Preprint, not peer reviewed

CUDA Rust on ThoxOS

An integration review of NVIDIA's two Rust GPU kernel tracks, now with the first THOX device qualification on an RTX 4060 Ti, a same-day comparison against the production CUDA C++ kernel, and the ThoxOS GPU layer that acts on both.

Tommy Xaypanya
Chief Technology Officer & Co-Founder
THOX.ai LLC
Corresponding author · ORCID 0009-0000-4425-9565

THOX.ai LLC · Cedar Park, Texas

ABSTRACT

NVIDIA's CUDA Rust offers two ways to write GPU kernels in Rust: cuda-oxide for SIMT and cutile-rs for Tile. We review both, map them onto ThoxOS™ hosts, and test the Tile track twice: on a host with no GPU, and on a THOX RTX 4060 Ti workstation (sm_89). Without a GPU, cutile 0.4.0 builds on 3 of 3 stable Rust toolchains given CUDA 13 headers including cuRAND, rejects 4 of 4 aliasing misuse cases at compile time, and exits with a typed driver error instead of panicking when no driver is present. On the workstation, our FP32 Tile RMSNorm kernel matches the CPU reference on all 6 model shapes we test. Against the production CUDA C++ kernel on the same host and day, the result splits by shape: on the full launch path the C++ side is 2.96x faster at 1 × 2048 and the Tile side 1.52x faster at 128 × 4096. Both figures are dominated by allocation, launch and copy-back, so we make no kernel-versus-kernel speed claim. A CUDA 13.0 toolkit builds the crate and passes host tests but cannot launch a Tile kernel, because the Tile IR assembler ships only with CUDA 13.2 and later. We describe the ThoxOS GPU layer that turns these results into a fallback order and test it on 8 virtual hosts. The Tile track is adopted for new NVIDIA kernel authoring behind that fallback; the production default stays on CUDA C++, Jetson is not yet run, and the SIMT track is reviewed from source only.

CONTENTS

Executive summary
1 Introduction
2 Background and related work
3 System and threat model
4 Design
5 Implementation
6 Evaluation
7 Discussion and limitations
8 Conclusion
9 Roadmap
A Appendix A: reproducibility
B Appendix B: notation
C Appendix C: device qualification protocol
D Appendix D: document control
References

Executive summary

NVIDIA now lets GPU kernels be written in Rust on two tracks: cuda-oxide for the classic SIMT model and cutile-rs for the newer Tile model [1]. ThoxOS™ already writes its systems layer in Rust, while its GPU kernels live in CUDA C++. Version 0.1.0 of this report measured what could be measured without a GPU. This edition adds the first device run of a THOX CUDA Rust kernel, a same-day comparison with the production CUDA C++ kernel on the same host, and the ThoxOS GPU layer that turns both into a kernel choice per host.

6/6
model shapes where the Tile RMSNorm kernel matches the FP32 CPU reference on sm_89 (tolerance 1.0e-5)
MEASURED ON DEVICE

Split
launch path, same host and day: C++ 2.96x faster at 1 × 2048, Tile 1.52x faster at 128 × 4096; no kernel-vs-kernel claim
CALCULATED FROM MEASURED MEDIANS

4/4
aliasing misuse cases rejected at compile time by cutile 0.4.0
MEASURED, GPU-LESS HOST

8/8
virtual hosts on which the ThoxOS GPU layer binds the expected kernel
SIMULATED

STATUS OF EVIDENCE

- **Measured on a device:** the FP32 Tile RMSNorm kernel on the THOX RTX 4060 Ti workstation (sm_89): parity on 6 shapes, 30-run benches on two shapes, and the CUDA 13.0 launch failure. Same-day CUDA C++ comparison on the same host.
- **Measured on a GPU-less host:** header and toolchain matrices, compile-time safety cases, no-driver behaviour and build cost (unchanged from v0.1.0).
- **Simulated:** the GPU layer's virtual lab of 8 hosts, two of which carry the workstation's measured facts.
- **Not run:** any Jetson or Orin device, any kernel-only Tile timing, any build of the SIMT track (source review only), FP16 and BF16.

PRODUCT IMPLICATIONS FOR THOXOS

The Tile track (cutile-rs) is adopted for new NVIDIA kernel authoring on qualified hosts, behind the GPU layer's fallback order. The production default does not change: the THOX edge-ops CUDA C++ kernels stay the default GPU path, and thox-rmsnorm stays the CPU path and numerical oracle. The case for adoption is correctness and authoring, not speed: the comparison splits by shape and carries a first-launch JIT cost. Jetson Orin stays on its current board support package until an Orin device run exists. ThoxMini, ThoxAir, ThoxClip and Apple silicon hosts have no NVIDIA GPU and are unaffected.

1 Introduction

GPU kernels are the last part of the THOX stack that is not written in Rust. THOX's microvisor, device orchestration and agent runtimes are Rust workspaces; the fused normalization kernels in the THOX edge-ops library are CUDA C++ with hand-managed pointers and launch geometry. NVIDIA's September 2026 announcement closes that gap with two tracks that mirror CUDA itself [1]. The SIMT track compiles Rust kernels through a custom rustc backend to PTX. The Tile track lets a kernel describe what one tile of data does and JIT-compiles it through CUDA Tile IR [2], an idea that descends from tile languages such as Triton [3].

Both tracks make the same safety claim: inputs are shared, outputs have exactly one writer, and Rust's ownership rules enforce that at the launch boundary. The research paper behind the Tile track proves its safe subset free of data races under the Tile IR memory model and reports near-cuBLAS GEMM throughput on B200 [4]. Those are NVIDIA's results on NVIDIA's hardware. THOX needs its own answers: does the toolchain fit ThoxOS build hosts, do the safety checks hold in our hands, how does a binary behave without a driver, does a THOX kernel run correctly on THOX hardware, and how does it compare with what ships today.

Contributions:

1. A dated review of CUDA Rust, including what changed in the weeks after the announcement (Section 2.2).
2. A requirement map from the two tracks onto ThoxOS NVIDIA targets (Section 3).
3. GPU-less experiments on the Tile track: header and toolchain matrices, 6 compile-time safety cases, the no-driver failure mode, and build cost (Sections 6.1 to 6.6).
4. The first device qualification of a THOX CUDA Rust kernel: an FP32 Tile RMSNorm on the THOX RTX 4060 Ti workstation (sm_89), with parity on 6 model shapes and 30-run benches, plus the finding that the CUDA 13.2 floor binds at launch time, not build time (Section 6.7).
5. A same-host, same-day comparison against the production CUDA C++ RMSNorm kernel, reported with its caveats (Section 6.8).
6. The ThoxOS GPU layer: device check, qualification registry, fallback order and diagnostics page, tested on a virtual lab (Sections 4.3, 5.2 and 6.9).

2 Background and related work

2.1 The two tracks

cuda-oxide is a rustc codegen backend. `#[kernel]` functions travel from Rust MIR through the Pliron IR framework and LLVM IR to PTX, and every other function goes to the standard backend [1]. Outputs are `DisjointSlice` values that hand each thread one element, and a `#[launch_contract]` lets the host validate the launch configuration against the kernel and the live device limits. Shared memory still requires `unsafe` [1].

cutile-rs works one level higher. A tile block runs the kernel body once as a single logical thread over one sub-tensor; the compiler decides how many hardware threads back it. Mutable outputs must be partitioned on the host, the partition fixes the launch grid, and the launcher takes ownership of every tensor until the work completes [1, 4]. The crate is on crates.io [5].

2.2 What changed since the announcement

Parts of the announcement were out of date within weeks. Table 1 compares it with the upstream repository at commit `2ffc19b` (2026-11-03) [6, 7]. We cite the repository, not the blog, for requirements.

Table 1. Announcement versus upstream, as reviewed on 2026-11-03.

Item	Announcement, Sep 8	Upstream, Nov 3
Home	NVlabs/cuda-oxide and NVlabs/cutile-rs	One repository, NVIDIA/cuda-rust; standalone cutile-rs archived read-only
SIMT toolchain	Pinned nightly (2026-04-03)	Pinned stable 1.99.1 with rustc-dev and llvm-tools
SIMT CUDA floor	CUDA toolkit 12.x or newer	CUDA toolkit 13.0 or newer, CUDA 13.x driver R580+
Tile CUDA	CUDA 13.3	13.3 recommended; Tile IR 13.2 floor for sm_80 to sm_89
Tile requirements	Stable Rust 1.89+, sm_80+, Linux	Unchanged; also probes the WSL2 driver shim (from 0.4.0)
Users named	HF Grout and mistral.rs	Grout README confirms cutile-rs; the mistral.rs README does not mention it

The last row is a discrepancy we flag rather than resolve: Grout describes itself as a testbed for LLM inference with cutile-rs [8], while the mistral.rs README we opened does not mention cuTile [9]. We cite the mistral.rs claim only as NVIDIA's statement.

2.3 Related work and positioning

Safe GPU programming with ownership types predates CUDA Rust. Descend extends a Rust-like borrow checker with views over grids, blocks and threads and reports no significant overhead against hand-written CUDA [10]. The Rust GPU ecosystem includes rust-cuda, a Rust-first PTX backend [11]; rust-gpu, which targets SPIR-V [12]; CubeCL, a multi-vendor kernel DSL [13]; and cudarc, safe host bindings for kernels written elsewhere [14]. NVIDIA's own ecosystem appendix positions cuda-oxide as complementary to all of these [15]. RLX shows a single Rust codebase acting as tensor compiler and runtime across many devices [16].

Compile-time safety is not correctness. Fixed-shape tolerance tests miss transcription bugs that a seeded fuzz oracle finds [17], and timing-perturbed testing plus proof finds races that random inputs miss [18]. THOX agents write kernels, so the type system is one layer of defense, not the only one. Our contribution is not a new safety mechanism. It is an adoption study: what these tracks mean for one Rust-first edge OS with mixed hardware.

Table 2. Positioning of Rust GPU options for ThoxOS.

Project	Model	Targets	Safety mechanism	ThoxOS fit
cutile-rs [5]	Tile, JIT via Tile IR	NVIDIA sm_80+	Ownership across launch; race-free safe subset [4]	Adopted for new NVIDIA kernels, behind fallback
cuda-oxide [6]	SIMT, AOT to PTX	NVIDIA	DisjointSlice, launch contracts; shared memory unsafe	Source review only; when warp-level control is needed
rust-cuda [11]	SIMT, PTX backend	NVIDIA	Rust-first API	Watch; early stage by its own account
CubeCL [13]	Embedded DSL, JIT	CUDA, ROCm, Metal, Vulkan, WebGPU, CPU	Portable subset	Cross-vendor, including Apple silicon
rust-gpu [12]	Shaders to SPIR-V	Vulkan	Rust semantics	Graphics, not compute kernels
cudarc [14]	Host bindings	NVIDIA	Safe host API	Launching existing PTX

3 System and threat model

ThoxOS runs on hosts with very different compute. Only the NVIDIA tier is in scope. Table 3 maps each THOX host class to the requirements in Table 1 and states what has been run on it.

Table 3. ThoxOS host classes against CUDA Rust requirements.

Host class	GPU and capability	OS and CUDA	Status
THOX RTX 4060 Ti workstation	RTX 4060 Ti 16 GB, Ada sm_89	Linux under WSL2; driver 630.20, CUDA toolkit 13.3.1	**Tile kernel device-qualified** (Section 6.7)
Jetson Orin NX 16 GB, current BSP	Orin, Ampere sm_87	L4T r36 / JetPack 6.2 ships CUDA 12.6.10 [19]	**Blocked**: both tracks need CUDA 13
Jetson Orin NX on JetPack 7.2.1	sm_87	Jetson Linux 39.2.1, CUDA 13.2.2 [20]	**Possible, not run**: Tile IR 13.2 covers sm_87; 13.3 features unavailable
Apple silicon	Apple GPU	macOS, Metal	Not applicable; Metal and CPU paths stay
ThoxMini, ThoxAir, ThoxClip	No NVIDIA GPU	ThoxOS on ARM SoCs	Not applicable; thox-rmsnorm CPU path stays

Requirements for adoption, each tied to a test:

- R1. ThoxOS build hosts compile kernels and host code without a GPU, so builds run in local checks.
- R2. The compiler rejects aliasing between a launch's outputs and its inputs, and between the host and in-flight outputs.
- R3. A binary on a host with no CUDA driver fails closed: a typed error, no panic, no load-time link to `libcuda`.

- R4. The edit-check loop after a kernel change stays short enough for agent and developer iteration.
- R5. The THOX NVIDIA targets (sm_87, sm_89) fall inside the supported matrix.
- R6. New kernels fall back to the production CUDA C++ and CPU kernels when the device, driver or qualification is not there.
- R7. No device claim is made without a 30-run protocol on the target, and no speed claim without a like-for-like comparison.

Threat model. The adversary is not an attacker but a mistake: a human or an agent writes a kernel launch that aliases buffers, reads an output while the GPU writes it, or ships a binary to a host whose driver or toolkit cannot run it. Bounds errors are out of scope for compile-time checking; the Tile track keeps runtime bounds checks unless a kernel opts out [4]. Supply chain risk covers crates.io packages, the CUDA toolkit headers bound at build time, and the `tileiras` assembler taken from the toolkit at run time.

4 Design

4.1 Where CUDA Rust sits in ThoxOS

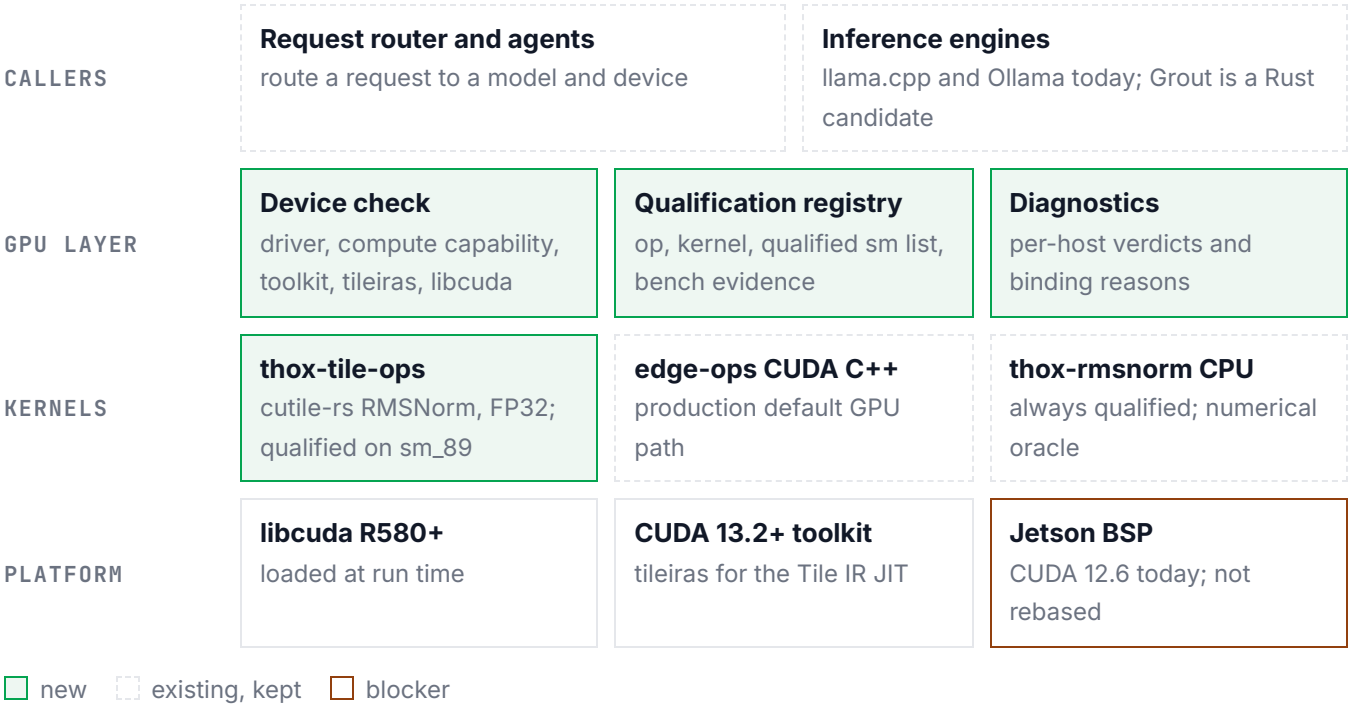


Figure 1. ThoxOS GPU layer. The device check and registry decide which kernel serves each operation; the production CUDA C++ and CPU kernels remain behind it.

Figure 1 shows the lane. CUDA Rust replaces nothing by default. A device check reads the driver, toolkit and device, a registry records which kernel is qualified on which compute capability, and existing kernels stay reachable. The cutile-rs runtime can launch external PTX or CUBIN modules alongside Tile kernels on the same stream [6], so the CUDA C++ kernels can move under one Rust host runtime before any of them is rewritten.

4.2 The ownership contract

Let I be the index set of an output tensor and let a launch partition it into tiles $S_1, \ldots, S_k$. The partition must cover the output without overlap, and tile block b may write only inside its own tile:

$$\bigcup_{b=1}^{k} S_b = I, \qquad S_a \cap S_b = \varnothing \ \ (a \neq b), \qquad W(b) \subseteq S_b \tag{1}$$

For the elementwise example, a length-N output partitioned into tiles of width B fixes the grid:

$$g = \frac{N}{B} = \frac{1024}{128} = 8 \tag{2}$$

PROPOSITION 1

If the host passes the output tensor into a launch by moving it into a partition, and passes each input either by value or by shared borrow, then no input of that launch aliases its output, and the host cannot read the output until the launcher returns it.

Proof sketch. Moving a value ends the caller's ownership; any later use is a use after move, which rustc rejects (E0382). Passing the same tensor as output and input requires either two moves of one value, rejected as above, or a move out of a value while a shared borrow of it is live, also rejected. Reading the output after the launch requires using the moved value, rejected again. The launcher returns ownership only when the device work completes, so the first legal host access follows synchronization. Together with the disjoint partition in Equation 1, writes within the launch touch disjoint tiles, and no read-write pair crosses input and output. The proof covers aliasing only; bounds and arithmetic correctness need runtime checks and testing.

Section 6.2 tests the four rejections this argument relies on. The paper behind the Tile track proves the stronger in-kernel race-freedom result [4]; our proposition is the host-side half that THOX code touches directly.

4.3 Selecting a kernel

Algorithm 1. Kernel selection in the ThoxOS GPU layer

```
report := device check (driver, compute capability, toolkit, cuRAND headers, tileiras, libcuda)
for op in [rms_norm (FP32)]:
  if Tile CAN RUN and registry qualifies the Tile kernel on report.sm: bind Tile
  else if the edge-ops CUDA C++ lane CAN RUN on report: bind edge-ops
  else: bind thox-rmsnorm CPU (always qualified)
  log the binding with the reason each earlier step failed
publish the report and bindings to the diagnostics page
```

A kernel enters the registry for a compute capability only after the Appendix C protocol passes there. A track that can run but holds no qualification for the host's compute capability never binds; the lab calls that state registered, not qualified. A GPU kernel never falls back silently inside its own launch: fallback is a bind-time decision, logged with its reason. Half-precision operations bind to the CPU path until a kernel meets the rounding contract in Section 7.

4.4 Security argument

The lane fails closed: without a driver the cutile runtime returns a typed error (Section 6.3) and selection falls through to CPU. Every crate version is pinned (`cutile = "=0.4.0"`), and the CUDA toolkit used for bindings is recorded at build time. The `unsafe` surface ThoxOS inherits is concentrated in the shared runtime that binds the driver API (Section 6.5), which is where review effort should go. Kernels in `thox-tile-ops` use only the safe subset.

5 Implementation

5.1 thox-tile-ops

The experiment harness (`exp/cuda_rust_experiments.py`) builds two small crates against cutile 0.4.0: a vector add taken from the announcement and a set of positive and negative compile cases. `thox-tile-ops` is a THOX crate with an FP32 RMSNorm Tile kernel, input validation, a CPU reference, host tests, device tests that are ignored by default, and a bench binary that implements the Appendix C protocol.

The kernel is FP32 only on purpose. The two existing THOX kernels disagree for FP16 and BF16: the edge-ops CUDA kernel rounds once after $x \cdot r \cdot w$, while thox-rmsnorm rounds $x \cdot r$ to the input type before multiplying by w. In FP32 the two contracts coincide, so `thox-tile-ops` takes no side until a half-precision contract is fixed (Section 7).

5.2 The ThoxOS GPU layer and its virtual lab

`thox-gpu-layer 0.1.0` is a dependency-free Rust crate that implements Algorithm 1. Its device check reads `nvidia-smi` (including the WSL2 path), the driver version, compute capability, the best CUDA toolkit present, the cuRAND header, the presence of `tileiras`, `libcuda` and `rustc`. Every failure degrades the report instead of panicking. The registry is seeded with one entry: `thox-tile-ops` RMSNorm, qualified on sm_89 by the run in Section 6.7. A diagnostics page is generated from a live probe and the lab profiles (Figure 5).

The virtual lab runs 8 host profiles through the same binder the real probe feeds, and `cargo test` asserts each binding. Two profiles carry the workstation's measured facts (before and after the CUDA 13.3.1 install); the other six are simulated from published requirements. Section 6.9 reports the matrix.

Table 4. Requirements and the tests that cover them.

Requirement	Test	Where
R1 build without GPU	E1 header matrix, E2 toolchain matrix, T1 check	Sections 6.1, 6.6
R2 aliasing rejected	E3 safety matrix, 4 negative and 2 positive cases	Section 6.2
R3 fail closed	E4 run without driver; T1 bench exit code; GPU layer probe on a GPU-less host	Sections 6.3, 6.6, 6.9
R4 short edit loop	E5 cold and incremental builds	Section 6.4
R5 target coverage	Device run on sm_89; sm_87 not run	Sections 3, 6.7
R6 fallback	GPU layer lab, 8 hosts; 3 crate tests	Section 6.9
R7 device and speed claims	30-run device protocol; same-day CUDA C++ comparison	Sections 6.7, 6.8

6 Evaluation

Two hosts. The GPU-less host (Sections 6.1 to 6.6, 6.9) is one x86_64 Linux VM with 2 vCPU, 7.8 GiB RAM and no GPU, with Rust 1.89.0, 1.97.0, 1.99.1 and CUDA toolkit headers from NVIDIA's PyPI wheels (CUDA 12.8, 13.3 and 13.4); crate under test cutile 0.4.0; upstream source read at commit `2ffc19b`. The device host (Sections 6.7 and 6.8) is the THOX RTX 4060 Ti workstation (sm_89): NVIDIA GeForce RTX 4060 Ti, compute capability 8.9, 16380 MiB, driver 630.20 (CUDA 13.4 user-mode driver), CUDA toolkit 13.3.1, Rust 1.99.1, Linux under WSL2, runs dated 2026-11-03.

6.1 Q1: can a GPU-less ThoxOS build host build it?

Table 5. Header matrix: `cargo check` of the vector-add crate on Rust 1.97.0, cold, by CUDA headers available (GPU-less host).

CUDA headers	Result	Reason	Seconds
No toolkit	**fails**	no toolkit found	20.8
CUDA 12.8	**fails**	toolkit too old (13.0+ required)	21.1
CUDA 13.3 without cuRAND headers	**fails**	cuRAND headers missing	21.7
CUDA 13.3	**builds**	builds	65.4
CUDA 13.4	**builds**	builds	64.6

Table 6. Toolchain matrix: cold `cargo check` with CUDA 13.3 headers (GPU-less host).

Rust	Result	Seconds
1.89.0	**builds**	74.0
1.97.0	**builds**	66.9
1.99.1	**builds**	67.4

cutile builds on every stable toolchain we tried, including the stated minimum 1.89 (3 of 3). It needs a CUDA 13 toolkit at build time because the shared bindings crate runs bindgen against `cuda.h`, and it also needs the cuRAND headers, even for kernels that never use random numbers. CUDA 12.8 headers fail with an explicit version message. No GPU and no driver are needed to build. R1 holds, with one cost: ThoxOS NVIDIA build images must carry CUDA 13 headers including cuRAND. Section 6.7 adds a second condition that a build check cannot see.

6.2 Q2: does the compiler reject aliasing misuse?

Table 7. Safety matrix on cutile 0.4.0, Rust 1.97.0. Each case was checked 3 times; results were identical across repetitions.

Case	What the host code does	Expected	Observed	Verdict
`p1_control`	Correct launch: partitioned output, two lazy inputs	compiles	compiles	**as expected**
`p3_shared_borrowed_input`	One materialized input borrowed for both shared parameters	compiles	compiles	**as expected**
`n1_alias_output_as_input`	Output tensor also passed as an input (blog case)	E0382	E0382	**as expected**
`n2_unpartitioned_output`	Mutable output passed without a partition	E0277	E0277	**as expected**
`n3_use_after_launch`	Host touches the output while the launch owns it	E0382	E0382	**as expected**
`n4_borrowed_input_aliases_output`	Output partitioned while a borrow of it is an input	E0382	E0382	**as expected**
`p2_shared_input_twice`	Same lazy input passed twice by value (reported, not a safety case)	compiles	E0382	**usability finding**

All 4 negative cases fail to compile with the expected error, and both positive controls compile. The blog's own example, passing the output as an input, fails with E0382 exactly as published [1]. An unpartitioned mutable output fails at the trait level (E0277) before borrow checking. These are the rejections Proposition 1 depends on, so R2 holds for the host side.

One case is a usability finding, not a safety one. Passing the same lazily created input to both read-only parameters fails with E0382, because the launcher takes lazy inputs by value. Materializing the tensor and passing two shared borrows compiles (`p3`). THOX code should materialize shared inputs before launch.

6.3 Q3: how does a binary fail without a driver?

We ran the release vector-add binary 3 times on the GPU-less host. Every run exited with status 1 and a typed driver error (code 3); none panicked. `ldd` shows no load-time dependency on `libcuda`, which the runtime opens on demand and searches across several paths, including the WSL2 shim (probed: yes). The binary is 12.2 MiB. R3 holds. The WSL2 probe matters because the THOX workstation builds and runs inside a WSL2 distribution.

6.4 Q4: what does it cost to build?

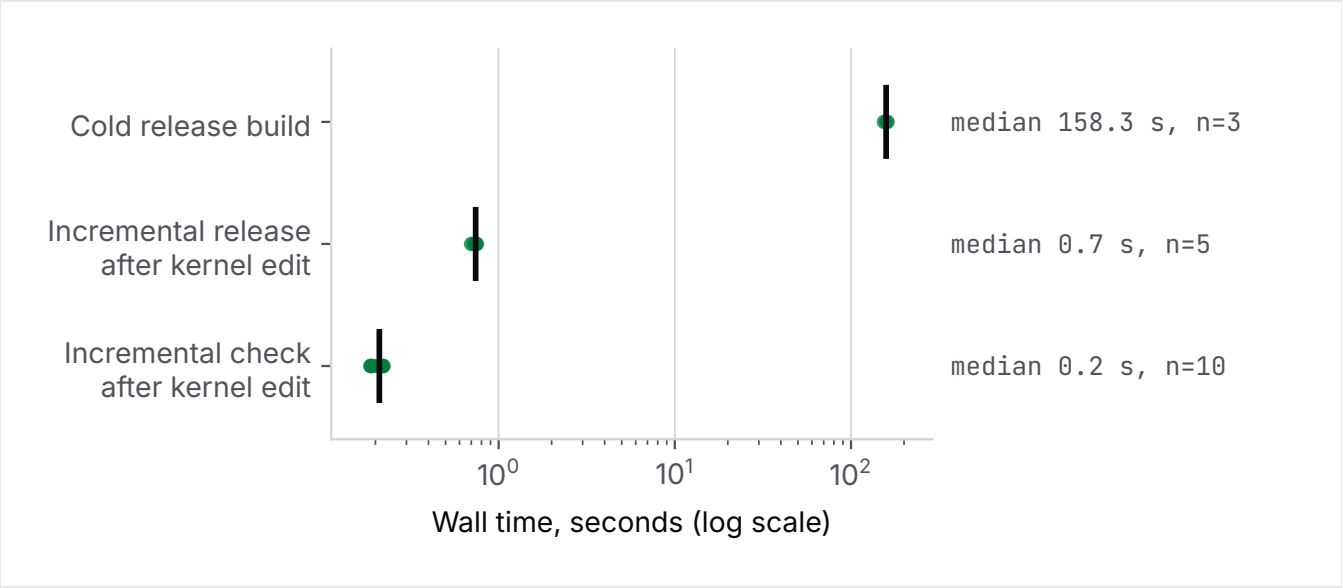


Figure 2. Build cost of the vector-add crate on 2 vCPU with no other load (GPU-less host). Points are repetitions; the bar marks the median. A kernel edit alternates the operand order in the kernel body, which changes the captured AST.

Figure 2 shows the edit loop. A cold release build takes a median 158 s (n=3, range 154 to 161 s). After a kernel edit, `cargo check` returns in a median 0.21 s (n=10) and a release rebuild in 0.74 s (n=5). Kernel compilation itself is deferred to the first launch on the device, so these numbers exclude JIT time; Section 6.7 measures that cost. R4 holds on this host.

6.5 Q5: what unsafe surface does ThoxOS inherit?

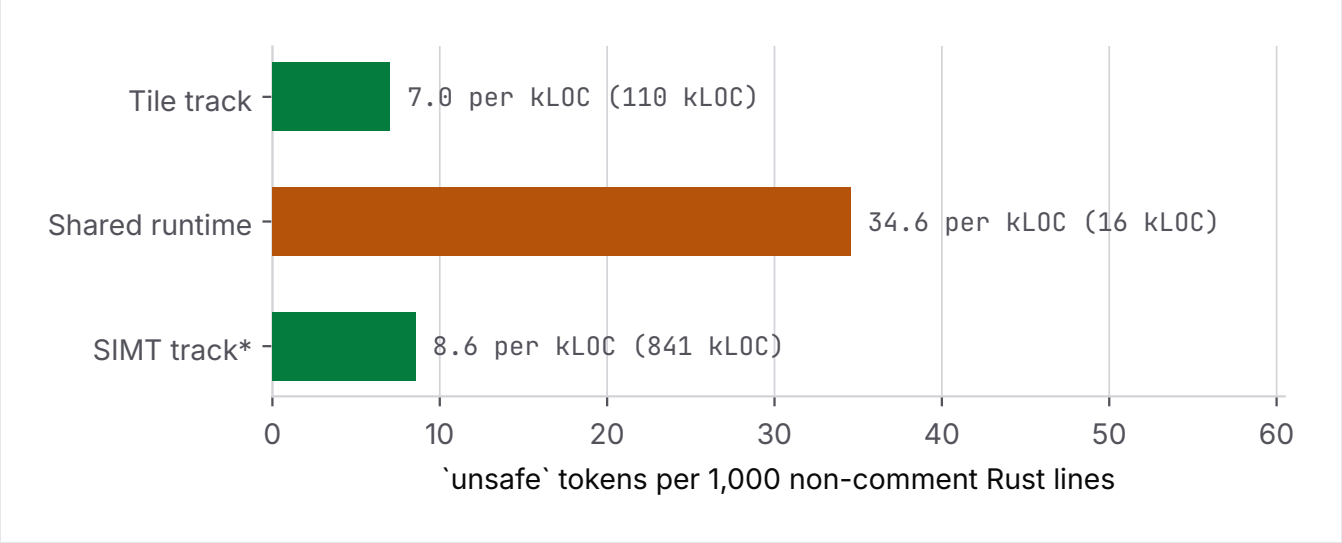


Figure 3. Density of the `unsafe` keyword in the upstream tree at the reviewed commit, read-only count over non-comment Rust lines. *The SIMT tree includes generated intrinsics, a vendored fuzzer and many examples, so its line count overstates hand-written code.

Figure 3 counts `unsafe` per thousand lines. The shared runtime that binds the driver API is the densest at 34.6 per kLOC across 16.4 kLOC, against 7.0 for the Tile track. That is expected for an FFI layer and it tells THOX where to spend review time. Two upstream details matter for THOX kernels. The upstream RMSNorm example carries a comment calling in-kernel `partition_mut` a safety leak if the partition is dropped and re-created, and the library's production norm kernels are `unsafe fn` with unchecked accesses for speed. Our kernel uses `partition_mut` once per launch and stays in the safe subset, but the leak is a property of the API, not something our code can rule out.

6.6 Q6: thox-tile-ops on a GPU-less host

`cargo check --all-targets` passes and `cargo clippy -D warnings` passes. The host tests pass 7 of 7, with 2 device tests ignored by design. The ignored no-driver test passes when run explicitly (1 passed), and the bench exits with code 2 and status `no_device`, as specified. Compiling the Rust host code does not compile the kernel, which happens through Tile IR on first launch; Section 6.7 is where it first ran.

6.7 Q7: does the Tile kernel run correctly on a THOX GPU?

On the THOX RTX 4060 Ti workstation (sm_89), with CUDA_HOME set to the 13.3.1 toolkit and a clean rebuild, the host tests pass 7 of 7 with 2 device tests ignored by default. Run explicitly, the device parity test `gpu_matches_reference_on_model_shapes` passes on all 6 model shapes (1 × 896, 1 × 2048, 1 × 4096, 8 × 2560, 32 × 3584, 128 × 4096): every output element matches the FP32 CPU reference within relative or absolute error 1.0e-5. This is the first THOX CUDA Rust kernel to run on a GPU.

Table 8. Tile RMSNorm bench on the THOX RTX 4060 Ti workstation (sm_89), FP32, 30 timed repetitions after 3 warmups. Timing is host wall clock around launch and synchronization, including output allocation and the copy back to host memory. First launch includes the Tile IR JIT.

Shape	Median	Q1	Q3	First launch	Max abs diff	Max rel diff	Deterministic
1 × 2048	0.0810 ms	0.0808 ms	0.0830 ms	186.4 ms	4.8e-7	3.0e-7	yes
128 × 4096	0.3118 ms	0.3090 ms	0.3169 ms	196.2 ms	4.3e-6	1.4e-6	yes

Both benches report status ok, parity pass and identical first and last outputs. The largest relative difference across the two benched shapes is 1.4e-6. The first launch takes 186.4 ms at 1 × 2048 and 196.2 ms at 128 × 4096, against fractions of a millisecond afterwards: the Tile IR JIT is a per-shape, first-call cost a serving engine must warm or cache.

> **FINDING: THE CUDA 13.2 FLOOR BINDS AT LAUNCH TIME, NOT BUILD TIME**
>
> Version 0.1.0 tested headers from CUDA 12.8, 13.3 and 13.4 and did not launch anything. On the workstation the first toolkit present was CUDA 13.0. With it, cutile built and the host tests passed, yet the first device launch failed with `failed to launch tileiras: No such file or directory`, surfaced as a typed Device error, no panic. The Tile IR assembler `tileiras` ships only with CUDA 13.2 and later toolkits. A CUDA 13.0 build therefore compiles and passes host tests while being unable to launch any Tile kernel. After installing CUDA 13.3.1 alongside 13.0, the same crate launched. A build check is not a readiness check: the GPU layer tests for `tileiras` directly (Section 6.9).

6.8 Q8: how does it compare with the production CUDA C++ kernel?

On the same host and day we ran the production edge-ops CUDA C++ `rms_norm_kernel` under the same protocol: FP32, eps 1e-6, identical generated inputs uploaded once outside the timing, one untimed first launch, 3 warmups, 30 timed repetitions, median and quartiles by the same method. The kernel's device code and launch geometry come verbatim from the production source (vector width 4, one block per row, 512 threads at 1 × 2048 and 1024 at 128 × 4096), compiled with `nvcc -O3 -arch=sm_89` from CUDA 13.3.1. The host driver is a bench harness: the library's PyTorch binding was not used, so a caller going through it would also pay operator-dispatch overhead that is not measured here.

Table 9. Launch-path comparison on the same host and day. Both sides time output allocation, launch, synchronization and copy back to host memory (C++ side: pageable host buffer). Ratios are quotients of the printed medians.

Shape	Tile median (Q1 to Q3)	C++ median (Q1 to Q3)	Faster launch path	First launch Tile / C++
1 × 2048	0.0810 ms (0.0808 to 0.0830)	0.0274 ms (0.0273 to 0.0278)	C++, 2.96x at 1 x 2048	186.4 ms / 0.86 ms
128 x 4096	0.3118 ms (0.3090 to 0.3169)	0.4726 ms (0.4624 to 0.4891)	Tile, 1.52x at 128 x 4096	196.2 ms / 1.11 ms

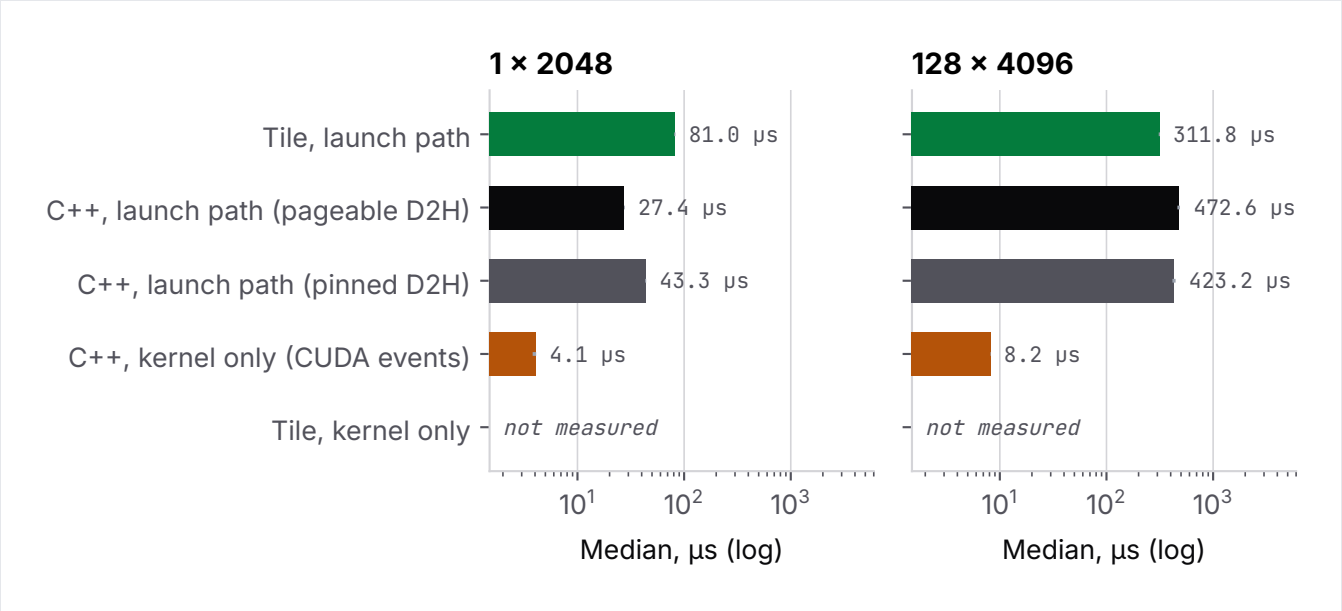


Figure 4. Medians with quartile bars, log scale, same host and day. The C++ kernel-only series uses CUDA events on a preallocated output; no Tile kernel-only series exists, so the figure supports no kernel-versus-kernel reading.

The result splits by shape (Table 9, Figure 4). At 1 × 2048 the C++ launch path is 2.96x faster; at 128 × 4096 the Tile launch path is 1.52x faster. These numbers come with caveats that belong next to them, not in a footnote:

- Both wall-clock figures are dominated by overhead. The C++ kernel alone, timed with CUDA events on a preallocated output, takes 4.1 µs at 1 × 2048 and 8.2 µs at 128 × 4096. The remaining tens to hundreds of microseconds on both sides are allocation, launch, synchronization and copy-back.
- No kernel-only Tile figure exists; the Tile bench never measured one. We therefore make no kernel-versus-kernel speed claim. What was compared is the full launch path each implementation presents today.
- First launches are not comparable: 186.4 ms and 196.2 ms on the Tile side include the Tile IR JIT; 0.86 ms and 1.11 ms on the C++ side are context and module load.
- Copy-back differs. The C++ series copies to pageable host memory; the buffer type the Tile runtime uses internally is not visible from its bench. At 128 × 4096 the output is 2 MiB, where this choice matters. A pinned-memory C++ series measured 0.4232 ms at 128 × 4096: it narrows the gap but does not close it.
- Deallocation differs: the C++ clock stops before `cudaFree`, while whether the Tile timing includes dropping its output tensor cannot be read from its bench. This favours the C++ side by an unmeasured amount. The allocators also differ.
- Small-shape ratios are approximate. A pinned C++ series at 1 × 2048 measured 0.0433 ms, slower than the pageable 0.0274 ms although its copy path is nominally faster; at 8 KiB of output the difference is process-to-process launch noise.
- Both sides ran under the same WSL2 driver shim on the same day, so the comparison is internally fair; absolute numbers may differ on bare-metal Linux.

Correctness agrees exactly. The C++ kernel passes parity at both shapes, and its maximum absolute and relative differences from the FP32 reference (4.8e-7 and 3.0e-7 at 1 × 2048; 4.3e-6 and 1.4e-6 at 128 × 4096) are identical to the Tile side's, which cross-checks both benches' data generation and references. The honest headline is a split result. The case for the Tile track rests on correctness and authoring, not on speed.

6.9 Q9: does the GPU layer bind the right kernel?

Table 10. GPU layer virtual lab. Every profile binds as expected (8 of 8); the crate's 3 tests assert the matrix.

Host profile	Facts	Binding	Reason
RTX 4060 Ti workstation, CUDA 13.3	measured facts	`tile`	Tile can run and is qualified on sm_89
Same workstation, CUDA 13.0 only	measured facts	`edge-ops`	Tile blocked: toolkit below 13.2, tileiras missing
Jetson Orin NX, JetPack 6 (current BSP)	simulated	`edge-ops`	Tile blocked: driver below R580, toolkit 12.6
Jetson Orin NX, JetPack 7.2.1 (evaluation)	simulated	`edge-ops`	Tile can run but holds no sm_87 qualification
Apple silicon Mac	simulated	`cpu`	No NVIDIA driver or GPU
Linux, no driver installed	simulated	`cpu`	No driver: fail closed with typed reasons
Ada workstation, CUDA 12.8, driver 570	simulated	`edge-ops`	Tile blocked: driver and toolkit below floor
GB10 (sm_121)	simulated	`cpu`	Tile not qualified on sm_121; sm_121 outside the edge-ops build range the lab encodes

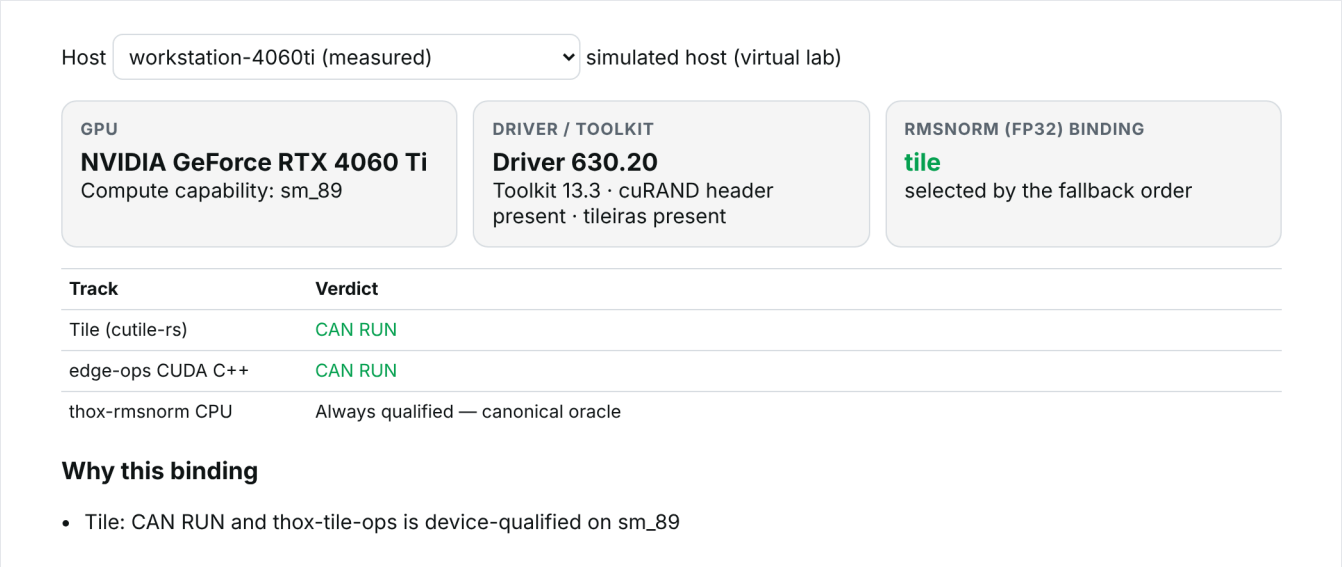


Figure 5. The GPU layer's diagnostics page, generated on the GPU-less paper build host, with the lab profile that carries the workstation's measured facts selected. The page labels it a simulated host because the binding is computed from recorded facts rather than a live probe.

Two real probes back the lab. On the workstation the probe reports the RTX 4060 Ti, driver 630.20, sm_89, toolkit 13.3 with `tileiras` present, and binds `tile`. On the GPU-less paper build host it binds `cpu` with typed reasons for every blocked step (Figure 5 shows the page). Binding here is the GPU layer's selection. THOX inference engines do not consult the GPU layer yet, and their production default remains the edge-ops CUDA C++ kernel (Section 7).

7 Discussion and limitations

What improves for ThoxOS. One language from microvisor to kernel means one cargo workspace, one review culture and one set of agent rules. The ownership contract turns a class of launch bugs that rarely reproduce into compile errors, which matters more because THOX agents now write code. Interop with external PTX lets the existing CUDA C++ kernels move under a Rust host runtime first. And the Tile track's architecture-agnostic source suits a fleet that mixes Ada desktops with Ampere modules.

7.1 Decisions of record

THOX recorded the following outcomes on 2026-11-03, after the device run and the comparison:

Table 11. Outcomes recorded by THOX after the device evidence.

Question	Outcome
Kernel ownership	The edge-ops library is the production home for GPU `rms_norm`, `fused_add_rms_norm` and `silu_and_mul`. thox-rmsnorm is the canonical CPU RMSNorm and the family's numerical oracle. thox-tile-ops lives in its own repository and is not merged into edge-ops.
Half precision	thox-tile-ops stays FP32. Any FP16 or BF16 kernel must match the CPU oracle with FP32 accumulation within a documented tolerance, with its rounding mode named.
Adoption gate	Met for thox-tile-ops RMSNorm on sm_89: device parity, 30-run benches, the same-day CUDA C++ comparison, driver and toolkit verified on the host, fail-closed behaviour kept. The Tile track is adopted for new NVIDIA kernel authoring on qualified hosts, behind the fallback order.
Production default	Unchanged. Changing it is a founder decision; the comparison is split by shape, the first-launch JIT costs 186 to 196 ms, and no kernel-only Tile timing exists.
Jetson	Not rebased. The current BSP and the sm_87 CUDA C++ kernels remain the Jetson path until a JetPack 7.x Orin device run passes the same protocol.
SIMT track	Source review only; no adoption and no proof-of-concept spend.

7.2 Negative results and open tensions

- Requirements moved within weeks of the announcement (Table 1). THOX should pin exact crate versions and re-review on every upgrade.
- Build images need CUDA 13 headers including cuRAND even when no kernel uses random numbers (Table 5), and launch needs CUDA 13.2 or later for `tileiras` (Section 6.7). The two floors differ, and only the second is visible at run time.
- Lazy inputs cannot be shared across two parameters without materializing them first (Section 6.2).
- The Tile IR JIT adds 186 to 196 ms to the first call per shape. A serving engine needs a warm-up or a persistent JIT cache before Tile kernels sit on a latency path.
- The selection rule in v0.1.0's protocol asked that the Tile median be no slower than the CUDA C++ median beyond quartile overlap before Tile could win selection. The gate THOX adopted requires the comparison to be run, not won. Under the v0.1.0 rule the result qualifies at 128 × 4096 and not at 1 × 2048. The GPU layer binds by qualification, not by shape, so on sm_89 it selects Tile at every shape. This is harmless while engines keep the CUDA C++ default, and it must be settled before any engine consults the GPU layer.
- Jetson Orin cannot run either track on the current L4T r36 BSP; JetPack 7.2.1 would allow the Tile track on paper, but CUDA 13.2.2 there lacks Tile IR 13.3 features such as allocation and gather or scatter views, and no Orin run exists.

Limitations. GPU-less host: one 2-vCPU x86_64 VM; 3 repetitions per safety case and per runtime run, 3 cold builds, 10 incremental checks and 5 incremental release builds, so timing spread there is indicative only. Device host: one RTX 4060 Ti under WSL2, one day; the six-shape parity result, the host test counts and the CUDA 13.0 launch failure come from the device run record, and the raw test log is not part of the evidence set; benches cover two of the six shapes. Not evaluated: a kernel-only Tile timing, a resident-output serving profile, FP16 and BF16, the SIMT track (building the upstream compiler was not possible in

the review environment), aarch64 and Jetson, Windows native and bare-metal Linux. No third-party performance figure in this report is a THOX measurement. This is a preprint and has not been peer reviewed.

8 Conclusion

CUDA Rust's Tile track fits ThoxOS's build and safety needs: it builds on stable Rust without a GPU, rejects the aliasing mistakes it promises to reject, fails closed without a driver, and its first THOX kernel matches the CPU reference on 6 model shapes on an RTX 4060 Ti. It is not faster across the board: against the production CUDA C++ kernel the launch path wins at 128 × 4096 and loses at 1 × 2048, and we cannot yet compare the kernels alone. THOX has adopted the track for new NVIDIA kernels behind a fallback order and kept the production default where it is. The next evidence that would change that decision is a kernel-only Tile timing, a warm-cache serving profile and an Orin run.

9 Roadmap

Table 12. Phases, status and exit evidence.

Phase	Status	Exit evidence
0 Review	**Done (v0.1.0)**	summary.json, safety matrix, crate compiles and host tests pass
1 Device qualification on sm_89	**Done (this edition)**	Parity on 6 shapes; 30-run JSON at 1 × 2048 and 128 x 4096; JIT first-launch time
2 Same-day CUDA C++ comparison	**Done (this edition)**	C++ series at both shapes, pageable, pinned and kernel-only
3 GPU layer	**Built; not consulted by engines**	Probe, registry, fallback, diagnostics; virtual lab passes
4 Kernel-only Tile timing and warm-cache profile	Open	Tile timing with resident output; JIT cache behaviour across restarts
5 Jetson path	Open	JetPack 7.x Orin run under the same protocol, power mode recorded; BSP decision after
6 FP16 and BF16	Open	Rounding study against the CPU oracle, then half-precision kernels
7 SIMT evaluation	Open	Only if a kernel needs warp-level control

A Appendix A: reproducibility

GPU-less environment: Rust 1.89.0: `rustc 1.89.0 (29483883e 2025-08-04)`, Rust 1.97.0: `rustc 1.97.0 (2d8144b78 2026-07-07)`, Rust 1.99.1: `rustc 1.99.1 (58cf8f9c3 2026-10-13)`. CUDA tools: 12.8: `Build cuda_12.8.r12.8/compiler.35583870_0`, 13.3: `Build cuda_13.3.r13.3/compiler.38244171_0`, 13.4: `Build cuda_13.4.r13.4/compiler.38855100_0`. Device host: `Build cuda_13.3.r13.3/compiler.38244171_0`, `rustc 1.99.1`.

```
# GPU-less experiments (v0.1.0), from the pack root
pip install --no-deps --target cuda13/wheels 'nvidia-cuda-runtime==13.3.*' 'nvidia-cuda-nvcc==13.3.*'
\
  'nvidia-cuda-crt==13.3.*' 'nvidia-nvvm==13.3.*' 'nvidia-cuda-nvrtc==13.3.*' nvidia-curand
rustup toolchain install 1.89.0 1.99.1 --profile minimal
python3 exp/cuda_rust_experiments.py --root . --cold-reps 3 --warm-reps 10 --safety-reps 3
python3 exp/thox_tile_ops_check.py --root . && python3 exp/analyze.py --root .
# Device protocol (Appendix C) on an sm_89 host with CUDA 13.3, inside thox-tile-ops
cargo test --release -- --ignored gpu_matches_reference_on_model_shapes
thox-tile-bench --rows 1 --hidden 2048 --reps 30 --warmup 3 --out bench-1x2048.json
thox-tile-bench --rows 128 --hidden 4096 --reps 30 --warmup 3 --out bench-128x4096.json
# Evidence set and paper (v0.2.0)
python3 exp/ingest_v020.py --root .
python3 wp/figures.py --root . && python3 wp/build.py --root .
node wp/katex_render.cjs wp/build/paper_raw.html wp/build/paper.html && python3 wp/pdf.py --root .
```

The device benches and the C++ series are copied into `exp/results/` unchanged in value; `exp/results/SANITIZE_MANIFEST.json` lists every file with the SHA-256 of the incoming file and of the copy, and the only rewrites are a neutral host label and build-machine path prefixes. The CUDA C++ bench harness embeds production kernel source and is not part of this pack. Harness corrections from v0.1.0 stand: the first CUDA 13.4 header set lacked cuRAND, a reinstall with `pip --target --upgrade` removed other headers, and the first no-device bench wrote unescaped JSON; Table 5 reports the corrected run.

B Appendix B: notation

Table 13. Symbols.

Symbol	Meaning
I	Index set of an output tensor
S_b	Tile owned by tile block b
$W(b)$	Indices written by tile block b
N, B, g	Output length, tile width, launch grid
x, w, r	RMSNorm input row, weight, inverse RMS

C Appendix C: device qualification protocol

1. Host record: GPU model, compute capability, driver version, CUDA toolkit version and the presence of `tileiras`, OS or WSL2 distribution, power mode for Jetson.
2. Build: `CUDA_HOME= cargo build --release` in `thox-tile-ops`; record the commit. A build alone does not show that a launch will work.
3. Parity: `cargo test --release -- --ignored gpu_matches_reference_on_model_shapes`; pass means relative or absolute error at most 1e-5 for every shape.

4. Timing: for each shape run `thox-tile-bench --rows R --hidden H --reps 30 --warmup 3 --out R_H.json` with no other GPU or heavy CPU work; keep the raw JSON. Report the first launch separately because it includes the Tile IR JIT.
5. Comparison: run the production CUDA C++ kernel on the same shapes, host and day with the same 30-run rule; report medians and quartiles side by side with the timing definition of each side, never a single best run, and add a kernel-only series on both sides before any kernel-versus-kernel statement.
6. Qualification: parity, the 30-run benches, the comparison and a verified driver and toolkit qualify the kernel for that compute capability in the GPU layer's registry. Qualification permits selection; it does not change a production default.

D Appendix D: document control

Edition	Date	Change
v0.1.0	2026-11-03	Internal preprint: review, GPU-less experiments, thox-tile-ops crate, device protocol.
v0.2.0	2026-11-04	Device qualification on sm_89 (Section 6.7) and the CUDA 13.2 launch-time finding; same-day CUDA C++ comparison with caveats (Section 6.8); ThoxOS GPU layer and virtual lab (Sections 4.3, 5.2, 6.9); decisions of record (Section 7.1); status, abstract, limitations and roadmap updated; author blocks with roles; public-edition sweep of internal identifiers; protocol moved to Appendix C and updated to the adopted gate; Template v1.0 design.

Report number THOX-TR-2026-11-03. Report number unchanged across editions. THOX.ai has no relationship with NVIDIA beyond being a customer of its products. Copyright 2026 THOX.ai LLC. THOX® and ThoxOS™ are marks of THOX.ai LLC. CUDA and NVIDIA are trademarks of NVIDIA Corporation.

References

[1] Sri Koundinyan, Melih Elibol, Jonathan Bentz. *Introducing CUDA Rust: Two Tracks for Writing GPU Kernels*. NVIDIA Technical Blog, 2026 (Sep 8). https://developer.nvidia.com/blog/introducing-cuda-rust-two-tracks-for-writing-gpu-kernels/

[2] NVIDIA. *Tile IR*. CUDA documentation, 2026. https://docs.nvidia.com/cuda/tile-ir/latest/index.html

[3] Philippe Tillet, H. T. Kung, David Cox. *Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations*. MAPL 2019 (co-located with PLDI 2019), 2019. https://pldi19.sigplan.org/details/mapl-2019-papers/1/Triton-An-Intermediate-Language-and-Compiler-for-Tiled-Neural-Network-Computations

[4] Melih Elibol, Jared Roesch, Isaac Gelado, Eric Buehler, Michael Garland. *Fearless Concurrency on the GPU*. arXiv:2606.15991v2 [cs.PL], 2026. https://arxiv.org/abs/2606.15991

[5] NVIDIA. *cutile 0.4.0 (published 2026-09-25, rust-version 1.89, Apache-2.0)*. crates.io, 2026. https://crates.io/crates/cutile

[6] NVIDIA. *NVIDIA/cuda-rust: NVIDIA's CUDA platform for Rust (commit 2ffc19b, 2026-11-03)*. GitHub repository, Apache-2.0, 2026. https://github.com/NVIDIA/cuda-rust

[7] NVIDIA. *NVlabs/cutile-rs (archived; development moved to NVIDIA/cuda-rust)*. GitHub repository, 2026. https://github.com/NVlabs/cutile-rs

[8] Hugging Face. *huggingface/grout: Testbed for LLM inference with cutile-rs*. GitHub repository, Apache-2.0, 2026. https://github.com/huggingface/grout

[9] Eric Buehler and contributors. *EricLBuehler/mistral.rs: Fast, flexible LLM inference*. GitHub repository, MIT, 2026. https://github.com/EricLBuehler/mistral.rs

[10] Bastian Köpcke, Sergei Gorlatch, Michel Steuwer. *Descend: A Safe GPU Systems Programming Language*. PLDI 2024 (doi:10.1145/3656411); arXiv:2305.03448, 2024. https://arxiv.org/abs/2305.03448

[11] Rust-GPU contributors. *Rust-GPU/rust-cuda: Ecosystem of libraries and tools for writing and executing fast GPU code fully in Rust*. GitHub repository, Apache-2.0 or MIT, 2026. https://github.com/Rust-GPU/rust-cuda

[12] Rust-GPU contributors. *Rust-GPU/rust-gpu: Making Rust a first-class language and ecosystem for GPU shaders*. GitHub repository, Apache-2.0 or MIT, 2026. https://github.com/Rust-GPU/rust-gpu

[13] Tracel AI. *tracel-ai/cubecl: Multi-platform high-performance compute language extension for Rust*. GitHub repository, MIT or Apache-2.0, 2026. https://github.com/tracel-ai/cubecl

[14] cudarc contributors. *chelsea0x3b/cudarc: Safe rust wrapper around CUDA toolkit*. GitHub repository, Apache-2.0 or MIT, 2026. https://github.com/chelsea0x3b/cudarc

[15] NVIDIA. *The Rust + GPU Ecosystem (cuda-oxide book appendix)*. CUDA Rust documentation, 2026. https://nvidia.github.io/cuda-rust/cuda-oxide/appendix/ecosystem.html

[16] Eugene Hauptmann, Nataliya Kosmyna. *RLX: A Unified Multi-Backend Tensor Compiler and Distributed Runtime in Rust*. arXiv:2609.37916v1, 2026. https://arxiv.org/abs/2609.37916

[17] Dipankar Sarkar. *The Correctness Illusion in LLM-Generated GPU Kernels*. arXiv:2606.20128v2, 2026. https://arxiv.org/abs/2606.20128

[18] Ashkan Vedadi Gargary, Guido Martínez, Sebastian Burckhardt, Gabriel Ebner, Abhinav Jangda, Madan Musuvathi, Tyler Sorensen. *RESOLVE: Language-Agnostic Validation of GPU Kernels Through Testing, Reduction, and Proof*. arXiv:2610.05683v1, 2026. https://arxiv.org/abs/2610.05683

[19] NVIDIA. *JetPack 6.2 Release Notes (Jetson Linux 36.4.3, CUDA 12.6.10)*. NVIDIA Jetson documentation, 2025. https://docs.nvidia.com/jetson/archives/jetpack-archived/jetpack-62/release-notes/index.html

[20] NVIDIA. *JetPack 7.2.1 downloads and notes (Jetson Linux 39.2.1, CUDA 13.2.2, Orin family)*. NVIDIA Developer, 2026. https://developer.nvidia.com/embedded/jetpack/downloads/archive-7.2.1.md