Gate Fusion Is Map Fusion

Martin Elsman
DIKU
University of Copenhagen
Copenhagen, Denmark
mael@di.ku.dk

Troels Henriksen
DIKU
University of Copenhagen
Copenhagen, Denmark
athas@sigkill.dk

Abstract

Most efficient state-vector quantum simulation frameworks are imperative. They work by having circuit gates operate on the global state vector in sequence and with each gate operation accessing and updating, in parallel, all (or large subsets of) the elements of the state vector. The precise access and update patterns used by a particular gate operation depend on which qubits the individual gate operates on.

Imperative implementations of state-vector simulators, however, often lack a more declarative specification, which may hinder reasoning and optimisations. For instance, correctness is often argued for using reasoning that involves bit-operations on state-vector indexes, which make it difficult for compilers to perform high-level index-optimisations.

In this work, we demonstrate how gate operations can be understood as maps over index-transformed state-vectors. We demonstrate correctness of the approach and implement a library for gate-operations in the data-parallel programming language Futhark. We further demonstrate that Futhark's fusion-engine is sufficiently powerful that it will ensure that consecutive gate operations on identical qubits are fused using map-map fusion. Moreover, we demonstrate that, using Futhark's uniqueness type system, state vectors may be updated in place. We evaluate the approach by comparing it with the state-of-the art state-vector simulators qsim and QuEST.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; • **Theory of computation** → **Quantum computation theory**; *Denotational semantics*; • **Software and its engineering** → *Functional languages.*

Keywords: quantum simulation, parallel programming, functional programming

ACM Reference Format:
Martin Elsman and Troels Henriksen. 2025. Gate Fusion Is Map Fusion. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '25), June 17, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3736112.3736143

ARRAY '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1927-1/25/06
https://doi.org/10.1145/3736112.3736143

1 Introduction

Fusion is a well-known concept in functional and parallel programming domains, where fused code offers essential performance gains over non-fused code [17, 24, 26]. A prototypical example of fusion is map-map fusion, which allows consecutive maps over two pure functions f and g to be fused into one map operations, as specified by the equation

$$\texttt{map}\ f\ o\ \texttt{map}\ g\ =\ \texttt{map}\ (f\ o\ g)$$

The equation suggests that first mapping a function g over an array and then mapping another function f over the result of the map is semantically equivalent to mapping the composed function $f\ o\ g$ over the array.

In the realm of quantum computing, simulators are based on the notion of applying *gates* (often represented as 2×2 complex matrices) to particular *qubits*, which are entities that may be *entangled* (i.e., implicitly coupled to) other qubits in the system. Due to entanglement, the state of the system grows exponentially with the number of qubits n in the system and may be modeled as a 2^n complex vector. As we shall see, applying a gate to a set of qubits in such a complex vector results in a complex access and update pattern, which may potentially affect all entries in the state vector.

Existing state-vector simulators, such as QuEST [22] and qsim [39], implement gate operations by operating imperatively on state vector entries identified by indices that are established using bit-level operations. Traditionally, compilers have difficulties analysing addresses established from bit-level operations, which makes it difficult for compilers to perform optimisations that, for instance, rely on the property that a gate operation uses an accesses pattern following a permutation of the state vector index space (e.g., parallelisation).

As an alternative to reasoning about gate access patterns using bit-level operations, we demonstrate how gate operations can be understood as maps over index-permuted state vectors in a data-parallel array language. The approach is based on what it means to apply a gate to a state vector (i.e., applying a large unitary matrix established from a series of tensor products to the state vector), from which efficient gate implementations are derived. Moreover, we establish that

gate-fusion, a notion developed for boosting quantum simulation performance, by merging consecutive gate operations [11], can be implemented using map-map fusion.

The development is performed in the context of the data-parallel array language Futhark [10, 18], which allows for being precise about array sizes, using a simple notion of dependent size types [3, 16]. The approach carries over, however, to any other functional array language.

The contributions are as follows:

1. We demonstrate how gate operations can be understood as maps over index-permuted state-vectors.
2. We present a Futhark library dqfut for writing quantum algorithms. The library may be used either as a library for end-users to write quantum algorithms or as a target language for higher-level quantum languages.
3. We demonstrate that Futhark's fusion-engine is sufficiently powerful that it fuses consecutive gate operations on identical qubits using map-map fusion.
4. Using Futhark's uniqueness type system, we demonstrate that state vector transformations (i.e., gate operations) are implemented in-place.
5. We show that dqfut in some cases is competitive to other quantum state-vector simulators and that simulation performance benefits from gate-fusion. Moreover, the experiments identify a set of weaknesses in Futhark's array indexing manipulations, which we consider future work to resolve.

In the following section, we introduce a small statement language for performing gate operations on a set of qubits. The language may be used for expressing the semantic operations of a quantum circuit but is more low-level and resembles languages such as cQUASM [25], OpenQUASM [7], the QuEST gate API [22], or the qsim gate language [39]. In Section 3, we derive a data-parallel interpretation scheme for gate statements by calculating how the semantics of statements corresponds to data-parallel operations over a state vector. Derivations are made for single-qubit gate operations, two-qubit swap gates, and so-called multi-controlled gates. The derivations lead to a set of Futhark functions to be used either by an end programmer or to be used as a backend for a higher-level quantum programming language. The section also demonstrates the derivation of a general swap-operation that serves to swap two arbitrary qubits. In Section 4, we demonstrate how gate fusion can be understood as map-map fusion, by showing the validity of the gate fusion rules using the properties of map-map fusion, transposition, and array flattening. In Section 5, we introduce the dqfut Futhark gate library and in Section 6, we provide benchmarks by comparing the performance of dqfut against QuEST [22] and qsim [39], under GPU execution, multi-core execution, and single-core execution. In Section 7, we present related work and in Section 8, we conclude and outline future work.

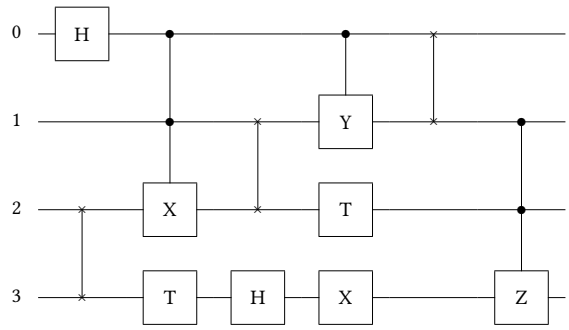

Figure 1. A random circuit. The gates `T`, `H`, and `X` on qubit 3 are subject to gate fusion.

2 Gate Semantics and Circuits

Qubits are referenced using positive integers (i.e., greater than or equal to 0) and are ranged over by q.

Gates and statements take the following form:

$$
\begin{array}{lcl}
g & ::= & \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \mid \texttt{H} \mid \texttt{S} \mid \texttt{T} \\
s & ::= & \texttt{gate}\ g\ q \mid \texttt{swap}\ q \mid \texttt{cntrl}\ n\ g\ q \\
 & \mid & s\ \texttt{;}\ s \mid \texttt{nop}
\end{array}
$$

Executing a statement s can have an effect on the entire underlying state, which may be the case even if the statement operates only on a single qubit. Whereas statements of the form `gate` g q have the effect of applying the gate g on qubit q, a statement `swap` q has the effect of swapping the two neighboring qubits q and $q+1$. Statements of the form `cntrl` n g q, where $n > 0$, have the effect of operating with the gate g on qubit $q{+}n$, provided the amplitudes of the qubits $q, \ldots, q+n-1$ are all 1. Finally, we support `nop` statements, which have no effect on the underlying state and which are useful for expressing various statement optimisations.

It is custom for quantum programmers to specify quantum programs using so-called circuit diagrams. In diagram notation, each qubit, in an n-qubit system, is represented as a horizontal line, numbered from 0 to $n-1$, single-gate operations are drawn as boxes, and swaps are drawn by connecting two lines with crosses. Multi-controlled gates are also drawn as boxes, but they are connected to the qubits that control the gates with vertical lines and bullets. An example random circuit is shown in Figure 1. It may be expressed by the following statement:

```
gate H 0 ; swap 2 ; cntrl 2 X 0 ; gate T 3 ;
swap 1 ; gate H 3 ; cntrl 1 Y 0 ; gate T 2 ; gate X 3 ;
swap 0 ; cntrl 2 Z 1
```

2.1 Qubits and Unitary Matrices

The basic building block of a quantum computer is a *qubit*, which can be modeled as a two-dimensional complex vector

$\begin{bmatrix}\alpha & \beta\end{bmatrix}^{\mathrm{T}}$ specifying a linear combination $\alpha|0\rangle + \beta|1\rangle$ of the basis vectors $|0\rangle$ and $|1\rangle$ such that $|\alpha|^2 + |\beta|^2 = 1$ (the *ket* $|0\rangle$ is defined as $\begin{bmatrix}1 & 0\end{bmatrix}^{\mathrm{T}}$ and $|1\rangle$ is defined as $\begin{bmatrix}0 & 1\end{bmatrix}^{\mathrm{T}}$).

A *single-qubit gate* operates on a single qubit and can be represented as a 2×2 *unitary* complex matrix U, meaning $U^\dagger U = UU^\dagger = I$ (norm-preserving and reversible).[1] Single-qubit gates include the *Pauli-gates*, X, Y, and Z, the *Hadamard gate* H, the T-gate, and the identity gate I.

$$X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \qquad Y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix} \qquad Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}$$
$$H = \tfrac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \qquad T = \begin{bmatrix}1 & 0\\ 0 & e^{i\pi/4}\end{bmatrix} \qquad I = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$$

The effect of "applying" the Hadamard gate H to a qubit in the $|0\rangle$ state puts the qubit in the *superposition* state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, which is the result of applying the matrix H to the vector $\begin{bmatrix}1 & 0\end{bmatrix}^{\mathrm{T}}$. Because H is its own inverse (i.e., it is *hermitian*), it follows that the effect of applying another H gate to the modified qubit puts the qubit back into the state $|0\rangle$.

A state of more qubits is modeled as a tensor product of the individual standard bases. Thus, a two-qubit state can expressed as a four-element complex vector $\begin{bmatrix}\alpha & \beta & \gamma & \theta\end{bmatrix}^{\mathrm{T}}$ that denotes a linear combination $\alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \theta|11\rangle$ of all combinations of basis vectors for the individual qubits, where $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\theta|^2 = 1$ (we write $|ab\rangle = |a\rangle \otimes |b\rangle$, for $a, b \in \{0, 1\}$, where $\otimes$ denotes tensor-product). The state space grows exponentially. For a three-qubit system, the state is expressed as an 8-element complex vector.

Single-qubit gates may be applied to particular qubits in sequence. To understand how an application of a single-qubit gate to a qubit affects the state vector, we first observe that the application happens through a series of tensor-products.

When A is an $m \times n$ complex matrix with elements a_{ij}, where $i \in [1; m]$ and $j \in [1; n]$, and B is a $p \times q$ complex matrix, the *tensor product* of A and B, written $A \otimes B$, is the $mp \times nq$ complex matrix

$$A \otimes B = \begin{bmatrix}a_{11}B & \cdots & a_{1n}B\\ \vdots & \ddots & \vdots\\ a_{m1}B & \cdots & a_{mn}B\end{bmatrix} \tag{1}$$

By treating a complex scalar as the 1×1 complex matrix containing the scalar, we have $1 \otimes A = A \otimes 1 = A$, for any complex matrix A. We also note here that $\otimes$ is associative and that if A and B are unitary complex matrices then $A \otimes B$ is also a unitary complex matrix. We further write I_n to denote 1 if $n = 0$ or $I \otimes I_{n-1}$ if $n > 0$.

Now, applying a single-qubit gate U (i.e., a unitary complex matrix) to qubit q of a k-qubit state vector v (of size 2^k), we are applying the unitary matrix $I_q \otimes U \otimes I_{k-q-1}$ to v. Similarly, applying a 2-qubit gate U to a state vector v (of size 2^k), we are applying the unitary matrix $I_q \otimes U \otimes I_{k-q-2}$ to v. An example 2-qubit gate is the 4×4 *SW* gate, which swaps two neighboring qubits:

$$SW = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

Another 2-qubit gate is the *controlled not* gate *CNOT*, which takes the following form:

$$CNOT = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}$$

It turns out that it is possible to define a *generalised control operator*, parameterised over an arbitrary 1-qubit gate U, written $C_n\, U$, which equals U if $n = 0$ and, provided $n > 0$, equals the block matrix on the form

$$C_n\, U = \begin{bmatrix}I_n & 0\\ 0 & C_{n-1}\, U\end{bmatrix}$$

With the gates we support here (more can be added easily), it is only the controlled gates that may introduce entanglement between qubits. Also notice that $C_1\, X = CNOT$ and that $C_2\, X$ is the controlled *CNOT* gate, also called the Tofolli gate [4].

2.2 Semantics

Whereas a single-qubit gate g is denoted, written $[\![g]\!]$, by its corresponding 2×2 unitary matrix, statements are denoted by a function $[\![\cdot]\!]_k : \mathbb{C}^{2^k \times 2^k}$, where k specifies the number of qubits in the system:

$$\begin{array}{lll}
[\![\texttt{gate}\ g\ q]\!]_k & = I_q \otimes [\![g]\!] \otimes I_{k-q-1} & (k > q)\\
[\![\texttt{swap}\ q]\!]_k & = I_q \otimes SW \otimes I_{k-q-2} & (k > q+1)\\
[\![\texttt{cntrl}\ n\ g\ q]\!]_k & = I_q \otimes C_n\,[\![g]\!] \otimes I_{k-q-n-1} & (k > q+n)\\
[\![s_1; s_2]\!]_k & = [\![s_2]\!]_k [\![s_1]\!]_k &\\
[\![\texttt{nop}]\!]_k & = I_k &
\end{array}$$

When s is some statement, we say that s is k-well-formed if $k > 0$ and $[\![s]\!]_k$ is defined (e.g., all side conditions in the semantics derivation hold).

A series of properties hold. First, notice that for all gates g and for all $k > q$, we have $[\![\texttt{cntrl}\ 0\ g\ q]\!]_k = [\![\texttt{gate}\ g\ q]\!]_k$. It is also straightforward to check that if $[\![s]\!]_k = U$ then $U : \mathbb{C}^{2^k \times 2^k}$.

3 Data-Parallel Interpretation

We now present an interpretation scheme that turns gate statements into data-parallel operations. The scheme can be used either for offline compilation or for interpreting gate operations directly as data-parallel operations.

[1] We use the notation $U^\dagger$ for the conjugated transpose of the unitary complex matrix U.

3.1 MiniFut

The underlying language that we rely on is a data-parallel array-language supporting regular-nested arrays, a set of data-parallel operations such as `transpose`, `flatten`, and `unflatten`, and a set of second-order array-combinators, such as `map`. Concretely, we base the development on MiniFut, a small subset of Futhark, a data-parallel language aimed for specifying data-parallel computations that, for instance, may be executed on GPUs. As Futhark, MiniFut supports the notion of size-types [3], which allows for programmers to express invariants about array sizes.

We assume denumerably sets of type variables, ranged over by α, and (program) variables, ranged over by x, n, and f. We use *bop* to range over traditional infix binary operations such as +, −, ∗, and ∗∗. MiniFut types (τ), expressions (e), and programs (p) take the following forms:

$$\begin{array}{lcl} \tau & ::= & \mathtt{c} \mid \mathtt{i} \mid \alpha \mid [e]\tau \mid (x:\tau_1) \to \tau_2 \\ e & ::= & c \mid i \mid x_\tau \mid f_\tau\, e_1 \cdots e_n \mid e \triangleright \tau \mid e_1\ bop\ e_2 \\ & \mid & \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid \mathtt{let}\ f\ x_1 \cdots x_n = e_1\ \mathtt{in}\ e_2 \\ p & ::= & \mathtt{def}\ f\ [n_1] \cdots [n_m]\ (x_1:\tau_1) \cdots (x_n:\tau_n) : \tau = e \\ & \mid & p_1\ p_2 \end{array}$$

A type τ is either the type of complex numbers `c`, the type of integers `i`, a type variable α, the type $[e]\tau$ of arrays of size e containing elements of type τ, or the type $(x:\tau_1) \to \tau_2$ of (dependent) functions, where we consider x bound in τ_2. We often write $\tau_1 \to \tau_2$ instead of $(x:\tau_1) \to \tau_2$ when x does not occur free in τ_2 and we consider types identical up to renaming of bound variables.

Expressions are either immediate constants (complex numbers or integers), variables x_τ, function applications $f_\tau\, e_1 \cdots e_n$, size-type casts $e \triangleright \tau$, binary operations $e_1\ bop\ e_2$, let-bindings, and local function bindings. We sometimes leave out the annotated types on expressions of the form x_τ and $f_\tau\, e_1 \cdots e_n$, when the variable type instantiations are clear from the context. Types, which may contain size-expressions, are considered equal only up to renaming of bound variables, which means, for instance, that the two types `[n * m]f64` and `[m * n]f64` are not considered equal. A size-type cast may be used, for instance, to type cast an expression e of type `[n]f64` into an expression of type `[m]f64`, which involves a runtime check to ensure that the runtime value of `m` equals the runtime value of `n`.

We assume a library of utility functions equipped with the following types:

$$\begin{array}{rcl} \mathtt{flatten} & : & \forall mn\alpha.[m][n]\alpha \to [m*n]\alpha \\ \mathtt{unflatten} & : & \forall mn\alpha.[m*n]\alpha \to [m][n]\alpha \\ \mathtt{transpose} & : & \forall mn\alpha.[m][n]\alpha \to [n][m]\alpha \\ \mathtt{map} & : & \forall n\alpha\beta.(\alpha \to \beta) \to [n]\alpha \to [n]\beta \end{array}$$

We see that `map` returns an array of the same size as the array it takes as argument. We also see that `transpose` interchanges the two outer dimensions of an argument array and that `flatten` and `unflatten` operations are guided by the types of the arguments.

Programs p allow for defining top-level functions that are polymorphic both in types (implicitly abstracting over free type variables) and in sizes (explicitly abstracting over program variables).

We define MiniFut variations of `flatten` and `unflatten` that operate on columns instead of rows, which eases some of the reasoning in the next section:

```
def vec [m][n] (x:[m][n]α) : [n*m]α =
  flatten(transpose x)
def unvec [m][n] (x:[m*n]α) : [n][m]α =
  transpose(unflatten x)
```

We note here that in Futhark, index operations such as `flatten`, `unflatten`, and `transpose` most often do not lead to allocation of new arrays as the operations are fused into operations, such as `map`, that consume values and return fresh values in recently consumed memory. Using properties of `map`, `transpose`, `flatten`, and `unflatten`, Futhark may effectively update such arrays in-place.

3.2 Deriving an Interpreter

We derive the functional data-parallel interpretation directly from the semantics by building on a relation between the mathematical definition of matrix multiplication and a functional implementation. In particular, from the definition of matrix multiplication, we observe the following equality, if we allow for A to be treated both as a matrix and as a function taking a column vector and returning a column vector:

$$AB \quad = \quad (\mathtt{map}\ A\ B^{\mathrm{T}})^{\mathrm{T}} \tag{2}$$

We shall further base the derivation on the well-known *vectorisation trick* [27], which establishes a relationship between the Kronecker tensor-product and matrix multiplication. Assuming $V = \mathtt{unvec}\ v$ and assuming V is appropriately compatible with A and B, we have

$$(A \otimes B)\ v \quad = \quad \mathtt{vec}(BVA^{\mathrm{T}}) \tag{3}$$

By combining Equations 3 and 2 and by utilising the property that $(AB)^{\mathrm{T}} = B^{\mathrm{T}}A^{\mathrm{T}}$, we obtain the following properties:

$$\begin{array}{rcll} (I_n \otimes B)\ v & = & \mathtt{vec}(BV) & (4) \\ & = & \mathtt{vec}((\mathtt{map}\ B\ V^{\mathrm{T}})^{\mathrm{T}}) & \\ (A \otimes I_n)\ v & = & \mathtt{vec}(VA^{\mathrm{T}}) & (5) \\ & = & \mathtt{vec}((AV^{\mathrm{T}})^{\mathrm{T}}) & \\ & = & \mathtt{vec}(\mathtt{map}\ A\ V) & \end{array}$$

By specialising to tensor products of square matrices we get the following more precise propositions:

Proposition 3.1 (Tensor-Id-Left). *For any positive n and m and for any B, v, and τ such that $B : \mathbb{C}^{2^m \times 2^m}$ and $v : [2^n * 2^m]\mathtt{c}$*

and $\tau = [2^n][2^m]\mathsf{c}$*, we have*

$$(I_n \otimes B)\ v \quad = \quad \texttt{flatten}(\texttt{map}\ B\ (\texttt{unflatten}_\tau\ v))$$

Proposition 3.2 (Tensor-Id-Right)**.** *For any positive n and m and for any A, v, and* τ *such that* $A : \mathbb{C}^{2^m \times 2^m}$ *and* $v : [2^m * 2^n]\mathsf{c}$ *and* $\tau = [2^n][2^m]\mathsf{c}$*, we have*

$$(A \otimes I_n)\ v \quad = \quad \texttt{vec}(\texttt{map}\ A\ (\texttt{unvec}_\tau\ v))$$

For a system of k qubits, the interpreter that we derive takes further a statement s and a complex state vector of size 2^k. We shall derive the interpreter $\mathcal{I}\,[s]_k\ v$ inductively, by utilising the following connection between the interpreter and the semantics, which says that for any statement s, integer $k > 0$, and state vector v of size 2^k, we have

$$\mathcal{I}\,[s]_k\ v \quad = \quad [\![s]\!]_k\ v \tag{6}$$

Here the juxtaposition of $[\![s]\!]_k$ and v to the right denotes matrix-vector multiplication.

We now derive the interpreter inductively over the structure of statements.

For `gate` statements, we have

$$\begin{aligned}
&\mathcal{I}\,[\texttt{gate}\ g\ q]_k\ v \\
&\quad = \quad ((I_q \otimes g) \otimes I_{k-q-1})\ v \\
&\quad = \quad \texttt{vec}(\texttt{map}\ (I_q \otimes g)\ (\texttt{unvec}\ v)) \\
&\quad = \quad \texttt{let}\ F\ u\ =\ \texttt{flatten}(\texttt{map}\ g\ (\texttt{unflatten}\ u)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ F\ (\texttt{unvec}\ v))
\end{aligned}$$

The last two steps utilise Proposition 3.2 and Proposition 3.1, respectively. Adding size types, we get the following:

$$\begin{aligned}
&\mathcal{I}\,[\texttt{gate}\ g\ q]_k\ (v : [2^k]\mathsf{c}) \\
&\quad = \quad \texttt{let}\ v\ =\ v \triangleright [(2^q * 2) * 2^{k-q-1}]\mathsf{c} \\
&\qquad\ \ \texttt{let}\ F\ u\ =\ \texttt{flatten}(\texttt{map}\ g\ (\texttt{unflatten}\ u)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ F\ (\texttt{unvec}\ v)) \triangleright [2^k]\mathsf{c}
\end{aligned}$$

Much similar to `gate` statements, for `swap` statements, we have

$$\begin{aligned}
&\mathcal{I}\,[\texttt{swap}\ q]_k\ v \\
&\quad = \quad ((I_q \otimes SW) \otimes I_{k-q-2})\ v \\
&\quad = \quad \texttt{vec}(\texttt{map}\ (I_q \otimes SW)\ (\texttt{unvec}\ v)) \\
&\quad = \quad \texttt{let}\ F\ u\ =\ \texttt{flatten}(\texttt{map}\ SW\ (\texttt{unflatten}\ u)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ F\ (\texttt{unvec}\ v))
\end{aligned}$$

Again, adding size types, we get:

$$\begin{aligned}
&\mathcal{I}\,[\texttt{swap}\ q]_k\ (v : [2^k]\mathsf{c}) \\
&\quad = \quad \texttt{let}\ v\ =\ v \triangleright [(2^q * 4) * 2^{k-q-2}]\mathsf{c} \\
&\qquad\ \ \texttt{let}\ F\ u\ =\ \texttt{flatten}(\texttt{map}\ SW\ (\texttt{unflatten}\ u)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ F\ (\texttt{unvec}\ v)) \triangleright [2^k]\mathsf{c}
\end{aligned}$$

For controlled gates, we have

$$\begin{aligned}
&\mathcal{I}\,[\texttt{cntrl}\ n\ g\ q]_k\ v \\
&\quad = \quad ((I_q \otimes C_n\ g) \otimes I_{k-q-n-1})\ v \\
&\quad = \quad \texttt{vec}(\texttt{map}\ (I_q \otimes C_n\ g)\ (\texttt{unvec}\ v)) \\
&\quad = \quad \texttt{let}\ F\ u\ =\ \texttt{flatten}(\texttt{map}\ (C_n\ g)\ (\texttt{unflatten}\ u)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ F\ (\texttt{unvec}\ v))
\end{aligned}$$

Adding size types, we get:

$$\begin{aligned}
&\mathcal{I}\,[\texttt{cntrl}\ n\ g\ q]_k\ (v : [2^k]\mathsf{c}) \\
&\quad = \quad \texttt{let}\ v\ =\ v \triangleright [(2^q * 2^{n+1}) * 2^{k-q-n-1}]\mathsf{c} \\
&\qquad\ \ \texttt{let}\ F\ u\ =\ \texttt{flatten}(\texttt{map}\ (C_n\ g)\ (\texttt{unflatten}\ u)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ F\ (\texttt{unvec}\ v)) \triangleright [2^k]\mathsf{c}
\end{aligned}$$

For sequential composition, using induction, we derive the following:

$$\begin{aligned}
\mathcal{I}\,[s_1; s_2]_k\ v \quad &= \quad ([\![s_2]\!]_k [\![s_1]\!]_k)\ v \\
&= \quad [\![s_2]\!]_k\ ([\![s_1]\!]_k\ v) \\
&= \quad [\![s_2]\!]_k\ (\mathcal{I}\,[s_1]_k\ v) \\
&= \quad \mathcal{I}\,[s_2]_k\ (\mathcal{I}\,[s_1]_k\ v)
\end{aligned}$$

Directly based on the above calculations, Figure 2 lists gate functions written as a Futhark library. Whereas Futhark syntax is close to MiniFut syntax, we see that Futhark supports (size-parameterised) type abbreviations and the possibility that function argument types and return types are annotated with *uniqueness* information (i.e., a star $*$). When an argument type is annotated with uniqueness information, a caller will ensure that the argument does not alias other values. Further, when a return type is annotated with uniqueness information, a caller will know that the result can alias only arguments that are marked as unique (and Futhark will check, for each function, that this property holds or a static error is signaled). Uniqueness annotations allow for Futhark to implement array updates in-place [18], using the construct e `with [`i`] =` e', which evaluates to the array resulting from evaluating e with the i'th index updated to hold the result of evaluating e'; in principle, the type of this function is `*[n]`$\alpha \to$ `i64` $\to \alpha \to$ `*[n]`α.

3.3 Swapping Arbitrary Qubits

A more general swap statement `swap` $q\ r$ swaps two arbitrary qubits q and r. Assuming $r > q$, such a statement can be implemented by a recursive V-shaped sequence (of length $2(r-q)-1$) of binary swap gates. We first define the parameterised *swap-layer operator* L^k_n, which creates a unitary matrix, of size $2^{n+1} \times 2^{n+1}$, for transforming $n > 1$ qubits by swapping qubits $k-1$ and k, where $0 < k \leq n$:

$$L^k_n \quad = \quad I_{k-1} \otimes SW \otimes I_{n-k} \tag{7}$$

For example, the layer L^2_2 equals $I \otimes SW$ and the layer L^2_3 equals $I \otimes SW \otimes I$.

We further define two parameterised *swap-sequence operators* Up^k_n and $Down^k_n$, for creating sequences of layers that either starts swapping qubits from the bottom, going up, or starts swapping from the top, going down:

```
type st[n] = [2**n]c
type gate = c → c → (c,c)
type q = i64

def gate [n] (g:gate) (q:q)
             (v:*st[n]) : *st[n] =
  let v = v :> *[(2**q*2)*2**(n-q-1)]c
  let g p = let (x,y) = g p[0] p[1] in [x,y]
  let f u = flatten (map g (unflatten u))
  in vec(map f (unvec v)) :> *st[n]

def swap [n] (q:q) (v:*st[n]) : *st[n] =
  let v = v :> *[(2**q*4)*2**(n-q-2)]c
  let g p = [p[0],p[2],p[1],p[3]]
  let f u = flatten (map g (unflatten u))
  in vec(map f (unvec v)) :> *st[n]

def cntrl [n] (c:i64) (g:gate) (q:q)
              (v:*st[n]) : *st[n] =
  let v = v :> *[(2**q*2**(c+1))*2**(n-q-c-1)]c
  let g p = let i = 2**(c+1)-2
            let (x,y) = g p[i] p[i+1]
            in p with [i] = x with [i+1] = y
  let f u = flatten (map g (unflatten u))
  in vec(map f (unvec v)) :> *st[n]
```

Figure 2. Implementations of the basic Futhark gate functions.

$$Down_n^k = \begin{cases} I_{n+1} & \text{if } k = 0 \\ L_n^k Down_n^{k-1} & \text{otherwise} \end{cases} \quad (8)$$

$$Up_n^k = \begin{cases} I_{n+1} & \text{if } k = 0 \\ L_n^{n-k+1} Up_n^{k-1} & \text{otherwise} \end{cases} \quad (9)$$

We note here that sequence composition is opposite than matrix-multiplication, which means that the first layer in a down-sequence, for instance, is a layer of the form L_n^1 for some n.

Similarly as for layers, down-sequences and up-sequences of the forms $Down_n^k$ and Up_n^k denote unitary matrices of size $2^{n+1} \times 2^{n+1}$.

We can now define a parameterised swap operator Sw_n for swapping qubits that are $n \geq 1$ qubits apart by combining an up-sequence with a down-sequence, as illustrated in Figure 3:

$$Sw_n = Down_n^n Up_n^{n-1} \quad (10)$$

The following propositions hold:

Proposition 3.3 (Down-layer). *For any $n \geq 1$ and k such that $0 \leq k < n$, we have $\mathrm{Down}_n^k = \mathrm{Down}_k^k \otimes I_{n-k}$.*

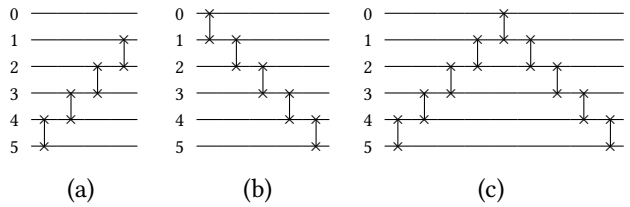

Figure 3. Swap circuits for (a) an Up_5^4 sequence, (b) a $Down_5^5$ sequence, and (c) a Sw_5 sequence.

Proposition 3.4 (Up-layer). *For any $n \geq 1$ and k such that $0 \leq k < n$, we have $\mathrm{Up}_n^k = I_{n-k} \otimes \mathrm{Up}_k^k$.*

Proposition 3.5 (Down-seq). *For any $n \geq 1$ and state vector v of type $[2^{n+1}]\mathsf{c}$, we have $\mathrm{Down}_n^n\ v = \mathtt{flatten}(\mathtt{unvec}_\tau\ v)$, where $\tau = [2^n][2]\mathsf{c}$.*

Proposition 3.6 (Up-seq). *For any $n \geq 1$ and state vector v of type $[2^{n+1}]\mathsf{c}$, we have $\mathrm{Up}_n^n\ v = \mathtt{flatten}(\mathtt{unvec}_\tau\ v)$, where $\tau = [2][2^n]\mathsf{c}$.*

Based on the above properties and Proposition 3.1, we have

$$\begin{aligned} Sw_n\ v &= Down_n^n(I \otimes Up_{n-1}^{n-1})\ v \quad (11) \\ &= \mathtt{let}\ up\ u\ =\ \mathtt{flatten}\ (\mathtt{unvec}_{\tau'}\ u) \\ &\quad\ \mathtt{let}\ dn\ u\ =\ \mathtt{flatten}\ (\mathtt{unvec}_\tau\ u) \\ &\quad\ \mathtt{in}\ dn(\mathtt{flatten}(\mathtt{map}\ up\ (\mathtt{unflatten}_\tau\ v))) \end{aligned}$$

where $\tau = [2^n][2]\mathsf{c}$ and $\tau' = [2][2^{n-1}]\mathsf{c}$.

We can calculate a map-nest for an operator that swaps two arbitrary qubits q and r, provided that $n = r - q > 0$ and $\tau = [2^{k-r-1}][2^{r+1}]\mathsf{c}$ and $\tau' = [2^q][2^{n+1}]\mathsf{c}$ and $\tau_1 = [2^n][2]\mathsf{c}$ and $\tau_2 = [2][2^{n-1}]\mathsf{c}$:

$$\begin{aligned} &\mathcal{I}\,[\mathtt{swap2}\ q\ r]_k\ (v : [2^k]\mathsf{c}) \\ &= ((I_q \otimes Sw_{r-q}) \otimes I_{k-q-(r-q)-1})\ v \\ &= ((I_q \otimes Sw_{r-q}) \otimes I_{k-r-1})\ v \\ &= \mathtt{let}\ n\ =\ r-q \\ &\quad\ \mathtt{in}\ \mathtt{vec}(\mathtt{map}\ (I_q \otimes Sw_n)\ (\mathtt{unvec}_\tau\ v)) \\ &= \mathtt{let}\ n\ =\ r-q \\ &\quad\ \mathtt{let}\ f\ u\ =\ (I_q \otimes Sw_n)\ u \\ &\quad\ \mathtt{in}\ \mathtt{vec}(\mathtt{map}\ f\ (\mathtt{unvec}_\tau\ v)) \\ &= \mathtt{let}\ n\ =\ r-q \\ &\quad\ \mathtt{let}\ f\ u\ =\ \mathtt{flatten}(\mathtt{map}\ Sw_n\ (\mathtt{unflatten}_{\tau'}\ u)) \\ &\quad\ \mathtt{in}\ \mathtt{vec}(\mathtt{map}\ f\ (\mathtt{unvec}_\tau\ v)) \\ &= \mathtt{let}\ n\ =\ r-q \\ &\quad\ \mathtt{let}\ sw\ u\ =\ SW_n\ u \\ &\quad\ \mathtt{let}\ f\ u\ =\ \mathtt{flatten}(\mathtt{map}\ sw\ (\mathtt{unflatten}_{\tau'}\ u)) \\ &\quad\ \mathtt{in}\ \mathtt{vec}(\mathtt{map}\ f\ (\mathtt{unvec}_\tau\ v)) \end{aligned}$$

```
def swap2 [k] (q:q)(r:q)(v:*st[k]) : *st[k] =
  let n = r-q
  let v = v :> *[(2**q*2**(n+1))*2**(k-r-1)]c
  let up (u:*[2**(n-1)*2]c) : *[2*2**(n-1)]c =
    flatten (unvec u)
  let dn (u:*[2*2**n]c) : *[2**n*2]c =
    flatten (unvec u)
  let sw (u:*[2**(n+1)]c) =
    let u :> *[2*(2**(n-1)*2)]c = u
    let w : *[2*(2*2**(n-1))]c =
      flatten(umap up (unflatten u))
    let w :> *[2*2**n]c = w
    in dn w :> *[2**(n+1)]c
  let f (u:*[2**q*2**(n+1)]c) =
    flatten (umap sw (unflatten u))
  in vec(umap f (unvec v)) :> *st[k]
```

Figure 4. Futhark code for a generalised swap function.

$$
\begin{aligned}
= \;& \texttt{let}\ n = r - q \\
& \texttt{let}\ up\ u = \texttt{flatten}\ (\texttt{unvec}_{\tau_2}\ u) \\
& \texttt{let}\ dn\ u = \texttt{flatten}\ (\texttt{unvec}_{\tau_1}\ u) \\
& \texttt{let}\ sw\ u = dn(\texttt{flatten}(\texttt{map}\ up\ (\texttt{unflatten}_{\tau_1}\ u))) \\
& \texttt{let}\ f\ u = \texttt{flatten}(\texttt{map}\ sw\ (\texttt{unflatten}_{\tau'}\ u)) \\
& \texttt{in vec}(\texttt{map}\ f\ (\texttt{unvec}_{\tau}\ v))
\end{aligned}
$$

Futhark code for the generalised swap operator is given in Figure 4.

An alternative specification of the swap2 statement specifies the statement in terms of two other statements, one that does an up-sweep and one that does a down-sweep:

$$\texttt{swap2}\ q\ r \quad = \quad \texttt{up}\ q\ r\ ;\ \texttt{down}\ (q+1)\ r \tag{12}$$

We can derive an implementation for up as follows, where $\tau = [2^{k-r-1}][2^{r+1}]\texttt{c}$ and $\tau' = [2^q][2^{n+1}]\texttt{c}$ and $\tau'' = [2][2^n]\texttt{c}$:

$$
\begin{aligned}
& \mathcal{I}\,[\![\texttt{up}\ q\ r]\!]_k\ (v : [2^k]\texttt{c}) \\
& \quad = \quad ((I_q \otimes Up^{r-q}_{r-q}) \otimes I_{k-q-(r-q)-1})\ v \\
& \quad = \quad ((I_q \otimes Up^{r-q}_{r-q}) \otimes I_{k-r-1})\ v \\
& \quad = \quad \texttt{let}\ n = r - q \\
& \qquad\ \ \texttt{in vec}(\texttt{map}\ (I_q \otimes Up^n_n)\ (\texttt{unvec}_{\tau}\ v)) \\
& \quad = \quad \texttt{let}\ n = r - q \\
& \qquad\ \ \texttt{let}\ f\ u = (I_q \otimes Up^n_n)\ u \\
& \qquad\ \ \texttt{in vec}(\texttt{map}\ f\ (\texttt{unvec}_{\tau}\ v)) \\
& \quad = \quad \texttt{let}\ n = r - q \\
& \qquad\ \ \texttt{let}\ f\ u = \texttt{flatten}(\texttt{map}\ Up^n_n\ (\texttt{unflatten}_{\tau'}\ u)) \\
& \qquad\ \ \texttt{in vec}(\texttt{map}\ f\ (\texttt{unvec}_{\tau}\ v)) \\
& \quad = \quad \texttt{let}\ n = r - q \\
& \qquad\ \ \texttt{let}\ up\ u = \texttt{flatten}(\texttt{unvec}_{\tau''}\ u) \\
& \qquad\ \ \texttt{let}\ f\ u = \texttt{flatten}(\texttt{map}\ up\ (\texttt{unflatten}_{\tau'}\ u)) \\
& \qquad\ \ \texttt{in vec}(\texttt{map}\ f\ (\texttt{unvec}_{\tau}\ v))
\end{aligned}
$$

```
def up [k] (q:q)(r:q)(v:*st[k]) : *st[k] =
  let n = r-q
  let v = v :> *[2**(r+1)*2**(k-r-1)]c
  let up (u:*[2**(n+1)]c) : *[2*2**n]c =
    let u = u :> *[2**n*2]c
    in flatten(unvec u)
  let f (u:*[2**(r+1)]c): *[2**q*(2*2**n)]c =
    let u = u :> *[2**q*2**(n+1)]c
    in flatten(umap up (unflatten u))
  in vec(umap f (unvec v)) :> *st[k]

def down [k] (q:q)(r:q)(v:*st[k]) : *st[k] =
  let n = r-q
  let v = v :> *[2**(r+1)*2**(k-r-1)]c
  let dn (u:*[2**(n+1)]c) : *[2**n*2]c =
    let u = u :> *[2*2**n]c
    in flatten(unvec u)
  let f (u:*[2**(r+1)]c): *[2**q*(2**n*2)]c =
    let u = u :> *[2**q*2**(n+1)]c
    in flatten(umap dn (unflatten u))
  in vec(umap f (unvec v)) :> *st[k]

def swap2 [k] (q:q)(r:q)(v:*st[k]) : *st[k] =
  down (q+1) r (up q r v)
```

Figure 5. Futhark implementation of the swap2 statement, including the definitions of up and down statements.

Similarly, we can derive an implementation for down as follows, where $\tau = [2^{k-r-1}][2^{r+1}]\texttt{c}$ and $\tau' = [2^q][2^{n+1}]\texttt{c}$ and $\tau'' = [2^n][2]\texttt{c}$:

$$
\begin{aligned}
& \mathcal{I}\,[\![\texttt{down}\ q\ r]\!]_k\ (v : [2^k]\texttt{c}) \\
& \quad = \quad ((I_q \otimes Down^{r-q}_{r-q}) \otimes I_{k-q-(r-q)-1})\ v \\
& \quad = \quad ((I_q \otimes Down^{r-q}_{r-q}) \otimes I_{k-r-1})\ v \\
& \quad = \quad \texttt{let}\ n = r - q \\
& \qquad\ \ \texttt{in vec}(\texttt{map}\ (I_q \otimes Down^n_n)\ (\texttt{unvec}_{\tau}\ v)) \\
& \quad = \quad \texttt{let}\ n = r - q \\
& \qquad\ \ \texttt{let}\ f\ u = (I_q \otimes Down^n_n)\ u \\
& \qquad\ \ \texttt{in vec}(\texttt{map}\ f\ (\texttt{unvec}_{\tau}\ v)) \\
& \quad = \quad \texttt{let}\ n = r - q \\
& \qquad\ \ \texttt{let}\ f\ u = \texttt{flatten}(\texttt{map}\ Down^n_n\ (\texttt{unflatten}_{\tau'}\ u)) \\
& \qquad\ \ \texttt{in vec}(\texttt{map}\ f\ (\texttt{unvec}_{\tau}\ v)) \\
& \quad = \quad \texttt{let}\ n = r - q \\
& \qquad\ \ \texttt{let}\ dn\ u = \texttt{flatten}(\texttt{unvec}_{\tau''}\ u) \\
& \qquad\ \ \texttt{let}\ f\ u = \texttt{flatten}(\texttt{map}\ dn\ (\texttt{unflatten}_{\tau'}\ u)) \\
& \qquad\ \ \texttt{in vec}(\texttt{map}\ f\ (\texttt{unvec}_{\tau}\ v))
\end{aligned}
$$

Notice here that the main difference between the implementations of up and down is the definition of τ'' in the two instances. Futhark versions of up and down appear in Figure 5.

4 Fusion

Gate fusion for statements can be specified by the following equations:

$$\texttt{gate}\ g\ q; \texttt{gate}\ g'\ q \quad = \quad \texttt{gate}\ (g' \circ g)\ q \tag{13}$$

$$\texttt{cntrl}\ n\ g\ q; \texttt{cntrl}\ n\ g'\ q \quad = \quad \texttt{cntrl}\ n\ (g' \circ g)\ q \tag{14}$$

$$\texttt{swap}\ q; \texttt{swap}\ q \quad = \quad \texttt{nop} \tag{15}$$

$$\texttt{gate H}\ q; \texttt{gate H}\ q \quad = \quad \texttt{nop} \tag{16}$$

Where as we can implement an explicit optimisation pass that takes statements as input and outputs an optimised set of statements, we shall see that map-fusion in MiniFut (and in Futhark), will take care of gate fusion. Futhark uses a fairly rich fusion scheme which allows for fusion of many operations, in order to reduce the amount of allocation needed. Here, we shall rely only on map-map fusion, which in MiniFut amounts to implementing the rule

$$\texttt{map}\ f'\ (\texttt{map}\ f\ e) \quad = \quad \texttt{map}\ (f' \circ f)\ e \tag{17}$$

The fusion engine also utilises the properties that, for any type $\tau = [m][n]\tau'$ and for any expression e of type τ, we have $\texttt{transpose}\ (\texttt{transpose}\ e) = e$ and $\texttt{unflatten}_\tau(\texttt{flatten}\ e) = e$. It follows that we have the following properties, provided f and g are type preserving:

$$(\texttt{flatten} \circ f \circ \texttt{unflatten}_\tau) \quad \circ \tag{18}$$
$$(\texttt{flatten} \circ g \circ \texttt{unflatten}_\tau) \quad =$$
$$\texttt{flatten} \circ (f \circ g) \circ \texttt{unflatten}_\tau$$

$$(\texttt{transpose} \circ f \circ \texttt{transpose}) \quad \circ \tag{19}$$
$$(\texttt{transpose} \circ g \circ \texttt{transpose}) \quad =$$
$$\texttt{transpose} \circ (f \circ g) \circ \texttt{transpose}$$

Further, from the definitions of unvec and vec and from 18 and 19, we have

$$(\texttt{vec} \circ f \circ \texttt{unvec}_\tau) \circ (\texttt{vec} \circ g \circ \texttt{unvec}_\tau) \quad = \tag{20}$$
$$\texttt{vec} \circ (f \circ g) \circ \texttt{unvec}_\tau$$

To show that Equation 13 is closed under interpretation, we calculate $\mathcal{I}\,[\![\texttt{gate}\ g\ q; \texttt{gate}\ g'\ q]\!]_k\ v$ as follows, where $\tau = [(2^q * 2)][2^{k-q-1}]\texttt{c}$ and $\tau' = [2^q][2]\texttt{c}$:

$$\begin{aligned}
&\mathcal{I}\,[\![\texttt{gate}\ g\ q; \texttt{gate}\ g'\ q]\!]_k\ (v : [2^k]\texttt{c}) \\
&\quad = \quad \texttt{let}\ v\ =\ v \triangleright [(2^q * 2) * 2^{k-q-1}]\texttt{c} \\
&\qquad\ \ \texttt{let}\ f\ u\ =\ \texttt{flatten}(\texttt{map}\ g\ (\texttt{unflatten}_{\tau'}\ u)) \\
&\qquad\ \ \texttt{let}\ v'\ =\ \texttt{vec}(\texttt{map}\ f\ (\texttt{unvec}_\tau\ v)) \triangleright [2^k]\texttt{c} \\
&\qquad\ \ \texttt{let}\ v'\ =\ v' \triangleright [(2^q * 2) * 2^{k-q-1}]\texttt{c} \\
&\qquad\ \ \texttt{let}\ f'\ u\ =\ \texttt{flatten}(\texttt{map}\ g'\ (\texttt{unflatten}_{\tau'}\ u)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ f'\ (\texttt{unvec}_\tau\ v')) \triangleright [2^k]\texttt{c}
\end{aligned}$$

By observing that $2^k = (2^q * 2) * 2^{k-q-1}$ and by rearranging bindings, we have

$$\begin{aligned}
&\mathcal{I}\,[\![\texttt{gate}\ g\ q; \texttt{gate}\ g'\ q]\!]_k\ (v : (2^q * 2) * 2^{k-q-1}) \\
&\quad = \quad \texttt{let}\ f\ u\ =\ \texttt{flatten}(\texttt{map}\ g\ (\texttt{unflatten}_{\tau'}\ u)) \\
&\qquad\ \ \texttt{let}\ f'\ u\ =\ \texttt{flatten}(\texttt{map}\ g'\ (\texttt{unflatten}_{\tau'}\ u)) \\
&\qquad\ \ \texttt{let}\ v'\ =\ \texttt{vec}(\texttt{map}\ f\ (\texttt{unvec}_\tau\ v)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ f'\ (\texttt{unvec}_\tau\ v')) \triangleright [2^k]\texttt{c}
\end{aligned}$$

Now, because map f and map f' are type preserving, it follows from 20 that we have

$$\begin{aligned}
&\mathcal{I}\,[\![\texttt{gate}\ g\ q; \texttt{gate}\ g'\ q]\!]_k\ (v : (2^q * 2) * 2^{k-q-1}) \\
&\quad = \quad \texttt{let}\ f\ u\ =\ \texttt{flatten}(\texttt{map}\ g\ (\texttt{unflatten}_{\tau'}\ u)) \\
&\qquad\ \ \texttt{let}\ f'\ u\ =\ \texttt{flatten}(\texttt{map}\ g'\ (\texttt{unflatten}_{\tau'}\ u)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ f'\ (\texttt{map}\ f\ (\texttt{unvec}_\tau\ v))) \triangleright [2^k]\texttt{c}
\end{aligned}$$

Finally, because map g and map g' are type preserving, we can apply 18 and Equation 17 to get

$$\begin{aligned}
&\mathcal{I}\,[\![\texttt{gate}\ g\ q; \texttt{gate}\ g'\ q]\!]_k\ (v : (2^q * 2) * 2^{k-q-1}) \\
&\quad = \quad \texttt{let}\ f\ u\ =\ \texttt{flatten}(\texttt{map}\ (g' \circ g)\ (\texttt{unflatten}_{\tau'}\ u)) \\
&\qquad\ \ \texttt{in}\ \texttt{vec}(\texttt{map}\ f\ (\texttt{unvec}_\tau\ v)) \triangleright [2^k]\texttt{c} \\
&\quad = \quad \mathcal{I}\,[\![\texttt{gate}\ (g' \circ g)\ q]\!]_k\ v
\end{aligned}$$

Similar reasoning can be used to demonstrate equations 14, 15, and 16.

5 The Gate Library Interface

The interface for the Futhark gate library appears in Figure 6. The library is implemented as a module (called mk_gates) parameterised over another module representing floating-point numbers, which, for instance, is used for the internal representation of complex numbers. Futhark has modules for representing different sizes of floating point values including f16, f32, and f64. Besides gate operations, the library features functionality for creating initial state vectors and functionality for reporting the probability distribution of result states. The library also includes a series of utility functions for composing gate operations, including functionality for repeatedly executing a statement.

As an example of how dqfut may be used, we present a version of Grover's algorithm written in Futhark using the dqfut library. Grover's algorithm can find the index of an element among n unsorted elements using only $O(\sqrt{n})$ gate operations [13]. A Futhark implementation of a version of Grover's algorithm that searches for the number 12 (and finds its index) is listed in Figure 7. In practice, when the number of qubits used is 19, the search space is 2^{19} and the number of basic gates applied for the particular instance of the search is 85.369.

Grover's algorithm works by first putting all qubits in a superposition state and then, repeatedly, applying an oracle and a diffusion operation. As a result, the solution is established after approximately $\frac{\pi}{4}\sqrt{n}$ iterations.

```
module type gates = {
  type c                -- complex numbers
  type q = i64          -- qubits
  type st[n] = [2**n]c  -- state vectors
  type^ stT[n] = *st[n] → *st[n] -- transform

  val gateX [n] : q → stT[n]
  val gateY [n] : q → stT[n]
  val gateZ [n] : q → stT[n]
  val gateH [n] : q → stT[n]
  val gateT [n] : q → stT[n]
  val cntrlX[n] : (m:i64) → q → stT[n]
  val swap  [n] : q → stT[n]
  val swap2 [n] : (q:q) → (r:q) → stT[n]
  ...

  -- Ket vectors
  type ket[n] = [n]i64
  val fromKet[n] : ket[n] → *st[n]
  val toKet      : (n:i64) → (i:i64) → ket[n]

  -- Distributions
  type dist[n] = [2**n](ket[n],f64)
  val dist    [n] : st[n] → dist[n]
  val distmax [n] : dist[n] → (ket[n],f64)

  -- Some utility functions
  val >*> 'a : (q → *a → *a) → (q → *a → *a)
               → (q → *a → *a)
  val |*>   'a 'b : *a → (*a → *b) → *b
  val >* 'a 'b 'c : (*a → *b) → (*b → *c)
                    → (*a → *c)
  val repeat   'a : i64 → (i64 → *a → *a)
                    → *a → *a
}
```

Figure 6. Futhark gate-function interface.

We report on the running time of Grover's algorithm in the next section.

6 Benchmarks

In this section we report the performance of dqfut against the established quantum simulators qsim [39] and QuEST [22]. We implement the same benchmarks in all three simulators: `ghz`, `grover`, and `qft`. Here `grover` is Grover's algorithm, which is described in Section 5. The `ghz` benchmark establishes the Greenberger–Horne–Zeilinger quantum state and is ported from the SupermarQ benchmark suite [40]. The `qft` benchmark is an implementation of the Quantum Fourier Transform [41].

```
import "dqfut"
open mk_gates(f64)

def grover_diff [n] : stT[n] =
     repeat n (gateH >*> gateX)
  >* gateH (n-1)
  >* cntrlX (n-1) 0
  >* gateH (n-1)
  >* repeat n (gateX >*> gateH)

def encNum [n] (i:i64) (s:*st[n]) : *st[n] =
  (loop (s,i) = (s,i) for n in n..>0 do
     if i % 2 == 0
     then (gateX (n-1) s, i/2)
     else (s,i/2)
  ).0

def oracle [n] i : stT[n] =
  encNum i >* cntrlZ (n-1) 0 >* encNum i

def grover (n:i64) (i:i64) : (ket[n], f64) =
  let k = 2**n |> f64.i64 |> f64.sqrt
       |> (*(f64.pi/4)) |> f64.ceil
       |> i64.f64
  let s = fromKet (replicate n 0)
      |*> repeat n gateH
      |*> repeat k (λ_ → oracle i
                      >* grover_diff)
  in dist s |> distmax
```

Figure 7. Implementation of Grover's algorithm.

6.1 Experimental Setup

We perform our experiments on a system with a pair of 24-core EPYC 7352 processors, where we run with both 1 and 48 threads. For multi-threaded qsim, we use a non-vectorised (but still parallel) simulator, which is implemented via OpenMP. The system is also equipped with an NVIDIA A100 GPU, which is supported by all used simulators. All floating-point arithmetic is done in double precision. We use version 0.29.29 of the Futhark compiler to compile dqfut.

6.2 Results

The raw runtime results are shown in Table 1, and the corresponding speedups in Figure 8.

While dqfut generally performs well for the simplest benchmark, `ghz`, performance is lacking for `grover` and `qft`. The reason is unrelated to gate fusion itself, but rather due to an unfortunate interaction with the Futhark compilers' array layout representation, which requires that the memory layout of a d-dimensional array is described with a d-dimensional *linear memory access descriptor* (LMAD) [29].

This can be used to allow certain index transformations without actually moving array values, by instead adjusting the LMAD. However, some of the compositions of reshapes and transposes used by dqfut result in layouts that cannot be represented as LMADs, and hence must be physically manifested in a representable memory layout. This is not an issue for gates that are syntactically adjacent, as fusion is in most cases able to remove the interspersed flatten/unflatten operations, but it *is* an issue when using the `repeat` operation, implemented via a sequential loop carrying a single-dimensional array, where the Futhark compiler insists that the loop state be representable with a single-dimensional LMAD. The majority of the run-time is thus spent performing such semantically redundant transpositions. Gate fusion is still important: on `grover` it provides roughly a 1.5 × speedup, and the impact would be larger were it not for the redundant transpositions. It is likely that future improvements to the Futhark compiler will allow it to eliminate many of the redundant manifestations.

Futhark's parallel CPU performance is also lacking relative to the GPU performance. This is largely due to Futhark's CPU scheduler being rather naive, especially on a NUMA system such as the one used for benchmarking, compared to the mature OpenMP scheduler used by qsim and QuEST.

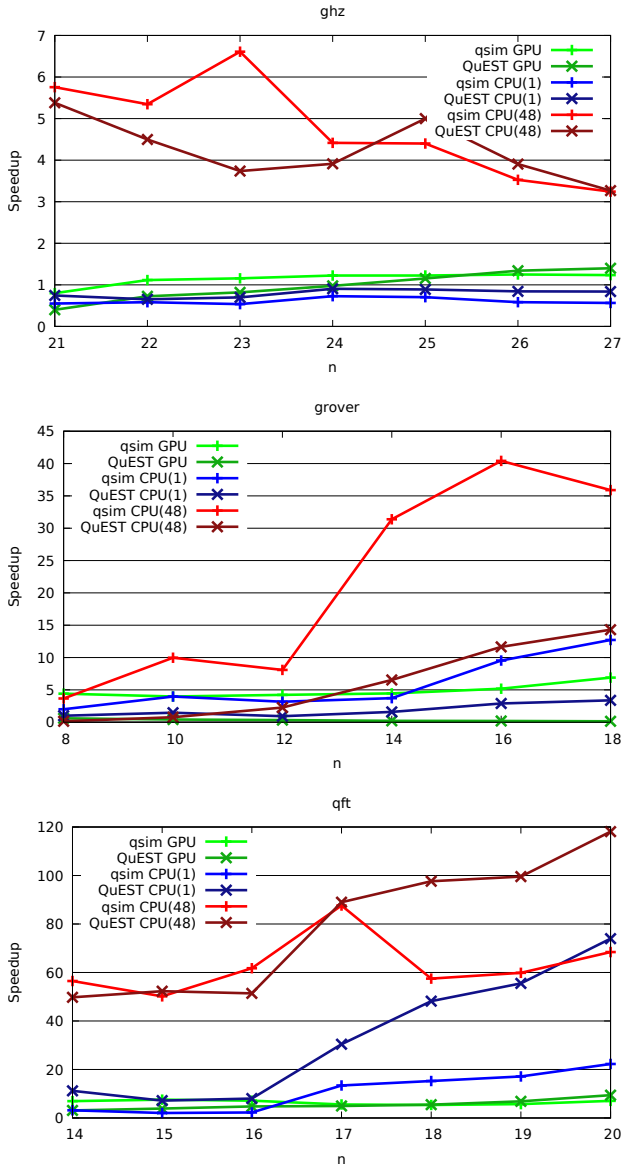

Figure 8. Plots showing the speedup of qsim and QuEST compared to our library on the three benchmarks `ghz`, `grover`, and `qft`. Points greater than one indicate that these tools are faster than our library.

7 Related Work

Many quantum computing text books, including [23, 30, 41], present the basics of quantum computing through the notions of qubits, gates, and diagrammatic circuits. Text books often give semantics to basic gates and composed circuits through the unitary matrices that the circuits denote and because simulators can be implemented by applying unitary complex matrices to state vectors, a naive simulator can be implemented in just a few lines of code [36]. Surprisingly, however, no previous work go the step further and demonstrate, as we do, the relationship between a state vector simulator and the semantics of circuits specified as unitary matrices. Related to (and in parallel to) the work presented here, we have also used the relationship to derive a purely functional interpreter for a circuit language declared as a recursive algebraic data type [9]. Whereas such an interpreter also avoids constructing explicit Kronecker products, it suffers from excessive state vector manipulations and data copying compared to the state-vector gate-operations derived here.

Some simulators keep the state space for a group of qubits separate as long as it is known that the group is ensured to be separable. Examples of simulators that follow such an approach are the Qiskit Aer state vector simulator [21], the Q# simulator [38], and the simulators for the Python-embedded domain-specific languages Qrisp [34] and ProjectQ [14, 19, 37]. Other examples include the simulators available in QCL [31] and in the Haskell-embedded framework Quipper [12]. Many of these frameworks uses static or dynamic techniques for implementing gate fusion [41, Section 2.5.5], [5, 11, 22, 35].

For non-general quantum computing there are ways to represent quantum states in less than $O(2^h)$ space. A well-known example includes the possibility for implementing

Table 1. Run-times in milliseconds for our three benchmarks `ghz`, `grover`, and `qft`, where n is the number of qubits, using our library and the simulators qsim and QuEST. The speedups relative to our library is shown in Figure 8.

	GPU			CPU (1)			CPU (48)		
n	dqfut	qsim	QuEST	dqfut	qsim	QuEST	dqfut	qsim	QuEST
					`ghz`				
21	1.2	1.5	3.0	163	296	219	49.5	8.6	9.2
22	2.9	2.6	4.0	257	441	394	85.0	15.9	18.9
23	5.9	5.1	7.2	520	969	744	154	23.3	41.2
24	12.0	9.8	12.3	1489	2046	1645	295	66.8	75.4
25	24.0	19.6	20.8	2920	4150	3284	550	125	110
26	50.0	40.0	37.3	5850	10024	6936	910	258	233
27	103	83.4	73.6	11967	21184	14265	1761	543	539
					`grover`				
8	7.5	1.7	12.5	1.2	0.6	1.2	2.2	0.6	15.6
10	16.3	4.1	39.3	11.9	3.0	8.1	32.9	3.3	42.7
12	39.3	9.3	120	65.2	20.6	70.3	283	35.0	125
14	93.9	21.1	447	684	184	430	2542	81.0	389
16	246	47.5	1298	10088	1058	3485	7759	192	666
18	786	114	5589	108478	8530	32092	39449	1099	2758
					`qft`				
14	11.8	1.7	3.8	81.7	25.8	7.3	418	7.4	8.4
15	13.6	1.8	3.5	123	60.0	17.2	617	12.3	11.8
16	15.7	2.2	3.3	313	138	39.1	704	11.4	13.7
17	17.8	3.2	3.6	2640	197	86.8	1139	13.0	12.8
18	22.5	4.2	4.1	8672	569	180	1523	26.5	15.6
19	34.3	6.0	5.0	17205	1005	310	2867	47.9	28.8
20	64.9	9.2	6.9	53023	2381	717	6247	91.3	52.9

efficient simulators for circuits composed from only Clifford-gates (Pauli-gates, H-gate, and $C\,X$ gate) [1]. Other possibilities for obtaining faster simulation algorithms for non-general quantum computing include restrictions of possible structures of circuits [28] and to use static analyses for identifying sets of qubits that can be simulated independently [2, 32]. Yet a possibility for optimisation is to use sparse representations of state vectors as implemented in the Q# simulator [15, 20].

Although many of these simulation frameworks are more elaborate than the simple framework we present here, none of the frameworks are demonstrated to be derived directly from a specification of the semantics of the operations.

Some tiling and memory considerations are treated in some related work that distribute the state vector on several devices [8]; it would be interesting to speculate on possible approaches to handling better these "skewed" access and update patterns.

8 Conclusions and Future Work

We have demonstrated that it is possible to implement a general simulator framework for quantum circuits in a functional data-parallel array language directly from the semantic specification of the state-vector gate operations.

There are plenty of possibilities for future work, including the possibility for improving the performance of the derived simulator both by improving the inner workings of the Futhark array-indexing representation and by using the framework as a backend for a higher-level quantum programming language that may benefit from circuit optimisations that are identified and exploited outside of the framework [6, 33, 42].

References

[1] Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Phys. Rev. A* 70 (Nov 2004), 052328. Issue 5. doi:10.1103/PhysRevA.70.052328

[2] Nicola Assolini, Alessandra Di Pierro, and Isabella Mastroeni. 2024. Abstracting Entanglement. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains* (Pasadena, CA, USA) *(NSAD '24)*. Association for Computing Machinery, New York, NY, USA, 34–41. doi:10.1145/3689609.3689998

[3] Lubin Bailly, Troels Henriksen, and Martin Elsman. 2023. Shape-Constrained Array Programming with Size-Dependent Types. In *Proceedings of the 11th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing* (Seattle, WA, USA) *(FHPNC 2023)*. Association for Computing Machinery, New York, NY, USA, 29–41. doi:10.1145/3609024.3609412

[4] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A.

Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Physical Review A* 52, 5 (Nov. 1995), 3457–3467. doi:10.1103/physreva.52.3457

[5] Srikar Chundury, Jiajia Li, In-Saeng Suh, and Frank Mueller. 2024. DiaQ: Efficient State-Vector Quantum Simulation. arXiv:2405.01250 [quant-ph] https://arxiv.org/abs/2405.01250

[6] Gavin E. Crooks. 2024. Quantum Gates - Gates, States, and Circuits. https://threeplusone.com/gates Tech. Note 014 v0.11.0 beta.

[7] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. arXiv:1707.03429 [quant-ph] https://arxiv.org/abs/1707.03429

[8] Jun Doi and Hiroshi Horii. 2020. Cache Blocking Technique to Large Scale Quantum Computing Simulation on Supercomputers. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE, 212–222. doi:10.1109/qce49297.2020.00035

[9] Martin Elsman. 2025. Deriving a Kronecker-Free Functional Quantum Simulator. In *Proceedings of the 2nd Workshop on Quantum Software* (Seoul, South Korea) *(WQS 2025)*.

[10] Martin Elsman, Troels Henriksen, and Cosmin E. Oancea. 2018. *Parallel Programming in Futhark*. https://futhark-book.readthedocs.io

[11] Jennifer Faj, Ivy Peng, Jacob Wahlgren, and Stefano Markidis. 2023. Quantum Computer Simulations at Warp Speed: Assessing the Impact of GPU Acceleration. arXiv:2307.14860 [cs.PF] https://arxiv.org/abs/2307.14860

[12] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 333–342. doi:10.1145/2491956.2462177

[13] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. doi:10.1145/237814.237866

[14] Thomas Haner, Damian S. Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High Performance Emulation of Quantum Circuits. In *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 866–874. doi:10.1109/sc.2016.73

[15] Thomas Häner, Damian S. Steiger, Mikhail Smelyanskiy, and Matthias Troyer. 2016. High performance emulation of quantum circuits. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 74, 9 pages.

[16] Troels Henriksen and Martin Elsman. 2021. Towards size-dependent types for array programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Virtual, Canada) *(ARRAY 2021)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3460944.3464310

[17] Troels Henriksen and Cosmin Eugen Oancea. 2013. A T2 graph-reduction approach to fusion. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. Association for Computing Machinery, New York, NY, USA, 47–58. doi:10.1145/2502323.2502328

[18] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. doi:10.1145/3062341.3062354

[19] Thomas Häner, Damian S Steiger, Krysta Svore, and Matthias Troyer. 2018. A software methodology for compiling quantum programs. *Quantum Science and Technology* 3, 2 (Feb. 2018), 020501. doi:10.1088/2058-9565/aaa5cc

[20] Samuel Jaques and Thomas Häner. 2022. Leveraging State Sparsity for More Efficient Quantum Simulations. *ACM Transactions on Quantum Computing* 3, 3, Article 15 (June 2022), 17 pages. doi:10.1145/3491248

[21] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum computing with Qiskit. doi:10.48550/arXiv.2405.08810 arXiv:2405.08810 [quant-ph]

[22] T Jones, A Brown, I Bush, and S Benjamin. 2019. QuEST and high performance simulation of quantum computers. *Scientific Reports* 9, 2019 (2019).

[23] Phillip Kaye, Raymond Laflamme, and Michele Mosca. 2007. *An Introduction to Quantum Computing*. Oxford University Press, Inc., USA.

[24] Ken Kennedy and Kathryn S. McKinley. 1993. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, Berlin, Heidelberg, 301–320.

[25] N. Khammassi, G. G. Guerreschi, I. Ashraf, J. W. Hogaboam, C. G. Almudever, and K. Bertels. 2018. cQASM v1.0: Towards a Common Quantum Assembly Language. arXiv:1805.09607 [quant-ph] https://arxiv.org/abs/1805.09607

[26] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream fusion, to completeness. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 285–299. doi:10.1145/3009837.3009880

[27] Hugo Daniel Macedo and José Nuno Oliveira. 2013. Typing linear algebra: A biproduct-oriented approach. *Science of Computer Programming* 78, 11 (2013), 2160–2191. doi:10.1016/j.scico.2012.07.012 Special section on Mathematics of Program Construction (MPC 2010) and Special section on methodological development of interactive systems from Interaccion 2011.

[28] Igor L. Markov and Yaoyun Shi. 2008. Simulating Quantum Computation by Contracting Tensor Networks. *SIAM J. Comput.* 38, 3 (Jan. 2008), 963–981. doi:10.1137/050644756

[29] Philip Munksgaard, Cosmin Oancea, and Troels Henriksen. 2023. Compiling a Functional Array Language with Non-Semantic Memory Information. In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages* (Copenhagen, Denmark) *(IFL '22)*. Association for Computing Machinery, New York, NY, USA, Article 2, 13 pages. doi:10.1145/3587216.3587218

[30] Michael A. Nielsen and Isaac L. Chuang. 2011. *Quantum Computation and Quantum Information: 10th Anniversary Edition* (10th ed.). Cambridge University Press, USA.

[31] Bernhard Omer. 2009. *Structured Quantum Programming*. Institute for Theoretical Physics Vienna University of Technology. http://tph.tuwien.ac.at/~oemer/doc/structquprog.pdf first version 26th May 2003.

[32] Simon Perdrix. 2008. Quantum Entanglement Analysis Based on Abstract Interpretation. In *Static Analysis*, María Alpuente and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 270–282.

[33] Jessica Pointing, Oded Padon, Zhihao Jia, Henry Ma, Auguste Hirth, Jens Palsberg, and Alex Aiken. 2024. Quanto: optimizing quantum circuits with automatic generation of circuit identities. *Quantum Science and Technology* 9, 4 (jul 2024), 045009. doi:10.1088/2058-9565/ad5b16

[34] Raphael Seidel, Sebastian Bock, René Zander, Matic Petrič, Niklas Steinmann, Nikolay Tcholtchev, and Manfred Hauswirth. 2024. Qrisp: A Framework for Compilable High-Level Programming of Gate-Based Quantum Computers. arXiv:2406.14792 [quant-ph] https://arxiv.org/abs/2406.14792

[35] Mikhail Smelyanskiy, Nicolas P. D. Sawaya, and Alán Aspuru-Guzik. 2016. qHiPSTER: The Quantum High Performance Software Testing Environment. arXiv:1601.07195 [quant-ph] https://arxiv.org/abs/1601.07195

[36] Robert Smith. 2023. A tutorial quantum interpreter in 150 lines of Lisp. https://www.stylewarning.com/posts/quantum-interpreter/ Blog post..

[37] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: an open source software framework for quantum computing. *Quantum* 2 (January 2018), 13. doi:10.22331/q-2018-01-31-49

[38] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL '18)*. ACM, 1–10. doi:10.1145/3183895.3183901

[39] Quantum AI team and collaborators. 2020. *qsim*. doi:10.5281/zenodo.4023103

[40] Teague Tomesh, Pranav Gokhale, Victory Omole, Gokul Subramanian Ravi, Kaitlin N. Smith, Joshua Viszlai, Xin-Chuan Wu, Nikos Hardavellas, Margaret R. Martonosi, and Frederic T. Chong. 2022. SupermarQ: A Scalable Quantum Benchmark Suite. arXiv:2202.11045 [quant-ph] https://arxiv.org/abs/2202.11045

[41] Colin P. Williams. 2011. *Explorations in Quantum Computing*. Springer-Verlag London. doi:10.1007/978-1-84628-887-6

[42] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022. Quartz: superoptimization of Quantum circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 625–640. doi:10.1145/3519939.3523433

Received 2025-04-01; accepted 2025-04-19