NVIDIA Ecosystem Award Submission

What the Duck

State Street x Classiq — Quantum Value at Risk Estimation

Team members: Jasper Sands, Leila Erhili, Azain Khalid, Shay Manor, Mallin Kumar

GitHub: github.com/ShayManor/WhatTheDuck

Problem statement

Value at Risk (VaR) is a core risk metric defined by a tail probability condition: find the threshold v such that $P(\text{loss} > v) = \alpha$. The challenge asked us to implement quantum amplitude estimation, specifically Iterative Quantum Amplitude Estimation (IQAE), to estimate these tail probabilities with improved asymptotic scaling versus classical Monte Carlo. The key goal was not only to run IQAE once, but to demonstrate and benchmark the advantage across multiple probability distributions that model different tail behaviors, under strict hackathon time constraints.

What was the hard part?

Two things made this hard.

1) Fair, high-throughput evaluation across distributions and hyperparameters

VaR is found via bisection over candidate thresholds, which requires repeatedly estimating $P(\text{loss} > \text{threshold})$. Across 8 distributions and multiple seeds, this becomes thousands of probability-estimation calls. To make any scaling claim meaningful, we needed enough trials, enough seeds, and consistent methodology across classical and quantum baselines, all within 24 hours.

2) Toolchain integration friction between Classiq circuit synthesis and NVIDIA execution

Classiq's compiled state preparation uses multiplexed constructions that are not directly importable into a standard OpenQASM gate set, and CUDA-Q could not ingest these constructs directly. We built a custom decomposition pipeline that iteratively expands unsupported operations until the circuit is expressed in a supported basis, then used `q-convert` to translate QASM3 into CUDA-Q kernels.

A second integration issue was recompilation overhead: naïvely compiling a new circuit for every bisection threshold is too slow. We fixed this by exporting the state-preparation circuit once from Classiq, then building the threshold comparator and oracle dynamically in CUDA-Q during bisection. This reduced end-to-end runtime dramatically because we reused a small set of precompiled circuits tens of thousands of times instead of resynthesizing dozens per VaR run.

How did you use NVIDIA software tools?

CUDA-Q was the execution engine for our quantum experiments. We used it to run repeated circuit executions with lower overhead than our alternative simulator stack, and to access GPU-accelerated statevector simulation via cuQuantum. We used `cudaq.sample()` for measurement-based probability estimation loops and used CUDA-Q's workflow to run many repeated shots efficiently.

For the classical baseline, we used native CUDA acceleration through PyTorch plus custom GPU kernels for fast sampling and importance sampling variants. This was important for fairness: we wanted classical baselines that were also strongly optimized, not a slow CPU Monte Carlo strawman.

What NVIDIA hardware did you use, and why?

- **H200 (141GB HBM3):** coordination and large hyperparameter sweeps; memory headroom for trial history and parallel evaluations
- **A100 80GB:** GPU statevector simulation for IQAE circuits and batched execution
- **A10/A30 (cluster):** large-scale validation runs across many distributions and seeds
- **A4000:** low-cost development and Docker iteration for quick debugging

Why these tools?

- **Performance:** CUDA-Q delivered significantly faster repeated execution than our baseline simulator approach for our workload pattern (many repeated circuit calls inside bisection and IQAE loops).
- **Native GPU integration:** cuQuantum-backed simulation let us scale experiments without rewriting quantum internals.
- **Flexibility:** CUDA-Q supported the execution patterns we needed for both probability estimation and repeated trials.

How did these tools benefit you during iQuHACK?

NVIDIA's GPU stack made the experiment measurable rather than purely theoretical. It enabled us to run hundreds of hyperparameter trials quickly, repeat across multiple distributions and seeds, and complete synthesis + optimization + validation + analysis in a few hours of wall-clock time across multiple GPU types. That throughput is what let us make a credible scaling comparison between classical Monte Carlo (sample complexity scaling) and IQAE (query complexity scaling) within hackathon constraints.

CUDA-Q alongside Classiq: interaction and friction

Classiq handled high-level circuit design and synthesis, especially state preparation. CUDA-Q handled execution at scale on NVIDIA GPUs. The friction was that Classiq's compiled circuits included operations not directly supported by CUDA-Q import pathways. We solved this with a custom decomposition pipeline to reduce unsupported multiplexed operations into a supported basis, then converted QASM3 to CUDA-Q kernels. We also avoided recompiling circuits during VaR bisection by separating reusable state preparation (compiled once in Classiq) from the threshold-dependent oracle (built dynamically in CUDA-Q). This architecture made repeated VaR evaluation fast enough to sweep parameters and validate results across multiple distributions.

Conclusion

Our pipeline was hardware-agnostic across multiple NVIDIA GPUs and avoided recompilation bottlenecks by separating reusable state preparation from threshold-dependent logic. This made large-scale validation feasible and enabled a fair, repeatable comparison between classical Monte Carlo and IQAE for VaR estimation.