translation for parallel programming. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, pp. 100965–100999, 2024. doi: 10.52202/079017-3202.

TOP500. TOP500 list – june 2026. https://www.top500.org/lists/top500/2026/06/, 2026. Accessed September 2026.

Paulius Velesko, Pekka Jääskeläinen, Henry Linjamäki, Michal Babej, Peng Tu, Sarbojit Sarkar, Ben Ashbaugh, Colleen Bertoni, Jenny Chen, Philip C. Roth, Wael Elwasif, Rahulkumar Gayatri, Jisheng Zhao, Karol Herbst, Kevin Harms, and Brice Videau. chipStar: Making HIP/CUDA applications cross-vendor portable by building on open standards. *International Journal of High Performance Computing Applications*, 2026. doi: 10.1177/10943420261423001.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 35, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html.

Jisheng Zhao, Huanzhi Pu, Shinnung Jeong, Chihyo Ahn, and Hyesoon Kim. CuLifter: Lifting GPU binaries to typed IR, 2026. URL https://arxiv.org/abs/2604.27486.

A Extended Related Work

Table 3 summarizes CUDA portability systems by the device-code representation they consume and produce.

Table 3: CUDA portability and translation systems. **Input** and **Output** denote the principal device-code representations consumed and produced. **Src.-free**: CUDA source not required. **LLM**: learned language-model-based translation. **Repair**: automated compiler-guided repair.

System	Input	Output	Src.-free	LLM	Repair
GPU Ocelot (Diamos et al., 2010)	PTX	Runtime target code	✓	–	–
CuPBoP-AMD (Chen et al., 2023)	NVVM IR	HIP-compatible LLVM IR	–	–	–
HIPIFY (Advanced Micro Devices, 2026)	CUDA source	HIP source	–	–	–
SCALE (Spectral Compute, 2026)	CUDA source	AMD device code	–	–	–
chipStar (Velesko et al., 2026)	CUDA/HIP source	SPIR-V	–	–	–
ZLUDA (Janik, 2026)	PTX	AMD module	✓	–	–
CuLifter (Zhao et al., 2026)	SASS	Typed LLVM IR	✓	–	–
CASS (Heakl et al., 2026)	SASS	RDNA3 assembly	✓	✓	–
GRIT (This work)	**CUBIN (SASS)**	**AMDGPU code object**	✓	✓	✓

B Problem Formulation Details

§3.1 summarizes GRIT's inference-time control problem; this section gives the full formalization.

Let u denote a KTU of a program. An inference policy $\pi = (e, K, m, N)$ is chosen once per program and applied to all of its KTUs: the reasoning effort $e \in \{\text{LOW}, \text{MEDIUM}, \text{HIGH}\}$, up to K repair turns per KTU, the context mode $m \in \{\text{COMPACT}, \text{RICH}\}$ (§3.3), and N independent samples of the whole program. The mode determines x_u, the input the model reads for u: its SASS together with the executable-derived facts that m selects. Given x_u and e, the model defines a distribution $p_M(y_u \mid x_u, e)$ over candidate AMDGPU translations y_u, which may be correct, incorrect, or invalid. A deterministic validator $V(y_u)$, available at inference time, accepts or rejects each candidate with diagnostics. The n-th sample takes one accepted candidate per unit, $\{y_u^{(n)}\}_u$, which the finalizer combines into a program $P^{(n)}$ (§3.7). A correctness oracle $O(P^{(n)})$, available only for evaluation, runs it on AMD hardware and compares its output with that of the original NVIDIA executable. V establishes legality and a few structural properties, not correctness: $O(P^{(n)}) = 1$ requires every $V(y_u^{(n)})$ to accept, but not the reverse.

C Experimental Setup Details

Inputs. For each workload, we record the selected NVIDIA architecture, CUBIN image, kernel roots, reachable helpers, SASS instruction and line counts, token counts, rich-context size, program arguments, and expected output. Binaries are compiled for SM89 (SM80 for the MI210 and CASS model runs) with `-Xptxas -O0`. When a fatbinary contains several architecture images, we select one complete image per kernel and do not count duplicate images as separate programs.

Gemini. Gemini 3.8 Flash runs at medium effort with a 65,536-token response limit and targets RDNA3 (gfx1100), or MI210 (gfx90a) for CDNA2. All runs share one prompt and the same deterministic backend. The prompt identifies SASS as authoritative, requires structured output, forbids invented resources, symbols, relocations, and constants, and contains no complete worked example.

CASS. The CASS paper reports 90.24% assembly compilation and 69.11% strict end-to-end accuracy for CASS-sm80-7B over 369 tasks (Heakl et al., 2026). We do not use these published numbers; we run CASS locally with checkpoint `ahmedheakl/cass-smA100-7b` (Heakl, 2025), a total context limit of 32,768 tokens, and sampling with temperature 0.7, top-$p = 0.8$, top-$k = 20$, repetition penalty 1.1, and seed 0. We reserve a 32-token safety margin, giving an output budget of $32{,}768 - n_{\text{in}} - 32$ tokens, where n_{in} is the number of input tokens.

CDNA2. The MI210 (CDNA2, gfx90a) is a datacenter-class AMD Instinct GPU, and we translate SM80 binaries from the matching NVIDIA datacenter generation. CDNA2 executes 64-lane waves, so each wave holds two CUDA warps and warp-level operations must stay within one 32-lane half, and it uses the older GFX9 instruction set. Only target-dependent parts change: the launch ABI in the context, the finalizer's kernel descriptor and metadata, the device runtime, and the target description in the prompt.

SCALE and ZLUDA. SCALE recompiles each program's CUDA source, both host and device code, with its own compiler into a complete gfx1100 executable that runs on SCALE's CUDA-compatible runtime; it therefore requires the full source and a proprietary, closed-source toolchain. ZLUDA runs the unmodified SM89 executables and JIT-compiles their embedded PTX, with a 240 s budget per program. We evaluate both on all three suites with the same oracles.

Runs and metrics. Each condition is run once over all programs, except the pass@3 condition, which uses three independent runs, the controller, which retranslates a program up to N times, and the response-limit study, which repeats 162 HeCBench programs three times. Translations run with up to 256 concurrent KTUs per experiment. End-to-end correctness uses all programs of a suite as the denominator, so incomplete generations count as failures.

D Reasoning Controller Details

This appendix describes the data, features, training protocol, and held-out results of the Reasoning Controller (§3.6).

Training data. The RC is trained on twelve fixed-configuration campaigns over the 375 programs, all translated to gfx1100 by the same pipeline in one pinned container on one GPU. Three of the twelve campaigns repeat the same configuration, which separates run-to-run variation from program difficulty. Each row of the training table is one (program, configuration) outcome, giving 3,175 rows, or 8.5 configurations per program on average. Treating the configuration as an input rather than a label lets the model estimate $P(\mathrm{pass} \mid \mathrm{program}, \mathrm{configuration})$ and exposes within-program contrasts, such as the same program at low and high effort. A recommendation is made by scoring every candidate configuration and choosing the highest.

Features. Program features are extracted from the recorded translation requests, which are exactly the units the model saw, so they are available for all 375 programs before any model call. An initial pool of 88 features was reduced to the 18 in Table 4. We removed, on principle rather than for performance, every feature that identifies the suite or the program: the suite itself, the oracle mechanism and oracle kind (each equivalent to the suite), the program name, and the executable size, which is unique to 73–100% of programs and let a tree memorize identities. Removing them left held-out accuracy essentially unchanged. Counters for tensor-core, texture, surface, warp-vote, indirect-branch, and SIMD-video instructions were dropped because no program in the corpus contains these instructions.

Protocol and models. The protocol was fixed before any model was fit. We split programs, not rows, into 283 development and 92 test programs, stratified on each program's pass rate (0.436 and 0.431). Model selection used 5-fold grouped cross-validation over development programs only, and the selected models were fit on all development programs and scored once on the test programs. Table 5 reports both stages. The random forest has 400 trees, a minimum of 5 samples per leaf, and 4 of the 18 features per split, with unbounded depth (median 16). The decision tree has depth 4, a minimum of 25 samples per leaf, and 15 leaves.

The tree is within noise of the forest on 92 programs, so the forest offers no clear accuracy advantage, and the tree remains fully interpretable.

Leakage checks. No program appears in both halves (283 + 92 = 375), and the test programs were not read during model selection. No single feature dominates: the strongest correlation with the outcome is 0.310, for the number of repair turns. With shuffled labels, held-out balanced accuracy is 0.491, against a chance level of 0.500; hidden leakage would keep it above chance.

Table 4: The 18 features of the Reasoning Controller: twelve describe the program and six describe the candidate configuration.

Program features	Configuration features
SASS input tokens	Reasoning effort
Rich-context input tokens	Repair turns
Largest-unit tokens (SASS)	Context mode (compact or rich)
Largest-unit tokens (rich)	Input tokens under the configuration
Largest unit as fraction of the limit	Largest-unit tokens under the configuration
Kernel count	Largest unit as fraction of the limit under the configuration
Maximum register count	
Maximum parameter count	
Global-memory instructions	
Local-memory instructions	
Double-precision instructions	
Call instructions	

Table 5: Pass/fail prediction for (program, configuration) pairs. Development scores are 5-fold grouped cross-validation; held-out scores are computed once on the 92 test programs.

	Development	Held out		
Model	Balanced acc.	Accuracy	Balanced acc.	ROC AUC
Majority class	0.500	0.569	0.500	–
Decision tree (depth 4)	0.750	0.791	0.789	0.847
Decision tree (depth 5)	0.757	–	–	–
Gradient boosting	0.797	–	–	–
Random forest	0.802	0.797	0.798	0.868

What the models learn. In the forest, the number of repair turns is the most important feature (mean decrease in impurity 0.131), ahead of every program property; the maximum register count (0.105) and reasoning effort (0.091) follow. Context mode carries little weight (0.006). The decision tree (Figure 5) splits first on program properties and then on configuration. For small, single-kernel programs (at most 23 registers), low effort passes 19% of the time, medium or high effort without repair 75%, and the same with three repairs 99%.

Deployment on held-out programs. For each of the 92 test programs, the forest chose a configuration and an attempt budget, and every attempt was a fresh translation; a program that passed was not retried. This used 173 translations in about 2.9 hours. Table 6 compares the chosen configurations with fixed and random ones on the same programs. The RC closes 33% of the gap between the best fixed configuration and a per-program oracle.

Table 6: Pass rate on the 92 held-out programs when each policy chooses one configuration per program.

Policy	Pass rate
Random configuration	0.462
Best fixed configuration	0.589
Reasoning Controller	0.663
Per-program oracle	0.815

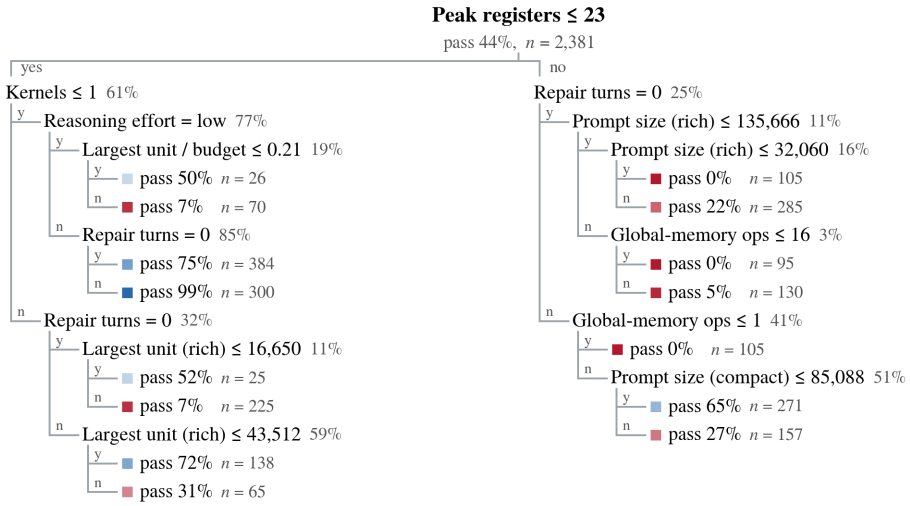


Figure 5: The depth-4 decision tree of the Reasoning Controller. Each leaf gives the pass rate of the (program, configuration) pairs that reach it.

End-to-end, 81 of the 92 held-out programs pass (88.0%), the figure in Table 2. The predicted attempt budget tracks difficulty (Figure 6). Programs advised one attempt passed every time, while those advised ten passed 16% of the time; the correlation between predicted and observed pass rates is 0.729.

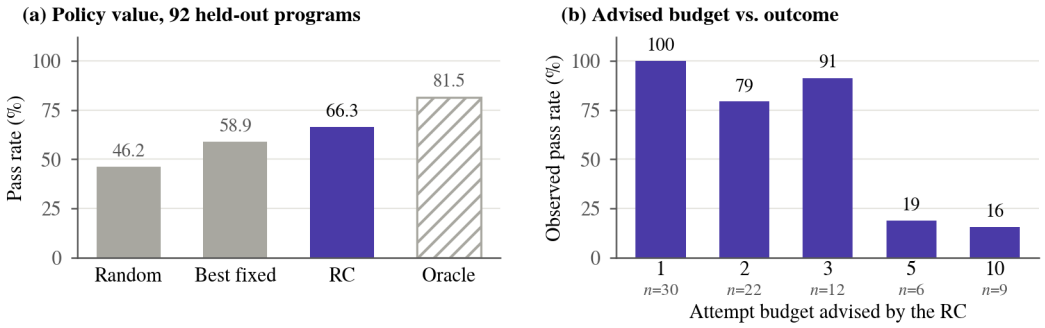


Figure 6: Held-out behavior of the Reasoning Controller. (a) Pass rate on the 92 held-out programs when each policy chooses one configuration per program: a random configuration, the best fixed configuration, the RC's choice, and a per-program oracle, which is an upper bound. (b) Observed pass rate of programs grouped by the attempt budget the RC advised, with the number of programs in each group.

Limitations. With 375 programs, the models are kept small, and the 92 test programs are too few to separate the forest from the tree. These test programs have now been used and cannot be rescored without bias. Part of the remaining error is irreducible: 20.5% of programs change verdict between identical runs (36.5% on HeCBench, 5.8% on CASS-bench-v2). Finally, all Rodinia runs in the training campaigns failed to start because of a harness fault, so the oracle recorded them as mismatches. The one held-out Rodinia program (`gaussian`) passes all three checks under the same configuration once the fault is fixed, and is counted as passing above. Because of these labels, the RC overestimates the difficulty of Rodinia programs: in a five-program probe, it advised 22 attempts in total, and four of the five passed on the first. Its ranking held, since the only failure (`lavaMD`) was the program ranked lowest, but its probabilities were pessimistic. The Rodinia labels should be regenerated on the fixed harness.

E RUNTIME PACKAGING

The finalizer produces a relocatable package. A shell wrapper places the package's own libraries on the loader path, points the runtime shim at the kernel map, and executes the unmodified host binary. That binary still issues CUDA calls, which the loader resolves against drop-in libraries carrying CUDA's sonames (`libcudart.so.12`, `libcublas.so.12`, and so on) that forward to HIP and native ROCm libraries, so host code is never recompiled or translated. A kernel map records, for each CUDA kernel, the AMD code object and symbol that replace it, and is the only file consulted at run time; the remaining manifests record build provenance and the finalizer's inputs. Running the package invokes neither the model nor the original NVIDIA binary.

F CORRECTNESS ORACLE DETAILS

§4.1 summarizes how each suite's correctness oracle O is evaluated; this section gives the exact benchmark admission criteria, per-program rules, and pass/fail handling.

Benchmark construction. CASS-bench-v2 (Heakl et al., 2026) has 400 test programs, of which 256 have paired sources and an NVIDIA binary with a device kernel; 190 of these pass the normalization admission test below. HeCBench (Jin & Vetter, 2023) contributes 170 programs from the `ivanradanov/rodinia` repository (Ivanov, 2016), which bundles HeCBench; their HIP versions are produced with `hipify`, and we keep programs with a valid single-GPU baseline and a kernel body in their SM80/SM89 binary. Rodinia (Che et al., 2009) contributes 15 classic Rodinia 3.1 programs from the same repository, omitting those it documents as racy, nondeterministic, unsupported, or failing to compile.

CASS-bench-v2 and HeCBench normalization. Admission itself confirms that the expected result is reachable on AMD hardware: a program is admitted only if its native CUDA and native HIP versions both succeed and print identical normalized output. Normalization removes run- and machine-dependent fields such as timing, performance values, and device banners; it never rounds floating-point results or sorts whole outputs. A few programs need a rule scoped to that program alone: for 11 CASS-bench-v2 programs, such a rule sorts a contiguous group of per-thread records or masks a printed address or device banner; for two HeCBench programs, it removes GPU-dependent error metrics. We checked each CASS-bench-v2 rule against all saved runs to confirm that it hides no missing output.

Rodinia result arrays. Built-in verifiers compare the translated program's device result arrays with arrays produced by the CUDA original, using Rodinia's original tolerances – stricter than matching printed output. The CUDA references must agree across two independent runs. The one clock-seeded program (`particlefilter`) runs with a fixed time on both platforms. A translation passes only if every expected verifier reports exactly one passing check; missing, duplicate, or failing reports count as failure.

G RUN-TO-RUN VARIATION

Figure 7 shows the per-program pass counts underlying the run-to-run variation discussed in §5.4: the same program, translated three times under identical settings, can succeed once and fail once, and this gain is concentrated in a smaller set of programs that pass in one or two of the three runs rather than spread evenly across all 375.

H BENCHMARK SIZE COMPARISON

Figure 8 compares the size of the three suites in §4.1. We measure each program as the complete SASS listing of its NVIDIA binary (`cuobjdump -sass`, SM80, `ptxas -O0`), tokenized with the CASS-7B tokenizer. SM89 listings of the same programs differ by less than 0.1% where both builds are available. CASS-bench-v2 programs are small: the median program has 6.5K tokens, and 13% exceed the 32K context of CASS-7B. HeCBench and Rodinia programs are much larger: their medians of 75.2K and 49.5K tokens are 11.5× and 7.6× that of CASS-bench-v2, and 79% and 67% of their programs exceed 32K. The complete listings of 7 HeCBench programs and 1 Rodinia

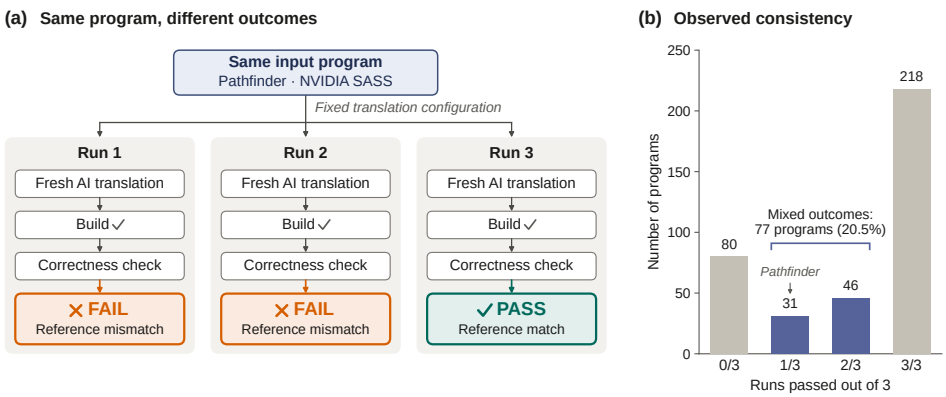


Figure 7: Per-program pass counts over three independent runs (pass@3, compact input, three repairs). (a) The same program (`pathfinder`), translated three times under identical settings. (b) Distribution of all 375 programs by the number of runs passed.

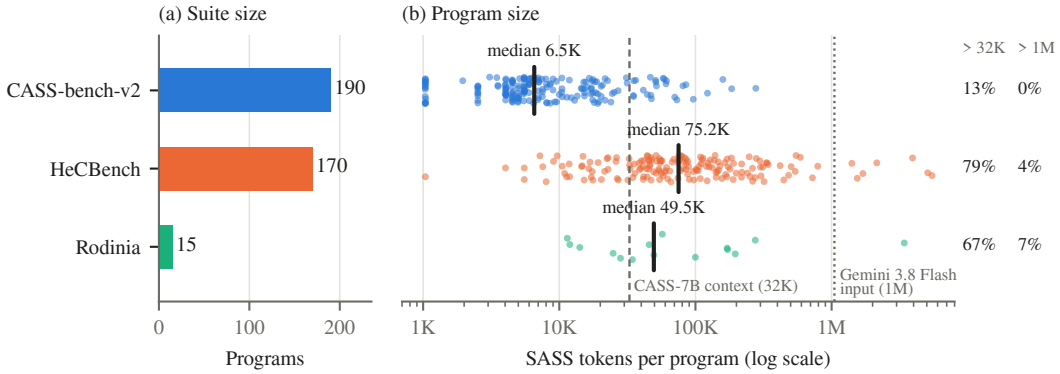


Figure 8: Benchmark size. (a) Programs per suite. (b) SASS tokens per program (log scale; one dot per program, bar at the median). The dashed and dotted lines mark the 32K context of CASS-7B and the 1M input limit of Gemini 3.8 Flash; the right columns give the share of programs above each. Tokens are counted with the CASS-7B tokenizer, so the Gemini line is approximate.

program also exceed the 1M-token input limit of Gemini 3.8 Flash; the largest HeCBench program has 5.5M tokens. CASS-bench-v2 therefore enables direct comparison with prior NVIDIA-to-AMD translation work, while HeCBench and Rodinia test whether translation scales to complete, realistic workloads. Their size is also one motivation for translating KTUs rather than whole programs (§3.2).

I 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. ZLUDA toolchain is available under the Apache 2.0 license as well. Scale toolchain is proprietary but freely available for research use. Our dataset and code will be released under Apache 2.0.