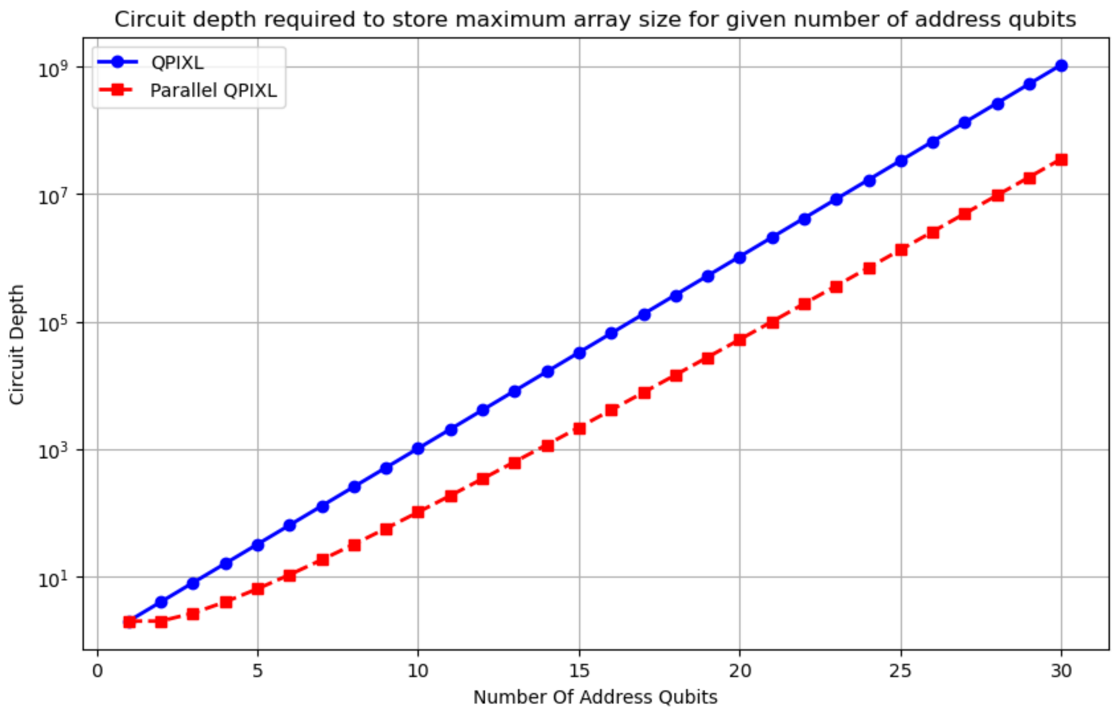
Qpixl_parallel_test

February 22, 2024

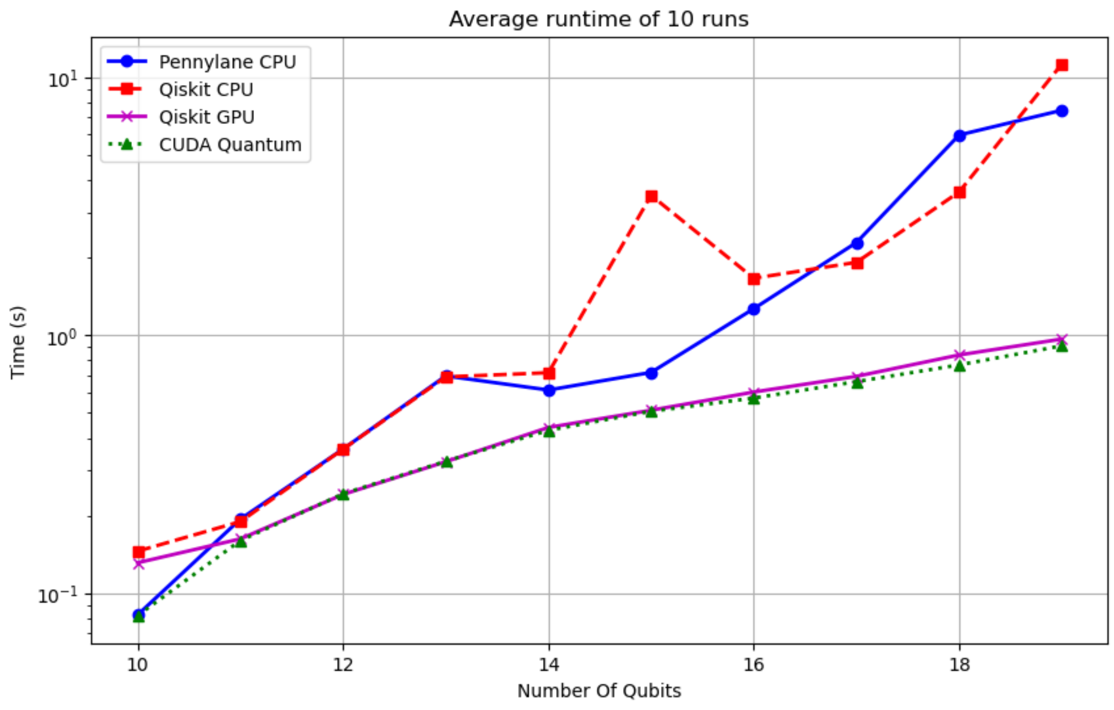
1 NVIDIA Prize Eligibility

1. In this project we implement an optimal (in depth) exact amplitude encoding scheme known as QPIXL and augment its capabilities by finding that it can have N different arrays simulatneously encoded for arrays that are of size 2^N. Furthermore, we implement parameterized versions of these circuits in CUDA quantum, pennylane and qiskit variants, so that they can be used for QML tasks with the Parallel QPIXL as an ansatz for state preparation.
2. The problem of amplitude embedding is of utmost importance for quantum processing of classical data. Having an efficient way to 'import' data into a quantum state, where it may be combined with other quantum enhanced features and then run through a QML algorithm or other time evolution would make for amazing research questions that go beyond full quantum or full classical ML paradigms, and perhaps form a combined quantum+classical data paradigm for QML (be it fully quantum or hybrid). Furthermore, it may be possible to encode more complex initial states for other algorithms, since the circuit structure is simple enough to preprocess classically, and the form of the state is exactly known, so that a quantum algorithm can continue on, via time evolution or other means, to compute observables of interest off of much better initial states. Of course, this is all exact and in the realm of fault tolerant quantum computing. Nonetheless, there are growing hints (such as those of QuEra), that fault tolerant computing may exist sooner than expected.
3. We use CUDA quantum and NVIDIA GPUs in the backend to process the very wide and deep circuits generated by this algorithm, and have written implementations of it for both standard data encoding and also as an 'ansatz' with variational parameters by assigning the angles via the `qc,vars = kernel(list)`, which can be found in the `QPIXL_CuQuantum` folder. Although the example notebook was run with the qiskit implementation, the statevector simulator was made to use the CuQuantum simulator. For the machine learning task, we modified the hybrid quantum neural network example on the MNIST dataset by embedding the data into the Parallel QPIXL algorithm and classify it with a tree tensor network classifier circuit appended to the end.
4. The performance results and scientific and numerical analysis are described in detail below, in such a way that the embedding is made clear, building up to the QML results. Suffice it to say that with the large states that are being dealt with, there was no way to simulate such circuits in a reasonable amount of time without NVIDIA GPUs. Below there is a figure showing the imporvement of parallelizing the data storage procedure to store an array of length 2^n, where n is the number of address qubits. Details of both QPIXL and this parallelization are dexribed in the rest of the PDF.



If we look at average runtime, we see, as expected, that the GPU outperforms CPU implementations of the simulators, but comparing CUDA quantum and the Qiskit GPU statevector simulator doesn't give a large difference at these qubit counts. The trend is that CUDA quantum somewhat outperforms Qiskit in this regard. All computations were run on the same machine with minimal other running programs.

```
<matplotlib.legend.Legend at 0x2190259fad0>
```



2 Parallel Exact Quantum Embedding

A series of python modules for implementing quantum amplitude embedding, which fulfills the challenge "Preparing for Battle".

The report goes through several embedding schemes we have come up with that are either good for simulating (simulator friendly encodings). The main feature is that we have implemented an embedding in qiskit for the FRQI-QPIXL framework (Amankwah et al., May 2022, https://www.nature.com/articles/s41598-022-11024-y) and have found a way to parallelize data input so that you can now embed N 2^N arrays in $2N$ qubits with the same circuit depth. This is included in three folders, one for a qiskit version, a pennylane version and a CuQuantum version. `qpixl.py` contains the original QPIXL algorithm, and `param_qpixl.py` contains its parameterized version that can be used to generate a NISQ friendly image feature map for QML amongst other things. The fies with `parallel` include the parallelized versions of the very saame codes.

- Contents
  - Introduction
  - QPIXL revisited
  - compression
  - parallel QPIXL
  - QML with full quantum embedding

3 QPIXL

3.1 Introduction

3.1.1 Quantum killers for NISQ

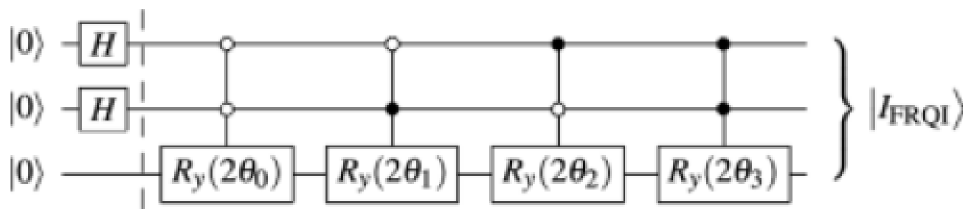
The depth (and connectivity) of a circuit completely determines how well it can be implemented on existing hardware. The killing blow to most algorithms is that they require fully connected and very deep circuits (which get decomposed to even deeper circuits with limited gatesets). Due to non-zero error rates, the probability that at least an error has occured throughout the run of a circuit eventually becomes 1. This can be mitigated, but at some point it can't be done. This is why short circuits for flexible data embeddings are so important. Assuming appropiate connectivity, it is possible to multiply the amount of data encoded within a constant depth.

3.1.2 Images in quantum computing

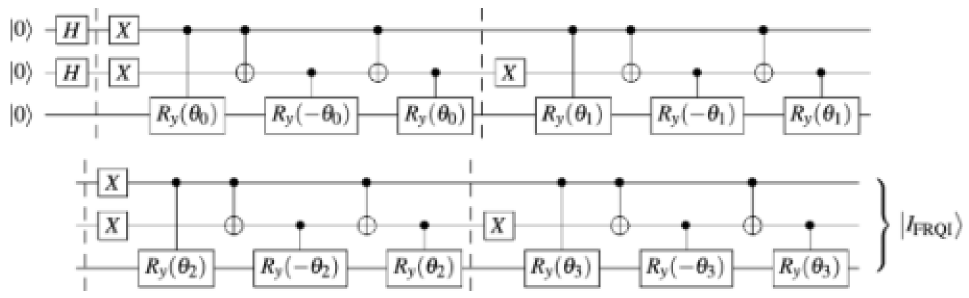
Although quantum computers have a more obvious to see advantage in quantum tasks, nonetheless it is thought that we can see some advantages in tasks involving classical data loaded onto a quantum computer. Although pictures may not be 'the' data-type that will see an advantage from quantum computing, it is nonetheless the case that a lot of data can be input in an image-like format, and studying pictoral algorihtms is definitely way easier on the eyes than pure data-driven tasks! Also, with a quantum state representing an image, you can see the results of any quantum transformation of the picture as a new picture! Of course, it needs to be seen from many 'angles', but maybe it can help with visualizing what is happening. Of course, images are just a stand in for arbitrary real valued data.

3.2 QPIXL algorithm

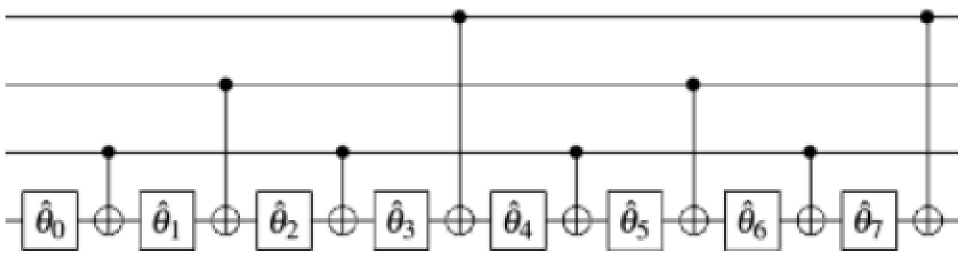
Why do we need another type of embedding in the mix? QPIXL is a framework to decompose popular image encodings such as FRQI, NEQR and their improved counterparts. It works by optimally decomposing the gates, and removing any 0 angles that are found. Thanks to the optimal decomposition the quantum gates can then be further reduced by removing pairs of CNOTS that used to be interweaved by rotation gates. They cancel out an become the identity. The FRQI embedding looks as follows:



but decomposed into CNOTS and rotations it looks like this!

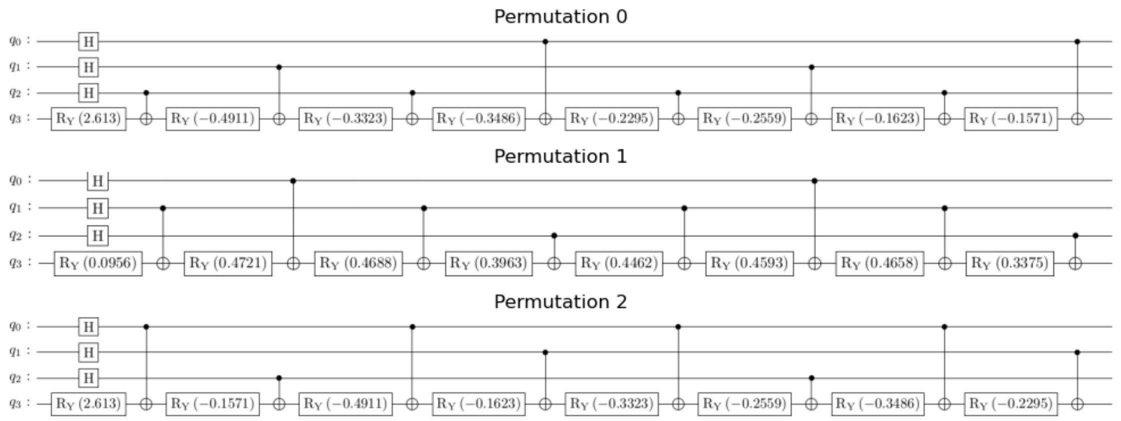


With QPIXL, the basic embedding, after transforming the image into an angle representation (using arctangent on the pixel values and a walsh hadamard transform) you have this much shorter decomposition! And this implies that we encode each of its elements into the amplitudes of the wave function.



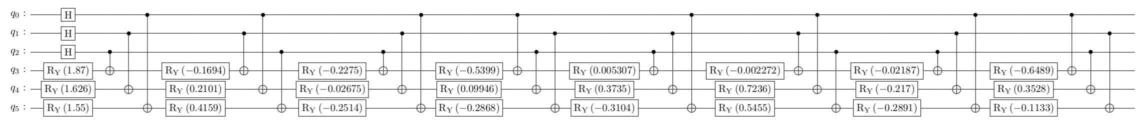
3.3 Implement multiple datasets in the same address qubits

The structure of QPIXL lends itself to a curious extension. Since the address qubits (all but the bottom qubit above) are each addressed by the data qubit (the lowest one in the figures above) individually. It is actually possible to add an additional data qubit (the one with the rotations) that then addresses a cyclic permutation of the qubits. This is shown below for some random data. This parallelization method was inspired by the paper (Balewski, J., Amankwah, M.G., Van Beeumen, R. et al., Sci Rep 14, 3435 (2024), https://doi.org/10.1038/s41598-024-53720-x).



3.4 Combining the three data streams into one!

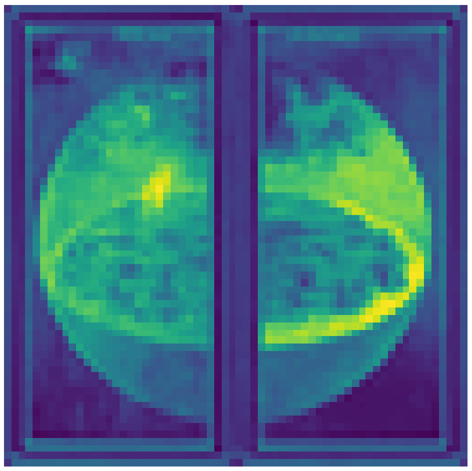
As you can see, and hopefully believe, the three different permutation never intersect at any slice of the circuit, and as such, you can combine all three data streams into a single one! This can be easily done in this code by giving an array with up to N elements, where N is the number of address qubits. Now, in the same depth you can encode N arrays, and the bigger the single array, the bigger the savings in depth, since the QPIXL algorithm is linear in depth with respect to the number of elements in the array (0 entries are actually ‘compressed’ out, so padding is always a valid strategy)



4 Encoding and decoding

Below, we show how you encode and then decode multiple images within one circuit. First you load and inspect your beautiful dataset

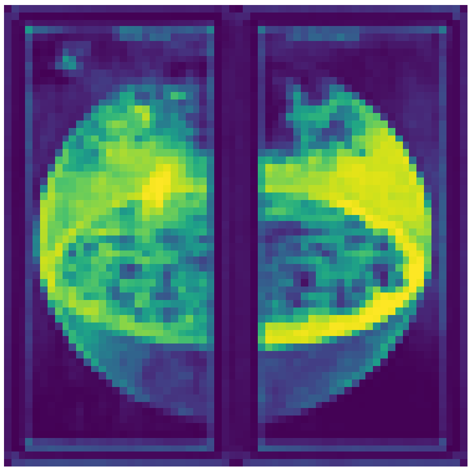
Now we can either pad or resize the data. Since these are images, we will make black and white copies with a size of 2^N pixels, which in this case is going to be 12 qubits, or 64*64 pixels.



Not quite as majestic as the originals, but they are quantum ready, so to the trained eye, the compressed versions are even more special. In the circuit, we need 12+2 qubits, 12 qubits address the 64*64 pixels, and the other two qubits encode the angles that represent the data. We encode the data as before

Unlike in qpixl that had a fairly straightforward encoding scheme, here we must trace out all data qubits except for the one that we want, and then permute the binary addresses back to what they would be before the cyclic permutation. For the implementation, you may inspect the code below, but suffice it to say that if you have D data qubits and you are interested in image d you would just take the partial trace

$$\rho_{\mathrm{d}} = \mathrm{Tr}_{D \neq d}[\rho_{\mathrm{full}}]$$



As you can see, the data has been successfuly recovered - the decoded data should in theory be exactly the same as the data put in, but the output images use 8bit color, so there is some rounding that would not exist with a slightly different implementation. The encoded state itself perfectly encodes the input data.

5 QML with fully quantum embedded data

With parallel QPIXL it is in theory possible to do bathced data upload for QML. For this task we have used the MNIST dataset, and attempted to encode the whole image onto the quantum computer via the circuit. The idea was to upload 10 ‘1’ images at once (since the closest power of two to the image size is 2^{10}) and likewise for the 0’s during training. Sadly, 20 qubits for just the data circuit + the QML part turned out to be too much, so we were unable to use this idea. The code is there in `QPIXL_QML`, but although it does run, more work is needed on the actual QML structure and important expectation values. It would be interesting to see how this compares to standard QPIXL and other embedding schemes.