QX: A High-Performance Quantum Computer Simulation Platform

N. Khammassi[1], I. Ashraf[2], X. Fu[3], C.G. Almudever[4] and K. Bertels[5]
QuTech, Computer Engineering Lab
Delft University of Technology
Delft, The Netherlands
[1]n.khammassi@tudelft.nl [2]i.ashraf@tudelft.nl [3]x.fu-1@tudelft.nl [4]c.garciaalmudever-1@tudelft.nl [5]k.l.m.bertels@tudelft.nl

***Abstract*—Quantum computing is rapidly evolving especially after the discovery of several efficient quantum algorithms solving intractable classical problems such as Shor's factoring algorithm. However the realization of a large-scale physical quantum computer is very challenging and the number of qubits that are currently under development is still very low, namely less than 15. In the absence of large size platforms, quantum computer simulation is critical for developing and testing quantum algorithms and investigating the different challenges facing the design of quantum computer hardware. What makes quantum computer simulation on classical computers particularly challenging are the memory and computational resource requirements. In this paper, we introduce a universal quantum computer simulator, called QX, that takes as input a specially designed quantum assembly language, called QASM, and provides, through agressive optimisations, high simulation speeds and large number of qubits. QX allows the simulation of up to 34 fully entangled qubits on a single node using less than 270 GB of memory. Our experiments using different quantum algorithms show that QX achieves significant simulation speedup over similar state-of-the-art simulation environment.**

I. Introduction

Since the early formulation of the idea of using quantum mechanical systems to store and process information [1] [2], the field of quantum computing is rapidly evolving especially after the elaboration of several efficient quantum algorithms solving intractable classical problems. For instance, the discovery of the Shors prime factorization algorithm [3] outlined the enormous potential of quantum computing, factoring prime numbers is the basis of several crypto-systems such as the widely used Rivest-Shamir-Adleman (RSA) security scheme [4]. Later the Grovers search algorithm [5] demonstrated quadratic speedup over its corresponding classical implementation. The discovery of these algorithms boosted the development of different physical quantum computer implementation using different technologies such as the Cavity Quantum Electrodynamics (QED) [6], trapped ions [7] or solid state systems [8].

Despite the rapid improvement of qubits coherence time and gate fidelity in different technologies, the current experimental platforms provide limited number of qubits and limited operational capabilities. When defining and building an architecture for a quantum computer, it is necessary to understand how to address and control larger numbers of qubits. As shown in Figure 1 and as described in [9], building a quantum computer involves implementing different functional layers. At the highest level, algorithm designers formulate quantum algorithms such as Shor's factoring algorithm, in a high level language that is designed to represent not only the quantum operations but also the classical logic which will always be necessary. A compiler will then translate those algorithms in the instruction set which can be executed on the quantum computer. Similarly to classical computers, the code generated by the compiler is at assembly level, and the assembler we have extended for this purpose is called Quantum Assembler (QASM) which is a quantum assembly language that has been initially defined for rendering quantum circuits in [10]. A micro-architecture will provide the hardware-based control logic needed to execute the instructions on the target qubit plane. These instructions are translated in micro-instructions and through the interface layer sent into the qubit plane.

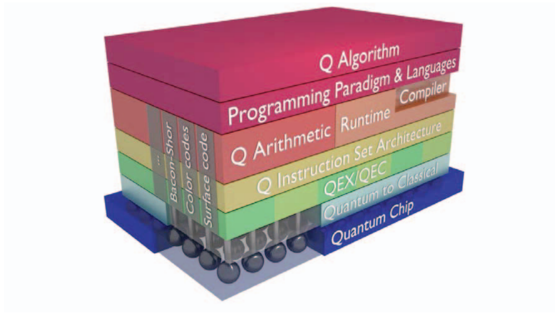


Fig. 1. Overview of quantum computer system stack.

In order to experiment, test and validate the architecture and in the absence of a large physical quantum computer capable of executing large quantum algorithms, quantum computer simulators are required for designing and testing these quantum algorithms. This way, it becomes feasible to verify their correctness and predict their behaviour on a real quantum computer under quantum noise. Requirements for the quantum computer simulator are the following: i) to allow evaluating the robustness of error correction codes in the presence of quantum noise, ii) to support the design, validation and correctness of quantum algorithms using a fully functional assembly level language allowing to describe quantum circuits, iii) to test and validate the potential of physical qubit layouts and qubit plane infrastructure with corresponding protocols and iv) to investigate hardware-software co-design to drive the development of both quantum computer hardware architecture and the software infrastructure including the quantum compiler and the execution support. These requirements therefore translate into the following features, as presented in this paper:

- A compact and expressive programming language, called QASM, allowing programmers to express quantum circuits in a straightforward way, including parallelism and binary control.
- An intuitive quantum programming interface allowing the programmer to design and test quantum circuits, algorithms and protocols.
- The ability to apply various quantum noise models.
- High performance quantum circuit simulation both in terms of memory consumption and wall clock simulation time.

The paper is organized as follows: we first introduce related work on quantum simulation and present the basics of the quantum circuit model. We then discuss the overall structure of the QX simulator and the QASM language. After discussing the need for error models, we report on the different optimizations implemented in QX and show the achieved performances compared to state-of-the-art simulators.

II. Related work

Several quantum computer simulators have been developed over the years and few are still supported. Essential features of quantum simulator are an expressive programming language, good performance and the capability of introducing quantum noise. JQuantum [11] and Jaquzzi [12] are interactive simulators incapable of handling a large number of qubits. They were developed more for educational purposes rather than high speed simulation. Several libraries were developed providing quantum functionality such as LibQuantum [13]. qHiPSTER [14] and JUMPIQCS [15] are two other simulation environment designed for distributed memory systems.

Another set of simulator platforms provides not only a simulation engine but also define a fully functional programming language such as QCL [16] and Lanq [17]. $LIQUi|\rangle$ [18] is a quantum circuit simulation platform recently released by Microsoft, it defines a language for programming quantum algorithms. We will use the $LIQUi|\rangle$ to benchmark the QX platform against this state of the art environment.

III. The Quantum Circuit Model

In contrast to the classical computers based on the well defined classical model of boolean logic, quantum computing can be implemented in different ways including adiabatic quantum computation [19], cluster state quantum computation [20] or topological quantum computation [21]. Nowadays, the quantum circuit model introduced by Deutsh [22] is the most common quantum computation model due to its analogy with classical circuits. Just like classical circuits are composed of logical gates operating on classical bits, the quantum circuit model is composed of quantum gates that operate on quantum bits referred to as qubits.

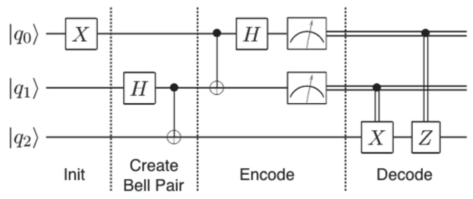


Fig. 2. The Teleportation Circuit.

A. Quantum Bit

The quantum bit or ”qubit” is the quantum analogue of the classical bit. A qubit is a two-level quantum-mechanical system. A classical bit can be only in the 0 or 1 states. However, quantum mechanics suggest that the qubit can be in a *superposition* of both states simultaneously, this property is fundamental to quantum computing. Thus, the pure qubit state can be represented as a linear combination of the basis state $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \equiv \begin{bmatrix}\alpha\\ \beta\end{bmatrix}; \quad |\alpha|^2 + |\beta|^2 = 1 \tag{1}$$

where α and β are complex numbers representing the amplitudes of each state.

B. Quantum Gates and Circuits

A quantum algorithm can be described as a circuit composed of a sequence of quantum operations applied on qubits to generate new quantum states. Gate operations are linear transformations which are applied to the quantum state and can be represented as unitary matrices.

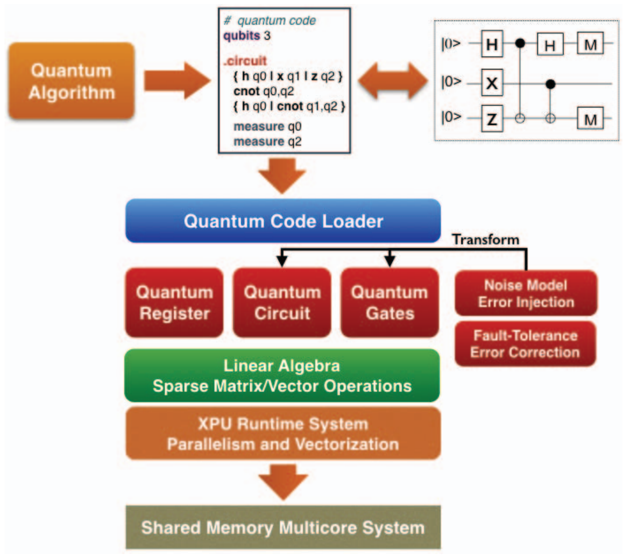
It has been demonstrated that a universal quantum computer can simulate any Turing machine [23] and any local quantum system [24]. A set of gates is called universal if they can be used to constitute a quantum circuit that can approximate any unitary operation to arbitrary accuracy.

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

$$X = \begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix} \quad Y = \begin{bmatrix}0 & -i\\ i & 0\end{bmatrix} \quad Z = \begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix} \tag{3}$$

$$CNOT = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix} \tag{4}$$

It has been proven that any unitary operation can be approximated to arbitrary accuracy by using only single qubit gates such as given in equations (2)(3) and the CNOT gate, as given in equation (4) [10]. In Figure 2, we show the famous teleportation circuit where the state of the q_0-qubit is teleported to q_2 by creating an entangled state between q_1 and q_2 and between q_0 and q_1. After measuring both q_0 and q_1, the state of q_2 will contain the state of q_0, after applying when necessary an X or Z gate depending on the measurement outcome.



```
# quantum code
qubits 3

.circuit
  { h q0 | x q1 | z q2 }
  cnot q0,q2
  { h q0 | cnot q1,q2 }
  measure q0
  measure q2
```

Quantum Algorithm
|0>
|0>
|0>
H
H
M
X
Z
M
Quantum Code Loader
Transform
Quantum Register
Quantum Circuit
Quantum Gates
Noise Model Error Injection
Fault-Tolerance Error Correction
Linear Algebra Sparse Matrix/Vector Operations
XPU Runtime System Parallelism and Vectorization
Shared Memory Multicore System

Fig. 3. Overview of the Quantum Computing Simulator Design

An important observation is that parallel quantum gates operating on different qubits can be combined or fused to form a larger unitary matrix by means of tensor products. In equation (5), two Hadamard gates are combined to form a single two-qubit gate. A sequence of quantum gates operating on the same qubit can be combined by multiplying their unitary matrices.

$$H \otimes H = \frac{1}{2}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{5}$$

Gates are applied to the quantum states through multiplying their unitary matrices by the input quantum state vector to obtain a new output quantum state. Fusing parallel gates through tensor products can generate very large matrices requiring thus significant memory resources. As more gates are combined, the density of the matrices will increase.

IV. QX: Design Overview

As shown in Figure 3, the QX simulator architecture is organized in a modular way such that future extensions are easy to introduce. The components are:

- The Quantum Code Loader which reads the input quantum code and creates the corresponding executable circuit. It can also load the execution parameters such as the error model to use and the quantum gate scheduling scheme.
- The Quantum Core includes the main simulation components: the qubits register and the quantum circuits.
- The Error Models inject discrete errors in the quantum circuit or alter quantum gates to simulate noisy execution.
- The Fault-Tolerance Module generates a fault-tolerant circuit based on the input faulty-circuit. Currently this module supports the Steane code [25] and will shortly support surface code [26]. A detailed discussion of this module is not provided due to space limitations.

```
qubits 3 # define quantum register of 3 qubits

.init    # first sub-circuit
   x q0
   display
.bell_pair  # create an EPR pair
   h q1
   cnot q1,q2
   display
.encode
   cnot q0,q1
   h q1
   measure q0
   measure q1
   display
.decode
   cx b1,q2   # binary-controlled X gate
   cz b0,q2   # binary-controlled Z gate
   display
```

Listing 1. Quantum Teleportation Circuit

- The Linear Algebra Layer implements the sparse matrix/vector operations needed for the simulation, focusing on efficiently using sparse operators.
- The XPU parallel execution runtime [27] allows the parallelization of the simulation on shared memory multicore systems both at thread and instruction levels.

V. The Quantum Assembly Code

A. *Qubits Creation*

1) *Quantum Register*

A Quantum Register is a set of qubits which is the analogue of a classical processor register. It contains the quantum state which holds the amplitudes of the different computational basis state. The programmer sets the Quantum Register size or the number of Qubits as in the example shown in Listing 1 which defines a quantum register of 3 qubits.

2) *Measurement Register*

When a quantum register is defined, a measurement register (MR) is implicitly created and linked to the quantum register. The MR is a classical register which contains the outcomes of the measurements of the different qubits. These outcomes can be used to control quantum operations. The bits of the MR are named respectively by default as b_0, b_1... and are associated with the measurement results of the qubits q_0, q_1... etc.

B. *Quantum Circuit*

Quantum algorithms can be represented as quantum circuits which specify the different operations on the qubits and their interactions. A quantum circuit is mainly composed of pure quantum gates such as the Hadamard, CNOT, and Pauli gates. We note that our QASM provides most of the common single-qubit, two-qubits and three-qubits gates. An essential component is the measurement of the state of the qubit which produces a classical bit. The later can be the result of the quantum computations or can be used to conditionally execute further quantum gates through binary-controlled gates such as in error correction circuits.

Listing 1 shows the teleportation circuit (shown in Figure 2) written in QASM. The circuit is split into four functional sub-circuits named 'init', 'bell_pair', 'encode' and 'decode' and

```
1  # define a quantum register of 7 qubits
2  qubits 7
3
4  .init  # initialization sub-circuit
5      measure q3 # measurement outcome in b3
6      cx b3,q3   # binary-controlled x gate using b3
7      {prepz q0 | prepz q1 | prepz q2}  # parallel gates
8      {x q3 | h q0 | h q1 | h q2 }
9      h q3
10 .grover(2) # 2 iterations loop
11     { x q2 | toffoli q0,q1,q4 }
12     toffoli q1,q4,q5
13     ...
14     { x q2 | h q0 | h q1 }
15     h q2
16 .result # measurement
17     { h q3 | measure q0 | measure q1 | measure q2 }
18     measure q3
19     display
```

Listing 2. Grover Algorithm Example

includes quantum gates, binary-controlled gates and measurement. In the following paragraphs, we present several QASM features.

C. Loop Support

The loop control structure is an essential component of many quantum algorithms such as the Grover's algorithm. For now, QASM provides support for the FOR-loop by simply adding the number of times a subcircuit needs to be executed right after the name of the subcircuit such as at line 10 of the Grover's algorithm example given in Listing 2.

D. Quantum Gate Scheduling

Before producing the final executable, any compiler will reorder the instructions to achieve the best performance in terms of resource usage and compute time. The same holds for quantum circuits where one tries to also reduce the overall execution time and to use the qubits in the most economical way. Both for simulation as well as real experiments, it is important to understand how the circuit latency is influenced by quantum noise.

QX allows to experiment with different noise models and timing parameters to study e.g. the robustness of the circuit to noise and how scheduling can improve that robustness. The scheduler also needs to be aware of possible hardware limitations such as the number of gates which can be executed simultaneously and to the possible trade-offs between execution speed and the decoherence of the qubits. The QASM allows the expression of parallelism between quantum gates. The gates enclosed within brackets are executed in parallel such as in the Grover's circuit example given in Listing 2.

E. Qubit Mapping

An important difference with classical computing is that qubits are mapped on a 2D lattice and that the logic is applied on the qubits using the quantum gates. Current quantum technologies require that qubits are placed next to each other when applying a multi-qubit gate. Qubit movement is performed using an expensive chained swap operations, thus mapping the qubits on the 2D lattice is critical for minimising qubit movements and improving data locality.

QASM offers a set of meta-instructions for placing or mapping a qubit to a given position on the qubit plane, such as

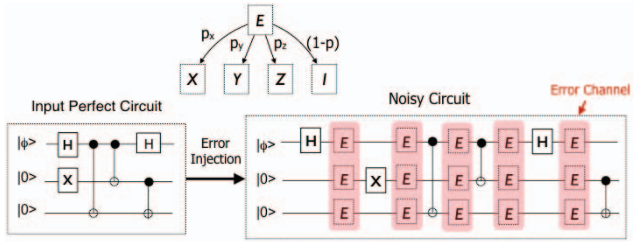


Fig. 4. The Symmetric Depolarizing Channel Implementation in QX.

map q_0**, 3, 4** where q_0 is mapped on position with coordinates x=3 and y=4. After the initial mapping, the qubits can be explicitly moved from one position to another by means of "swap" gates following the shortest path on the lattice. For instance, **mov** q_0**, 5, 1** moves q_0 to the new (x,y) coordinates. This feature aims to ease the investigation of optimal qubit mapping and qubit routing.

VI. The Depolarizing Channel

The quantum depolarizing channel is a standard model for noise in quantum systems [28]. For a d-dimensional system, the depolarising channel maps a qubit initially in state ρ into a linear combination of the state itself and the complete mixed state $\frac{I}{d}$ where I is the identity matrix and p is the probability of the qubit being depolarized.

$$\triangle_p(\rho) = (1-p)\rho + \frac{p}{d}I \tag{6}$$

The depolarizing channel can be defined by equation 7 where p_x,p_y and p_z are the respective probabilities of Pauli bit-flip, phase-flip and combined phase-bit-flip operators acting on the depolarized qubit. When p_x,p_y and p_z are equal then the depolarising channel is called symmetric.

$$\rho \rightarrow (1-p)\rho + p_x X\rho X + p_y Y\rho Y + p_z Z\rho Z \tag{7}$$

In the QX simulator, the depolarizing channel is implemented by injecting discrete errors during the different steps of the circuit in the form of quantum pauli gates with a probability of error p. The injected error can be a bitflip error or phase flip error or a combination of both of them. Figure 4 illustrates this error injection process: a perfect circuit is transformed into a noisy circuit through error injection at each time step.

VII. QX Optimisations and Experimental Results

In this section, we discuss the optimisations and compare QX to another state-of-the-art quantum computer simulator, namely $LIQUi|\rangle$ [18]. We explain how QX compares in terms of simulation speed, using different benchmarks. Our experimental hardware setup is based on a Linux SMP Platform with 2x Intel(R) Xeon(R) CPU E5-2683 with 396 GB of Memory.

A. Single Qubit Gate

As we saw in Section III, applying a single qubit gate to a quantum state vector corresponds to the multiplication of that complex vector by the tensor product of the gate matrix (applied to the target qubit) and the identity matrices (applied

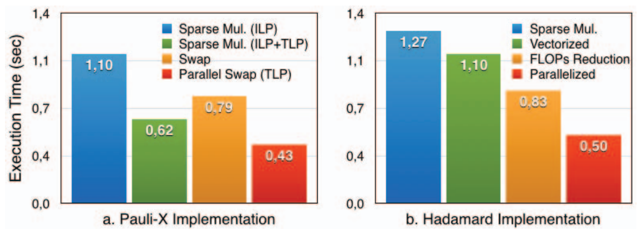


Fig. 5. (a) The effect the different optimizations on the execution time of a single qubit gate. (b) The swap-based Pauli-X Performance is compared to the multiplication-based implementation.

to the rest of the qubits). The resulting tensor matrix is highly sparse, and therefore allows a number of optimizations to reduce the memory consumption and the number of operations. In the following paragraphs, we describe how we can efficiently implement the basic complex arithmetics, then how to gradually reduce the floating point operations to speedup the execution.

1) Instruction-Level Parallelism

The first optimisation of the base version of the code is related to the vectorization of the complex arithmetic. Modern processors provide several vector instruction sets such as SSE, AVX and FMA intrinsics allow 2 and 4 ways double precision operations. Particularly, SSE3 provides special instructions to speedup complex arithmetic efficiently. We use the later instruction set in combination with the FMA extension which allows fused multiply-add operations, these instructions set are used to efficiently implement the sparse vector-matrix multiplication. The hand-vectorized implementation achieved a 24% improvement in execution time over the initial implementation automatically vectorized by the *GNU* compiler.

2) Floating Point Operations Reduction

As shown in section III of this paper, the matrices of the common quantum gates are known at compile time allowing for more specific optimizations. For instance, the elements of the Hadamard matrix are pure real complex numbers with null imaginary components allowing the reduction numbers of operations generated by a complex multiplication from 6 to 2 double precision operations. As shown in equations 4 and 5, some other matrices such as the Phase or Pauli-gates matrices are diagonal or counter-diagonal matrices allowing even less complex multiplications. Moreover, the application of the quantum gates is mainly composed of successive complex multiplication and addition allowing us to perform further optimization by fusing the complex multiplication and addition to achieve high performance and fewer memory accesses. The latest implementation reduced the overall single core execution time of the Hadamard gate operation by about 39% over the base sequential implementation. The effect of the different optimization steps on single qubit gate performance on a 28 qubits circuit are depicted in Figure 5.

3) Swap-Based Implementation

The Pauli-X gate is a common single-qubit gate which performs a bit-flip on the target qubit. If we refer to equation 3, we can observe that the unitary matrix of the Pauli-X gate is a permutation matrix, thus the tensor product of that matrix with an identity matrix is still a permutation matrix. One interesting

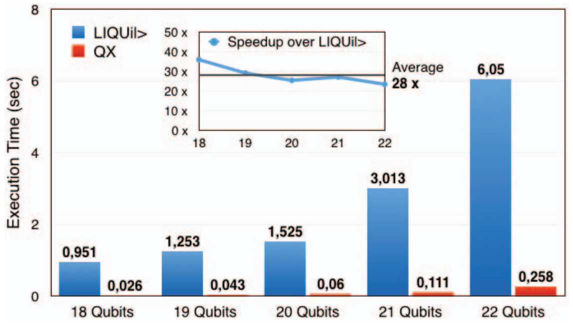


Fig. 6. Comparison of the performance of the QX and $LIQUi|\rangle$ simulators executing the QFT circuit on different number of qubits.

property of this type of matrix is that when multiplied by a vector, it simply permutes elements of that vector, thus the multiplication of such a matrix by our quantum state vector permutes a finite number of its elements. Therefore, the costly complex multiplication can be replaced by faster swap operations which reduce of the gate execution time by 40% as shown in Figure 5.

4) Thread-Level Parallelism

Thread-level parallelism is critical for achieving good scalability on multi-core systems. We parallelize the vector-matrix multiplication and the swap operations at thread-level using the *parallel_for* construct of the XPU parallel programming framework [27]. The XPU runtime detects dynamically the available processing units and distributes the workload over these units. Figure 5 shows the effect of 4 thread parallelism on single qubit gate performance reducing by half the execution time compared to the sparse multiplication.

B. Controlled Gate

Controlled single qubit gates reduce the number of operations by half for each control qubit. For instance, the CNOT and the Toffoli gates are respectively single and two qubits controlled Pauli-X gates which are widely used in many quantum algorithms. In their swap-based implementations, the number of swap operations can be reduced respectively by a factor of 2 and 4 and thus can reduce the simulation time.

C. Performance of Quantum Algorithms Simulations

In the following paragraphs, we compare the performance of our simulator to the $LIQUi|\rangle$ simulator through a set of popular quantum algorithms.

1) Quantum Fourier Transform

The Quantum Fourier Transform (QFT) is a fundamental part of many quantum algorithms, such as Shor's factoring algorithm [3] or the quantum phase estimation algorithm. The QFT is mainly composed of Hadamard gates and controlled phase shifts followed by a series of qubit swaps reversing the qubits. Figure 6 shows the achieved performance of the QX-simulator in comparison to the $LIQUi|\rangle$ simulator, our simulator achieves an average speedup of 14 times.

2) Entanglement Circuit

Quantum entanglement is a physical phenomenon that occurs when pairs of qubits interact in ways such that the quantum state of each qubit cannot be described independently.

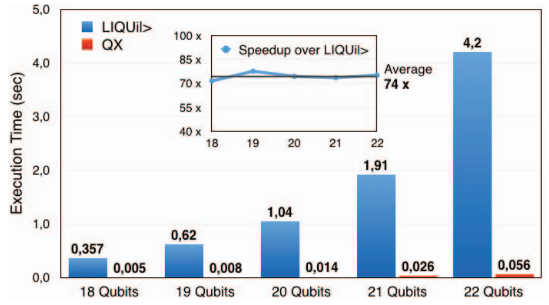


Fig. 7. Comparison of the simulation times of the Entanglement circuit by the QX-simulator and the $LIQUi|\rangle$ simulator.

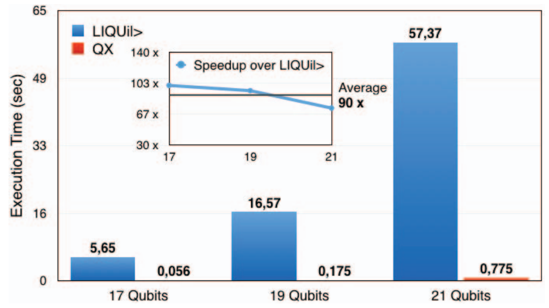


Fig. 8. Comparison of the simulation times of the Grover circuit using the QX-simulator and the $LIQUi|\rangle$ simulator.

The entanglement circuit is a common circuit in quantum information and is composed of a Hadamard gate and a set of CNOT gates followed by the qubits measurement. We compare the performance of the QX simulation of this circuit to the $LIQUi|\rangle$ simulator for different number of qubits. As shown in Figure 7, QX outperforms the $LIQUi|\rangle$ simulator and achieves an average of 46x speedup over it. We note that the $LIQUi|\rangle$ simulator compiles and optimizes the circuit before its execution, the compilation and optimization cost are included in the overall execution time.

3) *Grover Algorithm*

We implement Grover's search algorithm for different problem sizes (17,19 and 21 qubits). The simulation of the 21 qubits circuit requires the execution of 2388 gates including H,X,CNOT and Toffoli gates. We write the corresponding circuit using the QX's assembly code and the F# language used by $LIQUi|\rangle$. We compare the performance of the two simulation environments, Figure 8 shows that the QX-simulator outperforms $LIQUi|\rangle$ achieving a significant speedup of 95x for the 19 qubits circuit.

VIII. Conclusion

Quantum computer simulation exposes major challenges such as the significant memory and computation resources requirement and the design of a low level language to express quantum circuits. In this paper we introduced a fully functional assembly level language which allows the description of the quantum circuits and encapsulate their execution parameter. We described the implementation of the depolarizing noise model in our simulator and we showed how we can exploit the sparsity of the simulation data structures to reduce both the memory usage and the number of operations required for the simulation. Our benchmarks on various quantum circuits showed that our simulator achieves speedup ranging from 14 to 95 times over the state-of-the-art $LIQUi|\rangle$ simulator. Certain classes of quantum algorithms allow the development of additional specific optimizations which can further increase the speedup and diminish the memory requirements, allowing faster simulation of a larger number of qubits.

References

[1] R. P. Feynman, "Simulating physics with computers," *International journal of theoretical physics*, vol. 21, no. 6, pp. 467–488, 1982.
[2] P. Benioff, "Quantum mechanical hamiltonian models of turing machines," *Journal of Statistical Physics*, vol. 29, no. 3, pp. 515–546, 1982.
[3] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *SFCS*, 1994.
[4] R. L. Rivest *et al.*, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.
[5] L. K. Grover, "Quantum mechanics helps in searching for a needle in a haystack," *Physical review letters*, vol. 79, no. 2, p. 325, 1997.
[6] J.-M. Raimond *et al.*, "Manipulating quantum entanglement with atoms and photons in a cavity," *Reviews of Modern Physics*, vol. 73, no. 3, p. 565, 2001.
[7] C. Monroe *et al.*, "Demonstration of a fundamental quantum logic gate," *Physical review letters*, vol. 75, no. 25, p. 4714, 1995.
[8] J. R. Petta *et al.*, "Coherent manipulation of coupled electron spins in semiconductor quantum dots," *Science*, vol. 309, no. 5744, 2005.
[9] X. Fu *et al.*, "A heterogeneous quantum computer architecture," in *Computing Frontiers*, 2016.
[10] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge university press, 2010.
[11] A. de Vries, "jquantumquantum computer simulator," 2010.
[12] F. Schürmann, *Interactive quantum computation*. 2000.
[13] B. Butscher and H. Weimer, "libquantum. the c library for quantum computing and quantum simulation," 2013.
[14] M. Smelyanskiy, N. P. Sawaya, and A. Aspuru-Guzik, "qhipster: the quantum high performance software testing environment," *arXiv preprint arXiv:1601.07195*, 2016.
[15] D. B. Trieu, *Large-scale simulations of error prone quantum computation devices*, vol. 2. Forschungszentrum Jülich, 2009.
[16] B. Ömer, *Quantum programming in QCL*. na, 2000.
[17] H. Mlnarık, *Quantum programming language LanQ*. PhD thesis.
[18] K. M. S. Dave Wecker, "Liqui—¿: A software design architecture and domain-specific language for quantum computing." February 2014.
[19] E. o. Farhi, "Quantum computation by adiabatic evolution," *arXiv preprint quant-ph/0001106*, 2000.
[20] R. Raussendorf and H. J. Briegel, "A one-way quantum computer," *Physical Review Letters*, vol. 86, no. 22, pp. 5188–5191, 2001.
[21] A. Y. Kitaev, "Quantum error correction with imperfect gates," in *Quantum Communication, Computing, and Measurement*, pp. 181–188, Springer, 1997.
[22] D. Deutsch, "Quantum computational networks," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 425, no. 1868, 1989.
[23] D. Deutsch, "Quantum theory, the church-turing principle and the universal quantum computer," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 400, no. 1818, 1985.
[24] S. Lloyd, "Universal quantum simulators," *Science*, vol. 273, no. 5278, pp. 1073–1078, 1996.
[25] A. M. Steane, "Error correcting codes in quantum theory," *Physical Review Letters*, vol. 77, no. 5, p. 793, 1996.
[26] A. G. Fowler *et al.*, "Surface codes: Towards practical large-scale quantum computation," *Physical Review A*, vol. 86, no. 3, 2012.
[27] N. Khammassi *et al.*, "Mhpm: Multi-scale hybrid programming model: A flexible parallelization methodology," in *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication HPCC'12*, pp. 71–80, June 2012.
[28] E. Lieb and W. Thirring, "Inequalities for the moments of the eigenvalues of the schrödinger equations and their relation to sobolev inequalities," *Studies in Mathematical Physics: Essays in Honor of Valetine Bargmann*, pp. 269–303, 1976.