Deriving a Kronecker-Free Functional Quantum Simulator

Declarative Pearl

Martin Elsman[0000−0002−6061−5993]
mael@di.ku.dk

Department of Computer Science, University of Copenhagen,
Universitetsparken 5, DK-2100 Copenhagen, Denmark

Abstract. The effect of a quantum circuit on a quantum state can be expressed by specifying the unitary complex matrix that the circuit denotes. This notion of denotation can be expressed compositionally by inductively defining the denotation of sub-circuits and by relating the denotation of a composed circuit by composing the denotation of sub-circuits. Concretely, the state of an n-qubit quantum computer can be represented by a size 2^n complex vector and the denotation of an n-qubit quantum circuit can be expressed by a size $2^n \times 2^n$ unitary complex matrix. Based on the denotational semantics of a quantum circuit, it is straightforward, on a classical computer, to simulate the effect of the quantum circuit, simply by applying the unitary matrix, denoted by the circuit, on the incoming state vector. For small n, this strategy works well whereas for larger n (e.g., $n > 8$), the size of the unitary matrix becomes a limiting factor. A question emerges. Can we do better?
In this paper we derive a purely functional interpreter for quantum circuits that operates directly on a circuit without first constructing the unitary matrix that the circuit denotes. The interpreter takes as input an n-qubit quantum circuit and a complex state vector of size 2^n and outputs a resulting state vector also of size 2^n. We demonstrate the performance benefits of the approach and highlight some of its promises.

Keywords: Quantum simulation, semantics, vectorisation

1 Introduction

Most existing quantum-circuit simulation-frameworks are imperative in nature. They are centered around a programming model with operators that act on a set of qubits and implicitly update the underlying state space [3, 7, 9, 11, 15, 18, 28, 31, 32]. On the other hand, using a functional approach, it is possible to simulate the effect of a quantum circuit on the underlying state of a set of qubits by first establishing the unitary matrix that the circuit denotes and then applying that matrix to the state vector [30]. However, for general circuits, the space required by such a simulation approach is much larger than the space required to represent the state of the quantum computer. In general, the state of an n-qubit quantum

computer can be represented in $O(2^n)$ space on a classical computer, whereas the unitary matrix denoted by the n-qubit circuit occupies $O(2^{2n})$ space.

We show that it is possible to derive (i.e., calculate) a functional quantum-state simulator for a quantum circuit without representing the unitary matrix that the circuit denotes. We further demonstrate the performance benefits of the functional interpreter compared to the unitary matrix approach by presenting running times with the different simulators for an implementation of Grover's algorithm [14]. The approach is surprisingly novel and can, for instance, be used for generating specialised quantum-circuit interpreters.

2 Qubits, Quantum Circuits, 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 *identity gate* I, the *Pauli-gates* X, Y, and Z, the *Hadamard gate* H, and the T-gate [21,26,35]:

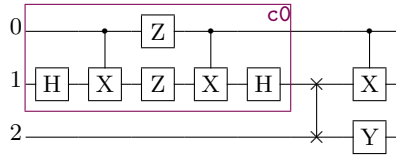
$$I = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} \qquad X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \qquad Y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix}$$

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

The effect of "applying" the Hadamard gate H to a qubit in the $|0\rangle$ state (i.e., the vector $\begin{bmatrix}1 & 0\end{bmatrix}^{\mathrm{T}}$) puts the qubit in the *superposition* state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. 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 can be composed sequentially and in parallel to form quantum circuits. A circuit can be *drawn* using a simple diagram notation with each qubit represented by a horizontal line, single-qubit operations drawn as boxes, and so-called swaps drawn by connecting two lines with crosses. Sequential composition corresponds to multiplying unitary matrices and parallel composition corresponds to forming the tensor-product of two unitary matrices. In the diagram notation, tensors are implicit and represented as vertical composition whereas sequential composition is horizontal. So-called *control gates* provide the

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




```
val c0 = I**H++C(X)++Z**Z++C(X)++I**H
val c1 = c0**I++I**SW++C(X)**Y
```

$$\begin{bmatrix} -i & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & i & 0 & 0 & 0 & 0 & 0 \\ 0 & -i & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & i & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -i & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & i \\ 0 & 0 & 0 & 0 & -i & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & i & 0 \end{bmatrix}$$

Fig. 1. A quantum circuit, its functional specification, and the complex unitary matrix that it denotes.

possibility that multiple qubits get *entangled*, meaning that it becomes impossible to decompose the state of the involved qubits into states of the individual qubits. Control gates are drawn by vertically connecting a gate on one qubit line with a circle on another qubit line. Figure 1 shows a quantum circuit of *height* 3 (i.e., number of qubits) composed from a series of basic gates and with qubits connected using control gates and swaps. For an in depth introduction to quantum circuits, consult [26].

Circuits can be represented using a functional language (e.g., Standard ML) type declaration:

```
datatype c = I | X | Y | Z | H | T | SW | C of c
           | Ten of c * c | Seq of c * c
```

Here `I` represents the identity gate, which together with `X`, `Y`, `Z`, `H`, and `T` are the basic one-qubit gates. Notice that, in diagrams, the identity gate is drawn as a horizontal line connecting the input directly to the output. The gate `SW` represents the two-qubit swap gate and `C` is a constructor for controlled circuits. Circuits are composed sequentially using the `Seq` constructor, provided the subcircuits operate on the same number of qubits, and in parallel using the `Ten` constructor. We use `**` as an infix-notation for `Ten`, which has higher precedence than the `++` infix-notation for `Seq`.

It is straightforward to extend the circuit type declaration with additional single-qubit and multi-qubit gates (e.g., S and R gates [26]). Swapping non-neighboring qubits can be modeled using a so-called *swap circuit* consisting of a sequence of swaps, each tensored with appropriate identity gates. Moreover, so-called *multi-controlled* gates can be modeled using nested control gates, perhaps generalised using swap circuits.

3 Semantics

The semantics of well-formed circuits is defined inductively as a function that takes a circuit of height n and returns a unitary complex matrix of size $2^n \times 2^n$ denoting the circuit. The function, which makes use of libraries for complex numbers and matrices (Appendices A and B), is listed in Fig. 2. The structure

```
fun sem (c:c) : mat =
  let val (c0, c1, ci) =
        (C.fromInt 0, C.fromInt 1, C.fromIm 1.0)
      val rsqrt2 = C.fromRe (1.0 / Math.sqrt 2.0)
      val eipi4 = C.exp(C.fromIm(Math.pi/4.0))
  in case c of
        I ⇒ fromIntM [[1,0], [0,1]]
      | X ⇒ fromIntM [[0,1], [1,0]]
      | Y ⇒ M.fromListList [[c0,C.~ ci], [ci,c0]]
      | Z ⇒ fromIntM [[1,0], [0,~1]]
      | H ⇒ M.fromListList [[rsqrt2,rsqrt2],
                           [rsqrt2,C.~ rsqrt2]]
      | T ⇒ M.fromListList [[c1,c0], [c0,eipi4]]
      | SW ⇒ fromIntM [[1,0,0,0], [0,0,1,0],
                      [0,1,0,0], [0,0,0,1]]
      | Seq(A,B) ⇒ matmul (sem B, sem A)
      | Ten(A,B) ⇒ tensor (sem A, sem B)
      | C A ⇒ control (sem A)
  end
```

Fig. 2. Declarative semantics of circuits.

`C` provides operations on complex numbers. For instance, the function `C.fromIm` takes a `real` value and turns it into a complex imaginary number. Auxiliary functions on complex matrices are defined in Appendix C, including functions for complex matrix multiplication (`matmul`), complex tensor products (`tensor`), functions for converting integer lists to complex matrices (`fromIntM`), and a generalised `control` function, of type `mat` → `mat`, following [5].

The semantics of all single-qubit gates and the swap gate are defined using by appropriate constant matrices. In alignment with the description in Section 2, the semantics of sequencing (i.e., `Seq`) is defined in terms of matrix multiplication, the semantics of tensoring (i.e., `Ten`) is defined in terms of tensor products, and the semantics of control gates (i.e., `C`) is defined in terms of a constructed control gate.

The `tensor` operation forms the tensor-product $A \otimes B$ of two matrices A and B of dimensions $m \times n$ and $p \times q$, respectively:

$$(A \otimes B)_{ij} = A_{(i/p,j/q)} B_{(i\%p,j\%q)} \tag{1}$$

Here we write A_{ij} or $A_{(i,j)}$ to denote the zero-indexed (i,j)-element of the matrix A (row-major). The result is a matrix of size $mp \times nq$.

We note here that the implementation of matrices is based on so-called *pull-arrays*, which use a functional representation, and which can be materialised by piping them (using the infix pipe operator `|>`) into row-major flat memory with the function `M.memoize`.

Based on the semantics, it is straightforward to define an evaluation function that, given a well-formed circuit of height h and a state vector of length 2^h

returns a state vector (also of length 2^h) representing the result of "applying the circuit" to the input:

```
fun eval (c:c, v:vec) : vec =
  M.matvecmul_gen C.* C.+ (C.fromInt 0) (sem c) v
```

The `eval` function makes use of a generic function for matrix-vector multiplication, available in the matrix-library `M`. As an example, calling the `eval` function on the circuit `c1` and a state corresponding to the ket $|101\rangle$ (i.e., the state $\begin{bmatrix}0&0&0&0&0&1&0&0\end{bmatrix}^{\mathrm{T}}$) results in the state $\begin{bmatrix}0&0&0&0&(-i)&0&0&0\end{bmatrix}^{\mathrm{T}}$, which entails that there is a 100 percent probability of finding the resulting system in the basis state $|100\rangle$.[2]

4 Kronecker-Free Circuit Interpretation

We now demonstrate that it is possible to evaluate a circuit directly on an input state vector and return an output state vector without first creating the unitary complex matrix that the circuit denotes. In particular, we shall avoid the intermediate representation of tensor products and thus avoid the state space expansion caused by (1).

We make use of the notion of *vectorisation* [22], which leads to the observation

$$(A \otimes B)\mathtt{vec}(V) = \mathtt{vec}(BVA^T) \tag{2}$$

where $\mathtt{vec}(V)$ is the vector formed by "stacking" the columns of V. When V is a matrix of size $m \times n$, the *vectorisation* of V, written $\mathtt{vec}(V)$, is the vector of length mn and values $\mathtt{vec}(V)_i = V_{(i\%m,i/m)}$. For a vector v of length mn, the *unvectorisation* of v, written $\mathtt{unvec}(v)$, is the matrix V of size $m \times n$ such that $\mathtt{vec}(V) = v$. Concrete implementations appear in Appendix C.

We now derive a function `interp` that acts as an interpreter working directly on a circuit. To ease the reasoning below, when c is some circuit, we write $[\![c]\!]$ to denote the unitary complex matrix that the circuit denotes (as obtained with `sem` c). Semantically, the interpreter has the property that for any well-formed circuit c of height h and complex vector v of length 2^h, we have

$$\mathtt{interp}\ c\ v = [\![c]\!]v \tag{3}$$

We derive the interpreter inductively on the structure of circuits, case by case, starting from 3 and from the property that $[\![c]\!]v =$ `eval (`c`,`v`)`. The resulting function `interp` is listed in Fig. 3.

The base cases are straightforward as the definitions for the base cases, including the swap gate, make directly use of the `eval` function. An exception is the case $c =$ `I` for which we simply return the input vector directly. For sequential composition (i.e., the `Seq` case), the derivation is as follows:

$$\begin{aligned}\mathtt{interp}\ (A\texttt{++}B)\ v = [\![A\texttt{++}B]\!]v &= ([\![B]\!][\![A]\!])v = [\![B]\!]([\![A]\!]v)\\ &= \mathtt{interp}\ B\ (\mathtt{interp}\ A\ v)\end{aligned} \tag{4}$$

[2] The high magnitude of basis state index 4 results in a measurement of the state $|100\rangle$. We use a big-endian encoding [21].

We derive 4 from the definition of the `sem` function in Fig. 2, from properties of matrix multiply, and by applying induction twice. For tensor composition (i.e., the `Ten` case), we have

$$\begin{aligned}
\texttt{interp}\ (A \texttt{**} B)\ v &= [\![A \texttt{**} B]\!] v \\
&= \texttt{vec}([\![B]\!] V [\![A]\!]^T) \ \text{ where } V = \texttt{unvec}\ v && (5) \\
&= \texttt{vec}(([\![A]\!]([\![B]\!]V)^T)^T) && (6)
\end{aligned}$$

Here we derive 5 from 2 and we derive 6 from the property that $(XY)^T = Y^T X^T$, for any compatible complex matrices X and Y. By induction and for any v' and v'' with appropriate lengths, we have

$$[\![A]\!] v' = \texttt{interp}\ A\ v' \tag{7}$$
$$[\![B]\!] v'' = \texttt{interp}\ B\ v'' \tag{8}$$

From 8, we have $[\![B]\!]V = (\texttt{map}\ (\texttt{interp}\ B)\ (V^T))^T$, which follows from the definition of matrix multiply, and thus

$$([\![B]\!]V)^T = \texttt{map}\ (\texttt{interp}\ B)\ (V^T) \tag{9}$$

Here `map` applies a function to each row (i.e., a vector) of a matrix. It follows from 6, 9, and by introducing a variable W, that we have

$$\begin{aligned}
\texttt{interp}\ (A \texttt{**} B)\ v = {} & \texttt{let}\ W\ \texttt{=}\ \texttt{map}\ (\texttt{interp}\ B)\ (V^T) \\
& \texttt{in}\ \texttt{vec}(([\![A]\!]W)^T)
\end{aligned} \tag{10}$$

From 7, we have $[\![A]\!]W = (\texttt{map}\ (\texttt{interp}\ A)\ (W^T))^T$, which follows from the definition of matrix multiply, and thus

$$([\![A]\!]W)^T = \texttt{map}\ (\texttt{interp}\ A)\ (W^T) \tag{11}$$

It now follows from 10, 11, and by introducing a variable Y, that we have

$$\begin{aligned}
\texttt{interp}\ (A \texttt{**} B)\ v = {} & \texttt{let}\ V\ \texttt{=}\ \texttt{unvec}\ v \\
& \texttt{let}\ W\ \texttt{=}\ \texttt{map}\ (\texttt{interp}\ B)\ (V^T) \\
& \texttt{let}\ Y\ \texttt{=}\ \texttt{map}\ (\texttt{interp}\ A)\ (W^T) \\
& \texttt{in}\ \texttt{vec}(Y)
\end{aligned} \tag{12}$$

For control gates (i.e., the `C` case), we leave it up to the reader to demonstrate that the result vector can be obtained by catenating the first half of the input vector with the result of interpreting the controlled circuit on the second half of the input vector, as shown in Fig. 3.

5 Interpreter Improvements

There are many opportunities for improving the derived interpreter, including the possibility of adapting the interpreter to use sparse representations of vectors. Another possibility is for the function always to test if the argument circuit

```
fun interp (c:c) (v:vec) : vec =
  case c of
    I ⇒ v
  | Seq(A,B) ⇒ interp B (interp A v)
  | Ten(A,B) ⇒
     let val V = unvec (pow2(height A),pow2(height B)) v
         val W = mapRows (interp B) (M.transpose V)
         val Y = mapRows (interp A) (M.transpose W)
     in vec Y
     end
  | C A ⇒ Vector.concat
             [vecTake (pow2(height A)) v,
              interp A (vecDrop (pow2(height A)) v)]
  | _ ⇒ eval (c,v)
```

Fig. 3. Kronecker-free quantum circuit interpreter. The functions `pow2` and `height` are easily defined.

is a tensor of identity gates, in which case the argument vector may be returned immediately. Moreover, the function may be refined to check if the argument vector contains only zeroes, in which case a zero-vector may be returned immediately. Also, it may be beneficial for the function to use the `sem` function for circuits up to some height and perhaps even cache unitary matrices denoted by sub-circuits that appear in the circuit more than once. Finally, it may be beneficial to optimise the circuit, using algebraic rewriting, and perhaps using a library of algebraic identities on circuits [36].

We end this section by suggesting two imperative simulation algorithms. The first one makes use of an execution scheme based on the functional interpreter. We observe that the functional interpreter always operates on a subpart of the state vector and that the vector returned is a modified version of that particular substate. Using these observations, we can derive an imperative execution function `exec` that, besides a circuit, takes an index function as argument. Instead of the vector that `interp` is passed as argument, the index function serves to describe how the vector values are obtained and how results can be written. In a functional language that supports mutable arrays, we can implement such an appropriate index function using a combination of bidirectional one-dimensional and two-dimensional pull-arrays. As we shall see in the next section, with such an implementation, we can achieve a three-times speedup, obtained primarily through memory-savings and reduced copying.

The second imperative simulation function we suggest (we call it `imp`) is a classical imperative simulation function that operates directly on a mutable state vector array [12]. For each gate in the circuit, operating on a particular qubit q and ordered from left to right, the simulation function performs a sequence of unitary matrix operations on appropriate indexes of the global state vector. Multi-qubit gates and (multi-)conditioned gates are handled using special index-

manipulations. We shall not here establish a formal link between the semantics of a circuit and simulating it using `imp`. Most existing simulators are implemented as variations of `imp`.

6 Performance Measurements

We have implemented Grover's search algorithm [14] in the circuit language and simulated it with the different simulation functions presented in the previous sections (`eval`, `interp`, `exec`, and `imp`) and with different sizes of the search space (different number of qubits). All measurements are performed on a 2023 MacBook Pro equipped with an Apple M2 Max processor, 32GB of memory, and macOS 15.1.1. Programs are compiled with MLton 20210117 [34] and time measurements reported are averages of 10 runs (with standard deviations below two percent).

Figure 4 shows the running times for simulating Grover's algorithm for the different simulation functions and with different sizes of the search space. We shall not discuss Grover's algorithm here but instead refer to the many good expositions, including Grover's original paper [14].

The circuit size grows with the size of the search space. With five qubits, the search space is 2^5 and the number of basic gates used by the algorithm is 157, which include an encoding of the search oracle. With 16 qubits, the algorithm uses 25.342 basic gates and with 19 qubits and a 2^{19} search space, 85.369 basic gates are used. For the 19-qubit case, a state-vector for the full system occupies 2^{19+4} bytes (a complex number occupies 16 bytes), which amounts to about 8Mb. Using the `eval` function, execution time grows quickly due to the space used to hold large unitary matrices and the time used for matrix processing. The `interp` function performs much better. The refined simulator `exec`, which operates on a global state vector using a combination of one-dimensional and two-dimensional mutable array-views, provides a further three-times speedup (`exec`). We consider it future work to close the performance gap up to the full imperative version `imp`, which is most likely caused by the interpretative overhead of manipulating indexes through compositions of index functions.

7 Related Work

The basics of quantum computing, including the notions of qubits, gates, and circuits have been described in a series of quantum computing textbooks [21,26,35]. The semantics of single-qubit gates is given using unitary matrices, which are composed sequentially using matrix multiplication and in parallel using Kronecker products (to form other unitary matrices). As we have shown, a naive interpreter can be implemented, directly based on the semantics, in just a few lines of code [30].

The notion of vectorisation, and in particular (2), is most thoroughly described by Macedo and Oliveira [22], but was earlier recognised as a tool in statistics by Magnus and Neudecker [23, 25]. In previous work, we have used

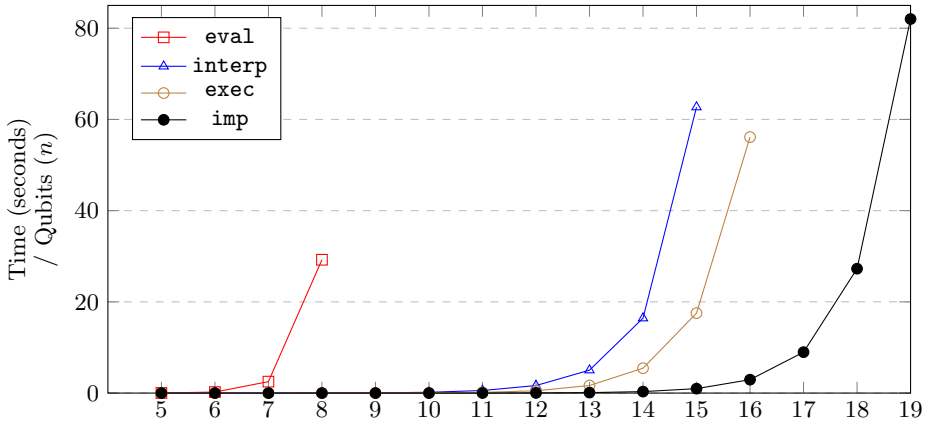


Fig. 4. Average execution times in seconds (over 10 runs with standard deviation less than two percent) for simulating Grover's algorithm with different simulators and different sizes of search problems (used qubits).

the notion of vectorisation for embedding a domain-specific language for gate operations in a data-parallel language [10]. No other previous work we know of demonstrate, as we do, using the notion of vectorisation, the relationship between an inductively defined state vector simulator and the semantics of circuits specified as unitary matrices. Unrelated to deriving an efficient interpreter for quantum circuits, vectorisation has been used for proving properties about so-called quantum strategies for homomorphism games [2].

Another particular strand of related work focuses on proving properties of quantum circuits, including proving that two circuits are semantically identical. Such work include work on model checking quantum circuits [8] and on using symbolic reasoning techniques [36].

In the functional programming community, continuations have been used as a technique for ensuring updates of entangled qubits [6]. In comparison, the technique we present here provides a direct approach to simulating quantum circuits.

8 Conclusion and Future Work

We have demonstrated that it is possible to implement a general simulator for quantum circuits in a functional language without generating tensor products for sub-circuits.

We are not claiming that the derived interpreter presented here is close in performance compared to state-of-art state vector simulators, such as qsim [33] and QuEST [20], which combine in-place state-vector updates with techniques for instruction-level vectorisation, gate fusion, and parallelism to achieve a performance that is magnitudes higher than the performance of the simple interpreter presented here.

There are plenty of possibilities for future work, including the possibility for combining the approach with parallelisation techniques [3, 7, 9, 11, 15, 32], with techniques for statically separating state spaces [4, 27], with techniques for gate fusion [17, 29], and with techniques for avoiding state space blow up when possible [1, 24], perhaps controlled using approximation techniques [16]. Preliminary experiments show promising results with respect to specialising the interpreter, using monadic GPU code generation [19].

Acknowledgments. Thanks to Fritz Henglein and Troels Henriksen for many fruitful discussions about this work and to Andrzej Filinski for encouraging publication of the work.

Disclosure of Interests. None.

References

1. Aaronson, S., Gottesman, D.: Improved simulation of stabilizer circuits. Phys. Rev. A **70**, 052328 (Nov 2004). https://doi.org/10.1103/PhysRevA.70.052328
2. Abramsky, S., Barbosa, R.S., de Silva, N., Zapata, O.: The Quantum Monad on Relational Structures. In: Larsen, K.G., Bodlaender, H.L., Raskin, J.F. (eds.) 42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017). Leibniz International Proceedings in Informatics (LIPIcs), vol. 83, pp. 35:1–35:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2017). https://doi.org/10.4230/LIPIcs.MFCS.2017.35
3. Amariutei, A., Caraiman, S.: Parallel quantum computer simulation on the gpu. 15th International Conference on System Theory, Control and Computing pp. 1–6 (2011), https://api.semanticscholar.org/CorpusID:18896951
4. Assolini, N., Di Pierro, A., Mastroeni, I.: Abstracting entanglement. In: Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains. p. 34–41. NSAD '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3689609.3689998
5. Barenco, A., Bennett, C.H., Cleve, R., DiVincenzo, D.P., Margolus, N., Shor, P., Sleator, T., Smolin, J.A., Weinfurter, H.: Elementary gates for quantum computation. Physical Review A **52**(5), 3457–3467 (Nov 1995). https://doi.org/10.1103/physreva.52.3457
6. Choudhury, V., Agapiev, B., Sabry, A.: Scheme pearl: Quantum continuations. In: Preprint for the 2022 Scheme Workshop (September 2022)
7. Claudino, D., Lyakh, D.I., McCaskey, A.J.: Parallel quantum computing simulations via quantum accelerator platform virtualization. Future Generation Computer Systems **160**, 264–273 (2024). https://doi.org/10.1016/j.future.2024.06.007
8. Do, P.N., Ogata, K.: Symbolic model checking of quantum circuits in maude. PeerJ Computer Science **10**, e2098 (2024). https://doi.org/10.7717/peerj-cs.2098
9. Doi, J., Horii, H.: Cache blocking technique to large scale quantum computing simulation on supercomputers. In: 2020 IEEE International Conference on Quantum Computing and Engineering (QCE). p. 212–222. IEEE (Oct 2020). https://doi.org/10.1109/qce49297.2020.00035

10. Elsman, M., Henriksen, T.: Gate fusion is map fusion. In: Proceedings of the 11th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming. ARRAY 2025, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3736112.3736143
11. Faj, J., Peng, I., Wahlgren, J., Markidis, S.: Quantum computer simulations at warp speed: Assessing the impact of gpu acceleration (2023), https://arxiv.org/abs/2307.14860
12. Fang, B., Özkaya, M.Y., Li, A., Çatalyürek, U.V., Krishnamoorthy, S.: Efficient hierarchical state vector simulation of quantum circuits via acyclic graph partitioning. In: 2022 IEEE International Conference on Cluster Computing (CLUSTER). pp. 289–300 (2022). https://doi.org/10.1109/CLUSTER51413.2022.00041
13. Gansner, E.R., Reppy, J.H.: The Standard ML Basis Library. Cambridge University Press, USA (2002), https://www.cambridge.org/core/books/standard-ml-basis-library/C67A8C88BF67CDE0BE47977B234A2C20
14. Grover, L.K.: A fast quantum mechanical algorithm for database search. In: Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing. p. 212–219. STOC '96, Association for Computing Machinery, New York, NY, USA (1996). https://doi.org/10.1145/237814.237866
15. Gutierrez, E., Romero, S., Trenas, M.A., Zapata, E.L.: Parallel quantum computer simulation on the cuda architecture. In: Proceedings of the 8th International Conference on Computational Science, Part I. p. 700–709. ICCS '08, Springer-Verlag, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69384-0_75
16. Hillmich, S., Zulehner, A., Kueng, R., Markov, I.L., Wille, R.: Approximating decision diagrams for quantum circuit simulation. ACM Transactions on Quantum Computing **3**(4) (Jul 2022). https://doi.org/10.1145/3530776
17. Häner, T., Steiger, D.S.: 0.5 petabyte simulation of a 45-qubit quantum circuit. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17, ACM (Nov 2017). https://doi.org/10.1145/3126908.3126947
18. Javadi-Abhari, A., Treinish, M., Krsulich, K., Wood, C.J., Lishman, J., Gacon, J., Martiel, S., Nation, P.D., Bishop, L.S., Cross, A.W., Johnson, B.R., Gambetta, J.M.: Quantum computing with Qiskit (2024). https://doi.org/10.48550/arXiv.2405.08810
19. Jones, N.D.: Transformation by interpreter specialisation. Science of Computer Programming **52**(1), 307–339 (2004). https://doi.org/10.1016/j.scico.2004.03.010, special Issue on Program Transformation
20. Jones, T., Brown, A., Bush, I., Benjamin, S.: Quest and high performance simulation of quantum computers. Scientific Reports **9**(2019) (2019). https://doi.org/10.1038/s41598-019-47174-9
21. Kaye, P., Laflamme, R., Mosca, M.: An Introduction to Quantum Computing. Oxford University Press, Inc., USA (2007), https://global.oup.com/academic/product/an-introduction-to-quantum-computing-9780198570493
22. Macedo, H.D., Oliveira, J.N.: Typing linear algebra: A biproduct-oriented approach. Science of Computer Programming **78**(11), 2160–2191 (2013). https://doi.org/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
23. Magnus, J.R., Neudecker, H.: The commutation matrix: Some properties and applications. The Annals of Statistics **7**(2079), 381–394 (March 2079), https://www.janmagnus.nl/papers/JRM005.pdf

24. Markov, I.L., Shi, Y.: Simulating quantum computation by contracting tensor networks. SIAM Journal on Computing **38**(3), 963–981 (Jan 2008). https://doi.org/10.1137/050644756
25. Neudecker, H.: The kronecker matrix product and some of its applications in econometrics. Statistica Neerlandica **22**(1), 69–82 (1968). https://doi.org/10.1111/j.1467-9574.1960.tb00619.x
26. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. Cambridge University Press, USA, 10th edn. (2011). https://doi.org/10.1017/CBO9780511976667
27. Perdrix, S.: Quantum entanglement analysis based on abstract interpretation. In: Alpuente, M., Vidal, G. (eds.) Static Analysis. pp. 270–282. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_18
28. Seidel, R., Bock, S., Zander, R., Petrič, M., Steinmann, N., Tcholtchev, N., Hauswirth, M.: Qrisp: A framework for compilable high-level programming of gate-based quantum computers (2024), https://arxiv.org/abs/2406.14792
29. Smelyanskiy, M., Sawaya, N.P.D., Aspuru-Guzik, A.: qhipster: The quantum high performance software testing environment (2016), https://arxiv.org/abs/1601.07195
30. Smith, R.: A tutorial quantum interpreter in 150 lines of lisp (July 2023), https://www.stylewarning.com/posts/quantum-interpreter/, blog post.
31. Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M.: Q#: Enabling scalable quantum computing and development with a high-level dsl. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. p. 1–10. RWDSL '18, ACM (Feb 2018). https://doi.org/10.1145/3183895.3183901
32. Tabakin, F., Juliá-Díaz, B.: Qcmpi: A parallel environment for quantum computing. Computer Physics Communications **180**(6), 948–964 (2009). https://doi.org/10.1016/j.cpc.2008.11.021
33. team, Q.A., collaborators: qsim (Sep 2020). https://doi.org/10.5281/zenodo.4023103
34. Weeks, S.: Whole-program compilation in mlton. In: Proceedings of the 2006 Workshop on ML. p. 1. ML '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1159876.1159877
35. Williams, C.P.: Explorations in Quantum Computing. Springer-Verlag London (2011). https://doi.org/10.1007/978-1-84628-887-6
36. Xu, M., Li, Z., Padon, O., Lin, S., Pointing, J., Hirth, A., Ma, H., Palsberg, J., Aiken, A., Acar, U.A., Jia, Z.: Quartz: superoptimization of quantum circuits. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 625–640. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3519939.3523433

A Appendix: SML Signature for Complex Numbers

```
signature COMPLEX = sig
  type complex
  val mk       : real * real → complex
  val fromRe   : real → complex
  val fromIm   : real → complex
  val fromInt  : int → complex
  val conj     : complex → complex
  val re       : complex → real
  val im       : complex → real
  val ~        : complex → complex          (* negation *)
  val +        : complex * complex → complex
  val -        : complex * complex → complex
  val *        : complex * complex → complex
  ...
  val sqrt     : complex → complex
  val exp      : complex → complex
  val abs      : complex → complex
  val pow      : complex * complex → complex
  val fmt      : StringCvt.realfmt → complex → string
  val toString : complex → string
end
```

B Appendix: SML Signature for Matrices

```
signature MATRIX = sig
  type α t
  val fromListList  : α list list → α t
  val dimensions    : α t → int * int
  val sub           : α t * int * int → α
  val map           : (α → β) → α t → β t
  val map2          : (α * β → γ) → α t → β t → γ t
  val transpose     : α t → α t
  val memoize       : α t → α t
  val tabulate      : int * int * (int*int → α) → α t
  val row           : int → α t → α vector
  val pp            : int → (α → string) → α t → string
  val ppv           : int → (α → string) → α vector
                      → string
  val matmul_gen    : (α*α→α) → (α*α→α) → α → α t
                      → α t → α t
  val matvecmul_gen : (α*α→α) → (α*α→α) → α → α t
                      → α vector → α vector
  ...
end
```

C Appendix: Auxiliary Complex Matrix Operations

```
structure M = Matrix
structure V = Vector (* Part of SML Basis Library  [13] *)

(* Matrix multiplication *)
fun matmul (t1:mat,t2:mat) : mat =
    M.matmul_gen C.* C.+ (C.fromInt 0) t1 t2 |> M.memoize

(* Tensor product of two complex matrices *)
fun tensor (a:mat,b:mat) : mat =
    let val (m,n) = M.dimensions a
        val (p,q) = M.dimensions b
    in M.tabulate(m * p, n * q,
                  fn (i,j) ⇒ C.* (M.sub(a,i div p,j div q),
                                  M.sub(b,i mod p,j mod q))
                 ) |> M.memoize
    end

(* Control matrix on square complex matrices *)
fun control (a:mat) : mat =
  let val n = M.nRows a
  in M.tabulate(2*n,2*n,
 (* 1 0 0 0 *)  fn (r,c) ⇒ if r >= n andalso c >= n
 (* 0 1 0 0 *)              then M.sub(a,r-n,c-n)
 (* 0 0 a b *)              else if r = c then C.fromInt 1
 (* 0 0 c d *)              else C.fromInt 0)
  end

(* List-to-matrix conversion *)
fun fromIntM (is:int list list) : mat =
  M.fromListList (map (map C.fromInt) is)

(* Row-major flattening and unflattening *)
fun flatten (m:mat) : vec =
  let val (r,c) = M.dimensions m
  in V.tabulate(r * c, fn i ⇒ M.sub(m,i div c,i mod c))
  end
fun unflatten (r,c) (v:vec) : mat =
  M.tabulate(r,c,fn (i,j) ⇒ V.sub(v,i*c+j))

(* Column-major flattening and unflattening *)
fun vec (m:mat) : vec = M.transpose m |> flatten
fun unvec (r:int,c:int) (v:vec) : mat =
  unflatten (r,c) v |> M.transpose

(* Take and drop elements from a vector *)
fun vecTake (n:int) (v:vec) =
  V.tabulate(n, fn i ⇒ V.sub(v,i))
```

```
fun vecDrop (n:int) (v:vec) =
  V.tabulate(V.length v - n, fn i ⇒ V.sub(v,i+n))

(* Map a function over the rows of a matrix *)
fun mapRows (f:vec→vec) (a:mat) : mat =
  List.tabulate(M.nRows a, fn i ⇒ f(M.row i a))
  |> M.fromVectorList
```