qjit function execution occurs asynchronously

Mohamed Hassan

July 1, 2024

1 Introduction

The goal of this challenge is to parallelize an interpreted programming model through asynchronous tasks. The dispatched asynchronous task returns immediately with a future data structure that will hold the result of the task's computation upon completion. A major concern for such an approach is data dependencies that may break functional correctness if not carefully maneuvered. The scenarios provided in this response implement the asynchronous tasks techniques safeguarded against data hazards.

2 Background

The implemented functions include both classical and quantum computation (hybrid functions) as requested. The functions include some rotation, CNOT, and Toffeli quantum gates in addition to matrix multiply operations to simulate longer execution times. The implemented scenarios test for various execution conditions and analyzes theoretical and actual speedup. All scenarios start by profiling a serial version of the function to evaluate the baseline performance. Serial function profiling is done by running the function 10 times, then dividing the total time by 10. This is to ensure a more accurate capture of performance eliminating the overhead of the first cold run.

3 Scenario A

This scenario tests running two parallel functions back to back with no data dependency between the two functions. The serial function's execution time is multiplied by 2 to account for running two functions. Theoretically this test should yield 2X speedup in parallel execution. The observed speedup is ~1.4X, which could be attributed to the fact that I am running on one device and the overhead of context switching is not insignificant.

4 Scenario B

This scenario runs the function four times, one serial function and three parallel functions. Since the serial function utilizes the main thread while the parallel ones are dispatched to separate threads, theoretically this should lead to a 4X improvement in performance. However, there is the added caveat that *"parallel_func_2"* uses the value returned from *"parallel_matmul"*. Hence this data dependence will have an impact on performance. The later *"parallel_func_2"* is preceded by *"future2.result()"* that blocks execution in the main thread until the result is ready to ensure the RAW data dependency hazard is resolved before invoking *"parallel_func_2"*. This scenario yields an actual speedup ~1.6X.

5 Scenario C

(I realize this test case was not in the code challenge, but it seemed like an opportunity to enhance performance in a safe way)

It was observed in the previous scenario that the main thread is blocked waiting to resolve a data dependency. However, what if downstream from that point in time there are other functions that could be run in parallel without any data dependence. This exposes another parallelism opportunity, where we only block the data-dependent parallel asynchronous task and go ahead with execution downstream. This is done using *"concurrent.futures.add_done_callback"* which creates a call back for the data-dependent function until the data it needs is ready. This approach will not block the main execution thread, continuing execution downstream

as long as there is no other data dependency. Since the test functions use multiple arguments (not just the future), *"functools.partial"* is used to pass more arguments to the call back function. This approach achieved ∼1.8X improvement in performance although it is running eight tasks instead of Scenario B's four tasks.

6 Comments

Two versions of each scenario are implemented, one with *qjit* and one without *qjit*. The reason for that is, upon testing, using *qjit* seems to always have a significant impact on performance. The performance results show that without *qjit* there is enhancement in performance, while with *qjit* there is a consistent degradation in performance. For the purposes of this coding challenge, I opted to implement non-qjitted versions of the scenarios to illustrate the parallel execution capabilities and pitfalls. The reported performance results are from the non-qjitted version of the code.

- For the non-jitted version, I created a decorative wrapper to conveniently call asynchronous parallel tasks. This wrapper didn't work with qjitted functions returning an error that the future is not a valid Jax type. This could be resolved by defining the wrapper in a way that conforms with PennyLane and Catalyst. Perhaps future work.
- I am running Apple M1 silicon, I am not sure if that hardware setup has any support issues (I remember seeing in Catalyst code some tests that include comments about Apple silicon not supported for some functions)
- It should be noted that if I had multiple devices available, a more significant speedup would have been observed for parallel execution.
- Sometime as the scenarios are run, the following error emerges:

  "*pennylane.QuantumFunctionError: All measurements must be returned in the order they are measured.*"

  The measurements are in fact returned in order, simply running the script again will yield a successful execution. (Albeit it may take two or three runs to get through)