Pre-Execution Preventability of Bugs in GPU-Accelerated Quantum Software Stacks: an Empirical Study

Tihomir Thomas Bicanic[1][0009−0003−1562−3497]

Trier University, Universitätsring 15, 54296 Trier
bicanic@uni-trier.de

Abstract. GPU-accelerated quantum software stacks combine heterogeneous hardware, vendor libraries, build toolchains, and framework integrations, creating failure modes that are difficult to diagnose. We ask which bugs in such stacks could, in principle, be prevented before execution. To study this question, we analyze N=196 GPU-relevant GitHub issues from two contrasting GPU-quantum stacks: NVIDIA CUDA-Q (n=159) and Aer GPU (n=37). Each issue is coded along three dimensions: stack layer (where the main cause occurs), bug type (the dominant manifestation), and a preventability class, CTClass (A=compile-time avoidable, B=pre-execution preventable, C=runtime-only). Overall, roughly one third of the issues are addressable before execution (A+B), whereas the majority are runtime-bound (C). Preventability is strongly layer-dependent: build-, deployment-, and environment-related issues are often pre-execution preventable, while runtime-adjacent issues are largely runtime-only. The two projects also show markedly different profiles: CUDA-Q is dominated by runtime-bound issues, whereas Aer GPU contains a substantially larger share of pre-execution-preventable cases. These results suggest that different parts of GPU-quantum stacks call for different prevention mechanisms, alongside continued runtime-oriented testing and observability.

Keywords: Quantum software · GPU acceleration · Empirical study · Bug classification · Preventability · Heterogeneous systems

1 Introduction

Modern computing systems increasingly rely on heterogeneous architectures. While the CPU previously dominated as the primary computing platform, today's systems use a variety of accelerators, especially GPUs. The current AI boom has made the advantages of such accelerators particularly visible and has further accelerated the adoption of heterogeneous CPU–GPU systems. However, this heterogeneity requires new programming paradigms, since memory management, data movement, and execution models differ fundamentally from classical CPU-centric approaches [22]. This means that software stacks grow in depth, and additional error-prone layers emerge at new levels. Hardware drivers, vendor libraries, build and packaging toolchains, language bindings, and framework

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

integrations add further complexity to software projects [3, 2]. This becomes particularly problematic when safety and abstraction guarantees do not consistently hold across all stack layers, allowing errors to propagate along the stack.

We observe this, for example, in systems that provide strong compile-time safety guarantees (e.g., Rust) but undermine them at CUDA interfaces (FFI), where unsafe code is required because CUDA and OpenCL do not provide the same guarantees [10].

In addition to such broken guarantees, the problems are amplified by the growing dependence on build and packaging ecosystems. Missing or incompatible dependencies, faulty build dependencies, as well as incompletely ported features can lead to build failures or integration problems. Particularly critical is that such errors sometimes occur only in specific hardware/software combinations, which complicates debugging and fixing [14].

To investigate the development described above, we focus in this paper on GPU-quantum stacks as a concrete specialization of heterogeneous systems and examine the following question: what proportion of relevant error classes can, in principle, be addressed before execution? The motivation is that early error detection becomes more important as stack depth increases, because debugging effort across multiple layers can become exponentially higher than for CPU-only programs [20]. Instead of reacting to errors at runtime, they should be intercepted where they arise.

While empirical studies have already identified bugs in quantum software, for example, Paltenghi & Pradel [15], who investigate high-level frameworks, or El Aoun [8], who classifies configuration, data type, and program-anomaly bugs, there has been little research on GPU-accelerated stacks and the potential for pre-execution error prevention.

To answer this question, we analyze bug issues from two GPU-quantum stacks: CUDA-Q and the GPU-related subset of Qiskit Aer. We describe them along three dimensions: the stack layer in which the primary cause occurs, the dominant bug type, and a preventability class. We distinguish three preventability categories: (A) bugs that can be eliminated at compile time through static analysis, (B) bugs that can be caught before execution via checks, constraints, or contracts, and (C) bugs that can only be addressed at runtime.

TypeSec [5] and GPUVerify [4] show that pre-execution prevention of selected bug classes is feasible in practice. The former prevents protocol errors at compile time, while the latter statically proves the absence of data races and barrier divergence. Based on this, we derive three research questions: (RQ1) Which bug patterns occur on which stack layers? (RQ2) How is CTClass A/B/C distributed overall and when comparing CUDA-Q and Aer GPU? (RQ3) What implications follow for static or preflight mechanisms compared to runtime checks? To answer these questions, we construct a manually coded dataset of GPU-relevant bug reports and analyze it along three complementary dimensions; Section 3 details the study design.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

2 Related Work

Empirical studies on quantum software bugs provide an important starting point for this work. For instance, the previously mentioned Paltenghi & Pradel [15] report 223 bugs in 18 quantum-computing platforms. They show that bugs are concentrated in compilation and optimization components, but they focus on CPU-based simulators. Nevertheless, their study provides initial insights into bug-class distributions, albeit without a GPU context.

El Aoun et al. [8] examined 125 quantum software projects and developed a broad bug classification. Their work provides useful background on bug categories, bug prevalence, and bug-fixing effort in quantum software. However, we do not use that classification directly, because our study focuses on GPU-accelerated stacks and on pre-execution preventability across stack layers.

Quetschlich and Di Matteo [18] provide an experience-based, qualitative characterization of quantum bugs based on a case sample and discuss typical debugging practices. Their work is useful for understanding how quantum bugs arise in practice, especially through interactions between multiple aspects. However, we do not adopt that characterization directly, because our study requires a coding scheme tailored to GPU-accelerated stacks and to pre-execution preventability across stack layers.

Tao et al. [19] present Giallar, an automated verification tool for quantum compiler passes. While this work does not provide a bug classification for GPU-accelerated quantum stacks, it is relevant to our study because it illustrates that selected correctness properties in quantum software can be checked before execution.

Also at the level of static analysis, Paltenghi & Pradel [17] present LintQ, a framework for analyzing Qiskit programs before execution. In their evaluation on 7568 real programs, the approach reports 91.0% precision, and 92.1% of the problems it finds are missed by the compared prior techniques.

Wang et al. [21] demonstrate with QDiff, a differential-testing framework for quantum software stacks (Qiskit, Cirq, PyQuil), that not all bugs can be detected before execution. The framework generates semantically equivalent variants of quantum programs, runs them on simulators and hardware, and compares deviations in the results.

We are also aware of work by Betts et al. [4] and Mai et al. [12] demonstrating static verification or analysis of GPU-specific bugs. While Betts et al. use a verification tool to analyze GPU kernels, Mai et al. rely on range checks and def-use analysis at compile time to ensure memory safety. Both works show that GPU-specific bugs can be addressed at compile time; however, they analyze isolated GPU kernels rather than the complex interactions across multiple stack layers that our RQ1 investigates.

The work by Betts, Mai, and Wang shows that bug prevention can be achieved, but always in isolated contexts (individual kernels, individual frameworks, individual abstraction layers). What has been missing so far is a holistic perspective. What do bug classes look like in a complete, GPU-accelerated software stack? Which bugs can be prevented pre-execution, and which cannot? Our answer is

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

not a new verification tool, but a rigorous empirical analysis. By systematically categorizing bug issues from CUDA-Q and Aer GPU by stack layer and CTClass (A/B/C), we provide quantitative evidence on how bug classes are distributed across GPU-accelerated quantum stacks and which of them are addressable before execution.

3 Study Design and Methods

To answer RQ1–RQ3, we construct and manually code a dataset of GPU-relevant bug reports from two contrasting GPU-quantum software stacks. Our study design combines three complementary coding dimensions. StackLayer captures where in the stack the primary cause resides, BugType captures the dominant manifestation of the issue, and CTClass captures whether the bug is, in principle, preventable before execution. The remainder of this section describes dataset construction, coding rules, adjudication, and the statistical analysis.

For this work, we examine bug reports from two established GPU-quantum software stacks. The stacks were selected as contrast cases because they differ in architecture, integration breadth, and role within the stack, making divergent bug profiles and CTClass distributions plausible. `NVIDIA/cuda-quantum` (CUDA-Q)[1] is an integrated framework that provides multiple targets. `Qiskit/qiskit-aer` (Aer GPU)[2] is designed as a quantum simulator that provides a GPU backend. CUDA-Q and Aer GPU therefore differ not only in capabilities, but especially in their position within the software stack (end-to-end framework vs. backend component). The selection is based on their public availability on GitHub and on the availability of enough GPU-relevant bug reports after screening for a comparative analysis.

3.1 Dataset Construction

We draw on GitHub Issues (bug reports) as our data source and extract the relevant entries. For both projects, GitHub Issues are the primary public channel for reporting and tracking bugs. For this purpose, we rely on Python scripts that run GitHub searches with documented search qualifiers (including `is:issue`, `label:bug`, `created:`) and store the results as a dataset snapshot. For the analysis, we consider only the initial bug description, not the subsequent discussion, in order to base the coding on the information available at reporting time and to avoid later diagnosis or fix information influencing the labels. We then conducted a manual screening to exclude non-bugs (e.g., feature requests) and off-topic threads. For Qiskit Aer, we include only GPU-related issues in the final sample; these issues are marked as GPU-relevant in the dataset, while non-GPU-related issues are excluded during screening. We consider issues GPU-relevant if their symptom explicitly refers to CUDA/cuStateVec/the GPU backend, missing GPU

[1] https://github.com/NVIDIA/cuda-quantum
[2] https://github.com/Qiskit/qiskit-aer

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

libraries, or GPU resource problems. The final dataset comprises $N = 196$ GPU-relevant bug reports: 159 issues from `NVIDIA/cuda-quantum` (creation period 2023-03-20 to 2025-09-10) and 37 GPU-relevant issues from `Qiskit/qiskit-aer` (2023-02-07 to 2025-10-29). We recorded open/closed status at the time of export. The resulting dataset is unbalanced (CUDA-Q: 159; Aer GPU: 37), reflecting repository-specific issue availability after GPU-relevance filtering; we therefore interpret cross-project contrasts as analytical comparisons rather than population-level estimates.

Selection Procedure. Selection consisted of (1) query definition, (2) screening, and (3) deduplication. Additional random sampling was not necessary because the resulting set could be covered in full.

3.2 Dimensions and Definitions

Each issue is described along three complementary dimensions, because no single dimension is sufficient for our research questions:

1. **StackLayer:** the layer in which the primary root cause of the bug resides (e. g., backend library, framework integration, high-level API/framework logic, build/deploy/environment, runtime/framework runtime).
2. **BugType:** the dominant cause of the bug (e. g., API/usage/logic, backend/framework integration, build/install/packaging, configuration/ environment, performance/numerics).
3. **CTClass**[3]**:** classification of whether the bug can, in principle, be prevented upfront:
   - **A** (*compile time*): The bug can be eliminated at compile time via static analysis or type rules.
   - **B** (*pre-execution*): The bug can be caught before execution (preflight, constraints, contracts), but is not *trivially captured by static analysis*.
   - **C** (*runtime-only*): The bug is substantially runtime- or environment-dependent and therefore cannot be reliably prevented before execution.

For CTClass B, we additionally record a subtype B1/B2 to distinguish different mechanisms. B1 denotes metadata- or compatibility-based checks (e. g., support matrices, version/architecture constraints), whereas B2 denotes contract-/guard-/typestate-adjacent checks that express stronger, non-trivial preconditions. We introduced this split because CTClass B proved too broad during initial coding. We want to interpret the resulting prevention mechanisms separately. This prevents a substantial share of CTClass B cases from collapsing into an unspecific category and yields a finer, interpretable resolution for the analysis.

[3] CTClass = Compile-Time Possibility Class

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

Decision Rules for CTClass To reduce subjectivity, we apply the following decision rules: (1) If there is a clear API/sequence violation or a deterministic violation of a consistency condition that can be checked without runtime information → A; (2) if the bug can be detected or blocked by deterministic pre-execution checks (e. g., dependency/configuration validation, capability checks, contracts) → B (B1/B2); (3) if the occurrence depends substantially on runtime behavior, hardware state, timing, numerical instability, or resource availability → C. Borderline cases are documented in the full codebook, which also includes representative examples for the coding categories; to avoid optimistic assignments, we apply a conservative tie-break rule (C > B2 > B1). Table 1 shows exemplary borderline cases.

3.3 Coding Process, Rationales, and Borderline Cases

In addition to the categories, we store three short textual rationales per issue to improve label transparency: Bug reason, Stack reason and Class reason (CTClass justification incl. B1/B2). These decisions are derived manually from the issue description (including embedded logs/stack traces) and serve as an audit trail for later replication and error analysis.

Example (Issue)	**Alt.**	**Why borderline / final rule**
CUDA-Q #920: MPI not included in Python wheels	B1↔C	API appears to be available, but the implementation is missing from the artifact; conservatively treated as an artifact/runtime issue ⇒ C.
CUDA-Q #2628: cudaErrorIllegalAddress	B2↔C	Guards are conceivable, but illegal-address errors are often timing-/context-dependent in asynchronous multi-GPU settings ⇒ C.
CUDA-Q #1464: CNOT degradation in lowering	B↔C	Pre-execution prevention would require a semantic compiler-equivalence check; conservatively ⇒ C.
CUDA-Q #2279: mid-circuit measurement semantics	A↔C	Syntactically valid, but backend/semantic wrong results; conservatively ⇒ C.
CUDA-Q #875: ASTBridge spec adherence	B2↔C	User is spec-compliant, but the internal bridge/translation violates the spec; conservatively ⇒ C.

Table 1. Exemplary borderline cases and the conservative adjudication rule (C > B2 > B1)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

3.4 Inter-Coder Agreement and Adjudication

CTClass labels were assigned independently by the author and a second coder independent of the author, and then finalized by consensus. We report an observed agreement of $P_o = 84.6\%$ and Cohen's $\kappa = 0.70$ [13, 7]. The final data set therefore contains consensus labels for CTClass.

3.5 Analysis Plan and Statistics

To answer the research questions, we construct contingency tables for project and CTClass, StackLayer and CTClass, and BugType and CTClass. As an effect size for associations in contingency tables, we use Cramér's V (range $[0, 1]$) [1]. Because expected cell counts are small in several contingency tables and the asymptotic χ^2 approximation can then be unreliable [6], we additionally assess significance via a permutation test (5000 permutations, fixed seed) by randomly permuting CTClass labels under the null hypothesis and comparing the resulting χ^2 statistic to the observed value [9]. We summarize effect sizes and permutation-test p-values in Table 2.

3.6 Reproducibility and Artifacts

We document the GitHub queries, the codebook (including decision rules), the labeled data set (issue URLs/IDs + categories + rationales), and the analysis scripts used to generate contingency tables, effect sizes, and permutation tests.

All artifacts are publicly available under the Apache License 2.0 at:

- Repository: https://github.com/TheBuccaneer/gpu-quantum-bug-study
- Persistent archive (DOI): https://doi.org/10.5281/zenodo.18326741

Cross-tabulation	χ^2	df	p_{perm}	V	min. expected
Project × CTClass	43.21	2	0.0002	0.4695	2.27
StackLayer × CTClass	77.01	8	0.0002	0.4432	1.10
BugType × CTClass	60.41	8	0.0002	0.3926	0.61

Table 2. Effect sizes and permutation tests (5000 permutations) for the main cross-tabulations

4 Results

In this section, we report the central findings of our analysis of $N = 196$ GPU-relevant bug reports (CUDA-Q: $n = 159$, Aer GPU: $n = 37$). We structure the results along the research questions: (RQ1) their patterns across StackLayer and

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

BugType, (RQ2) the distribution of classes (CTClass) overall and by project, and (RQ3) the breakdown of CTClass B into B1/B2. This highlights which parts of GPU-quantum stacks contain bugs that can, in principle, be detected before execution and where bugs predominantly arise only at runtime.

4.1 RQ1: CTClass Across Stack Layers

Figure 1 breaks down CTClass by StackLayer. The distribution varies substantially across layers and shows that pre-execution preventability is not evenly distributed throughout the stack. CTClass B is particularly pronounced in Build/Deploy/Environment ($N = 33$): CTClass B (78.8%, 26/33), CTClass C (18.2%, 6/33), and CTClass A (3%, 1/33). These findings support the intuition that bugs related to installation, deployment, dependencies, and hardware/software setup can often be caught by deterministic checks (e. g., version/API compatibility, capability checks, dependency constraints).

In contrast, Runtime/Framework-Runtime ($N = 18$) is entirely CTClass C (100%, 18/18). Bugs in this layer arise only during execution and cannot be reliably ruled out upfront. Framework-Integration ($N = 61$) is also strongly dominated by CTClass C (85.2%, 52/61), with CTClass B (13.1%, 8/61) and CTClass A (1.6%, 1/61). In Backend-Library ($N = 34$), CTClass B is higher (29.4%, 10/34), but still below CTClass C (67.6%, 23/34). High-Level-API/Framework-Logic ($N = 50$) is more heterogeneous by comparison: CTClass A (18%, 9/50), CTClass B (18%, 9/50), and CTClass C (64%, 32/50). Overall, the layer results show that bugs detectable upfront are especially prevalent in build-adjacent areas, whereas runtime-adjacent layers almost exclusively contain CTClass C.

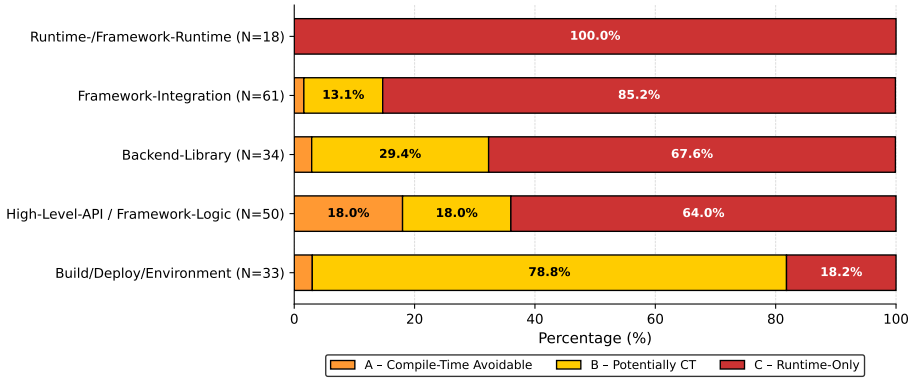

Fig. 1. Overall CTClass distribution by Stack Layer

4.2 RQ1: CTClass Across Bug Types

Figure 2 breaks down CTClass by BugType and shows a similar picture. Some bug types are detectable upfront, whereas others are almost exclusively runtime-

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

only. Config/Environment bugs ($N = 10$) are clearly dominated by CTClass B (80%, 8/10), while CTClass C accounts for only 20% (2/10).

Build/Install/Packaging bugs ($N = 24$) are also predominantly CTClass B (70.8%, 17/24) and 25% CTClass C (6/24). Together, these two categories form a group of bug types that can be reduced particularly well via preflight and constraint mechanisms. On the other hand, some bug types are almost entirely assigned to CTClass C. Performance/Numerics bugs ($N = 34$) fall into CTClass C in 91.2% of cases (31/34), reflecting the strong runtime dependence of such problems (e. g., hardware/toolchain-dependent numerical instabilities, performance issues). Backend/Framework integration bugs ($N = 53$) are likewise strongly C-dominated (83%, 44/53), while only 15.1% fall into CTClass B (8/53). The largest category, API/Usage/Logic bugs (high-level) ($N = 75$), is less clear-cut: CTClass C constitutes the majority (64%, 48/75), but 22.7% fall into B (17/75) and 13.3% into A (10/75). This category, in particular, shows that even in GPU-quantum stacks, a share of bugs could be addressed via static rules or pre-execution checks.

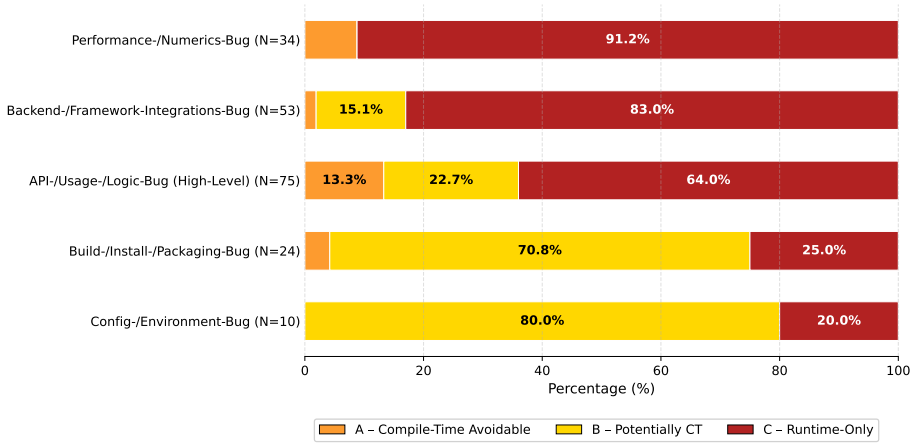

Fig. 2. Overall CTClass distribution by Bug Type

4.3 RQ2: Distribution of Preventability Classes

Figure 3 shows the distribution of CTClass overall (A) and split by project (B). Across all issues, CTClass C dominates with 131/196 (66.8%). CTClass B comprises 53/196 (27%), while CTClass A is rare with 12/196 (6.1%). Thus, in our data set, roughly one third of cases (A+B) are, in principle, preventable or detectable before execution. The majority, however, concerns bugs that are runtime-dependent.

The projects exhibit markedly different profiles (Fig. 3B). In CUDA-Q, most issues fall into CTClass C (121/159, 76.1%), while CTClass B accounts for

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

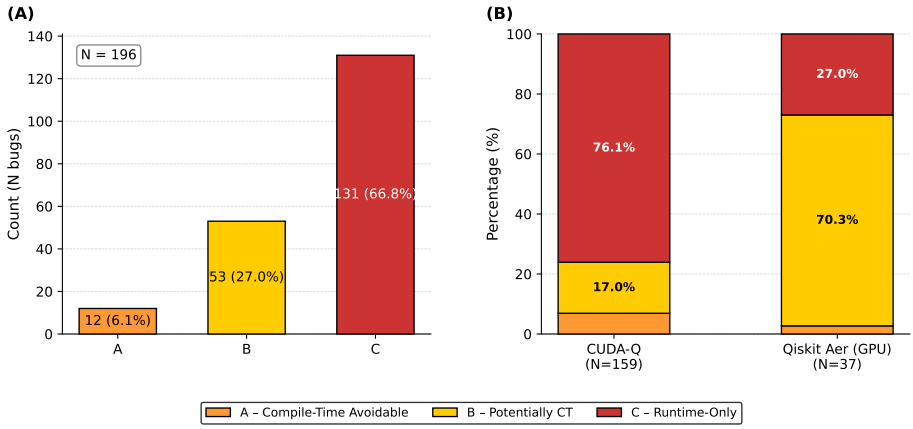

Fig. 3. CTClass distribution overall (a) and by project (b)

27/159 (17%) (CTClass A: 11/159, 6.9%). In Aer GPU CTClass B is dominant: 26/37 (70.3%) fall into B, whereas CTClass C accounts for only 10/37 (27%) (CTClass A: 1/37, 2.7%). This shift is consistent with the contrast between the two systems as an end-to-end framework (CUDA-Q) versus a backend component/adapter (Aer GPU): while CUDA-Q includes many runtime-adjacent integration and execution bugs, Aer GPU issues are more concentrated in configuration and compatibility aspects that can be checked upfront.

4.4 RQ3: Breakdown into B1/B2

Figure 4 breaks down all CTClass B issues ($n = 53$) by B1/B2 subtypes. Overall, B1 (config/metadata constraints) is slightly more common at 56.6% (30/53) than B2 (contracts/typestate) at 43.4% (23/53). However, the composition differs in projects. In CUDA-Q, 66.7% of B cases fall into B2 (18/27) and 33.3% into B1 (9/27). In Aer GPU, the pattern is reversed: B1 strongly dominates (80.8%, 21/26), whereas B2 accounts for only 19.2% (5/26). Thus, the stacks differ not only in the frequency of CTClass B, but also in the nature of potentially catchable bugs. Aer GPU mostly exhibits deterministic compatibility/configuration checks (B1), while CUDA-Q more often reflects preconditions and guard-/contract-adjacent mechanisms within CTClass B (B2).

5 Discussion

Our results show that pre-execution detectability is present in GPU-quantum stacks, but depends on the architecture, degree of integration, and the affected stack layer. In particular, CUDA-Q and Aer GPU differ substantially in the distribution of classes A/B/C and in the B subtypes. These differences show that CUDA-Q and Aer GPU have developed different bug profiles. They arise

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

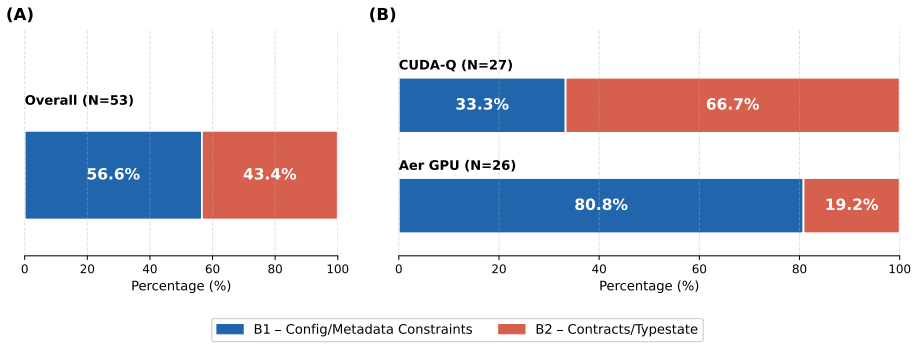

Fig. 4. Decomposition of CTClass B into B1 and B2 overall and by project

from their respective roles in the stack (integrated framework vs. specialized adapter) and the typical challenges of GPU deployment.

5.1 Differences Between CUDA-Q and Aer GPU

The strong contrast between CTClass C in CUDA-Q and predominantly CTClass B in Aer GPU can be explained by the systems roles. Aer GPU acts as a GPU-capable backend component within a larger ecosystem. Accordingly, GPU-relevant issues concentrate on (deterministic) preconditions (drivers/libraries, version and capability mismatches, build/packaging), i. e., topics that can be summarized as pre-execution checks. This pattern is consistent with observations in the literature on heterogeneous platforms, where deployment and integration problems are described as recurring challenges [2, 3, 11]. By contrast, CUDA-Q is an integrated framework that bundles different backends into an end-to-end workflow. As a result, errors more frequently arise that become visible only during execution or through the interaction of multiple components (CTClass C), for example in framework integration, runtime, and performance/numerics.

5.2 Layer and Bug Type Patterns for Prevention

Layer analysis (Fig. 1) can be interpreted as follows. Build/Deploy/Environment is clearly dominated by CTClass B, whereas runtime-adjacent layers contain almost exclusively CTClass C. This observation is also confirmed in other heterogeneous contexts. Zhang et al. [23] report that about 48% of failures in deep-learning jobs due to platform interactions rather than code logic. These results suggest that substantial prevention opportunities lie in preflight mechanisms for installation, configuration, and capability checks (B1). At the same time, the strong dominance of CTClass C in Framework-Integration and Runtime indicates that a subset of problems must be made detectable not primarily through additional static checks, but through robust runtime testing, monitoring, and differential checks.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

A similar picture emerges from the bug types (Fig. 2): Build/Install and Config/Environment are predominantly CTClass B, whereas Performance/Numerics is almost exclusively CTClass C. This separation is particularly plausible for GPU-quantum stacks, because numerical deviations, performance problems, or toolchain-/hardware-dependent effects only appear under real conditions. This also confirms what Paltenghi and Pradel discuss for quantum platforms more generally: many bugs do not manifest as crashes, but as unexpected outputs and are therefore difficult to detect purely statically [15]. The experience-based classification by Quetschlich and Di Matteo likewise indicates that quantum bugs often arise from complex interactions and that debugging strategies cannot be cleanly assigned to a single bug class [18]. Our results support this perspective for GPU stacks: where CTClass C dominates, it is more effective to invest in oracle-/differential testing and runtime diagnostics than in stricter preflight rules.

5.3 Implications of B1/B2 Prevalence

The B1/B2 breakdown (Fig. 4) makes the difference between deterministic checks (B1) and non-trivial protocols (B2) visible. That Aer GPU is strongly B1-heavy within Class B argues for measures such as explicit support matrices (CUDA/cuStateVec versions), automatic validation of the runtime environment, and CI-based preflight suites that block faulty configurations before execution. By comparison CUDA-Q shows more frequent B2 cases within CTClass B. This suggests that in an integrated framework, not only the *whether* (is the environment compatible?) but also the *how* (was an allowed API/resource protocol followed?) becomes central.

There are precedents for this in adjacent areas. TypeSec demonstrates that protocol errors in GPU APIs can be encoded via typestates, with state transitions enforced at compile time [5]. GPUVerify shows that important GPU-kernel properties, such as the absence of races and barrier divergence, can be checked statically [4]. Honeycomb likewise demonstrates the use of static validation to control GPU execution [12]. Our B2 cases therefore suggest that GPU-quantum stacks may benefit from more protocol-aware APIs, such as typestate-based designs, guards, or contracts, rather than focusing only on the classic configuration layer.

5.4 CTClass C Testing and Diagnostic Strategies

The strong dominance of CTClass C in runtime-adjacent layers and in Performance/Numerics motivates testing approaches that do not rely on a single correct reference result. QDiff provides one relevant example through differential testing of quantum software stacks [21]. Differences across backends, drivers, and compilers can lead to faulty results that standard tests do not detect. Accordingly, multi-backend comparisons, metamorphic testing [16], and regression checks over result distributions appear particularly relevant for surfacing C-heavy bug types.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

At the same time, Class C should not be misunderstood as unavoidable: even if problems cannot be reliably prevented before execution, they can be debugged much faster through better observability (logging, telemetry, deterministic reproduction scripts, automatic capture of environment metadata). Here, our results connect to challenges discussed in the literature, such as interpreting probabilistic outputs and the knowledge gap between quantum and classical system stacks [8].

5.5 Threats to Validity

The dataset is unbalanced (Aer GPU $n = 37$ vs. CUDA-Q $n = 159$), and the GPU-relevance filtering for Aer GPU may bias the observed cross-project differences, in particular the comparatively high share of CTClass B in Aer GPU. We therefore interpret cross-project contrasts as analytical comparisons rather than population-level estimates. At the same time, this selection is substantively justified, because Qiskit Aer, as a simulator, also contains many non-GPU-specific issues, and comparable GPU-quantum stacks often do not provide an open and sufficiently detailed issue history. Extending the study to additional GPU-quantum stacks (e. g., further simulators or backends) would improve generalizability and help assess whether the observed patterns, including the layer-to-B1 tendency, remain stable. A further limitation is that CTClass C groups together heterogeneous runtime-dependent phenomena; a finer-grained subdivision (e. g., numerical instability vs. resource- or timing-related issues) could support more targeted testing strategies in future work.

6 Conclusion

We studied the pre-execution preventability of bugs in GPU-quantum software stacks by analyzing 196 GPU-relevant GitHub issues from CUDA-Q (n=159) and Aer GPU (n=37). Our classification along StackLayer, BugType, and CTClass shows that roughly one third of the observed issues are, in principle, addressable before execution (A+B), while runtime-dependent failures dominate overall. Preventability is strongly layer-dependent: Build/Deploy/Environment is largely pre-execution preventable, whereas Runtime/Framework-Runtime is entirely runtime-bound. The two projects also exhibit sharply different profiles, consistent with their roles as an end-to-end framework and a backend adapter. To make CTClass B more actionable, we divide it into B1 (compatibility/metadata preflight checks) and B2 (contract-/protocol-like guards). The resulting B1/B2 split suggests that different prevention mechanisms may be appropriate for different parts of the stack, while CTClass C underlines the continued importance of runtime-oriented testing and observability. Future work includes extending the dataset to additional GPU-quantum stacks, triangulating issue-based labels with fixes and reproduction artifacts, and exploring automated B1/B2 checkers to assess how much of the observed bug space can be addressed in practice.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

Disclosure of Interests

The author declares that they have no competing interests to declare that are relevant to the content of this article.

References

1. Agresti, A.: Categorical Data Analysis. Wiley (2002). https://doi.org/10.1002/0471249688
2. Andrade, H., Lwakatare, L.E., Crnkovic, I., Bosch, J.: Software challenges in heterogeneous computing: A multiple case study in industry. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). pp. 148–155 (05 2019). https://doi.org/10.1109/SEAA.2019.00031
3. de Andrade, H.S., Schroeder, J., Crnkovic, I.: Software Deployment on Heterogeneous Platforms: A Systematic Mapping Study . IEEE Transactions on Software Engineering **47**(08), 1683–1707 (2021). https://doi.org/10.1109/TSE.2019.2932665, https://doi.ieeecomputersociety.org/10.1109/TSE.2019.2932665
4. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: Gpuverify: a verifier for gpu kernels. SIGPLAN Not. **47**(10), 113—-132 (Oct 2012). https://doi.org/10.1145/2398857.2384625, https://doi.org/10.1145/2398857.2384625
5. Bicanic, T.T.: Compile-time resource safety for gpu apis: A low-overhead typestate framework. In: Tagungsband des FG-BS Herbsttreffens 2025. p. 10–18420. Gesellschaft f"ur Informatik eV (2025)
6. Cochran, W.G.: Some methods for strengthening the common chi-square tests. Biometrics **10**(4), 417–451 (1954)
7. Cohen, J.: A coefficient of agreement for nominal scales. Educational and Psychological Measurement **20**(1), 37–46 (1960). https://doi.org/10.1177/001316446002000104
8. El Aoun, M.R.: Empirical Studies of Quantum Programming Issues. Master's thesis, Polytechnique Montréal (2022), https://publications.polymtl.ca/10482/
9. Good, P.I.: Permutation, Parametric, and Bootstrap Tests of Hypotheses. Springer, 3 edn. (2005). https://doi.org/10.1007/b138696
10. Li, Z., Wang, J., Sun, M., Lui, J.C.S.: Detecting cross-language memory management issues in rust. In: Atluri, V., Di Pietro, R., Jensen, C.D., Meng, W. (eds.) Computer Security – ESORICS 2022. pp. 680–700. Springer Nature Switzerland, Cham (2022). https://doi.org/10.1007/978-3-031-17143-7_33
11. Macho, C., McIntosh, S., Pinzger, M.: Automatically repairing dependency-related build breakage. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 106–117 (2018). https://doi.org/10.1109/SANER.2018.8330201
12. Mai, H., Zhao, J., Zheng, H., Zhao, Y., Liu, Z., Gao, M., Wang, C., Cui, H., Feng, X., Kozyrakis, C.: Honeycomb: Secure and efficient GPU executions via static validation. In: 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). pp. 155–172. USENIX Association, Boston, MA (Jul 2023), https://www.usenix.org/conference/osdi23/presentation/mai
13. McHugh, M.: Interrater reliability: the kappa statistic. Biochemia Medica pp. 276–282 (01 2012). https://doi.org/10.11613/BM.2012.031

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5

14. Mens, T., Decan, A.: An overview and catalogue of dependency challenges in open source software package registries. In: Proceedings of the 23rd Belgium-Netherlands Software Evolution Workshop (BENEVOL). CEUR Workshop Proceedings, vol. 3941, pp. 160–176 (2024), https://arxiv.org/abs/2409.18884, accepted for BENEVOL24
15. Paltenghi, M., Pradel, M.: Bugs in quantum computing platforms: an empirical study. Proc. ACM Program. Lang. **6**(OOPSLA1) (Apr 2022). https://doi.org/10.1145/3527330, https://doi.org/10.1145/3527330
16. Paltenghi, M., Pradel, M.: Morphq: Metamorphic testing of the qiskit quantum computing platform. In: Proceedings of the 45th IEEE/ACM International Conference on Software Engineering. pp. 2619–2631. ICSE 2023 (2023). https://doi.org/10.1109/ICSE48619.2023.00217
17. Paltenghi, M., Pradel, M.: Analyzing quantum programs with lintq: A static analysis framework for qiskit. Proc. ACM Softw. Eng. **1**(FSE) (Jul 2024). https://doi.org/10.1145/3660802, https://doi.org/10.1145/3660802
18. Quetschlich, N., Di Matteo, O.: An experience-based classification of quantum bugs in quantum software. Computing **107**, 193 (2025). https://doi.org/10.1007/s00607-025-01547-3, https://doi.org/10.1007/s00607-025-01547-3
19. Tao, R., Shi, Y., Yao, J., Li, X., Javadi-Abhari, A., Cross, A.W., Chong, F.T., Gu, R.: Giallar: push-button verification for the qiskit quantum compiler. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 641–656. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523431, https://doi.org/10.1145/3519939.3523431
20. Wang, J.: Hardware-Assisted Software Testing and Debugging for Heterogeneous Computing. Ph.D. thesis, University of California (2024), https://escholarship.org/uc/item/9dh545t9
21. Wang, J., Zhang, Q., Xu, G.H., Kim, M.: Qdiff: differential testing of quantum software stacks. In: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering. p. 692–704. ASE '21, IEEE Press (2022). https://doi.org/10.1109/ASE51524.2021.9678792, https://doi.org/10.1109/ASE51524.2021.9678792
22. Yang, W., Zhang, C., Pan, M.: Understanding the topics and challenges of gpu programming by classifying and analyzing stack overflow posts. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1444–1456. ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3611643.3616365, https://doi.org/10.1145/3611643.3616365
23. Zhang, R., Xiao, W., Zhang, H., Liu, Y., Lin, H., Yang, M.: An empirical study on program failures of deep learning jobs. In: Proceedings of the 42nd International Conference on Software Engineering. pp. 1159–1170. ICSE 2020 (2020). https://doi.org/10.1145/3377811.3380362

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29918-5_5