ATPL Project

Matin Nafar (qhw109) & Kacper Zdrada (kmn339) & Simon Vinding Brodersen(rzn875)
Project Advisor: James Avery

January 19, 2026

Contents

1 Introduction

This paper presents the design, implementation, and benchmarking of a high-performance quantum circuit simulator specialised to execute Clifford circuits with GPU-acceleration. Due to the exponential growth of the state space, general quantum computation is classically infeasible. However, the Gottesman-Knill theorem proves that the specific subset of Clifford circuits can be simulated efficiently (in polynomial time) on a classical computer. Using this theorem, we developed a Futhark-based simulator that uses the bit-vector stabiliser formalism. Paired with a Haskell interface, we combine high-level functional control, with low-level parallel efficiency. Finally, we demonstrate that our implementation was able to simulate even up to 96000 qubits, and compare this to a much lower maximum for existing state-vector approaches in the way of a CPU Haskell simulator, and the GPU-accelerated industry leading Qiskit framework.

The primary issue that lies within simulating quantum circuits on classical hardware is that the state-vector model requires both $O(2^n)$ scaling with regards to time and space. However, there are many quantum algorithms, specifically error-correction protocols, that rely heavily on Clifford circuits. The Gottesman-Knill theorem establishes that these circuits can be simulated efficiently using stabliser formalism, which reduces the space complexity from exponential to quadratic.

We adopt a hardware-accelerated approach using the bit-vector model described, creating a custom simulator in the functional language Futhark, which compiles down to heavily optimised GPU-code. With the usage of a $(2n + 1) \times (2n + 1)$ tableau that tracks the stabilisers and destabilisers of the quantum state, we maximise parallalelism by implementing the Hadamard, Phase, and CNOT gates with a span of $O(1)$, and measurements with a span of $O(\lg n)$.
The Haskell interface created for this project exposes GPU execution for HQP programs that compile to the supported stabilizer instruction set; programs that cannot be lowered are rejected and must be handled separately by the caller.
Our implementations were benchmarked extensively against existing state-vector solutions - particularly the University of Copenhagen's CPU-based Haskell framework, and IBM's Qiskit Aer library (that too entails GPU acceleration). The results show that our implementation is massively superior, with successful simulations of 96000 qubits and 100000 gates performed within roughly 20 seconds.

Within the rest of the paper the mathematical background will be established; the Futhark implementation will be discussed; its correctness will be verified; the Haskell interface will be explained; the benchmarking tests and analysis will be presented; and future work will be reflected upon.

2 Background

This section will introduce some of the relevant mathematical background theory used within this project.

2.1 The State Vector Model

The standard model of quantum computing involves representing an n qubit state via a unit vector in an n-dimensional complex Hilbert space. For a single qubit, a generalisation of its state, $|\psi\rangle$, can be written as a linear combination of the basis vectors $|0\rangle = \begin{bmatrix}1\\0\end{bmatrix}$ and $|1\rangle = \begin{bmatrix}0\\1\end{bmatrix}$ as follows:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$$

where both α and β are complex numbers (probability amplitudes) that satisfy $|\alpha|^2 + |\beta|^2 = 1$.

However, with an n-qubit system, the resulting state space is the tensor product of all the individual Hilbert spaces. This results in the following generalisation of an n-qubit system [8]:

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle$$

where α_i is a complex amplitude.

Hence, it can be seen that the memory requirement for an n-qubit system will require $O(2^n)$ memory due to the need for keeping track of 2^n complex amplitudes.

In order to transform an n-qubit quantum state $|\psi\rangle$, quantum logic gates are applied to the state. Within this formalism these are represented by unitary matrices U which have a dimensionality of $2^n \times 2^n$. In order to apply this logic gate, the state vector is multiplied by the matrix as follows:

$$|\psi_1\rangle = U |\psi\rangle$$

With this operation, each of the 2^n probability amplitudes may change, and although a general matrix multiplication would require $O(4^n)$ time, standard quantum gates are sparse, allowing the update to be performed in $O(2^n)$ time. These asymptotics for both memory and time represent a significant challenge on simulating quantum circuits efficiently on classical computers.

2.2 Clifford Circuits

Clifford circuits however are a specific subsection of quantum circuits that can be simulated efficiently within polynomial time on a classical computer.

Clifford circuits are quantum circuits that are only composed of Clifford gates, which are a set of operators that normalise the Pauli Group. In other words, they map the Pauli operators I, X, Y, Z to other Pauli operators under conjugation. These Clifford gates can entirely be generated by the Hadamard, Phase, and Controlled-NOT gate. However, it is worth noting that Clifford gates do not allow for universal quantum computation [6].

The Gottesman-Knill Theorem [7] states that a quantum circuit using only the following elements can be simulated efficiently on a classical computer:

1. Preparation of qubits in computational-basis states
2. Clifford gates
3. Measurement in the computational basis

2.3 Stabiliser Formalism and the Bit Vector Model

The following formalism is primarily derived from the work of Aaronson and Gottesman [1].

Stabilisers: For a unitary matrix U to stabilise some quantum state $|\psi\rangle$, $|\psi\rangle$ must be an eigenvector of U with eigenvalue 1, or algebraically if $|\psi\rangle = U |\psi\rangle$.

Stabiliser States: Now, any quantum state that can be created via a Clifford circuit is called a stabiliser state. These stabiliser states are unique in the fact that they can be represented using n unitary matrices that stabilise them. Furthermore, these states will be stabilised more specifically by a unique set of n independent Pauli operators. Hence, instead of tracking 2^n complex amplitudes, they can be simply be represented through these n Pauli operators. Each one of these Pauli stabilisers can in turn be written as a tensored Pauli string, with the Pauli gate for each qubit being represented using 2 bits.

Bit-Vector Representation: This brings us to the bit-vector representation of stabiliser quantum states. In order to store the n stabilisers it is required to have n rows of information. As there are then n qubits, and the Pauli gate for each can be written using 2 bits, along with an extra bit to keep track of the phase of the stabiliser each row will have $2n + 1$ entries. Hence, the state can be represented with a $n \times (2n + 1)$ binary matrix, reducing the space complexity from $O(2^n)$ to $O(n^2)$.

Gate Operations: When performing operations on the quantum state, it is simply enough to update the stabilisers, and the three Hadamard, Phase, and Controlled-NOT gates can be shown to only require $O(n)$ time. This is because each gate requires simple $O(1)$ operations on each row, such as swapping bits or performing logical XOR operations, and as there are n rows, this results in a linear runtime.

Measurement Operations: However, the issue in complexity arises with the measuring of the state. It can be decided within $O(n)$ time whether the measurement of a certain qubit will yield an outcome that is certain or deterministic. This is still dominated by a factor of $O(n^2)$ required to update the matrix if the measurement outcome is random, and even worse still of $O(n^3)$ if it is deterministic.

However, by doubling the memory to also keep track of the n destabilisers, it can be shown that updating the matrix will take $O(n^2)$ time regardless of whether the measurement is random or deterministic. Destabilisers are also Pauli operators that anticommute with their own stabilisers, and commute with every other stabiliser. Hence, the final size of the binary matrix to represent the quantum state is $2n \times (2n + 1)$.

3 Implementation

Our implementation closely follows the description in [1], where we focus on maximizing the parallel nature of the implementation. This implementation can be found in the accompanying file `definitions.fut`.

Our implementation uses 0 indexing, as opposed to the papers 1 indexing, which means our tableu is the following:

$$\left(\begin{array}{ccc|ccc|c}
x_{00} & \cdots & x_{0(n-1)} & z_{00} & \cdots & z_{0n} & r_0 \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\
x_{(n-1)0} & \cdots & x_{(n-1)(n-1)} & z_{(n-1)0} & \cdots & z_{(n-1)(n-1)} & r_{n-1} \\
\hline
x_{n1} & \cdots & x_{n(n-1)} & z_{n0} & \cdots & z_{n(n-1)} & r_n \\
\vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\
x_{(2n-1)0} & \cdots & x_{(2n-1)(n-1)} & z_{(2n-1)0} & \cdots & z_{(2n-1)(n-1)} & r_{2n-1} \\
x_{(2n)0} & \cdots & x_{(2n)(n-1)} & z_{(2n)0} & \cdots & z_{(2n)(n-1)} & r_{2n}
\end{array}\right) \quad (1)$$

This was done to play nicer with the default indexing found in most languages including futhark. Note, that the first n rows $[0 : n-1]$ is tracking the destabilizers, the next n rows $[n - 2n - 1$ are tracking the stabilizers and the final row is a scratch row.

3.1 Initial tableu

Any gate sequence is started off by the initial_tableu, which can be seen defined below:

```
type t = i8
type tab [n] = [2 * n + 1][2 * n + 1]t

def initial_tableu (n: i64) : tab [n] =
  let tmp = 2 * n + 1
  in tabulate_2d tmp tmp (\i j ->
                                      if j == tmp - 1 || i == tmp - 1
                                      then -- r_i and last row
                                           0
                                      else if i == j
                                      then -- diagonal
                                           1
```

```
                                        else 0)
```

This initial tableu represents the state $|0 \ldots_{n-2} 0\rangle$, i.e. it is the zero state given n qubits. Subsequent preparation of qubits before a main gate circuit is left for the user, which means that a user always starts off from this initial tableu. The function is able to compute all cells individually, and as such the span of the function is $O(1)$ and the work of it is $O(n^2)$.

3.2 The CNOT gate

The CNOT gate can be seen defined below:

```
def CNOT [n] (tableu: *tab [n]) (a: i64) (b: i64) : *tab [n] =
  let tmp = 2 * n
  let indices =
    map (\i ->
            [(i, tmp), (i, b), (i, n + a)])
         (iota tmp)
    |> flatten
  let values =
    map (\i ->
            let zia = tableu[i][n + a]
            let zib = tableu[i][n + b]
            let xia = tableu[i][a]
            let xib = tableu[i][b]
            let ri = tableu[i][tmp]
            in [ri ^ xia * zib * (xib ^ zia ^ 1i8),
                xib ^ xia, zia ^ zib])
         (iota tmp)
    |> flatten
  in scatter_2d tableu indices values
```

This function takes ownership of the current tableu (denoted by the *tab [n]), such that we do not have to copy the tableu on every gate usage. This is a common attribute between all the gates. Given qubit index a and b, we can calculate the indices this gate wishes to update, before calculating the values these indices should be given. This way we do not introduce any inconvinient race conditions. At last, we can use a single scatter call to update all the values inside the tableu. Given that we map over $2 * n$ elements and scatter with $3 * 2 * n = 6n$ values, then the total span of this algorithm is $O(1)$ and work $O(n)$.

3.3 The Clifford gates

The Hadamard and Phase gate follow almost exactly the same strategy when updating the tableu. The only difference being the calculations needed to update the tableu. As such, they have the same exact work and span as the CNOT gate. In fact given that these 3 gates can fully simulate the entire Clifford group, then for any gate within the Clifford group, we would be able to do the simulation with exactly the same work and span. As of now the other Clifford group operators are not implemented, but chaining together Hadamard, Phase and CNOT in a specific manner would give the same result as any such gate. However, for performance such gates could be implemented in the future, something we will come back to later.

```
def Hadamard [n] (tableu: *tab [n]) (a: i64) : *tab [n] =
  let tmp = 2 * n
  let indices = map (\i ->
         [(i, tmp), (i, a), (i, n + a)]
    ) (iota tmp) |> flatten
  let values =
    map (\i ->
            let ri = tableu[i][tmp]
            let xia = tableu[i][a]
            let zia = tableu[i][a + n]
            in [ri ^ xia * zia, zia, xia])
         -- swap in regards to indices, which are ri, xia, zia
         (iota tmp)
    |> flatten
  in scatter_2d tableu indices values

def Phase [n] (tableu: *tab [n]) (a: i64) : *tab [n] =
  let tmp = 2 * n
  let indices = map (\i -> [(i, tmp), (i, a + n)]) (iota tmp)
                |> flatten
  let values =
    map (\i ->
            let ri = tableu[i][tmp]
            let xia = tableu[i][a]
            let zia = tableu[i][n + a]
            in [ri ^ xia * zia, zia ^ xia])
         (iota tmp)
    |> flatten
  in scatter_2d tableu indices values
```

3.4 Measurement

The measurement implementation is a bit more involved. The measurement uses two helper functions, which are defined below:

```
local
def g (x1: t) (x2: t) (z1: t) (z2: t) =
  if x1 == z1 && x1 == 0
  then 0
  else if x1 == z1 && x1 == 1
  then z2 - x2
  else if x1 == 1 && z1 == 0
  then z2 * (2 * x2 - 1)
  else if x1 == 0 && z1 == 1
  then x2 * (1 - 2 * z2)
  else 0

local
def rowsum [n] (tableu: tab [n])
               (h: i64)
               (i: i64) : ([n*2+1](i64, i64), [n*2+1]i8) =
  let max_idx = (size n) - 1
  let tot_sum = map (\j -> g tableu[i][j] tableu[i][j + n] tableu[h][j]
                tableu[h][j + n]) (iota n) |> reduce (+) 0
  let res =
    (2 * tableu[h][max_idx] + 2 * tableu[i][max_idx] + tot_sum) % 4
  let v = if res == 0 then 0 else 1
  let is = map (\j -> [(h, j), (h, j + n)]) (iota n) |> flatten
  let vs = map (\j -> [tableu[i][j] ^ tableu[h][j], tableu[i][n + j] ^
  tableu[h][n + j]]) (iota n) |> flatten
  in (is ++ [(h, max_idx)], vs ++ [v])
```

The g function is quite self explanatory, and the rowsum function sets r_h to 0, if:

$$\left(2r_h + 2r_i \sum_{j=0}^{n-1} g(x_{ij}, z_{ij}, x_{hj}, z_{hj}\right) \mod 4 \equiv 0 \tag{2}$$

and set r_h to 1 otherwise. Then, for $j \in \{1, \ldots, n\}$ we update $x_{hj} = x_{hj} \oplus x_{ij}$ and $z_{hj} = z_{hj} \oplus z_{ij}$.

The function header is provided below, along with some initial calculation.

```
def Measurement [n] (eng: rng_engine.rng)
                    (tableu: *tab [n])
                    (a: i64) : (rng_engine.rng, *tab [n], t) =
  let (p, xpa) =
    reduce_comm (\(p1, xp1a) (p2, xp2a) ->
                  if xp1a == 1 && xp2a == 1
                  then if p1 <= p2
                       then (p1, xp1a)
                       else (p2, xp2a)
                  else if xp1a == 1
                  then (p1, xp1a)
                  else if xp2a == 1
                  then (p2, xp2a)
                  else (0, 0))
                (0, 0)
    <| map (\p -> (p, tableu[p][a])) (n...2 * n - 1) -- both inclusive
```

This function computes the measurement of qubit a. The first thing we do is compute the minimum index p, for which $x_{pa} = 1$. If we find a p, for which $x_{pa} = 1$, then we know that the outcome is non-deterministic, and if no such p exists, then it is deterministic.

Non-deterministic case: This case can be seen implemented below:

```
...
in if xpa == 1
 then -- Call rowsum for all i in {1...2n}
       let filtered_is = filter (\i -> i != p && tableu[i][a] == 1)
                                (iota (2 * n))
       let (is1, vs1) =
         map (\i ->
                rowsum tableu i p)
             (filtered_is)
         |> unzip
       let tmp1 = scatter_2d tableu (is1 |> flatten) (vs1 |> flatten)
       -- set (p - n) row equal to pth row
       let is2 = map (\i -> (p - n, i)) (iota (size n))
       let vs2 = map (\i -> tmp1[p][i]) (iota (size n))
       let tmp2 = scatter_2d tmp1 is2 vs2
       -- set pth row 0 except rp is 0 or 1 50/50 and zpa = 1
       let (eng1, rand_val) = rand_i8.rand (0, 1i8) eng
       let is3 = map (\i -> (p, i)) (iota (size n))
       let vs3 =
         map (\i ->
                -- zpa
                if i == (n + a)
                then 1
```

```
                  else -- rp
                  if i == (2 * n)
                  then rand_val
                  else 0)
              (iota (size n))
        let tmp3 = scatter_2d tmp2 is3 vs3
        in (eng1, tmp3, rand_val)
...
```

The first thing we do is call the rowsum function, for all i, where $i \neq p$ and $x_{ia} = 1$. Now because we know that the `is` are independent, then they will update different values in the tableu. We have made the decision to make the rowsum function return the indices together with the values it wants to update. This allows us to gather them together for all the `is` in one go, and we can do a single scatter call instead of multiple.

Afterwards, we update the $p - n$ row to be equal to the pth row, set the pth row to 0 except r_p, which is 0 or 1 with a 50/50 chance and set z_{pa} to 1. Note, that the measurement function also takes in a random engine as input, and if it is used, then we return a new random engine. This is futharks version of randomness (based on the one in cpp), which makes it such that the outcomes are independent by giving us a new engine once it is used.

This part of the algorithm first calls filter, which is what dominates the span of the algorithm, giving it a span of $O(\lg n)$. The reason for this is, that while rowsum does have nested maps, nested maps in futhark are parallelised [4] via flattening. This means, that it is only the work of the algorithm which dominates this case, and it happens to be $O(n^2)$.

Deterministic case: This case is provided below:

```
...
else let max_idx = (size n) - 1
  let is1 = map (\i -> (max_idx, i)) (iota (size n))
  let vs1 = replicate (size n) 0
  let tmp1 = scatter_2d tableu is1 vs1
  -- Filter indices where x_ia == 1
  let filtered_is = filter (\i -> tmp1[i][a] == 1) (iota n)
  -- Compute all contributions in parallel using INITIAL tmp1 state
  let (phases, x_vals, z_vals) =
    map (\i ->
           let idx = i + n
           let tot_sum =
             map (\j ->
                    g tmp1[idx][j]
                      tmp1[idx][j + n]
                      tmp1[max_idx][j]
                      tmp1[max_idx][j + n])
                 (iota n)
             |> reduce (+) 0
           let res = (2 * tmp1[max_idx][max_idx]
                      + 2 * tmp1[idx][max_idx]
                      + tot_sum) % 4
           let phase_contrib = if res == 0 then 0 else 1
           let x_contribs = map (\j -> tmp1[idx][j]) (iota n)
           let z_contribs = map (\j -> tmp1[idx][n + j]) (iota n)
           in (phase_contrib, x_contribs, z_contribs))
         filtered_is
    |> unzip3
  -- XOR all contributions together
  let final_phase = reduce (^) 0 phases
  let final_x = map (\col -> reduce (^) 0 col) (transpose x_vals)
  let final_z = map (\col -> reduce (^) 0 col) (transpose z_vals)
  -- Update row max_idx with final values
  let is2 = map (\i -> (max_idx, i)) (iota (2 * n))
            ++ [(max_idx, max_idx)]
  let vs2 = ((final_x ++ final_z ++ [final_phase]) :> [2 * n + 1]i8)
  let tmp2 = scatter_2d tmp1 is2 vs2
  in (eng, tmp2, final_phase)
```

In the deterministic case, we start of by setting the row $2n$ to 0 (note $max\_idx = 2 * n + 1$), and then we find the indices for which $x_{ia} = 1$. In the paper they would again call rowsum on each of the indices. However, this would lead to sequential behaviour, as each index would attempt to write to the same location. Instead, we compute the contributions that each index would have on the overall result, and then we xor all of the values together. This way we are able to compute the result in a parallel manner improving performance. Afterwards, we can simply scatter the final values to their location and return.

This part of the measurement is also dominated by the filter and reduce calls. The nested reduce is converted via. flattening as well [4], so the span of this part of the algorithm also happens to be $O(\lg n)$. The work for this part of the algorithm still happens to be $O(n^2)$, as we are mapping over all the indices inside the map to calculate the contributions.

Conclusion: This leads us to the final analysis for the Measurement gate. We can see, that it happens to be asymptotically the most expensive gate both in terms of span of $O(\lg n)$ and work being $O(n^2)$ as was referenced in the original paper [1]. However, with an impressive span of only $O(\lg n)$, the implementation should be quite efficient at computing many gates with a lot of qubits on systems with a lot of cores to handle the work.

3.5 The simulate function

The final function for the futhark implementation is provided below:

```
def simulate [n] (seed: i32)
                 (num_qubits: i64)
                 (gates: [n]i64)
                 (cQ: [n]i64)
                 (tQ: [n]i64) : ([][]i8, []i8) =
  let zipped_gates = zip3 gates cQ tQ
  let num_measurements = reduce (+) 0 <|
                         map (\(gate, _, _) ->
                              if gate == 0 then 1 else 0
                         ) zipped_gates
  let (tableu, _, _, measurements, _) =
    loop (tableu, i, rng, measurements, measurement_count) =
         (initial_tableu num_qubits,
          0,
          rng_engine.rng_from_seed [seed],
          replicate num_measurements 0,
          0)
    while i < n do
      let (gate, control, target) = zipped_gates[i]
      in if gate == 0
         then let (new_rng, new_tab, measurement) =
                  Measurement rng  tableu control
              in (new_tab
                 , i + 1
                 , new_rng
                 , measurements with [measurement_count] = measurement
                 , measurement_count + 1
                 )
         else if gate == 1
         then (Hadamard tableu control,
               i + 1,
               rng,
               measurements,
               measurement_count)
         else if gate == 2
         then (Phase tableu control,
               i + 1,
               rng,
               measurements,
               measurement_count)
         else if gate == 3
         then (CNOT tableu control target,
               i + 1,
               rng,
               measurements,
               measurement_count)
         else -- do nothing
              (tableu, i + 1, rng, measurements, measurement_count)
  in (tableu, measurements)
```

This function takes in a random seed (used to create reproducible results), the number of qubits to simulate, along with a list of gates (gates), control qubits (cQ) and target qubits (tQ). In the case of unitary gates, then the target qubit (tQ) is garbage data and is ignore, it is instead the control qubit (cQ) that is used for the unitary gates.

This function first initializes the empty tableu via. the initial_tableu function. It then sequentially loops through all of the gates and calls the respective function. The gates are represented by 64-bit integers for more seamless integration with outside function calls, such as calling the implementation from Haskell. The conversion between integers and gates is summarized in the table below:

Integer	Gate
0	Measurement
1	Hadamard
2	Phase
3	CNOT
Other	Do nothing

3.6 Final asymptotics

As the simulate function runs sequentially over all of the gates, then in the worst case, when all gates are measurements, we would end up with the span of $O(m \lg n)$ and work of $O(mn^2)$, where m is the number of gates and n is the number of qubits. This sequential loop is a hard hit to the parallel performance, but it is something we believe could be improved upon, which we will come back to in the future work section.

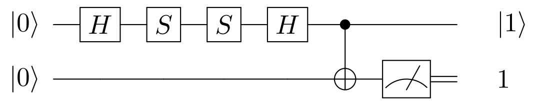
Further, as we do not expect there to be a large amount of measurements, for circuits that contain no measurements we expect the asymptotics to be span $O(m)$ and work $O(\max(mn, n^2))$, as the initial tableu has work $O(n^2)$ which in the case we have just a few gates would be more costly than computing the gates in terms of work.

4 Verification

To verify our implementation a variety of different quantum programs were implemented and tested manually. Some of the more interesting ones are the ones seen below:

4.1 Testing CNOT functionality

First, we wanted a very simple program, which uses each of the gates once to verify that our implementation is able to handle a sequence of gates. The simple program we came up with is the following.



(3)

This simple gate initializes the first qubit to be in the deterministic state $|1\rangle$. A subsequent CNOT gate would then flip the initial $|0\rangle$ of the second qubit to be a 1 as well.

Results: When running this simple circuit, we do indeed see, that our measurement produces a 1. Further, for the argument of correctness, we can see that the final tableu once the circuit has finished is the following:

$$\text{Tableu: } = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{4}$$

After manual Gaussian elimination on the stabilizers:

$$= \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \tag{5}$$

As noted in earlier sections, the first two rows store the destabilizers for the qubits, rows 2 and 3 store the stabilisers and the last row is the scratch row we are using for computations. When checking for the measurement, we are looking at cells [3,1], [3,2], [4, 1], [4, 2] (1-indexing at this point) and checking if either contains a 1. In this case none of them do, so we know that the measurement outcome is deterministic. We then update our scratch row using the afore mentioned xor reduction, which then stores our measurement result in the bottom right corner of the matrix, which just happens to be 1.

The second matrix included has done Gaussian reduction on the two stabilizer rows (3-4), which shows that these two rows have full rank which is what the traditional method would do to check for deterministic result. The result would then be stored within these two rows.

For example for row 3, we have the bit string "00101". The first bit together with the third bit give us the needed information about the first stabilizer, here we see that we have a "01", which indicates that we do not have an X and we do have a Z. Therefore, this would be the Z stabilizer. Using the same logic for bit 2 and 4, we have that this corresponds to the I. At last, the final bit indicates our phase, which in this case is a 1, meaning that our phase is negative. Thus, this row corresponds to $-ZI = -Z = |1\rangle$.

If we do the same walk through for row 4, then we find that this corresponds to $-IZ = -Z = |1\rangle$, so the tableu correctly contains the expected qubit information.

4.2 Teleportation

The teleportation program is a little more involved than that of the previous case. As such, the testing was instead done by implementing the teleportation given in [1], where we first initialize qubit 0 by using a Hadamard on it.

This gives us the following program:

Listing 1: Futhark implementation of the teleportation program

```
def teleportation (seed: i32) : ([][]i8, []i8) =
  let (tg, tc, tt) =
    ( [1i64, 3, 3, 1, 0, 0, 3, 3, 3, 1, 3, 1]
    , [1i64, 1, 0, 0, 0, 1, 0, 1, 4, 2, 3, 2]
    , [0i64, 2, 1, 0, 0, 0, 3, 4, 2, 0, 2, 0]
    )
  let (gates, cQ, tQ) =
    ([1] ++ tg ++ [0],
     [0] ++ tc ++ [4],
     [0] ++ tt ++ [0])
  in simulate seed 5 gates cQ tQ

def estimate_teleportation (n: i64) : f64 =
  let measurement seed =
    let tmp = teleportation (i32.i64 seed) |> (.1)
    in tmp[2]
  let mean =
    map (\i -> f64.i8 <| measurement i) (iota n) |> reduce (+) 0
    |> (/ (f64.i64 n))
  in mean
```

First, tg are the teleportation gates, which summarize to [H, CNOT, CNOT, H, M, M, CNOT, CNOT, CNOT, H, CNOT, H], tc is the control qubit for CNOT gates and tt is the target qubit. As before, if we have a unitary gate, then tc is the qubit we are changing, and tt is garbage.

We prepend an initial H on qubit 0 and append a measurement on qubit 4, as this program should teleport the qubit 0 to qubit 4. The method for testing this circuit was to create the function `estimate_teleportation`, which takes in a number of times it should run teleportation, and reports the mean value measured over the runs. Note, that on every run, we have a different seed for the random generation[1].

When I ran this code 150 times, then I found that the average measurement outcome was 0.50 with two digit precision. While this does not guarantee that we correctly teleported the qubit 0, one could increase the number of iterations until they are convinced beyond a reasonable doubt, or they could do an analysis similar to before to determine that the teleportation was successful.

5 Haskell Interface

The Futhark `simulate` kernel is accessed from Haskell through a thin two-stage interface that separates circuit compilation from backend execution. At the front, an HQP program expressed in the interpreted stabilizer language is compiled into a compact instruction stream whose operands are plain qubit indices. At the back, the instruction stream is transferred to the Futhark-generated runtime through the C API produced by `futhark {cuda,opencl} --library`, and the resulting measurements and final tableau are copied back into Haskell data structures. The design goal is that any part of the execution that can be expressed in a data-parallel form is delegated to Futhark, while Haskell performs only orchestration, validation, and translation between representations.

[1]in this case, we are just taking seeds from [0...n]

5.1 Compilation from HQP to a backend instruction stream

The module `HQP.QOp.GPU.Compile` translates HQP programs into the concrete gate encoding expected by the Futhark kernel. The compiler traverses the HQP program, extracts the circuit width, and lowers each step into a sequence of instruction records. Each instruction stores an opcode together with up to two qubit operands, matching the backend interface that expects a control index and a target index, where the target is ignored for single-qubit gates and measurements. Only stabilizer operations supported by the backend are emitted. If the program contains unsupported gates, compilation fails and returns a `Left` value.

This design enables a safe hybrid execution strategy in which programs are dispatched to either the GPU or CPU backend based on the result of the compilation phase, as demonstrated in the executable `HybridGPUCPU.hs`. Unsupported programs are thus rejected during compilation to the GPU backend rather than causing runtime failure.

The backend operates on integer opcodes rather than algebraic data types. This ensures stable data transfer across the FFI boundary and avoids any dependence on Haskell-level constructors at runtime. In the current encoding, measurement, Hadamard, Phase, and CNOT are represented by small integers, which align with the dispatcher logic in the Futhark `simulate` kernel.

5.2 FFI layer and calling the Futhark library

The execution layer resides in `HQP.QOp.GPU.RunFuthark`. Futhark generates a C API consisting of a context configuration, a context, array constructors, entry points, and array readback functions. The Haskell wrapper imports these symbols through the FFI, including `futhark_context_config_new`, `futhark_context_new`, `futhark_entry_simulate`, the relevant `futhark_new_*` array constructors, `futhark_values_*` functions, and `futhark_free_*` destructors.

At runtime, the wrapper allocates a Futhark context using a `bracket`-based resource scope, ensuring clean-up even if execution fails. The instruction stream is packed into three contiguous `i64` vectors for opcodes, control-qubit indices, and target-qubit indices. This avoids opaque parameter warnings that arise when passing arrays of tuples through the Futhark C API. The vectors are uploaded into device-managed arrays using `futhark_new_i64_1d`, after which the wrapper invokes `futhark_entry_simulate`.

The Futhark program returns both the final tableau and measurement outcomes. For CUDA and OpenCL backends, execution and readback are asynchronous, so the interface explicitly synchronizes using `futhark_context_sync`. Runtime errors are retrieved via `futhark_context_get_error` and surfaced as Haskell exceptions. Measurement results are read by querying array shape with `futhark_shape_i8_1d` and copying values using `futhark_values_i8_1d`. The tableau is read similarly using the 2D variants. All allocated Futhark arrays are freed after use.

The interface exposes multiple convenience entry points. A measurement-only call returns just the measurement vector and discards the tableau, which is useful when benchmarking large circuits. A combined call returns both measurements and the final tableau, which is intended as the primary interface when the final state is required for post-processing.

5.3 Build integration

The Futhark code is compiled as a library and linked into the Haskell executable. The compiler is invoked with `--library`, producing `hqp_gpu.c` and `hqp_gpu.h` under a local `futhark-gen` directory. The Cabal configuration includes this C file under `c-sources` and adds `futhark-gen` to `include-dirs`. When using GPU backends, the executable is linked against the system CUDA or OpenCL libraries depending on the selected backend. This arrangement keeps the Futhark kernel as a single source of truth while allowing Haskell to invoke it as an ordinary C library.

5.4 Result representation and inspection

Measurement outcomes are returned as storable vectors of `Int8`, matching the `i8` output type of the Futhark kernel. The tableau is returned as a flat `Int8` vector together with its row and column counts, using row-major layout where element (r, c) resides at index $r \cdot \texttt{cols} + c$. This representation enables constant-time indexing without additional copying. For debugging, utilities are provided to print only a prefix of the tableau, which is essential for large circuits where the tableau grows quadratically in the number of qubits.

6 Benchmarking

This section will provide a detailed insight into how the performance of our Clifford-simulator was analysed and provide comparisons to existing frameworks.

6.1 Goal

The goal of the project was to compare the performance of the parallelised bit-vector implementation to the provided quantum simulation framework, as well as the industry-leading framework Qiskit. In theory we aim to demonstrate the practical implications of the Gottesman-Knill Theorem, which states that any Clifford Circuit can be simulated efficiently (in polynomial time) on a classical computer.

6.2 Experimental Setup

Hardware All benchmarks were ran on the hardware found in table 1.

Table 1: Hardware Test Environment Specifications

Component	**Specification**
	Host System (CPU)
Processor Model	2 × AMD EPYC 7352 ”Rome”
Architecture	x86_64 (Zen 2)
Core Configuration	48 Physical Cores (24 per socket)
Thread Count	96 Logical Threads (SMT Enabled)
Topology	2 NUMA Nodes
	Accelerator (GPU)
Model	NVIDIA A100-PCIE-40GB
Architecture	NVIDIA Ampere (Compute Capability 8.0)
Memory	40 GB HBM2e
Interface	PCIe Gen 4.0 x16
Driver / CUDA Version	570.86.15 / 12.8

Comparison Frameworks: Our main aim is to compare between the simulation of the two different quantum state representations - ours being a bit-vector representation, and the comparison being a state-vector representation. In order to further enhance our performance however, we take advantage of GPU parallelisation. Thus we compare to two separate frameworks:

1. **ATPL/HQP Haskell Framework v3:** This is provided by the Department of Computer Science at the University of Copenhagen. It allows for CPU-simulations of Quantum Circuits using state-vector representations and matrix algebra.

2. **Qiskit Aer:** This is an open-source industry-leading quantum simulator for circuits written in the Qiskit framework, provided by IBM Research. Whilst it involves a Python interface, it runs on a highly-optimised C++ backend. This framework allows for both CPU and GPU simulation of quanutm circuits using state-vector representations, and thus we choose to use the GPU to provide a comparison to our own implementation.

6.3 Methodology

Test Data Generation: The test data was generated using a simple Futhark script, allowing for completely randomised Clifford circuits up to n qubits and g Clifford gates. We set parameters for these values of n and g, and create test data for every possible combination. All generated circuits begin from the zero state. An example dataset file can be found below:

Listing 2: Example of a Benchmark Input File ($q = 5$ Qubits)

```
-223823371i32
5i64
[1i64, 1i64, 1i64, 2i64, 2i64, 3i64, 1i64, 1i64, 1i64, 3i64]
[1i64, 0i64, 1i64, 1i64, 1i64, 1i64, 1i64, 1i64, 0i64, 0i64]
[2i64, 3i64, 4i64, 3i64, 3i64, 4i64, 4i64, 4i64, 4i64, 3i64]
```

1. The first line is a 32-bit integer seed that is used by all frameworks and implementations in order to keep any randomised results (particularly to do with probabilistic measurement) reproducible.

2. The second line is a 64-bit integer that specifies the number of qubits used.

3. The third line is a g-length list of 64-bit integers that are bound between 0 and 3 and they represent the Clifford gates that are being applied in order in the circuit (**0**: Measurement, **1**: Hadamard, **2** Phase, **3**: Controlled-NOT).

4. The fourth line is a g-length list of 64-bit integers. The i-th integer in this list specifies to which qubit the i-th gate in the previous list is being applied. In the case that the i-th gate is a Controlled-NOT gate then this value is the control qubit.

5. The fifth line is a g-length list of 64-bit integers. This is a redudant row whose data is ignored only except when the i-th gate is a Controlled-NOT and requires two qubits, in which case the integer is the target qubit.

Transpilation This standard data format is passed into custom transpilers that are able to convert it into actual system instructions/quantum circuits that the different frameworks can interpret and execute.

The Futhark implementation takes in each of the lines of the dataset file as parameters, and then runs a sequential loop on each instruction and applying the changes to the tableau. Thus, it effectively acts as an interpreter.

A snippet of the Haskell transpiler can be found below:

Listing 3: Transpiling Dataset into HQP

```
interpretOperation :: Int -> OperationInput -> Step
interpretOperation qubit_num (opcode, qubit1, qubit2)
    | opcode == 0 = Measure [qubit1]
    | opcode == 1 =
        Unitary (Id qubit1 ⊗ H ⊗ Id (qubit_num - 1 - qubit1))
    | opcode == 2 =
        Unitary (Id qubit1 ⊗ R Z (1/2) ⊗ Id (qubit_num - 1 - qubit1))
    | opcode == 3 = cnotHelper qubit_num qubit1 qubit2
    | otherwise = error "Invalid opcode"
```

Again the program takes in the entire dataset file, and converts it into HQP's Program type, which then gets evaluated. There is no circuit optimisation that occurs.

The Qiskit transpiler takes in the entire dataset file, and converts it into a QuantumCircuit object. This is then transpiled even further through Qiskit's library to be able to run on the Aer backend, performing circuit optimisations in the meantime. This is then ran on the backend.

For fairness, all preprocessing steps of this transpilation are not included in the benchmark - only the final simulation is measured.

Benchmarking Software

1. **Futhark:** In order to test our Futhark implementation we used the built-in Futhark benchmarking software (`futhark bench`), which only measures the the runtime of the entry point and does no take into account compilation and parsing of the data file. However, it does factor in the time taken to copy the input arrays from CPU RAM to GPU VRAM and the time to copy the measurement result back to CPU RAM. The benchmark runs one warmup run that is discarded, proceeded by 10 runs that are taken into account, and an unknown further amount of runs until a measurement of sufficient statistical quality is reached, returning the arithmetic mean [3].

2. **Haskell HQP:** For the HQP Haskell implementation, we use the Criterion benchmarking library. Once again, all the preprocessing steps to convert the dataset file into HQP's `program` is done outside of any timing. The function that is actually benchmarked is the `runSimulation` function, which involves running the `evalProg` function (actual simulation of the quantum circuit), and also the retrieval of the first element of the resulting state vector.

   The latter step is done to prevent problems with Haskell's lazy evaluation and to ensure that the final result and full circuit is evaluated. As a quantum program is a sequential fold, evaluating just one value in the final state vector is enough to guarantee that all gate steps have been processed. Given that we use full normal form evaluation at the end, just retrieving this one value is computationally inexpensive, and yet allows us to gather the most accurate timing, as we do not need to traverse the full 2^n length state vector. Further type wrappers had to be applied, for example to the infinite list of random numbers, as normal form would never finish evaluating this, so we only check that that the list exists when evaluating the `InfiniteRNG` type.

   The software runs the function multiple times, and returns the arithmetic mean too [2].

3. **Qiskit:** For the Qiskit implementation, we use the `pyperf` benchmarking library. Just like with the other two implementations, the preprocessing into a `QuantumCircuit` object and then the further Qiskit Aer backend transpilation is not taken into account within the benchmark. The only function that is timed is `run_simulation` which involves simulating the circuit and also retrieving the final state vector.

   Pyperf spawns a single worker to calibrate the benchmark, and then 20 further workers that each do a warm up benchmark, discard it, and run three benchmarks each of their own. An arithmetic mean is then taken [5].

6.4 Results

Having run all the benchmarks, the empirical results can be found below:

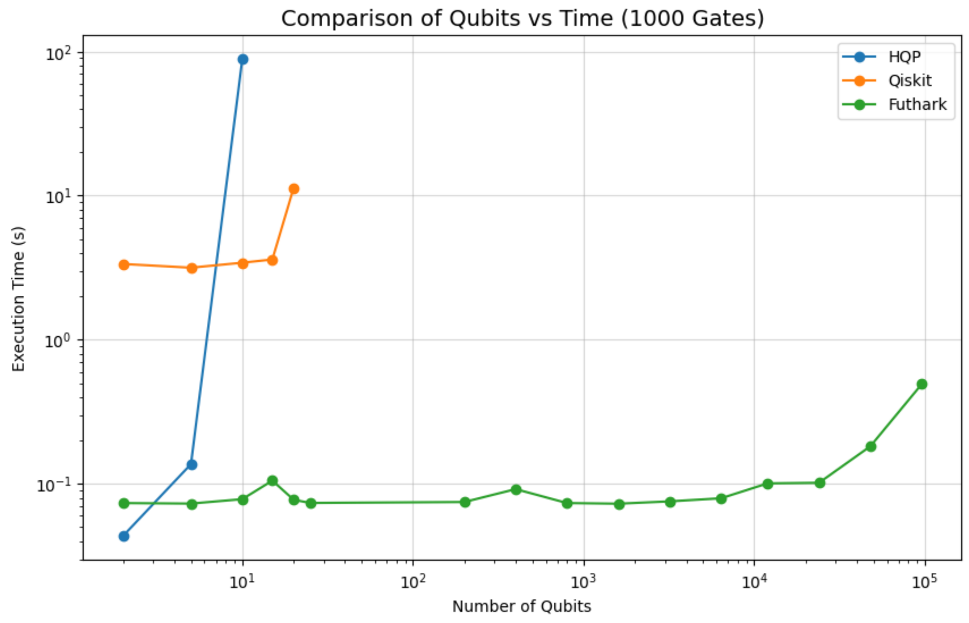


Figure 1: Graph showing the runtime against the number of qubits for all three implementations, with the number of gates set at 1000. Note that both x and y axis use log scaling.

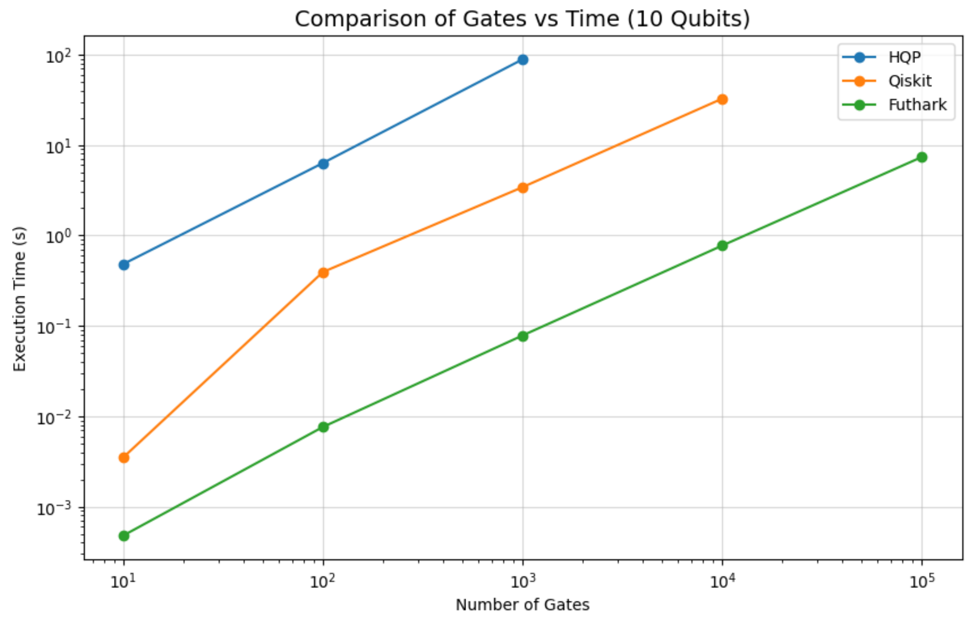


Figure 2: Graph showing the runtime against the number of gates for all three implementations, with the number of qubits set at 10. Note that both x and y axis use log scaling.

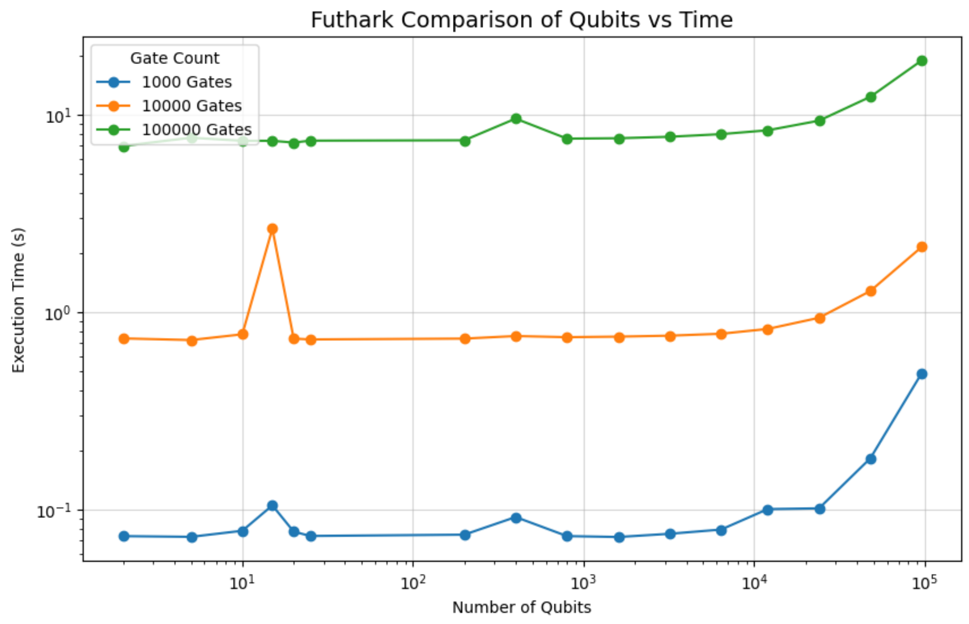


Figure 3: Graph showing the runtime against the number of qubits for the Futhark implementation for varying numbers of gates. Note that both x and y axis use log scaling.

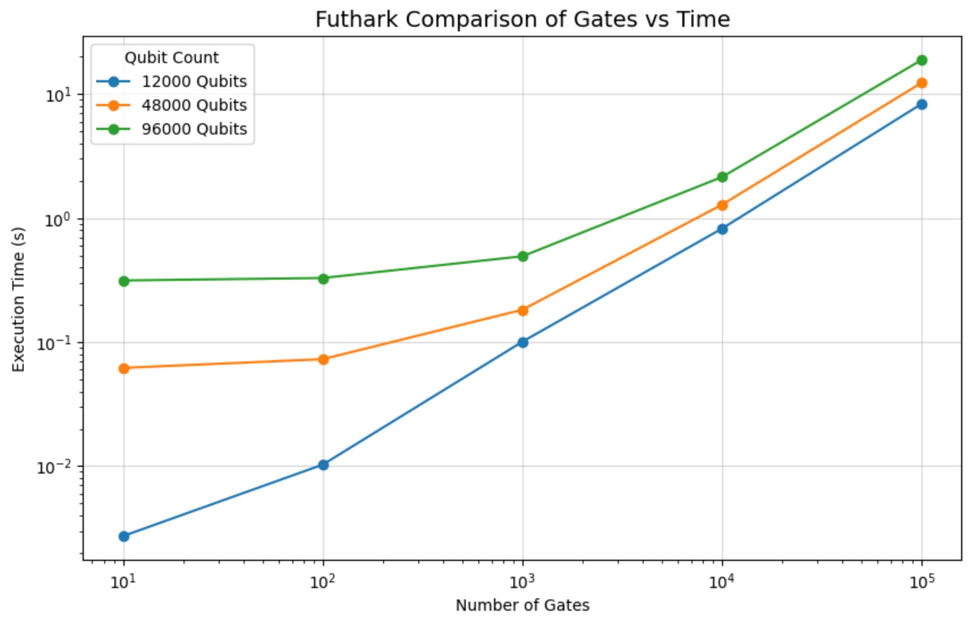


Figure 4: Graph showing the runtime against the number of gates for the Futhark implementation for varying numbers of qubits. Note that both x and y axis use log scaling.

6.5 Discussion

Comparing Futhark vs HQP vs Qiskit: As can be seen from both figure 1 and 2, the Futhark implementation outperforms both the HQP and Qiskit frameworks significantly. The reason for why there is a limited number of data points on both graphs, for both HQP and Qiskit, relative to our Futhark implementation, is because as the gate and qubit counts were increased, the runtime became so large that it was no longer feasible to continue the simulation. It was unknown as to when or if the simulation would end, and so the decision was taken to take any further benchmarks. On the other hand, our Futhark implementation scaled much better, drastically outperforming both Haskell and Qiskit, with the largest benchmark we did at 96,000 qubits and 100,000 gates, which ended up having an average runtime of about 18.9 seconds, a factor of 4.5x lower than the HQP implementation took to run at 10 qubits and 1000 gates for comparison.

Looking further at figure 1 it can be seen that the HQP implementation performs best at 2 qubits, even faster than our Futhark implementation. This is most likely because there is the least amount of startup overhead, given that the Haskell file is pre-compiled into machine-binary. On the other hand, given that both the Futhark and Qiskit implementations are GPU-accelerated, there is a constant overhead associated with the latency between moving data between the CPU and the GPU aswell as launching the kernels on the GPU itself.

As the number of qubits increases, the HQP runtime immediately begins to spike, being three orders of magnitude higher by the time 10 qubits are being simulated. It was at this point that any further benchmarks were no longer feasible. The Qiskit implementation has relatively constant scaling until 15 qubits, before it too starts experiencing spikes, and further benchmarks past 20 qubits were not feasible. It does however begin to perform better than HQP at 10 qubits, and this is is most likely associated to the GPU-acceleration that Qiskit uses, whereas HQP is CPU-bound.

Continuing on to looking at how the implementations scale with increasing the gate count, but a set number of qubits, we look to figure 2, where we can see a linear pattern across all three. This is again to be expected, as all cases run the gates in a linear fashion. Again, the same trend of HQP performing the worst continues, followed by Qiskit, and the fastest implementation being the Futhark one. Qiskit and Futhark both perform better than HQP due to the inherent acceleration provided by a GPU, while futhark also outperforms the Qiskit version.

Analysing Futhark Performance: Our Futhark performance scales very well, linearly in regards to the gate count and comparably in the qubit count.

Initially, as can be seen in figure 3, there is a clear constant scaling with the increase in number of qubits, all the way until about 12000, regardless of the number of the gates. This is the expected result from the aforementioned span of the algorithm. In our implementation section, we argued that non-measurement gates could be simulated with $O(1)$ span and the measurement could be simulated with $O(\log n)$ span. As such, this flat section of the graph is when the GPU available is not utilizing all of the cores, so more qubits simply get allocated to other cores to do the work, with a small non-constant factor likely coming from the measurement gates.

It can then be seen that around 10^3, there is an elbow point where the execution time begins to increase more drastically. Given that this sudden change happens for all gate counts, it is evident that around 10000 qubits is where GPU resources begin to run out, leading to the work asymptotics dominating the runtime.

Another observable is random spikes in runtime during the constant period, such as for 10000 gates at 15 qubits, or at 400 qubits and 100000 gates. Given that our data generation was completely random, it is possible that these results are influenced due to a probabilistic influx of measurement gates within those sequences. As these measurements have worse span and work asymptotics, this could explain the flucutations.

Finally, we can see that the scaling for the number of gates in figure 4 starts out to be sub-linear, before all qubit counts converge to be linear at about 1000 gates onwards. This behaviour again is to be expected, given that the main `simulate` function has a sequential loop within it, meaning the runtime is to be expected $O(g)$. Here we are not entirely sure why we see a sub-linear section before the more linear scaling, as we would expect the span to be inherently linear. One theory could be, that because we are using compiled input (meaning the data is compiled into the binary) futhark might be able to do some loop unrolling or other compiler improvements for the smaller gate numbers, that it does not do once the loop becomes too large.

7 Future Work

7.1 Interface Improvements

The current implementation of the interface is minimal. It executes a stabilizer circuit in one invocation of the Futhark backend and returns the resulting measurements and final tableau. While the design is sufficient for testing and benchmarking, it leaves some problems for using it to execute these stabilizer circuits.

A natural extension to resolve this is incremental execution. Currently the entire gate sequence is transferred to the backend and simulated in a single call. Since the tableau is returned, the interface could be generalized to support partial execution, where a circuit is split into segments that are executed independently. The final tableau of one segment could then be passed back as the initial tableau for the next segment. This would enable streaming execution, check pointing, and hybrid workflows where parts of the a circuit are simulated on the GPU while other are analysed or transformed on the host.
An extension to the just mentioned approach would be that the tableau returned by the backend fully encodes the stabilizer state and could, in principle, be converted back into an HQP-level representation. Such a transformation would allow the GPU simulator to act not only as an execution engine but also as an optimization or analysis pass, producing simplified stabilizer descriptions that can be reinserted into the interpreted pipeline. As mentioned earlier currently the programmer has to make a 'hyrbrid' run function, this in the future should be part of the interface and as mentioned should feel seemless to the user where they do not have to alter their HQP program to work with the Futhark backend.

Another improvement could be to change the priority of the interface from clarity and correctness to minimizing data movement. For large circuits, transferring the full tableau back to the host is costly and slows down runtime. Future version could expose finer-grained control over what data is returned, such as performing post processing directly in Futhark before read back. This would allow the interface to scale more efficiently as circuit size grows.

7.2 Circuit Optimisations - Circuit Depth

The current state of our Futhark implementation will run every single gate, regardless of whether it is redundant or not. As part of the preprocessing step, we could explore circuit-level optimisations that reduce the gate depth (reducing the number of gates executed).

Gate Cancellation: One of the primary ways in which this could be achieved is by passing over the entire gate instruction set, and performing gate cancellation. This is where we would be able to detect and remove self-inverse sequences. Some examples of these include $H_i \circ H_i = I$ or $S_i^4 = I$ or $CNOT_{i,j} \circ CNOT_{i,j} = I$. As the identity operator keeps the state equivalent, it does not need to be simulated, and so by performing these operations we effectively eliminate the gates.

Gate Reordering: A further avenue that could be explored in preprocessing is gate rearrangement. If two gates commute, which is true if they act on disjoint qubits, then the order of applying the gates does not yield a different answer. Hence, by reordering the sequence of gates, we could make it so that further gate cancellation could be achieved.

Gate Fusion: A final optimisation could be made due to the nature of only working with Clifford gates. For any single qubit, there are only 24 possible unique Clifford operations, regardless of the length of any sequence of single-qubit qubit gates. Thus, we would be able to again reduce the number of passes over the tableau by reducing the gate depth at the preprocessing step. By implementing all 24 unique operations within the `definitions.fut` file, we would be able to send any one of 24 opcodes, if it can be detected that some sequence of gates can actually be reduced to just a single operation.

7.3 Gate Batching

One of the primary performance bottlenecks identified previously is that the main `simulate` function within our Futhark implementation uses a sequential loop to process each gate. So while the implementation will still efficiently run a single gate using parallelism, consecutive gates will have to be synchronised. The reason why this is required is, that two gates could act on the same qubit. In such a case, we would need to first calculate the effect of the first gate before we can start the calculation of the next.

In order to optimise this what can be explored is gate batching. This would involve preprocessing the data in such a way that multiple gate operations can be applied to the tableau at the same time. This would significantly improve on the overall parallelism of the implementation, and would improve our theoretical span for the implementation. There are two factors however that need to be taken into consideration with gate batching.

1. The first is that because each GPU thread is operating on a single row, any operations that can be batched need to be independent of the other rows. The two single-qubit Hadamard and Phase gates have this trait, and applying them to a single qubit just means each row is updated in the exact same way. The CNOT gate however applies changes to two rows, but they are still local and independent of each other, meaning it too fits this criteria. However, the measurement involves scanning, performing calculations, and updating all rows within the tableau. Thus, you cannot batch a measurement with any other gates as you cannot be sure whether a different thread will be executing a gate update, at the same time as another thread is executing a measurement. All measurements must be within their own batch, and performed sequentially with the guarantee that the previous batch of updates has finished. Hence, the maximum size of a batch is strictly limited to the longest sequence of non-measurement operations.

2. The second is that operations that can be batched also need to be qubit-independent. The reason for this is that whilst each thread can apply multiple updates to a single row at the same time, the order of these updates matters, because most Clifford operations on the same qubit are not commutative. Hence, the gates within each batch must be acting on different qubits, so that each thread can update multiple points within each row regardless of timing. It is therefore also true that the maximum size of a batch is strictly limited to the longest sequence of independent-qubit operations.

However, this gate batching could be a significant performance improvement, especially when couple together with circuit optimisations such as gate reordering. Clifford gates allow gate re-ordering, which would transform some of the gates of the program, but would allow us to create an implementation that first maximizes the batch sizes, and then is able to launch these batches sequentially. The proposed idea would thus include a sort of program transformation before it is passed along to the futhark simulation backend.

Even further, this gate reordering could perhaps be done in futhark itself, as with a lot of gates, a lot of the data processing could be done in parallel. This is a tale as old as time, how however long one might work, there will always be a good idea left to work on.

Whilst measurements themselves are a limiting factor that cannot be improved, moving around disjoint gates between measurements such that batches could be as large as possible would certainly improve the overall parallism.

8 Conclusion

In this project, we successfully developed a high-performance Clifford circuit simulator by leveraging the stabilizer formalism and the Gottesman-Knill Theorem. By transitioning from the traditional state-vector model—which suffers from exponential complexity $O(2^n)$ - to a bit-vector representation, we reduced the space complexity to $O(n^2)$ for an n-qubit system. This transition allowed us to simulate a specific but significant subset of quantum operations with much higher efficiency on classical hardware.

Key Acheivements:

- **GPU-Parallelized Implementation:** By utilizing Futhark, we implemented parallel execution of Clifford gates (Hadamard, Phase, and CNOT) on the GPU, achieving a span of $O(1)$ for these operations.
- **Efficient Measurement Logic:** We addressed the high complexity of measurement by parallelizing the rowsum operations. In both deterministic and non-deterministic cases, our implementation achieved a span of $O(\lg n)$, significantly improving performance for high-qubit-count circuits.
- **Haskell-Futhark Integration:** We designed a robust two-stage interface where Haskell serves as the high-level orchestration and validation layer, while Futhark handles the intensive data-parallel computation via a C API.
- **Verification:** The simulator's correctness was verified through manual analysis of stabilizer states and the implementation of a quantum teleportation protocol, which yielded the expected average measurement outcome of 0.50 over multiple iterations.

Future Outlook: While our implementation demonstrates the practical implications of the Gottesman-Knill Theorem, a primary bottleneck remains in the sequential processing of the gate instruction stream, leading to a total work of $O(mn^2)$ for m gates. As discussed, future work could significantly mitigate this through gate batching, gate reordering, and circuit depth optimization (such as gate fusion and cancellation).

Ultimately, this project provides a solid foundation for a high-performance, GPU-accelerated stabilizer simulator that bridges the gap between functional high-level orchestration and low-level hardware optimization.

References

[1] Scott Aaronson and Daniel Gottesman. "Improved simulation of stabilizer circuits". In: *Physical Review A* 70.5 (Nov. 2004). ISSN: 1094-1622. DOI: 10.1103/physreva.70.052328. URL: http://dx.doi.org/10.1103/PhysRevA.70.052328.

[2] Criterion Documentation. *Criterion*. URL: https://hackage.haskell.org/package/criterion.

[3] Futhark Documentation. *Futhark Bench*. URL: https://futhark.readthedocs.io/en/latest/man/futhark-bench.html.

[4] Futhark documentation. *Regular Flatenning*. URL: https://futhark-book.readthedocs.io/en/latest/regular-flattening.html.

[5] Pyperf Documentation. *Pyperf Run A Benchmark*. URL: https://pyperf.readthedocs.io/en/latest/run_benchmark.html.

[6] Wikipedia. *Clifford Gate*. URL: https://en.wikipedia.org/wiki/Clifford_gate.

[7] Wikipedia. *Gottesman–Knill Theorem*. URL: https://en.wikipedia.org/wiki/Gottesman%E2%80%93Knill_theorem.

[8] Wikipedia. *Quantum State*. URL: https://en.wikipedia.org/wiki/Quantum_state.