Reverse Engineering the CUDA Software Stack: An Independently Implemented GPU Machine-Code Toolchain for the NVIDIA GB10

M. Drake Stapleton
Independent Researcher, Aien, Springfield, MO, USA
drake@aienos.com

Version 2, October 26, 2026

Abstract

I describe CUDA++, a research-stage GPU machine-code toolchain and runtime for the NVIDIA GB10 in a DGX Spark. It provides an execution path for demonstrated workloads without NVIDIA's proprietary CUDA user-space compiler, runtime, driver library or mathematics libraries. It is not a drop-in implementation of the CUDA language, APIs or library ecosystem. The path still uses NVIDIA's open Linux kernel modules and signed GPU firmware. I built the first-party implementation by directing AI coding agents, using public interface headers, vendor tools applied offline to my own probe kernels, Mesa's NAK/NVK sources, published technical references and hardware experiments. The implementation includes a driver client of about 900 lines, a kernel builder with 59 intermediate-representation instruction kinds, register allocation and direct launch-descriptor construction. Archived results include exhaustive GPU/CPU-sequence parity for nine declared FP32 operations, a 1,048,576-element vector-add run whose code and output hashes match its host-simulator record, and a live 35B model-serving process with no matched CUDA-library mappings. Numerical parity is distinct from correctly rounded mathematics: the contracts include error bounds and a restricted SIN/COS domain. Model-output evidence is limited to specified prompts and includes a known near-tie token difference. Throughput and launch timings are reported as workload-specific observations, not physical ceilings. This revision corrects the earlier scope, runtime-gating and evidence terminology; distinguishes historical results from the October 26 source snapshot; and identifies missing M16 raw logs that prevent full re-audit of that qualification. No new hardware measurements are introduced.

Keywords: GPU computing; reverse engineering; machine-code generation; device drivers; CUDA; Blackwell; reproducibility

1 Scope and contribution

The conventional NVIDIA CUDA software stack supplies compilers, a runtime, a user-space driver library and domain libraries. My goal was to implement the path needed by my workloads without depending on those proprietary user-space components. CUDA++ communicates with NVIDIA's open Linux kernel modules and generates GPU machine code directly. This is a bounded engineering result on one NVIDIA DGX Spark, using the GB10 target identified by the toolchain as `sm_121`, rather than a claim that the whole CUDA ecosystem has been replaced.

The main contributions are an RM/UVM driver client, an encoder and kernel builder for a subset of the GB10 instruction set, command-list construction, and a collection of host tests and

archived chip experiments. The implementation and historical measurements are public, but their evidence is not uniform: some results have raw logs and hashes, some are summaries, and some planned capabilities have no hardware qualification. I distinguish these cases throughout.

Version 1 was published on October 22, 2026 [1]. I preserve its historical results where the evidence supports them and correct broader interpretations. The historical CUDApp snapshot used here is `232332422bfd`; the later snapshot is `fbbf3af40db8`. Full hashes and immutable source links are in Section 7. A result measured on an earlier implementation is not automatically a qualification of either snapshot.

2 Sources and implementation method

2.1 Public interfaces and prior work

NVIDIA's open kernel-module headers define the RM ioctl interfaces, object classes and UVM controls used by the client. CUDA++ includes 67 interface headers from release 580.173.02 with their NVIDIA MIT notices. These user-space interface definitions should be distinguished from the combined Linux kernel module, which NVIDIA licenses under MIT/GPLv2 [3, 4].

I used `nvcc` 13.0.88 and `cuobjdump -sass` offline on small CUDA C++ kernels written for probing. Changing one register, immediate, predicate or modifier at a time gave candidate encoding rules. The research use of vendor tools is separate from the resulting execution path, which does not require them. Some optional host cross-checks still use `nvdisasm` when installed; that test dependency has not been retired.

Mesa's NAK encoder and latency/QMD definitions, and NVK's compute-dispatch sequence, supplied hardware facts and implementation references. The repository attributes these uses and reports that no Mesa source code was copied [3, 5]. NVIDIA's PTX documentation and Binary Utilities instruction-set references also informed the work. It is too broad to call SASS wholly undocumented: NVIDIA publishes mnemonic-level references and disassembly tools. Recovering the binary fields and scheduling details needed here still required source inspection and experiments [6].

The tensor-arithmetic model also draws on published numerical research, explicitly credited in its source. The model header identifies the work of Fasi, Higham, Mikaitis and Pranesh [2] as a basis for candidate accumulation models [7]. Hardware observations test those candidates; agreement with a finite probe set is not a complete mathematical specification of the tensor unit.

2.2 Authorship and provenance

I set the architecture, directed AI coding agents (Claude, Codex, Grok and Gemini), and selected or rejected their first-party implementation work. The repository records this AI-assisted method and the identities in its commit metadata. NVIDIA's retained headers and other acknowledged third-party material are not my authorship.

Use of AI tools. I directed this work and am responsible for its content. AI coding agents (the Claude, Codex, Grok and Gemini models named above) wrote most of the first-party code and ran the hardware measurements under my direction. AI writing assistants, including Anthropic Claude models used through the omp agent harness, drafted and revised the text of this paper under my direction. No AI tool is an author. Every number in this paper is tied to a public log, commit or pull request listed in the bibliography.

The extraction and later provenance audits report scoped checks of commit identities, notices, credential patterns and binary content. They do not prove the origin of every line, authenticate

every historical account, or constitute a legal clean-room certification [8]. Accordingly, this revision uses "independently implemented" in the title and states the actual sources and boundaries rather than implying a separately certified clean-room process. I did not disassemble, decompile or patch proprietary NVIDIA programs; the documented oracle procedure inspected output generated from my own probes. The historical extraction audit reports removal of a vendor-generated disassembly listing from the published CUDApp cut; source repositories and extracted histories must not be conflated.

3 The execution path

3.1 Driver client and memory

The client in `nvrm/nvrm.c` is 908 lines in the reviewed October 26 snapshot. It uses `/dev/nvidiactl`, a GPU device file and `/dev/nvidia-uvm`. The implemented sequence is [9]:

1. Allocate the RM client, device and subdevice; query the class list and require the expected compute, GPFIFO and usermode classes.
2. Create a GPU virtual-address space and register it with UVM, which manages the page tables.
3. Allocate system-memory buffers and map each at corresponding CPU and GPU virtual addresses. For these allocations there is one underlying shared-memory buffer rather than separate host and device copies.
4. Create the channel group, context share and channel. A GPU-uncached 64 KiB allocation holds the GPFIFO ring and USERD page; the compute object is created beneath the channel.
5. Write an 8-byte ring entry, issue the required memory barriers, update GPPut and ring the usermode doorbell with a 32-bit store.
6. Poll a completion word under bounded wait rules. An uncertain completion leads the session to retain potentially live resources and refuse further work on that session.

The established enqueue-and-ring fast path has no system call. Initialization, allocation, diagnostics and other runtime operations are outside that statement. Cacheability and CPU/GPU visibility are explicit platform requirements; the current path requires GPU-uncached storage for host-polled synchronization words. This is a contract for the tested driver/chip path, not a universal rule for all GPU memory systems.

The M16 qualification programs implement doorbell-withheld/doorbell-rung controls, process-map inspection and two-channel checks. However, the imported M16 summary references a raw-evidence directory and a superseded receipt absent from CUDApp. Its historical PASS fields remain data, not independently re-audited proof. The October 26 README explicitly labels current native qualification `NOT_RUN` and requires a new authorized run with retained raw logs [10]. Other archived chip experiments are discussed separately below.

3.2 Machine-code generation and submission

The `BlackwellIROpcode` enumeration contains 59 instruction kinds. The implementation includes virtual registers, register allocation, instruction encoders and QMD construction. This is an implemented subset, not the complete hardware ISA. Golden-word tests and optional disassembler cross-checks cover particular encodings; they do not by themselves establish every operand combination or scheduling behavior on hardware [11].

The fixed unsigned-integer vector-add encoder contains 32 128-bit instructions: 512 bytes, or 128 32-bit words. The later `hello_gpu` floating-point example uses the kernel builder and has its own archived code hash. Keeping these examples distinct avoids treating every vector-add record as the same program.

The BF16 tensor probe defines 2,048 cases with operand classes intended to distinguish candidate HMMA accumulation models. The source permits separate confirmation seeds and a dropped-accumulator negative control. This supports a repeatable model-fitting method, not exhaustive characterization of all BF16 matrices [7].

Command lists supply program addresses, arguments, QMDs, cache operations and launch commands. Source comments attribute dispatch ordering to Mesa NVK. Actual timings depend on barriers and workload configuration; Section 5.4 gives the measured scope rather than one universal launch cost.

3.3 What the zero-CUDA checks establish

The repository contains several distinct mechanisms [12]:

- `tools/zero-cuda-gate.sh` checks selected ELF dependencies and imported symbols, with a synthetic negative control.
- The M16 qualification executable inspects its process mappings for `libcuda`/`libcudart` and fails that test when detected.
- Several submit wrappers populate optional diagnostic fields for linkage, symbols and process mappings. They do not make those fields a mandatory pre-launch refusal: callers can omit the execution-info output, and the returned flags are not themselves launch-admission conditions.

Thus I do not claim a universal runtime gate that refuses any NVIDIA library in every calling process. A process-map snapshot establishes what was mapped at its observation time, subject to the stated matching rule; an ELF check establishes the inspected dependency surface. Neither removes the kernel modules or firmware, and neither is a universal proof against later dynamic loading.

4 Evidence classes used in this paper

Each quantitative statement below carries one of four labels. **MEASURED (chip)**: read from a raw log or receipt produced on the GB10 and archived at the linked commit. **MEASURED (host)**: read from a host or simulator log. **REPORTED**: stated in a pull-request description or summary file without an archived full transcript. **DERIVED**: computed from measured values by a stated formula; I rechecked each derived value from the logged parameters. This revision adds no new measurement; no binary was rebuilt to confirm a recorded binary digest.

5 Archived results and their limits

5.1 Numerical closure

The E1 closure summary records 380 reduction cases for SUM, MAX, MIN and MEAN, 148 load/store specifications, and a 26-test Gate 5 result. It also combines exhaustive GPU/CPU-sequence parity for nine FP32 operations [13]. The earlier four-operation run used Omega `4863803a47f2`; the later five used `fb36109d39bd`. The second commit sits on a feature branch and is not an ancestor

Table 1: Quantitative claims, their evidence class and their artifact.

Claim	Class	Artifact
380 reduction cases, 148 load/store specifications, Gate 5 26 tests, all PASS	MEASURED (chip), summary	[13]
2^{32} inputs per operation, nine operations, 0 mismatches, 0 unwritten	MEASURED (chip), raw logs	[14]
Declared maximum ULP per operation (Table 2)	contract, host-measured	[15]
1,048,576 elements, 0 wrong, matching code and output hashes	MEASURED (chip and simulator)	[16]
No mapped CUDA library in the serving process	MEASURED (process snapshot)	[17]
32/32 on three prompts, 31/32 on a fourth, 24/24 HTTP tokens	MEASURED (chip)	[18]
115.14 and 115.36 TFLOPS	DERIVED from measured median times	[19]
195.1 and 207.7 GB/s	DERIVED from measured times, modified tree	[19]
25.09 to 31.97 μs and 2.68 to 9.68 μs per added launch	DERIVED from measured spans	[20]
94/94 submit gate, 54 swap checks	REPORTED (host)	[21]

of the Omega main branch; the closure instead records that binaries rebuilt on merged main (`40d1ea378617`) equal the chip binaries by digest. Closure records equality of relevant earlier kernel digests with the candidate, and separates the merged-main host rerun from the chip execution. The closure file's name is the SHA-256 of its canonical content without the digest field; that digest, and the digests of the two transcendental receipts and their raw logs, recompute exactly. Its digests are integrity checks, not authenticity.

The raw transcendental logs report $2^{32} = 4{,}294{,}967{,}296$ input encodings per operation with zero mismatches and unwritten outputs [14]. The comparison target is the frozen `omega_math_*` CPU sequence, including exceptional-value rules. This is bitwise implementation parity, not a claim that all nine mathematical functions are correctly rounded over their usual real domains.

Table 2: Declared numerical contracts accompanying the exhaustive parity claim. The error bounds are those reported by the CPU numerical contract, not a new accuracy measurement.

Operation	Declared max ULP	Important boundary
EXP2, LOG2	2	Base-two operations; specified exceptional-value behavior.
SIGMOID, TANH	3	Defined saturation and signed-zero rules.
ERF, GELU	3	GELU uses the erf form; specified tail behavior.
SIN, COS	2	Admitted domain $\lvert x\rvert \le 2^{22}$; outside it returns canonical NaN.
RSQRT	0	Correct-rounding contract for positive inputs; specified zero, infinity and negative-input results.

The contract specifies operation order, rounding environment, canonical NaNs and subnormal treatment. Its mathematical-accuracy comparison uses a binary128 reference with a documented hard-to-round caveat; the nonzero declared bounds include a margin. An exhaustive encoding sweep includes intentional NaN outputs outside the admitted trigonometric domain. Natural-base EXP and LOG are not GB10 kernels in this closure and must not be inferred from EXP2 and LOG2 [15].

5.2 Vector-add correspondence

The October 22 `hello_gpu` chip log records 1,048,576 checked FP32 additions, zero wrong results and zero never-written values, with exit code zero. The simulator record reports the same result. Both name machine-code SHA-256 `6248b8ed3380...` and output SHA-256 `72d0c1b29287...`; the complete hashes are in the linked logs [16]. This is a concrete correspondence for one program, input construction and recorded run. It is not a standing qualification of all later commits or all input arrays.

5.3 Model serving and output comparison

The October 21 live-serving snapshot names a Qwen3.6-35B-A3B-FP8 model directory at revision `95a723d0`, the server executable hash and its command line. It lists the C library, mathematics library, `libgcc_s` and loader, with zero mappings matching its CUDA-library pattern. It also records the NVIDIA 580.173.02 open kernel driver and open NVIDIA device files [17]. This supports the observed process-level user-space dependency claim, not numerical-output parity by itself.

Separate October 21 motor evidence compares token choices with the project's Rust CPU forward pass [18]. Under the documented fixed 0.1-logit near-tie rule, three 32-token teacher-forced prompts match 32/32 token IDs; a fourth matches 31/32, with its one mismatch inside the permitted band. The fourth raw receipt explicitly records nonidentical goldens. A distinct HTTP gate records one 24-token greedy completion matching all 24 reference token IDs. These are bounded checks, not bit-identical logits or universal model equivalence.

The same evidence preserves an initial adaptive-band gate as withdrawn: deriving the band from the run's own error let a deliberately broken implementation pass. The fixed-rule rerun includes a failing negative control. It also documents a later prose-prompt failure outside the band. These exclusions matter when interpreting the selected passing examples. The process-map snapshot and output gates name different executable identities; the later live binary must not silently inherit an earlier binary's output qualification.

5.4 Performance observations

The October 13 tensor probe uses HMMA.16816.F32.BF16 and reports median times over five trials, with 4,096 floating-point operations counted per HMMA instruction ($2 \times 16 \times 8 \times 16$). Throughput is CTAs $\times$ warps per CTA $\times$ 4,096 iterations $\times$ accumulator chains $\times$ 4,096 operations, divided by the median time. Its clean-tree log records 115.36 TFLOPS for 384 CTAs of 4 warps with 16 chains (median 3,574.3 μs) and 115.14 TFLOPS for 384 CTAs of 8 warps with 16 chains (median 7,162.2 μs); both recompute from those parameters. These are achieved probe throughputs, not a demonstrated physical upper bound [19].

The bandwidth log records 195.1 GB/s for a 96 MiB distinct working set and 207.7 GB/s for 384 MiB; repeated access to the same data gives different delivered rates. GB/s here is decimal bytes per nanosecond, while the source's working-set labels use binary-size arithmetic. The bandwidth run records `dirty=3`, a nonzero working-tree change count, while the tensor run four minutes earlier records `dirty=0`. The commit identifier alone cannot reconstruct the modified working tree. These observations should not be promoted to exact reproducible ceilings without the corresponding source changes and experimental conditions [19].

The October 14 launch-cost archive has two receipts. Run 1 records four failed checks and verdict FAIL; its timing samples do not make it a passing qualification. Run 2 has 11 checks, zero failures and verdict PASS. For that run, the recorded incremental wait-for-idle launch estimates range from 25.09 to 31.97 μs; the no-barrier estimates range from 2.68 to 9.68 μs. The configuration

uses $n = 16{,}384$, $k = 2{,}048$ and 20 repetitions. These estimator values depend on barrier policy and workload, and should not be generalized to every dependent launch [20].

5.5 Host tests and evidence strength

The October 22 submit-path PR reports an earlier 94/94 gate; the swap-command PR reports 54 passing host checks after rebase. These are host results reported in PR descriptions, not new chip experiments or immutable full test transcripts [21]. Mutation tests show that specified deliberately broken variants are detected; passing them is useful evidence, not proof that all faults are excluded.

The E1 closure explicitly describes its digests as integrity checks, not authenticity. I therefore use *hash-addressed receipts*, not "signed chip receipts." A Git commit signature also does not turn a measurement into hardware attestation. Receipt authenticity requires a stated signer, trust model and verification procedure separate from content hashing.

6 Coverage and open qualification

Table 3 summarizes the supported scope and known limits. Several were already partial or missing in the October 22 roadmap; they are not merely later regressions [22].

Table 3: Scope of the reviewed toolchain, not a CUDA compatibility guarantee.

Component	Evidence and boundary
Code generation	Native GB10 subset, 59 IR kinds and QMD builder; no complete CUDA-language or ISA claim.
Driver client	RM/UVM implementation exists; imported M16 raw evidence is incomplete; current native qualification is NOT_RUN.
Runtime	Memory, synchronous launch and command lists; general streams/events and process isolation remain qualification work.
Math libraries	Model-shaped matrix multiplication and attention; general any-shape coverage, scans and sorts remain incomplete.
FFT and RNG libraries	Roadmap marks cuFFT/cuRAND replacements missing.
Disassembly	Optional vendor cross-checks remain; independent disassembler work is not a completed CUDApp replacement.
Kernel and firmware	NVIDIA open kernel modules and signed firmware remain in the demonstrated path.
Portability	One demonstrated GB10/Linux/driver combination; no qualification of arbitrary GPUs, operating systems or driver releases.

The October 26 first-run guide calls CUDA++ research-stage and not a drop-in replacement for all CUDA. Its compatibility preflight is wrapper-only: `make doctor` checks target and driver, and `make hello-gpu` runs it first. Direct library callers and binaries do not have the same driver-version/PCI-ID admission guard. READY is a diagnostic result, not current-commit chip qualification [23].

A replacement kernel driver remains work in progress; this paper does not claim that it has displaced NVIDIA's modules. Recovery after a fault, multi-process safety and broader library coverage need their own retained receipts. A timeout reported to the host does not establish that the GPU has stopped or that resources can safely be reused. Current first-party CUDApp is primarily C; planned migration work must not be described as already complete. No cross-vendor backend or standards-conformance result is established here.

7 Reproducibility and licensing

7.1 Frozen source references

The following full hashes define this revision's evidence-reading snapshots; they do not replace the different execution commits recorded inside individual experimental receipts:

- Historical CUDApp: 232332422bfdc31c8918085b50f13c1e18ab702e.
- Reviewed CUDApp: fbbf3af40db8f02195603dd69c94140758975d4f.
- Reviewed Omega evidence: 3364a1dfd0ca50c1b82bf19ba34932d8bc10a7a9.
- Reviewed Core model evidence: bf15e82a44fbfa04cbffdafce433d4cd5ed66df4.

The bibliography links to files at those hashes, including raw logs where available. Readers should record the execution commit, source modifications, compiler and flags, artifact hashes, machine and driver identity, commands, exit codes and exclusions for each reproduced result. A build/test PASS, simulator PASS, physical-chip PASS and independently replicated PASS are different evidence categories. A skipped or unavailable gate is NOT RUN, never PASS.

7.2 Reproduction workflow and missing evidence

Begin with a detached checkout at the chosen snapshot and the corresponding first-run guide. Record `git rev-parse HEAD`, working-tree status, OS/architecture and compiler version. Host build/tests, the simulator and ELF dependency gate do not require a new hardware experiment. The full suite has host-architecture restrictions: an x86-64 build failure on AArch64 memory-ordering instructions must be reported, not bypassed and called a pass [23].

Physical GB10 tests require compatible driver/kernel state and an authorized quiet window that does not interrupt a serving workload. Retain raw output, negative controls and source/binary hashes. Missing M16 raw logs prevent complete re-audit of the imported qualification; they do not demonstrate that the historical experiment did not occur. Likewise, a protocol can be runnable while its original result remains incompletely archived. The dirty bandwidth run needs its exact source modifications for exact reconstruction. No new hardware run or full historical reconstruction was performed for this revision.

7.3 License snapshots

Version 1 correctly described the historical CUDApp release as MIT. That historical grant remains available at `232332422bfd`. In the October 26 snapshot, first-party material is offered under Apache-2.0 WITH LLVM-exception, subject to any explicit file-specific notices; NVIDIA headers and other third-party material retain their own terms. Later contributions do not become MIT merely by being copied onto the historical branch. The transition and bounded provenance review are documented separately [24, 8].

8 Conclusion

CUDA++ demonstrates a readable path from selected programs to GB10 machine-code execution without the proprietary CUDA user-space components used by the conventional stack. The strongest evidence is attached to particular source versions, workloads and measurements. The implementation does not yet provide complete CUDA compatibility, general library coverage, universal runtime

admission or qualification of every later revision. Publishing those limits alongside the working encoder, driver client and experimental records makes the engineering result more useful and more reproducible.

Revision statement

Version 2 corrects the complete-replacement and mandatory-runtime-gate claims; narrows model-output equality; adds the FP32 numerical domains and error-contract distinction; replaces signature/ceiling language with the actual receipt and measurement scope; identifies missing M16 evidence and dirty-tree reconstruction limits; and separates historical MIT licensing from the later license. It changes the subtitle to avoid implying a certified clean-room process, and labels each quantitative claim by evidence class (Section 4). Version 1 remains the historical publication. This revision introduces no new chip performance or correctness measurements.

Acknowledgements

I acknowledge Mesa's NAK/NVK developers and NVIDIA's open kernel-module team for the public work on which this implementation relies. Their contributions, notices and the distinction between included material and technical references are documented in the repository.

References

[1] M. D. Stapleton. *Reverse Engineering the CUDA Software Stack: A Clean-Room GPU Machine-Code Toolchain for the NVIDIA GB10*. Version 1, October 22, 2026. https://doi.org/10.5281/zenodo.23578453 (all versions: https://doi.org/10.5281/zenodo.23578452).

[2] M. Fasi, N. J. Higham, M. Mikaitis and S. Pranesh. Numerical behavior of NVIDIA tensor cores. *PeerJ Computer Science* 7:e330, 2021. https://doi.org/10.7717/peerj-cs.330.

[3] CUDA++. Third-party material and notices; historical notices.

[4] NVIDIA. *Open GPU kernel modules*, release 580.173.02. Versioned source; retained license text.

[5] CUDA++. Documented implementation method. Historical method statements are subject to the evidence qualifications in this revision.

[6] NVIDIA. *CUDA Binary Utilities 13.0*, instruction-set references and disassembly tools. https://docs.nvidia.com/cuda/archive/13.0.0/cuda-binary-utilities/index.html.

[7] CUDA++. BF16 accumulation model and prior-work attribution; 2,048-case probe and negative control.

[8] CUDA++. Historical extraction audit; later bounded provenance audit.

[9] CUDA++. RM/UVM client implementation; driver walkthrough; session and uncertain-state handling.

[10] CUDA++. Imported M16 summary; current README evidence limitation; qualification executable.

[11] CUDA++. IR and register definitions; encoder and fixtures; fixed integer vector-add table; QMD builder.

[12] CUDA++. Historical submit diagnostics; current submit diagnostics; ELF dependency gate.

[13] CUDA++. E1 closure summary and constituent digests. Omega: closure scope and exclusions.

[14] Omega. Four-operation exhaustive raw log; five-operation exhaustive raw log.

[15] Omega. Frozen CPU transcendental contract, domains, ULP metric and oracle limitations. Its initial GPU-status text is superseded for the documented operations by the later closure receipts, not by this revision.

[16] CUDA++. October 22 chip log; simulator logs; historical PR 2.

[17] Omega. October 21 live-server process-map, binary and driver record.

[18] Aien Sovereign Core. October 21 motor gates, withdrawals and exclusions; fourth-prompt raw result; 24-token HTTP result; CPU forward-reference description.

[19] CUDA++. Clean-tree tensor log; dirty-tree bandwidth log; probe source at the execution commit `5faa0f70`; later copy in CUDApp.

[20] CUDA++. Launch-cost run 1 (FAIL); run 2 (PASS).

[21] CUDA++. PR 4, reported earlier 94/94 submit gate; PR 3, reported 54/0 swap host checks. PR descriptions are mutable summaries.

[22] CUDA++. Historical roadmap; October 26 roadmap.

[23] CUDA++. First-run and evidence guide; runtime API and admission boundary; platform portability contract.

[24] CUDA++. License transition and historical MIT access; historical MIT license; reviewed current license.