Connection Science

ISSN: (Print) (Online) Journal homepage: informahealthcare.com/journals/ccos20

Toward cost-effective quantum circuit simulation with performance tuning techniques

Nai-Wei Hsu, Chuan-Chi Wang, Chia-Hsin Hsu, Chia-Heng Tu & Shih-Hao Hung

To cite this article: Nai-Wei Hsu, Chuan-Chi Wang, Chia-Hsin Hsu, Chia-Heng Tu & Shih-Hao Hung (2024) Toward cost-effective quantum circuit simulation with performance tuning techniques, Connection Science, 36:1, 2349541, DOI: 10.1080/09540091.2024.2349541

To link to this article: https://doi.org/10.1080/09540091.2024.2349541

© 2024 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group.

Published online: 09 May 2024.

Submit your article to this journal

Article views: 273

View related articles

View Crossmark data

Full Terms & Conditions of access and use can be found at
https://informahealthcare.com/action/journalInformation?journalCode=ccos20

CONNECTION SCIENCE
2024, VOL. 36, NO. 1, 2349541
https://doi.org/10.1080/09540091.2024.2349541

OPEN ACCESS Check for updates

Toward cost-effective quantum circuit simulation with performance tuning techniques

Nai-Wei Hsu [a], Chuan-Chi Wang [a], Chia-Hsin Hsu [a], Chia-Heng Tu [b] and Shih-Hao Hung [a,c,d]

[a]National Taiwan University, Taipei, Taiwan; [b]National Cheng Kung University, Tainan, Taiwan; [c]Mohamed bin Zayed University of Artificial Intelligence, Abu Dhabi, United Arab Emirates; [d]National Center for High-performance Computing, Hsinchu City, Taiwan

ABSTRACT

Quantum circuit simulation is a popular approach to evaluating novel quantum algorithms before a physical quantum computer is available. Unfortunately, the simulation is often done with the *full-state* quantum circuit simulation scheme, and a huge memory space required by a full-state simulator is a limiting factor for the simulation of a larger qubit system. In this work, in order to support the simulation of a broadened qubit system, storage devices are introduced into the full-state quantum circuit simulation. A *vertical qubit simulation design* is proposed for the storage-based, full-state quantum circuit simulation, including qubit representation, threading model, and parallel state manipulations over storage devices. An empirical method of simulator parameter tuning is developed to achieve higher simulation performance. Our experimental results show that compared with the state-of-the-art memory-only simulator (QuEST), the storage-based simulation can achieve a 61x higher *cost-delay ratio* and can simulate a 39-qubit system on a commodity computer. The encouraging results indicate that our proposed simulator can help scale the full-state simulation for larger quantum circuits and achieve higher performance via the performance tuning method.

ARTICLE HISTORY
Received 18 July 2023
Accepted 25 April 2024

KEYWORDS
Quantum computing; quantum circuit simulation; parallel computing; performance tuning

1. Introduction

Quantum computing is a rapidly developing research field with substantial distinctions from classical computing. Quantum computers operate on the principles of quantum mechanics, in contrast to classical computers that follow boolean logics. In general, quantum machines can perform certain computations exponentially faster than their classical counterparts, a phenomenon referred to as quantum supremacy. This capability has the potential to revolutionise various day-to-day applications, such as cryptography (Xiao et al., 2021), financial modelling (Fu et al., 2022; Ren et al., 2023), and machine learning (Kalmet et al., 2020; Patel et al., 2018). Moreover, quantum computing can solve problems that

CONTACT Chia-Heng Tu chiaheng@ncku.edu.tw National Cheng Kung University, No.1, University Road, Tainan City 701, Taiwan (R.O.C.)

This article has been corrected with minor changes. These changes do not impact the academic content of the article.

© 2024 The Author(s). Published by Informa UK Limited, trading as Taylor & Francis Group.
This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited. The terms on which this article has been published allow the posting of the Accepted Manuscript in a repository by the author(s) or with their consent.

pose challenges for classical computers (Hu et al., 2023; P. Liu & Huang, 2022; Lu, 2021). This potential to change our everyday applications is driving the development of innovative quantum algorithms.

Recently, several industrial companies have entered the field of quantum computing. However, the current impracticality of these quantum computers arises from their limited scale and vulnerability to noise interference (Chow et al., 2021). In contrast, quantum circuit simulation (QCS) offers a promising alternative, demonstrating the capability to produce noise-free results and thereby mitigating the challenges associated with assessing and validating novel algorithms for quantum circuits.

Typical QCS, which employs the Schrödinger-style *full-state* simulation (Alexander et al., 2020; Jones et al., 2019; Smelyanskiy et al., 2016; Steiger et al., 2018) to produce intermediate results for validation and debug purposes, demands a substantial memory space during the simulations. This poses a significant challenge as the required memory space grows exponentially (Y. A. Liu et al., 2021; Raedt et al., 2007) with deeper quantum circuits or larger qubit sizes. For an N-qubit circuit, it requires $O(2^N)$ memory space to keep the states. In response to the demand, existing works rely on large-scale supercomputers for state storage. For instance, the Fugaku supercomputer (Okazai et al., 2020) comprises 158,976 machine nodes and 4.7 petabytes of memory to accommodate the simulations of 48-qubit circuits. Unfortunately, the supercomputer-based approaches place an entry barrier for average users, thereby impeding the promotion of the development and accessibility of quantum computing.

As a remedy for the growing memory space problem, we propose a storage-based QCS that adopts the full-state simulation and extends the memory space with secondary memory, such as solid-state drives (SSDs). This innovative storage-based solution can be deployed on commodity machines, leveraging the parallel computing power released by many-core processors. Additionally, as storage devices typically offer larger capacities than memories, our approach effectively extends the simulation scale. For example, while QuEST (Jones et al., 2019) estimates a qubit limit of 33-qubit on our hardware platform with 256GB of main memory, our storage-based simulator achieves the 39-qubit QCS with 16TB SSD on the same machine.

To further show that the storage-based simulator is a better design alternative, Table 1 compares the specifications and performance data between the typical memory-based and our proposed storage-based QCS, where the experiments are done on our hardware platform listed in Section 5.1, and QuEST is used to represent the state-of-the-art (SOTA), typical QCS. The bottom row of Table 1 indicates that the storage-based QCS is nearly 80 times more cost-effective than the memory-based alternative while the former is 0.4x to 0.7x slower than the latter (in terms of *delay* that is depicted as the averaged simulation time for the Hadamard gate). A longer delay introduced by the storage-based simulator is mainly attributed to a smaller bandwidth for accessing solid-state drives (that is nearly a quarter of the memory bandwidth). In order to further evaluate the combined impact of cost and delay, the cost-delay product (CDP) (Roloff et al., 2017) is used as a figure of merit to quantify the overall performance delivered by a QCS. The performance data shows that the cost-performance efficiency of the storage-based QCS is up to 61x better than memory-based solutions.

In this paper, we propose a systematic modelling approach to enable larger scale quantum circuit simulations on commodity computers equipped with storage devices. This

Table 1. Comparison of the tradeoffs between the memory- and storage-based QCS.

	Memory Capacity	Bandwidth (GB/s)	Cost (USD/GB)	Delay (ms/H gate)	Cost Delay Product (CDP)
Memory-based QCS over DRAM HX436C18FB3K2/64 (DDR4)	32 GB	**28.8**	7.1	**0.3** ∼ **305.6**	2.2 ∼ 2,191.1
Storage-based QCS over NVMe SSD SFYRD/2000G (PCIe 4.0)	**2 TB**	7.0	**0.09**	0.7 ∼ 398.4	**0.06** ∼ **35.8**
Ratio (Memory/SSD QCS)	0.015	4.1	79.6	0.4 ∼ 0.7	36.6 ∼ 61.1

Note: A lower CDP means a better cost efficiency for quantum circuit simulation.

modelling approach adopts a vertical qubit simulation design, taking into account characteristics from the underlying parallel computing hardware and file system software to the simulation of an *N*-qubit quantum system. This vertical design aims to address a challenging issue in the storage-based quantum circuit simulations. Specifically, when multiple simulation threads access quantum state files concurrently, it is crucial to strike a balance between higher computing power and reduced file I/O overheads. To achieve this balance, we develop a qubit representation for indexing quantum gate files and a threading model for throttling the computing power. Furthermore, we parameterise this qubit indexing mechanism and threading model to respect the characteristics of the vertical system stack. We provide a formal definition of the indexing mechanism and threading model, and offer a workflow for identifying a proper setting of the parameters. This enables the storage-based quantum circuit simulation to fully leverage the potential of the underlying parallel hardware. An additional module for profiling and logging is included to assist users in tracing simulation results and optimising the quantum algorithm when necessary. To the best of our knowledge, this architecture-aware vertical design, specially devised for the storage-based variant of quantum circuit simulations, is the first of its kind seen in the literature in this field since typical memory-based variants do not involve intensive file operations. The contributions achieved by this work are listed as follows.

(1) A novel QCS with storage devices is proposed, achieving higher cost efficiency than the memory-based alternative. To the best of our knowledge, the proposed solution is the first storage-based QCS featuring tunable parameters to enlarge the qubit simulation limit and pursue higher simulation performance. The proposed simulator does not rely on specialised hardware support, making it readily adoptable by average users. More about the uniqueness of our proposed work is discussed in Section 2.
(2) A vertical qubit simulation design is proposed for the storage-based, full-state QCS of an *N*-qubit system, including qubit representation, threading model, and parallel state manipulations over storage devices. The empirical performance tuning method for tweaking the simulator parameters is presented to facilitate the practical use of our proposed simulator on different hardware platforms.
(3) A profiling and logging module is embedded in our proposed full-state simulator to capture the performance statistics of simulated quantum gates, including counts of simulated gates and the elapsed time of each simulated gate. An illustration of the collected performance data is presented in Section 3.1.

2. Background and related work

This section briefly introduces the essential information of full-state quantum circuit simulations, including the data structure used to represent a quantum state (Section 2.1) and the mathematical representation of gate operations for the simulations (Section 2.2). Furthermore, in order to further model the noises, the density matrix representation is introduced in Section 2.3. Tensor network states are another way to represent quantum states and an important variant is matrix product states, which is introduced in Section 2.4. The existing full-state simulators and the comparison of our work to the existing work are presented in Section 2.5.

2.1. State vector

A state vector is a mathematical construct to represent a quantum state in a physical system, and it is also known as a wave function. A state vector contains the complete information about a quantum system, and it can be used to compute the probabilities of the possible outcomes measured in the system.

In the context of an N-qubit quantum system, the state vector is typically denoted as $|\Psi\rangle = \Sigma_{i=0}^{2^N-1} \alpha_i |i\rangle$, where α_i represents the probability amplitude of state $|i\rangle$. It can be represented as a column vector in a Hilbert space, which is a complex vector space endowed with an additional mathematical structure for describing quantum systems. When a measurement is performed on a quantum system, the state vector falls down to one of the possible measurement outcomes. The measurement process can be defined mathematically by a projection operator, which can be expressed as a matrix projecting the state vector onto the measurement outcome.

2.2. Gate operation

A gate operation is a building block of quantum circuits. Specifically, a quantum circuit consists of a sequence of gate operations and qubit data, where the operations are performed on the data across different qubits to accomplish a high-level application logic. From the mathematical point of view, a gate operation, which can be represented as a matrix operation, is a tensor product performing on one or more qubits and changes the quantum states of the qubits. For instance, a single-qubit gate operation G applied on the jth qubit of the state vector can be denoted as $G_j|\Psi\rangle := I_{N-1} \otimes \cdots \otimes I_{j+1} \otimes G \otimes I_{j-1} \otimes \cdots \otimes I_0 |\Psi\rangle$, where G_j is an $2^N * 2^N$ matrix multiply on the state vector. In a general view, this matrix multiplication can be represented as follows.

$$\begin{bmatrix} \alpha'_{\star\cdots\star 0_j \star\cdots\star} \\ \alpha'_{\star\cdots\star 1_j \star\cdots\star} \end{bmatrix} = G \begin{bmatrix} \alpha_{\star\cdots\star 0_j \star\cdots\star} \\ \alpha_{\star\cdots\star 1_j \star\cdots\star} \end{bmatrix}$$

where ★ refer to any bit value when represent index i of α_i in binary representation. Furthermore, when considering a two-qubit unitary gate U_{jk} for updating the state vector on

both the jth and kth qubits, the matrix form can be rewritten as follows.

$$\begin{bmatrix} \alpha'_{\star\cdots\star 0_j \star\cdots\star 0_k \star\cdots\star} \\ \alpha'_{\star\cdots\star 0_j \star\cdots\star 1_k \star\cdots\star} \\ \alpha'_{\star\cdots\star 1_j \star\cdots\star 0_k \star\cdots\star} \\ \alpha'_{\star\cdots\star 1_j \star\cdots\star 1_k \star\cdots\star} \end{bmatrix} = U_{jk} \begin{bmatrix} \alpha_{\star\cdots\star 0_j \star\cdots\star 0_k \star\cdots\star} \\ \alpha_{\star\cdots\star 0_j \star\cdots\star 1_k \star\cdots\star} \\ \alpha_{\star\cdots\star 1_j \star\cdots\star 0_k \star\cdots\star} \\ \alpha_{\star\cdots\star 1_j \star\cdots\star 1_k \star\cdots\star} \end{bmatrix}$$

2.3. Density matrix

A density matrix is another form to represent the quantum state. It is a Hermitian, positive semi-definite matrix for characterising the state of a mixed quantum state, which is represented by a weighted sum of pure states and the weights correspond to the probabilities of finding the system in each of the pure states. Given a set of N-qubit quantum systems with a state vector $|\Psi_s\rangle = \Sigma_{i=0}^{2^N-1}\alpha_i|i\rangle$, the corresponding density matrix can be defined as $\rho_s = |\Psi_s\rangle\langle\Psi_s|$ describing the same quantum system. The key advantage of the density matrix is that it can not only be used to describe systems that are not in pure states, such as systems that are subject to decoherence or other forms of environmental noise, but help to investigate entanglement properties of composite quantum systems. To apply a gate as the view of the density matrix, it can be leveraged by the Choi-Jamiolkowski isomorphism (Choi, 1975) as follows.

$$\begin{aligned} \rho' &= \Sigma_s p_s G_i(\Sigma_{j=0}^{2^N-1}\Sigma_{k=0}^{2^N-1}\alpha_{j,k,s}|j\rangle\langle k|)G_i^\dagger \\ &\rightarrow \Sigma_s p_s G_{i+N}^\dagger G_i(\Sigma_{j=0}^{2^{2N}-1}\alpha_{j,s}|j\rangle) \end{aligned}$$

2.4. Matrix product state

A matrix product state uses *tensors* to characterise quantum gates and their states. The product of N tensors represents the quantum states of an N-qubit system. Each tensor contains an external index s and the bond indices denoted by α with a sufficient dimension as follows.

$$|\Psi\rangle = \sum_{\alpha} A^{s_1}_{\alpha_1} A^{s_2}_{\alpha_1\alpha_2} \ldots A^{N}_{\alpha_{N-1}}$$

A one-qubit gate is a tensor with two external indexes for the quantum gates of the desired operations. A jth target qubit of the gate represents the operations contracts an external index of the gate tensor and the jth external index of the quantum state tensor. In this case, the contraction has negligible overhead without any change in the dimension of bond indices. Taking a two-qubit gate as an example, we assume that the two-qubit gate is performed on the nearby qubit j and $j+1$, and two tensors: $A^{s_j}_{\alpha_j\alpha_{j+1}}$ and $A^{s_{j+1}}_{\alpha_{j+1}\alpha_{j+2}}$ are contracted with the gate tensor. Then, the tensor performs the singular value decomposition to split the tensor with two external indices back to the product of two tensors with one external index. This process would increase the dimension of bond indices, leading to the expansion of the tensor and the storage space.

As the entanglement increases with the growth of the qubit number, the accuracy of the matrix product state could be degraded, and the dimension of bound indices could exhibit exponential growth as the state vector. Despite these drawbacks, the matrix product state

is one of the popular approaches for compressing the storage space required by quantum computations.

2.5. Related work

Many quantum circuit simulators have been developed to facilitate quantum computing using classical computers. Among other data representations employed in quantum circuit simulations, the state vector-based approach is a favored choice as it can track quantum states throughout the simulation (this is referred to as the full-state simulations). Traditionally, full-state simulators rely on the main memory to store these state vectors, but the size of the available memory limits the scale of quantum circuit simulations. As a remedy, a storage-based scheme has been recently proposed to handle larger-scale quantum circuit simulations on a single computer. The following subsections introduce the memory- and storage-based simulators, as well as the tools that offer support for the understanding, analysis, and testing of quantum circuits. To distinguish our proposed simulator from existing literature, the differences between our work and the state-of-the-art work are highlighted.

2.5.1. Memory-based quantum circuit simulators

Numerous memory-based quantum circuit simulators have been developed to aid in the design, testing, and debugging of quantum algorithms using classical computers. Current efforts of the simulations fall into two broad categories: typical quantum computing and hybrid quantum-classical computing. The former concentrates on developing quantum algorithms for quantum computers, whereas the latter harnesses the strengths of both classical and quantum computers to tackle intricate problems, such as materials science, drug discovery, financial modelling, and artificial intelligence.

Simulators for typical quantum computing. qsim (team & collaborators, 2020) is a full-state quantum circuit simulator using state vectors to keep quantum states during the simulation. It follows the Schrödinger algorithm to simulate quantum gates, which is referred to as a gate-by-gate simulation scheme in this work. To boost simulation performance, it incorporates the optimizations of the underlying CPU, including gate fusion, CPU multithreading with OpenMP, and vector operations (e.g. AVX instructions). The integration with Cirq simplifies the task of performing quantum circuit simulations with qsim.

qHiPSTER (Smelyanskiy et al., 2016), which is short for Quantum High Performance Software Testing Environment, is a quantum circuit simulator that can take advantage of distributed computation to overcome the memory size limitation (for keeping the quantum states). qHiPSTER is able to work on supercomputers, such as the Stampede supercomputer. Different optimizations have been implemented to accelerate the simulation speed, including vectorisation, multi-threading, cache blocking, and overlapping communication with computation.

Qulacs (Suzuki et al., 2021) is a C++/Python library for quantum circuit simulations with and without noisy modelling. The library has been built with features for fast simulations, such as circuit compression, CPU multithreading support, and GPU multithreading support. Quantum++ (Gheorghiu, 2018) is a C++ library for quantum computing for multicore CPUs. Similar to other full-state-based simulators, Quantum++ is able to simulate the evolution of quantum qubits in a pure state. To accelerate the simulation time, a mixed-state approach can be adopted by using density matrices to represent the quantum states.

ProjectQ (Steiger et al., 2018) is an open-source software framework. It has a compiler framework with different back-ends for various purposes, such as running quantum circuits on quantum hardware/simulator and drawing circuit diagrams. Furthermore, it also includes a quantum circuit simulator that is able to help evaluate quantum algorithms (less than 30 qubits) on a personal computer. The Quantum Exact Simulation Toolkit (QuEST) (Jones et al., 2019) is one of the fastest quantum circuit simulators, and it supports different representations of the quantum states, such as state vectors and density matrices. It has CPU/GPU multithreading support on a single machine node and can also run in a distributed environment with multiple computer nodes.

Cirq (Developers, 2021) is a Python-based library developed by Google Quantum AI to facilitate quantum computing in the noisy intermediate-scale quantum (NISQ) era. The developed quantum programs are able to run on a variety of quantum hardware and simulators, especially on Google's quantum computing services. Cirq is capable of working with various tools to explore the potential of quantum computing. For example, the Cirq programming framework is able to incorporate research libraries and tools, such as PennyLane and TensorFlow Quantum. It can also link to the development tools, such as QUEKO (Tan & Cong, 2020), and high-performance quantum circuit simulators, such as qsim, Qulacs, and cuQuantum.

cuQuantum (cuQuantum development team, 2023) is a software development kit developed by NVIDIA for quantum computing. It provides different front-ends for programmers, including Python and C++. Two crucial components are provided for C++/Python users: cuStateVec for state vector computations and cuTensorNet tensor network computations. The cuQuantum implementation is able to take advantage of multi-GPU hardware on a single machine node or multiple machine nodes (a distributed environment). It is adopted by several quantum circuit simulators for GPU accelerations, such as Cirq, QuEST, Qiskit, and qsim.

Simulators for hybrid quantum-classical computing. Qiskit (Alexander et al., 2020) is a software development kit developed by IBM Quantum. Similar to Cirq, Qiskit provides circuit libraries for quantum programming, a transpiler to translate quantum code into optimised circuits to run on various quantum hardware/simulators. In addition to IBM's quantum computers and simulator (Aer), Qiskt has the back-ends to generate the circuit for other quantum computers, such as AQT and IonQ systems. For developing quantum applications, Qiskit offers tools for quantum machine learning (e.g. integrating with PyTorch), optimisation, and chemistry.

PennyLane (Bergholm et al., 2022) is a Python framework facilitating the development and optimizations of quantum and quantum-classical algorithms. It incorporates popular machine learning frameworks, such as PyTorch and TensorFlow, for quantum-aware operations. For example, PennyLane extends the capabilities of automatic differentiation techniques from these frameworks for performing gradient descent operations that are associated with quantum computations. It also provides several quantum simulators to facilitate the development of hybrid quantum-classical algorithms.

With the aim of efficient simulations of quantum-classical algorithms, TensorCircuit (Zhang et al., 2023) is a Python-based quantum software framework built on top of machine learning frameworks TensorFlow, PyTorch, and Jax. It provides support for quantum hardware and simulator and offers hybrid deployment choices among CPU, GPU, and

QPU. The TensorCircuit simulator is based on tensor network contraction and comes with support for CPU/GPU multithreading.

2.5.2. Storage-based quantum circuit simulators

The available main memory bounds the full-state quantum circuit simulations, as the required memory space increases exponentially with the number of simulated qubits. In order to meet the demand for memory space for more extensive scale simulations, a common strategy is to distribute the quantum states across different machine nodes (K. De Raedt et al., 2007; Häner & Steiger, 2017; LaRose, 2018; Niwa et al., 2002; Trieu, 2009). This distributed approach can be built upon commodity computers or supercomputers.

Alternatively, a storage-based simulation scheme has been proposed to take advantage of the secondary memories to scale the quantum circuit simulations at a lower cost (Chung, 2022; Park et al., 2022), compared with the memory-based distributed simulation scheme. SnuQS (Park et al., 2022), a simulator developed in parallel to our proposed one (Chung, 2022), demonstrates that the hard disk drives and solid-state drives (SSDs) can be utilised to store the states of a larger-scale quantum circuit simulation (e.g. 42 qubits). The quantum states are separated into sub-states spread across the SSDs. Like the existing memory-based simulators, SnuQS is built with the support for CPU/GPU multithreading. It also proposes a sub-circuit partitioning algorithm to lower the frequency of I/O accesses to the underlying RAID-0-enabled storage devices.

2.5.3. Supplement tools

Various types of tools are developed to facilitate quantum programming and performance analysis, such as visualisation toolkit statistical analysis methods (Suau et al., 2021; Wu et al., 2020), and debugging tools (Huang & Martonosi, 2019; Metwalli & Meter, 2022). The visualisation tools depict the quantum circuit organisation specified in a given quantum program (or a description file) to provide a better understanding of the input program. Sometimes, the visualisation tools provide the user interfaces to interactively edit the visualised circuits and convert the edited version into the circuit format for further execution (simulation). To provide comprehensive support for quantum programming and simulation, a quantum circuit simulator could integrate visualisation tools, statistical analysis modules, and debugging tools to facilitate quantum computing.

2.5.4. Comparison with the SOTA work

This work proposes a storage-based scheme to scale quantum circuit simulations. The scheme enables the simulation of larger-scale quantum systems (e.g. a 39-qubit system) on a single classical computer. Thus, average quantum program developers are able to develop quantum algorithms at a lower cost with a commodity computer. The advantage of using a storage-based simulation scheme is further provided in Section 5.5. Furthermore, our work was done initially in 2022 (Chung, 2022), which makes it one of the pioneers works for storage-based simulations, alongside SnuQS. While our work shares the foundational concept of utilising storage devices for larger-scale simulations with SnuQS, it significantly diverges in its focus on hardware independence and optimisation strategies, which are detailed in the following paragraphs.

Hardware independence. SnuQS requires hardware-assisted RAID-0 support, where a state datum is divided into segments that are distributed across the RAID-0 storages,

enhancing the I/O subsystem's throughput. In contrast, our simulator does not require any specific hardware support and offers better per-gate simulation efficiency. This is because each state datum can be independently accessed from a storage device by a simulation thread. The performance comparison is further presented and discussed in Section 5.2.

Optimization strategy. SnuQS focuses on the algorithmic-level optimizations of an input quantum program for simulation. This includes a circuit optimiser to refine the input quantum circuit before simulations and a sub-circuit partitioning to reduce the I/O commands issued to the I/O devices. Conversely, our work employs a *vertical qubit simulation design* ranging from qubit state representation, thread model (Section 3.2), and data allocation for parallel simulation (Section 3.3), to parallel I/Os (Section 4.1) and I/O latency reduction (Section 4.3). The corresponding optimisation strategy is proposed for the storage-based design. Furthermore, we offer a methodology to fine-tune the overall simulation performance, considering the various configurations available for the storage-based quantum circuit simulations (Section 4.4).

Our work and SnuQS address different aspects of the storage-based quantum circuit simulations, making them complementary technologies. Integrating SnuQS optimizations into our simulator would further improve the simulation performance. For instance, when the massive multithreaded simulation is bounded by I/O accesses, especially for a larger-scale quantum system, our direct I/O scheme can be applied to shorten the I/O latency further.

3. SSD-based quantum circuit simulation

An overall architecture for the storage-based quantum circuit simulation is outlined in Section 3.1. Section 3.2 presents our proposed indexing mechanism, including formal qubit representation for parallel simulation, the parameters for the indexing mechanism, a workflow to determine the indexing parameters, and an example to better display the concept of the mechanism. Section 3.3 describes the parallel simulation paradigm, both inherited from the qubit representation.

3.1. Architecture

The three-layer architecture of the proposed quantum circuit simulator is illustrated in Figure 1. The simulator accepts the user inputs to perform the quantum circuit simulation, and the quantum states are kept in the storage devices and buffered by the memory. The three layers are described as follows.

(1) ***User inputs.*** The inputs to the simulator include a quantum circuit file and a configuration file. The quantum circuit file can be easily transformed from the popular OpenQASM format (Cross et al., 2017) to specify the involved quantum gates in the circuit file by our open-source code. The configuration file provides the settings for the simulations, without the recompilation of the simulator to take effect. In particular, the configuration file can be used to turn on/off the noise model, to set up the simulation parameters (e.g. number of threads), to enable/disable the performance profiling during the simulation, and to log the execution status/intermediate states during the simulation.

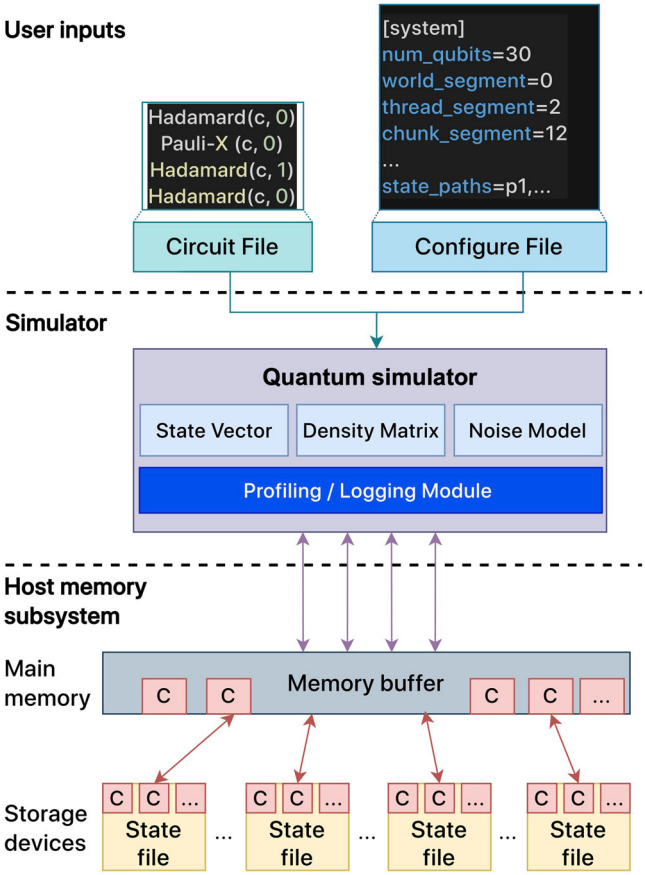

Figure 1. The architecture of the quantum circuit simulator.

(2) ***Simulator.*** The proposed simulator adopts the Schrödinger-style algorithm to perform a gate-by-gate simulation (H. De Raedt et al., 2019; Raedt et al., 2007). Given an N qubit circuit, the simulator uses 2^N complex amplitudes in the storage devices, and these state data are retrieved when needed. They are buffered in the main memory and updated in place during the gate-by-gate simulation. A matrix representation to represent state vectors or density matrix is used to facilitate the simulation. The matrix representation helps the traversing of quantum programs since it provides a clear view of quantum states.

A profiling/logging module is built within the simulator to analyse the simulation performance of the simulated quantum gates. The module is able to collect dynamic performance data, such as the simulated gate sequence, the counts of simulated gates, and the simulation time for each gate. Later, the quantum circuit optimisation(s) can be judiciously performed based on the collected dynamic performance data. This module can further broaden the optimizations to be performed, as a complement to the static optimizations performed by typical quantum compilers. The example output of the profile data is shown in Figure 2 for the simulation of the H-CNOT-H-X gate sequence.

```
Qubit 0: H-CNOT-H-X
[H]:    Freq: 2 times; Avg.: 93.5 us
[X]:    Freq: 1 time;  Avg.: 70.0 us
[CNOT]: Freq: 1 time;  Avg.: 86.2 us
[H-CNOT]: Freq: 1 time; Avg.: 184.4 us
[CNOT-H]: Freq: 1 time; Avg.: 175.0 us
[H-X]:    Freq: 1 time; Avg.: 163.5 us
[H-CNOT-H]: Freq: 1 time; Avg.: 273.2 us
[CNOT-H-X]: Freq: 1 time; Avg.: 245.0 us
-----
Qubit 1: H-CNOT
[H]:    Freq: 1 time; Avg.: 87.5 us
[CNOT]: Freq: 1 time; Avg.: 83.2 us
```

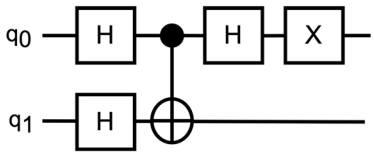

Figure 2. An example output of the profiling data offered by our simulator.

(3) ***Host memory subsystem.*** The states of qubits are stored in the *state files* in the storage devices. These files are opened as ordinary files and are accessed via the standard Linux I/O system calls. Memory buffers are allocated by the simulator for buffering the qubit states for further manipulations. Threads are created with the OpenMP library in the simulator to manipulate the quantum states in parallel. The basic units for content updates of the qubit state are called *chunks*, which are transferred back and forth between the secondary memory and the main memory, as illustrated in Figure 1. The following subsections introduce the method used for partitioning the qubit states (e.g. into chunks) and the scheme for parallel qubit state processing.

3.2. System modelling

The key aspect for architecture-aware parallel quantum circuit simulations with SSD storages is an indexing methodology of $2^{\mathcal{N}}$ quantum states. This approach is similar to the application of dynamic hashing for persistent memory (Nam et al., 2019), which aims to establish a tunable data transfer size between the main memory and SSD. The indexing parameters can be specified by the configuration file described in Section 3.1 for starting the quantum circuit simulations. In the following paragraphs, the indexing mechanism is introduced first, together with an associated threading model. Next, the workflow for determining the indexing parameters is presented. Last, an illustrative example is given to better present the concept of our indexing mechanism.

Qubit indexing mechanism. To simulate an N-qubit quantum system, three segment parameters, F, M, and C, are used to index the 2^N quantum states. The N-qubit quantum system is run by 2^T simulation threads. The relevant descriptions are provided in Table 2. The relationship between the three parameters and N qubit quantum states is defined by Equation (1), where the three segment indexes can point to an N-qubit quantum state.

$$N = F + M + C \tag{1}$$

The methodology to determine proper values for T, F, M, and C is described as follows. Especially, these values are changed according to the scale of the quantum circuit simulation, i.e. the size of N, and the available hardware resource for parallel simulations, i.e. the number of hardware threads.

Table 2. The parameters used by the proposed quantum circuit simulator.

Parameter	Description
N	The total number of qubits to be simulated.
F	The number of bits for file segment, representing a total of 2^F state files.
M	The number of bits for middle segment, representing a total of 2^M chunks.
C	The number of bits for chunk segment, representing a total of 2^C quantum states.
T	A total number of 2^T threads for quantum circuit simulations.

Table 3. Configuration of hardware and software for experiments.

Name	Description
CPU	AMD Ryzen Threadripper PRO 3995WX 64-Cores@2.7 GHz(64C/128T)
RAM	HX436C18FB3K2/64, 256 GB(8 × 32 GB) DDR4 3600 MHz
Storage	Kingston SFYRD/2000 G, 16 TB (8∗2 TB on PCIe 4.0 bus)
OS	Ubuntu 20.04 LTS(kernel version 5.15.0-58-generic)
Compiler	GCC version 9.4.0
API	OpenMP version 4.5

- *The value of T* is determined by underlying classical processors (CPUs) for quantum circuit simulations, as defined in Equation (2). For instance, given sixty-four physical cores in the system, as our experimental environment listed in Table 3, T should be set equal to (or larger than) six to maximise utilisation of the processor with 2^6 cores. A larger T is possible, for example, $T > 6$ that could be enabled by turning on the hyperthreading feature of the manycore processor. However, a larger T would further push the I/O bandwidth since each thread is responsible for accessing its assigned quantum states from storage devices. Empirical analysis is needed to determine a proper value for a balance between the computing rate and the data transmission rate. An empirical workflow is presented below, and the experimental results of finding good settings are presented in Section 5.3.

$$T = \begin{cases} HWThreadsNum, & \text{if } N \geq HWThreadsNum \\ N, & \text{otherwise} \end{cases} \tag{2}$$

- *The value of F* determines the number of quantum state files used in the simulation, 2^F. Moreover, F is associated with T to conform to our arrangement: *one-quantum-state-file-per-thread*, which is a computer architecture dependent threading style based on data parallel model. As will be presented in Section 4.1, we find that a state file dedicated to a simulation thread delivers the best performance on the Linux file system. In this case, the hardware specifications, precisely the number of physical cores, can serve as a reference for the quantum state indexing (F). This means that the value of F should be equal to T to minimise system overhead. Each thread is tasked with updating the quantum states stored in its respective state file when the effect of a quantum gate is simulated in parallel. To adapt to the quantum system with a varying value of N, the equation below defines F.

$$F = \begin{cases} T, & \text{if } N \geq T \\ N, & \text{otherwise} \end{cases} \tag{3}$$

- *The values of C and M* are defined in Equations (4) and (5), respectively. C is associated with the size of the quantum states to be transmitted in each system call for file accessing,

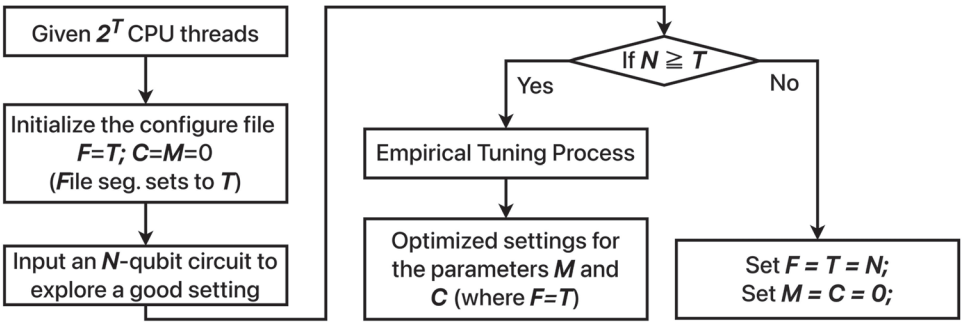

Figure 3. An empirical workflow to determine proper settings for the *F*ile, *M*iddle, and *C*hunk segments for an *N*-qubit system whose representation is shown in Figure 4.

where the size is denoted as 2^C. *M* implies the number of data transfers required for each file, expressed as 2^M. Under the *one-quantum-state-file-per-thread* execution style, the time required by each single file I/O operation has an important influence on the simulation performance, since the simulation engine cannot proceed with the quantum state updating process without the required quantum states. Hence, the value of *C* should be determined first before we decide the value for *M*. Furthermore, the performance of a file I/O operation is highly dependent on the organisation and implementation of the file system, for example, the *page cache* mechanism and the data block size of a file. A clearer delineation is provided in Section 4.2. In order to deliver the best simulation speed, an empirical search is necessary to determine a proper *C* value, and is presented in the following paragraph.

$$C = \begin{cases} EmpiricalSearch(C), & \text{if } N \geq T \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

$$M = \begin{cases} N - F - C, & \text{if } N \geq T \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

Parameter value search. The parameter tuning process for determining the partition of an *N*-qubit is illustrated in Figure 3. Given a manycore processor platform with 2^T threads to simulate an *N*-qubit system, if $N < T$ (suggesting that the computing power exceeds the necessary computation), the parameters *F* and *T* should be set to *N* while *M* and *C* are set to zero. This means that a simulation thread is assigned to a state file to handle each *targ* qubit, and the number of active simulation threads is equal to 2^N. If $N \geq T$, *F* is set to *T* as previously stated, and the values of *C* and *M* should be determined through an empirical process that explores all combinations of *C* and *M* to find the best settings.

An illustrative example. A quantum state of an *N*-qubit system can be accessed through the above indexing mechanism. An illustrative example of a 11-qubit data is depicted in Figure 4. The first three qubits in *F*, belonging to the file segment, form the state file name. The following three qubits are the *M* segment served as the chunk index further point to a specific chunk. The rest of the qubits in *C* represent the offset to the exact quantum state;

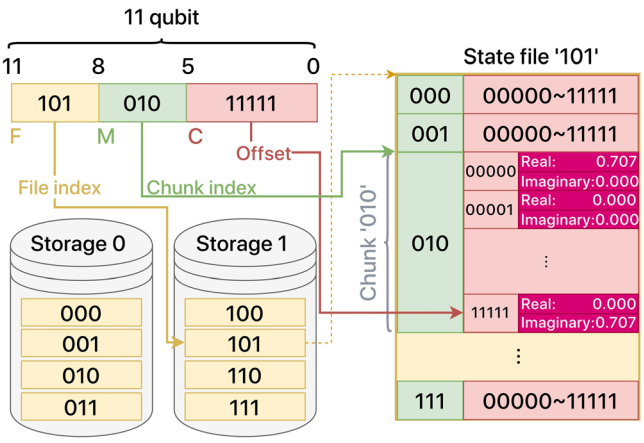

Figure 4. An example of the representation of qubits, where the 11-qubit is partitioned into three segments, **F**ile, **M**iddle, and **C**hunk segments that can be used to point to the corresponding quantum state stored in the state file.

a quantum state is represented by a complex amplitude with the real and imaginary numbers. The number of quantum states accessed by a simulation thread is 2^C. If a complex amplitude is stored as two eight-byte double-precision numbers, the size of a chunk segment is 2^{C+4}. To facilitate the parallel execution on the different host environments for the best performance, the size of the three segments *F*, *M*, and *C* can be set up through the configuration file of our simulator after their values are determined by the above process.

3.3. Qubit representation for parallel simulations

The partitioning of an *N*-qubit into the three segments facilitates the accessing of the quantum states by the parallel threads. Using a single qubit gate simulation as an example. When threads attempt to do a gate operation, they encounter one of the following three situations according to the position of the *targ*. Additionally, as will be explained in Section 4.2, each thread has access to a distinct state file, which leads to the best performance. An algorithm should be developed to automatically identify the matched file for each thread, which is done by Algorithm 1 and is described later.

Algorithm 1 The pairing algorithm is proposed to determine corresponding pairs of the basis vectors for each thread.

```
uint32_t targShift ← F-(N-targ);
uint64_t targSegment ← 1 << targShift;
base ← tIdx + ((tIdx >> targShift) << targShift);
correspond ← (base) ⊕ targSegment;
```

targ falls within a middle segment. The states to be referenced by the simulator are at the positions of a middle segment. As the corresponding pairs of basis vectors are distributed across different chunk segments within the same state file, the contents of the related chunk segments are buffered in the allocated memory regions for the simulation

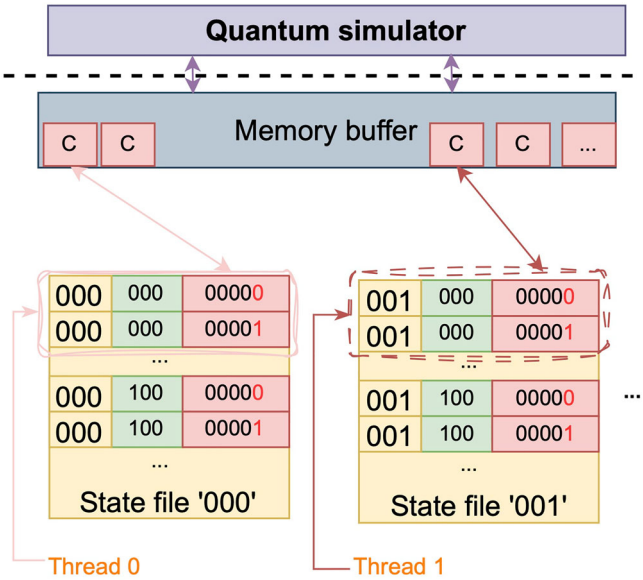

Figure 5. For the 11-qubit system as an example, the qubit representation and the corresponding accesses to the qubit states by the threads, where *targ* is within a middle segment.

need. As shown in Figure 5, Thread 0 is required to copy chunks with bit strings "000" and "100" to memory, as indicated by the *targ* parameter.

targ falls within a file segment. The target qubits are at the positions of a file segment, which involves the accesses across different state files. The related states of the dispersed chunk segments should be brought to the memory buffer first before they can be updated. When accessing the chunks across different files, it is important to orchestrate the thread accessing chunks in order to avoid the lock-step simulation among threads (for accessing chunks in the same file). Figure 7 illustrates that both thread 0 and thread 1 need to access the chunks within the two files, namely "000" and "001". Thread 0 is responsible for handling the chunks of the middle segment of "000", whereas thread 1 is for the chunks within the middle segment "100". The above assignment scheme is able to assign the proper file index of each thread without the need for additional thread management or thread barriers since the Linux file system is responsible for handling the access order of the chunks within the same state file. This assignment scheme is achieved by Algorithm 1, which is introduced as follows.

Algorithm 1 is used by each simulation thread for calculating the index of the state file (e.g. "000" for thread 1 in Figure 7, where the first half-paired of state vectors is located, based on the *tIdx* (the identifier of each thread), *targ*, and bit shifting operations (lines 1–3). This algorithm is achieved by using binary numbers to index the files and threads. In this case, each simulation thread can calculate the file location for its own use by itself, avoiding the need for thread communications/synchronisations. Similarly, the second-half state vectors can be obtained by using the previously calculated intermediate value along with an XOR operation (line 4). It is noteworthy that in situations where a middle segment exists, necessitating the division of *tidx* by 2 to make thread access the file in pairs. For further

information, kindly consult the open-source code. Lastly, by utilising the proposed segmentation and algorithms, the simulator can effectively execute diverse analyses and extend its functionality to include additional qubit gates such as CNOT, Toffoli, and others in a prompt manner.

4. Implementation remarks

In comparison to the memory-based quantum circuit simulations, which rely heavily on the virtual memory systems supported by underlying operating systems, the storage-based simulations involve file reads/writes that are more sophisticated than simple memory reads/writes since the secondary memory usually has a higher accessing latency and various technologies have been developed in order to achieve high performance for file operations. The following subsections give the design considerations that are critical to the development of a high-performance storage-based quantum circuit simulator. In particular, the parallel I/O operations and the file system support for a high-performance file I/O are discussed in Section 4.1. The configuration of the file system organisation would affect the delivered simulation performance and is discussed in Section 4.2. The direct I/O support is discussed in Section 4.3 to further shorten the data accessing latency when the I/O bandwidth becomes the system bottleneck (it happens when simulating a large qubit system). The configuration of the three segments is vital to the simulation performance and is discussed in Section 4.4.

4.1. Parallel I/O reads and writes

The parallel quantum circuit simulation is facilitated largely by the parallel IO operations done between the main memory and the storage devices, as shown in Figures 6 and 7,

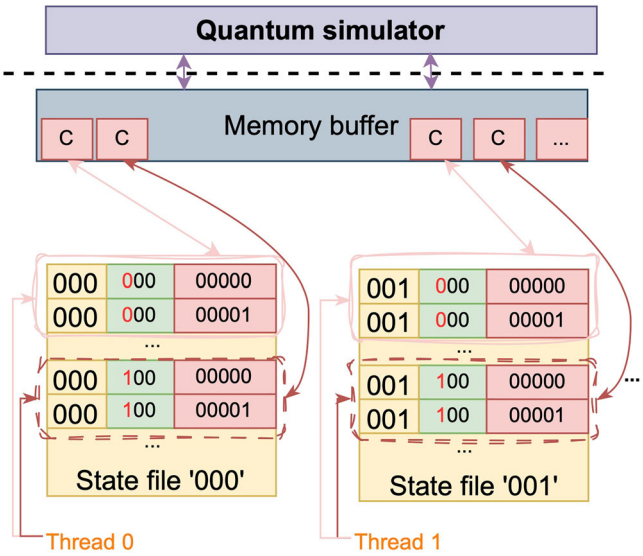

Figure 6. For the 11-qubit system as an example, the qubit representation and the corresponding accesses to the qubit states by the threads, where *targ* is within a chunk segment.

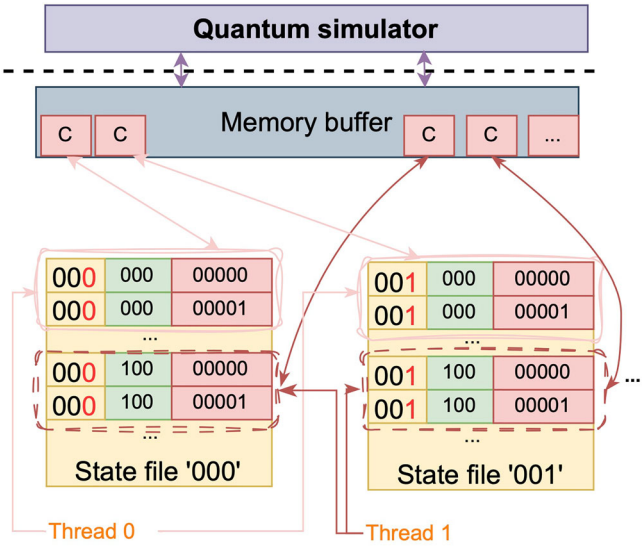

Figure 7. For the 11-qubit system as an example, the qubit representation and the corresponding accesses to the qubit states by the threads, where *targ* is within a file segment.

where the states of qubits are kept in the state files and the main memory is served as the buffer for the quantum circuit simulator to operate on. Instead of using the read/write functions provided by the standard C library, such as `fread` and `fwrite`, the Linux standard system calls `pread` and `pwrite` are used to handle the accesses to the state files in the multithreaded environment. While accessing to these files, their file contents are buffered in the *page cache* maintained by the underlying operating system to shorten the data access time. However, the access time degrades when the data size grows larger than that of the page cache. In this case, extra efforts should be made, as will be discussed in Sections 4.3 and 5.2.

The system calls can access the same file concurrently, thanks to their thread-safe support. Nevertheless, the implicit synchronisation mechanism is implemented to ensure the correctness of the concurrent accesses to the same file, which hinders the delivered performance of the parallel simulation. Therefore, as shown in the previous section, our design assigns a file to a single thread whenever possible to alleviate the synchronisation overhead. The experimental results are presented and discussed in Section 5.3.

4.2. File access efficiency

The qubit states are kept in the state files. While doing the simulation, the qubit states should be pulled out of the state files into the memory buffers, or should be pushed into the storage devices. The data blocks of the to-be-processed file kept in the file system are either pulled or pushed from or to the storage devices. In Linux systems, where the fourth extended file system *Ext4* is considered as a default file system in Linux systems, the addresses to the data blocks are stored in the *inode* data structure and the address pointers (known as *extents*) refer to a range of contiguous physical blocks in the storage devices. Based on our experimental analysis, a small inode size tends to decrease the delivered I/O

performance. This is because (1) more inodes are created to index the actual file contents, which decreases the storage device size allocated for the file data; (2) the *extent tree* traversal is more frequent to find the physical block addresses. Hence, a larger inode size is desired to provide a larger storage space for file contents and less extent tree traversal time. To this end, the two options `largefile` and `largefile4` can be used to enlarge the inode size into 1 and 4 MiB, respectively.

4.3. Direct IO

The virtual file system (VFS) of Linux systems provides a unified interface for user-space file I/O operations without considering the heterogeneity of the underlying I/O devices. The VFS exposes different interfaces to user-space applications for file I/O. For example, a standard I/O interface leverages the buffering/caching mechanisms in between a user-space application and storage hardware (left of Figure 8), whereas a direct I/O interface (right of Figure 8) reduces the caching mechanism in the hope to provide a shorter data accessing latency (especially for an I/O bound system) and to avoid potential data consistency issue. The selection of a standard I/O or a direct I/O for handling the file operations is controlled by the flag `O_DIRECT` as the parameter when opening the files.

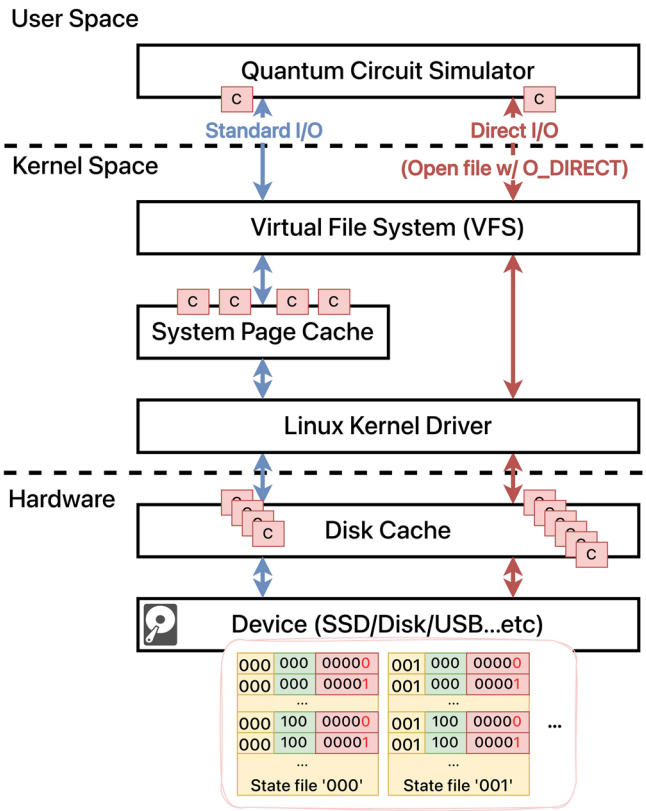

Figure 8. Illustration of the accesses to the quantum state files by our proposed simulator through a standard I/O (left) or a direct I/O interface (right).

Nevertheless, neither of the methods can consistently deliver the best performance for a quantum circuit simulator that could be used to simulate different qubit sizes (e.g. N could be 5, 10, or 40).

Under such a circumstance, it would be necessary for the simulator to be aware of the workload, and perform the simulation accordingly to achieve higher performance. Our empirical studies suggest that when simulating a smaller-scale qubit system (e.g. $N = 5$), the standard I/O interface can be used as the main memory is often large enough to accommodate the page cache data. On the other hand, direct I/O is desired when handling a larger-scale qubit system to provide a shorter accessing latency (since the page cache miss rate is high due to the capacity miss). We have implemented the simulator with the two I/O methods to achieve a high-performance simulation under different circumstances. However, in practical applications, there are some potential issues while applying the direct I/O solution. Further results about how to configure the simulator into the standard I/O or the direct I/O mode are presented in Section 5.2. An automatic mode switching mechanism is possible by monitoring the page cache miss rate at runtime; this is part of our future work.

4.4. Parameter tuning tips

To unleash the performance offered by the multicore hardware and the memory subsystem, it is important to use proper parameter settings for the quantum circuit simulations. The important parameters and their effects on the performance are listed as follows. ***Chunk segment (C)***. As the simulator operates on the state data chunk by chunk, the chunk size should be determined judiciously to reduce the number of system calls for file I/O. These chunks are buffered in the memory and a proper chunk size setting (to ensure that the chunks can be accommodated by the memory buffer) results in better performance.

File segment (F). As described in Section 4.1, concurrent reads/writes from/to a state file would incur the thread synchronisation overhead. To eliminate the overhead, it is possible to let a thread handle one or more state files. Nevertheless, increasing the number of files would inevitably add the burden to the underlying system to maintain the multiple files (for a thread). Furthermore, when there are multiple storage devices, the distribution of the state files onto the devices could also be taken into account in order to achieve the best performance. ***Thread segment (T) and simultaneous multithreading (SMT) effect.*** A larger number of threads could potentially improve simulation speed by allowing more work to be done in parallel. Nevertheless, it would incur higher thread management costs, especially when the thread number is larger than the physical hardware core number. Thus, it is crucial to set the T value properly to deliver the best simulation performance. When the T value is considered, the SMT feature would be taken into account as one of the possible options for a larger T. SMT is a hardware multithreading support found in modern processors and is a useful technique for improving system throughput by allowing more hardware logical threads to run on a physical hardware core simultaneously. Nevertheless, the outcome of enabling the SMT feature is unpredictable, depending highly on the runtime system status.

As a whole, observing that improper different parameter settings could lead to poor performance, as an example, our experimental results in Section 5.3 provide the insight (by showing the procedure) to find a proper setting for the quantum circuit simulation on our experimental environment.

5. Experimental results

The experiments that have been conducted to showcase the effectiveness of our proposed methodology are presented in this section. The environment that is used for the experiments is described in Section 5.1. The quantum gate level performance evaluation of our proposed QCS is presented in Section 5.2, and the performance results are compared with the state-of-the-art tools, QuEST and SnuQS, where the former is a memory-based QCS and the latter is a storage-based QCS. The experimental results for the empirical turning process (illustrated in Table 3) are exhibited in Section 5.3 to show a set of good parameters for our experimental platform. These parameters are used in the quantum circuit level performance evaluation of our proposed QCS in Section 5.4, and the results are compared with QuEST to shed light on the potential research direction in the future.

5.1. Experimental setup

A multithreading quantum circuit simulation environment has been built to provide many working threads to manipulate the quantum states during the simulations. The simulation environment is built upon the AMD Threadripper processor with 64 processor cores (128 logical hardware threads enabled by the SMT technology), as listed in Table 3, where the processor has a large last-level CPU cache of 256 MB. For the memory subsystem of the machine, the simulation hardware has eight 32 GB DDR4 memory modules, consisting of a total of 256 GB main memory. Besides, to support the simulation of a larger qubit size, the eight storage devices form a total of 16 TB of secondary memory, each of the storage device having up to 7.0 GB/s of memory bandwidth over the PCIe 4.0 bus (*x*4 lanes; 8 GB/s; larger than the theoretical SSD bandwidth). The multithreaded simulation is enabled by the OpenMP library (ver. 4.5), and the simulator is built by GCC ver. 9.4.0, running on top of Ubuntu Linux 20.04 LTS.

5.2. Gate-level performance evaluation and validation

We conduct a series of comparative analyses between our simulator and the state-of-the-art simulators, including QuEST (Jones et al., 2019) and SnuQS (Park et al., 2022). Moreover, the simulation results of our proposed simulated are discussed to validate the effectiveness of our simulator.

Comparing with QuEST. Figure 9 demonstrates that the storage-based simulation scheme has a comparative performance against the memory-based solution. To ensure fairness and consistency, each measurement of the Hadamard gate is the average execution time derived from 10 runs, removing the highest and lowest observations, and *double* precision floating point numbers are adopted. The experiments run with the qubit sizes, ranging from 21 to 39.

For instance, when $N = 26$, it takes 24 ms to do the simulation with our storage-based approach, whereas it costs 21 ms to do the same simulation by QuEST (the state-of-the-art memory-based simulator), meaning that the storage-based method is 0.88 times as fast as the memory-based counterpart. It is important to note that as the number of qubits grows, the memory-based approach cannot handle the simulation effectively and can lead to unknown behaviours. In our setup, when $N = 34$, the state-of-the-art work cannot respond

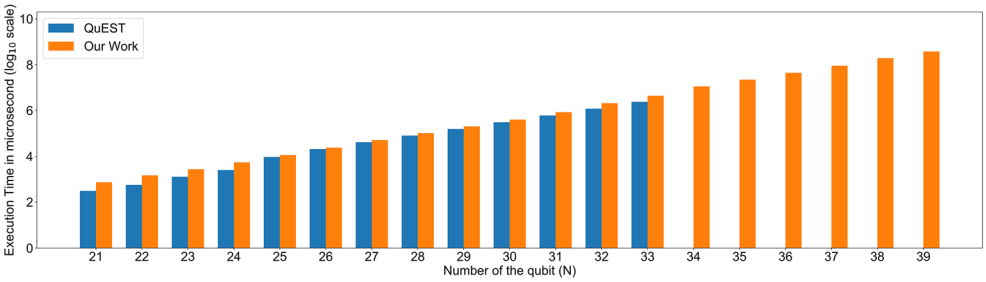

Figure 9. A comparison between QuEST and our simulator is performed for the Hadamard gate simulation in microseconds, with a range of qubits from 21 to 39.

as expected because the state data size exceeds the primary memory size, and the operating system is busying in *swapping* out data from memory to the secondary storage to make room for the requested data.

This phenomenon renders the memory-based scheme unsuitable for handling a large qubit simulation since it is bound by the size of the system memory. On the other hand, our proposed scheme relaxed the constraint on the memory size, augmenting the simulation of a larger qubit system by adding more SSDs; in this experiment, our approach is able to simulate up to a 39-qubit system. Given that the size of SSD is often several orders of magnitude larger than that of memory (62.5 times larger in our experimental environment), the storage-based solution can support the development of important quantum algorithms, e.g. quantum approximate optimisation algorithms (Farhi et al., 2014; Zhou et al., 2020), which require a larger qubit system to produce a more accuracy result.

Comparing with SnuQS. SnuQS (Park et al., 2022) is the other one of the pioneer works for the storage-based quantum circuit simulation, which can support the simulation of up to 41-qubit quantum programs with larger RAID-0 enabled SSDs (e.g. taking 1.44 hours to handle the 41-qubit H gate simulation). Unfortunately, we were unable to reproduce the results reported in the paper by Park et al. (2022). This is due to the absence of the asynchronous block I/O implementations in their opened source code,[1] and the asynchrony is an essential part of their design for overlapping the parallel computation and the transfers of quantum states. As an alternative, an approximate method is adopted to give a rough comparison with SnuQS. The simulation time of the H gate for a 39-qubit system is approximated by scaling down the 41-qubit simulation time linearly, and it turns out that it takes about 0.36 hours to do the 39-qubit simulation.[2] On the other hand, our proposed simulator costs about 0.11 hours to do the H gate simulation for a 39-qubit system, and we believe our approach is comparable to the state-of-the-art work. As this work adopts an orthogonal approach for storage-based simulation, our methodology can work with the SnuQS methods to achieve an even higher simulation performance. More about comparing our method and SnuQS's method is provided in Section 2.5.

Verifying the Simulation Results. Our QCS is implemented in C language, conforming to the IEEE-754 standard to represent the floating-point numbers that are used to model the quantum operations. We have developed a gate-level benchmark to validate the results of our developed QCS. The correctness is checked successfully by comparing the simulation results of different gates produced by our QCS and QuEST in a gate-wise comparison

manner. Specifically, the absolute difference between the outputs reported by the two simulators is less than 10^{-9} for each tested quantum gate.[3] As quantum gates and circuits can be characterised by their unitary matrix properties,[4] the correctness of the results of our QCS is also validated by the criteria of the unitarity cheques. Besides, the quantum circuit level validation is done successfully on the quantum circuits for QFT and five-level QAOA, and the related performance is presented in Section 5.4.

5.3. Parameter tuning

To fully harness the potential of the underlying hardware for the storage-based quantum circuit simulations, the modelling parameters should be selected judiciously. The parameter tuning process should be performed for each new hardware/software platform to avoid underfitting or overfitting issues. The subsequent paragraphs use the most common gate, Hadamard gate, as an example workload to evaluate the delivered simulation performance. The reported performance data are the average ten-run simulation time, excluding outliers.

Chunk segment. Figure 10 illustrates the simulation times under different chunk segment sizes ($C = 8, 10, 12, 14, 16$) and qubit sizes ($N = 23, 26, 29, 32$) combinations, where there are 64 threads ($T = 6$), each accessing a respective file ($F = 6$). Overall, when the chunk segment is set to twelve, the simulations deliver the best performance. Simply choosing a value could result in poor performance. For example, it is intuitive to use the file system block size, which is 4,096 bytes in our system, as the chunk segment size ($C = 8$).[5] But, as illustrated in the figure, when the chunk segment is set to eight, its performance is up to 5.4x slower than the performance for the setting of twelve (chunk size $= 64$ KB), in the experiment of $N = 26$. According to our preliminary analysis, the drop in the elapsed time from the chunk segment size of 4 KB to 64 KB could be attributed to the decrease in the read/write function calls while handling a large number of state files. Specifically, the simulation requires four times more instructions to be executed (and two times more context switches between the user- and kernel-mode) when the chunk segment size is 4 KB than

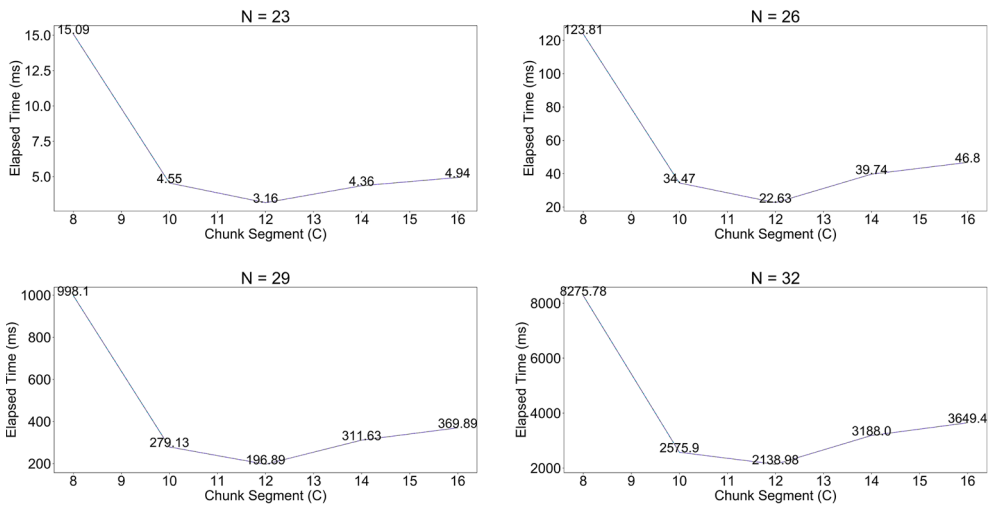

Figure 10. Ultizing 64 cores to explore a proper chuck segment value.

Table 4. Evaluating the performance impact of different file segments (unit: s).

	31	32	33	34	35
$C = 12, F = 3, T = 6$	1.52	5.51	17.55	65.23	132.24
$C = 12, F = 6, T = 6$	0.99	2.90	6.14	29.36	45.72
[Largefile]$C = 12, F = 3, T = 6$	1.21	2.66	5.29	46.10	95.75
[Largefile]$C = 12, F = 6, T = 6$	**0.96**	**2.13**	**4.26**	**21.92**	**40.71**

when it is 64 KB. It is important to note that the delivered simulation performance is affected by several factors, e.g. the file system block size, the virtual file system caching mechanism, the virtual memory management (e.g. the aggressiveness of swapping out memory pages), the caching mechanism of the storage devices, and the block size of the storage devices. Consequently, it requires the empirical tuning process to find a proper setting when the simulations run on a new computer.

File segment. To experiment with the synchronisation overhead of the concurrent accesses to the state files, as described in Section 4.1, this experiment uses 64 threads ($T = 6$) to perform the concurrent reads from 8 ($F = 3$) and 64 ($F = 6$) files, respectively, with and without the *largefile* setting.

Table 4 shows that the synchronisation overhead is eliminated by allowing each thread to access a file (the settings with $F = 6$), and the performance is improved by a factor of 2.8 ($N = 35$) and 2.3 ($N = 35$, largefile). Note that when multiple threads access the same file, each thread has to wait for its turn to access the file descriptor to avoid an inconsistent state, and this synchronisation overhead can be removed by using the one-file-for-one-thread scheme.

Thread segment. To discover a better T value, this experiment tests the T values ranging from six (64 threads) to ten (1024 threads). The performance data suggest that the simulation system delivers the best when the T value is agreed with the physical core number (64). Turning on the SMT feature ($T = 7$; via turning off the SMT switch in the system BIOS) does not help the performance. Further increasing the thread number ($T > 6$) would hurt the performance since the simulation system is bounded by I/O bandwidth when T is six, according to our preliminary analysis. Using a larger number of threads would incur a higher context-switching overhead, and the increased I/O requests made by the threads contend with the I/O transfer channels.

Direct IO. Table 6 lists the performance delivered by the storage-based simulations with and without the use of direct I/O for state data reads/writes under the setting of 64 threads ($F = T = 6$) and 64 KB ($C = 12$) of the chunk size. As Section 4.3 mentions, simply using a specific I/O method cannot consistently deliver the best performance. When the state data can be accommodated by the main memory ($N \leq 33$), the standard I/O method is 2× faster than the direct I/O alternative since the state data (maintained by the page cache) are hit on the memory. On the other hand, the direct I/O method performs 2× better than the standard I/O method for a larger qubit size ($N > 33$) because it provides direct access to the underlying storage devices without incurring the virtual memory management overhead (caused by the page cache for not being able to handle a large data). It is important to note that the empirical data shown in Table 5 suggest the quantum circuit simulation is an I/O bound workload. Under such a circumstance, the direct I/O can reduce the I/O latency without the intervention of the caching mechanism. This mechanism should be enabled for the simulation of an extensive qubit system.

Table 5. Evaluating a proper thread segment value (unit: s).

Qubit	$T = 6$	$T = 7$	$T = 8$	$T = 9$	$T = 10$
31	**1.0**	1.1	1.3	1.3	1.2
32	**2.1**	2.3	2.9	2.8	3.0
33	**4.3**	4.8	5.6	5.7	6.4
34	**21.9**	27.0	28.7	29.1	27.4
35	**40.7**	76.4	88.1	85.5	83.8
36	**97.4**	148.2	189.5	174.6	190.4
37	**186.5**	293.5	366.1	376.5	363.2
38	**541.1**	761.8	803.4	766.2	729.0
39	**1127.3**	1190.9	1559.3	1462.2	1460.7

Table 6. The elapsed time of the standard and direct IO (unit: s).

Qubit	Memory Requirement	Standard IO	Direct IO	Relative Performance
31	32 GiB	**1.0**	2.0	0.5
32	64 GiB	**2.1**	3.5	0.6
33	128 GiB	**4.3**	6.0	0.7
34	256 GiB	21.9	**11.4**	**1.9**
35	512 GiB	40.7	**22.4**	**1.8**
36	1 TiB	87.3	**44.7**	**2.0**
37	2 TiB	182.1	**91.4**	**2.0**
38	4 TiB	377.3	**197.1**	**1.9**
39	8 TiB	764.2	**385.0**	**2.0**

5.4. Circuit-level performance evaluation

The quantum circuits of two classical-quantum algorithms, quantum Fourier transform (QFT) (Coppersmith, 2002) and quantum approximate optimisation algorithm (QAOA) (Farhi et al., 2014; Zhou et al., 2020), are used to demonstrate the delivered performance of our QCS. Table 7 lists the elapsed time of the quantum circuit simulation of QFT and QAOA delivered by our QCS (denoted as SSD[6]) and by QuEST, the state-of-the-art memory-based QCS. Different qubit sizes, ranging from 24 to 39, have been tested. It is important to note that there is a surge of the elapsed time when $N = 27$ because the size of the quantum state files is larger than the L3 cache size of the manycore processor. Furthermore, while the memory and IO bandwidths (as listed in Table 1) account for the performance difference between the memory- and SSD-based solutions, our proposed qubit indexing and performance tuning mechanisms (in Sections 3.2 and 4.4 respectively) help close the performance gap,[7] resulting in less than two times the performance gap between the two simulators for the five-level QAOA circuit. On the other hand, there is a four-fold performance gap between QuEST and our QCS for QFT, since the state-of-the-art work has an ad-hoc optimisation specialised for the CPhase gates in QFT.[8] The above results demonstrate the efficiency of our implemented QCS, and a hybrid scheme (switching between the memory and SSD scheme) is under development, which can help accelerate the simulation performance when necessary. In addition, the specialised optimizations (e.g. the CPhase optimisation done in QuEST) could be further implemented in our QCS to further boot simulation performance.

5.5. Scaling out storage-based quantum circuit simulations

To facilitate larger quantum circuit simulations, the three limitations of the storage-based simulation should be analysed and addressed: storage capacity, PCIe bus bandwidths and

Table 7. The quantum circuit simulation time on QFT and five-level QAOA (unit: s).

	QFT		5-level QAOA	
Qubit	QuEST	SSD	QuEST	SSD
24	0.2	1.8	2.1	8.8
27	6.5	20.8	72.8	96.7
30	59.9	187.1	707.3	878.9
33	544.9	2,848.3	6,806.2	12,788.4
36	–	26,568.4	–	133,461.0
39	–	243,090.0	–	1,226,450.2

lanes, and I/O access. The first two factors determine the maximum qubit size achievable on a single machine node. For instance, the simulation of a 39-qubit quantum system is supported by our machine[9] listed in Table 3. Duplicating machines can achieve a larger-scale quantum circuit simulation. For example, two of the listed machines can support 40 qubits, and a cluster of sixteen machines with 256 TB SSDs can handle 43 qubits.

When a computer cluster is used for quantum circuit simulations, inter-node communications for quantum state exchange often dominate the performance. To mitigate this, data caching mechanisms (Doi & Horii, 2020) are crucial for alleviating the communication overhead. These caching mechanisms allow each computer node to maximise local memory updates to reflect the effects of the simulated quantum gate operations, significantly enhancing simulation efficiency.

Similarly, the caching concept can be applied to the processor cache level to address the third limitation and to accelerate the simulation efficiency on a single node. We are developing a block-wise quantum state updating mechanism to increase the reuse of processor-cached data, reduce the I/O accesses, and boost the simulation performance. In preliminary tests, compared to the gate-by-gate simulation scheme, the blocked data access achieves 38-fold and 43-fold reduction of external data accesses for 35-qubit QFT and QAOA programs, respectively.[10] Thanks to the blocked access mechanism, the external I/O writes can be significantly reduced, and thus, the wearing-out issue for the SSDs can be alleviated significantly.

Cost Advantage of the Storage-based Alternative. Storage-based quantum circuit simulations offer significant cost advantages compared to the memory-based approaches. Considering a 43-qubit quantum circuit simulation, running this with a traditional memory-based tool would require a supercomputer like *Taiwania 2* (ranked 20th in TOP500, 2018). However, Taiwania 2 is at least 500 times the cost compared to our storage-based cluster solution.[11] Storage-based simulations make them the preferred choice for advancing quantum applications.

6. Conclusion

In this work, the storage-based quantum circuit simulator is proposed to provide a more cost-effective alternative to the memory-based scheme. The proposed approach is up to 61 times more efficient than the memory-based counterpart, regarding the cost-delay product. A qubit representation is developed, together with a threading model, which helps the simulation of different-qubit-size systems efficiently with different parameter configurations. Moreover, a performance-tuning strategy and empirical results have been proposed and discussed. The proposed storage-based simulation is

a hardware-independent approach, and the qubit size supported by the simulator is bounded by the volume sizes of the adopted SSDs and the maximum bandwidth of the I/O devices from the underlying hardware. Therefore, our approach presents a flexible solution to meet different cost/performance requirements of different quantum research groups.

Notes

1. SnuQS is available at https://github.com/mcrl/SnuQS.
2. Based on the experimental results provided by SnuQS the simulation time grows log-linearly with the increase of the simulated qubit size.
3. The following quantum gates are tested: H gate, S gate, T gate, X gate, Y gate, Z gate, P gate, Unitary 1-qubit gate, CX gate, CY gate, CZ gate, CP gate, Unitary 2-qubit gate, SWAP gate, and Unitary U3 gate.
4. It presents the requirement that performing matrix multiplication between a gate and its conjugate transpose must be an identity matrix, expressed as $U^{\dagger}U = UU^{\dagger} = I$.
5. When C is eight, it means the chunk size of 4,096 bytes ($2^{8+4} = 4{,}096$). Please refer to Section 3.2 for details.
6. In this experiment, the setting of the parameters is as follows, $T = 6$, $F = 6$, $C = 12$, and $M = N-18$, where the direct I/O is turned on when $N > 30$.
7. With the two mechanisms, quantum states can be kept in the memory without incurring file I/O operations, provided that there is sufficient available memory.
8. QuEST is optimised for the diagonal matrix computations, representing the CPhase gates frequently used in QFT.
9. A PCIe adaptor is used to mount eight SSDs.
10. The experimental results assume that the simulation of each quantum gate can incur one external data access in the gate-by-gate simulation scheme.
11. The cost of the computer cluster with sixteen machine nodes and 256 TB of SSDs is around $128,000 USD.

Disclosure statement

No potential conflict of interest was reported by the author(s).

Funding

This work was supported by National Science and Technology Council (the grant numbers, 113-2119-M-002-024 and 111-2221-E-006-116-MY3).

ORCID

Nai-Wei Hsu http://orcid.org/0009-0009-7481-2515
Chuan-Chi Wang http://orcid.org/0000-0002-8267-8071
Chia-Hsin Hsu http://orcid.org/0009-0007-7232-1195
Chia-Heng Tu http://orcid.org/0000-0001-8967-1385
Shih-Hao Hung http://orcid.org/0000-0003-2043-2663

References

cuQuantum development team, T. (2023, April). cuquantum.: Zenodo. https://doi.org/10.5281/zenodo.7806810

team, Q.A. & collaborators. (2020 September). qsim.: Zenodo. https://doi.org/10.5281/zenodo.4023103

Alexander, T., Kanazawa, N., Egger, D. J., Capelluto, L., Wood, C. J., Javadi-Abhari, A., & McKay, D. C. (2020, August). Qiskit pulse: Programming quantum computers through the cloud with pulses. *Quantum Science and Technology*, *5*(4), Article 044006. https://doi.org/10.1088%2F2058-9565%2Faba404

Bergholm, V., Izaac, J., Schuld, M., Gogolin, C., Ahmed, S., Ajith, V., Alam, M., Alonso-Linaje, G, Akash-Narayanan, B., Asadi, A., Arrazola, J., Azad, U., Banning, S., Blank, C., Bromley, T., Cordier, B., Ceroni, J., Delgado, A., Matteo, O., … Killoran, N. (2022). Pennylane: Automatic differentiation of hybrid quantum-classical computations.

Choi, M. D. (1975). Completely positive linear maps on complex matrices. *Linear Algebra and Its Applications*, *10*(3), 285–290. https://doi.org/10.1016/0024-3795(75)90075-0

Chow, J., Dial, O., & Gambetta, J. (2021). IBM Quantum breaks the 100–qubit processor barrier. https://research.ibm.com/blog/127-qubit-quantum-processor-eagle.

Chung, Y. H. (2022). *Enlarging quantum circuit simulation and analysis with non-volatile memories* [Unpublished master's thesis]. National Taiwan University. No. 1, Sec. 4, Roosevelt Rd., Taipei 106216, Taiwan (R.O.C.).

Coppersmith, D. (2002). An approximate fourier transform useful in quantum factoring.

Cross, A. W., Bishop, L. S., Smolin, J. A., & Gambetta, J. M. (2017). Open quantum assembly language. arXiv. https://arxiv.org/abs/1707.03429.

De Raedt, H., Jin, F., Willsch, D., Willsch, M., Yoshioka, N., Ito, N., Yuan, S., & Michielsen, K. (2019). Massively parallel quantum computer simulator, eleven years later. *Computer Physics Communications*, *237*, 47–61. https://doi.org/10.1016/j.cpc.2018.11.005

De Raedt, K., Michielsen, K., De Raedt, H., Trieu, B., Arnold, G., Richter, M., Lippert, T., Watanabe, H., & Ito, N. (2007). Massively parallel quantum computer simulator. *Computer Physics Communications*, *176*(2), 121–136. https://doi.org/10.1016/j.cpc.2006.08.007

Developers, C. (2021, March). *Cirq. : Zenodo*. Retrieved from 2021, March. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors. https://doi.org/10.5281/zenodo.4586899

Doi, J., & Horii, H. (2020, October). Cache blocking technique to large scale quantum computing simulation on supercomputers. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE. https://doi.org/10.1109/QCE49297.2020.00035

Farhi, E., Goldstone, J., & Gutmann, S. (2014). A quantum approximate optimization algorithm.

Fu, J., Cao, B., Wang, X., Zeng, P., Liang, W., & Liu, Y. (2022). BFS: A blockchain-based financing scheme for logistics company in supply chain finance. *Connection Science*, *34*(1), 1929–1955. https://doi.org/10.1080/09540091.2022.2088698

Gheorghiu, V. (2018, December). Quantum++: A modern C++ quantum computing library quantum++: A modern C++ quantum computing library. *PloS One*, *13*(12), e0208073. https://doi.org/10.1371%2Fjournal.pone.0208073

Häner, T., & Steiger, D. S. (2017, November). 0.5 petabyte simulation of a 45-qubit quantum circuit. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. https://doi.org/10.1145%2F3126908.3126947

Hu, J., Xu, X., Hao, J., Yang, X., Qiu, K., & Li, Y. (2023). Microservice combination optimisation based on improved gray wolf algorithm. *Connection Science*, *35*(1), Article 2175791. https://doi.org/10.1080/09540091.2023.2175791

Huang, Y., & Martonosi, M. (2019, June). Statistical assertions for validating patterns and finding bugs in quantum programs. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM. https://doi.org/10.1145%2F3307650.3322213

Jones, T., Brown, A., Bush, I., & Benjamin, S. (2019, July). QuEST and high performance simulation of quantum computers. *Scientific Reports*, *9(1), 10736, 1-11*.

Kalmet, P. H. S., Sanduleanu, S., Primakov, S., Wu, G., Jochems, A., Refaee, T., Ibrahim, A., Hulst, L. V., Lambin, P., & Poeze, M. (2020). Deep learning in fracture detection: A narrative review. *Acta Orthopaedica*, *91*(2), 215–220. https://doi.org/10.1080/17453674.2019.1711323

LaRose, R. (2018). Distributed memory techniques for classical simulation of quantum circuits.

Liu, P., & Huang, W. (2022). A graph algorithm for the time constrained shortest path. *Connection Science*, *34*(1), 1500–1518. https://doi.org/10.1080/09540091.2022.2061916

Liu, Y. A., Liu, X. L., Li, F. N., Fu, H., Yang, Y., Song, J., Zhao, P., Wang, Z., Peng, D., Chen, H., Guo, C., Huang, H.Wu, W., & Chen, D. (2021, November). Closing the "quantum supremacy" gap. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM. https://doi.org/10.1145%2F3458817.3487399

Lu, T. C. (2021). CNN convolutional layer optimisation based on quantum evolutionary algorithm. *Connection Science*, *33*(3), 482–494. https://doi.org/10.1080/09540091.2020.1841111

Metwalli, S. A., & Meter, R. V. (2022). A tool for debugging quantum circuits.

Nam, M., Cha, H., Choi, Y.-R., Noh, S. H., & Nam, B. (2019). Write-optimized dynamic hashing for persistent memory. In *Proceedings of the 17th Usenix Conference on File and Storage Technologies* (p. 31–44). USENIX Association.

Niwa, J., Matsumoto, K., & Imai, H. (2002, December). General-purpose parallel simulator for quantum computing. *Physical Review A*, *66*(6), Article 062317. https://doi.org/10.1103/PhysRevA.66.062317

Okazai, R., Tabata, T., Sakashita, S., Kitamura, K., Takagi, N., Sakata, H., Ishibashi, T., Nakamura, T., & Ajima, Y. (2020). Supercomputer Fugaku CPU A64FX realizing high performance, high density packaging, and low power consumption. https://www.fujitsu.com/global/documents/about/resources/publications/technicalreview/2020-03/article03.pdf.

Park, D., Kim, H., Kim, J., Kim, T., & Lee, J. (2022). SnuQS: Scaling quantum circuit simulation using storage devices. In *Proceedings of the 36th ACM International Conference on Supercomputing*. Association for Computing Machinery. https://doi.org/10.1145/3524059.3532375

Patel, H., Thakkar, A., Pandya, M., & Makwana, K. (2018). Neural network with deep learning architectures. *Journal of Information and Optimization Sciences*, *39*(1), 31–38. https://doi.org/10.1080/02522667.2017.1372908

Raedt, K. D., Michielsen, K., Raedt, H. D., Trieu, B., Arnold, G., Richter, M., Lippert, T., Watanabe, H., & Ito, N. (2007, January). Massively parallel quantum computer simulator. *Computer Physics Communications*, *176*(2), 121–136. https://doi.org/10.1016%2Fj.cpc.2006.08.007

Ren, Y., Ren, Y., Tian, H., Song, W., & Yang, Y. (2023). Improving transaction safety via anti-fraud protection based on blockchain. *Connection Science*, *35*(1), Article 2163983. https://doi.org/10.1080/09540091.2022.2163983

Roloff, E., Diener, M., Carreño, E. D., Moreira, F. B., Gaspary, L. P., & Navaux, P. O. (2017). Exploiting price and performance tradeoffs in heterogeneous clouds. In *Companion Proceedings of the10th International Conference on Utility and Cloud Computing* (pp. 71–76). Association for Computing Machinery. https://doi.org/10.1145/3147234.3148103

Smelyanskiy, M., Sawaya, N. P. D., & Aspuru-Guzik, A. (2016). qhipster: The quantum high performance software testing environment.

Steiger, D. S., Häner, T., & Troyer, M. (2018, January). ProjectQ: An open source software framework for quantum computing. *Quantum*, *2*, 49. https://doi.org/10.22331%2Fq-2018-01-31-49

Suau, A., Staffelbach, G., & Todri-Sanial, A. (2021). qprof: a gprof-inspired quantum profiler.

Suzuki, Y., Kawase, Y., Masumura, Y., Hiraga, Y., Nakadai, M., Chen, J., K. M. Nakanishi, Mitarai, K., Imai, R., Tamiya, S., Yamamoto, T., Yan, T., Kawakubo, T., Nakagawa, Y. O., Ibe, Y., Zhang, Y., Yamashita, H., Yoshimura, H., Hayashi, A., & Fujii, K. (2021, October). Qulacs: A fast and versatile quantum circuit simulator for research purpose. *Quantum*, *5*, 559. https://doi.org/10.22331/q-2021-10-06-559

Tan, B., & Cong, J. (2020, July). Optimality study of existing quantum computing layout synthesis tools. *IEEE Transactions on Computers*, *70*(9), 1363–1373.

TOP500 (2018, November). NOVEMBER 2018 — TOP500. November 2018 — top500. https://www.top500.org/lists/top500/2018/11/.

Trieu, D. B. (2009). *Large-scale simulations of error-prone quantum computation devices*. (Dr. (Univ.), Universität Wuppertal, Jülich). Retrieved from (Record converted from VDB: 12.11.2012; Universität Wuppertal, Diss., 2009) https://juser.fz-juelich.de/record/7578.

Wu, C. H., Hsieh, C. Y., Li, J. Y., & Li, J. C. M. (2020). qATG: Automatic test generation for quantum circuits. In *2020 IEEE International Test Conference (ITC)* (pp. 1–10). IEEE.

Xiao, L., Xie, S., Han, D., Liang, W., Guo, J., & Chou, W. K. (2021). A lightweight authentication scheme for telecare medical information system. *Connection Science*, *33*(3), 769–785. https://doi.org/10.1080/09540091.2021.1889976

Zhang, S. X., Allcock, J., Wan, Z. Q., Liu, S., Sun, J., Yu, H., Yang, X.-H., Qiu, J., Ye, Z., Chen, Y.-Q., Lee, C.-K., Zheng, Y.-C., Jian, S.-K., Yao, H., Hsieh, C.-Y., & Zhang, S. (2023, February). TensorCircuit: A quantum software framework for the NISQ era. *Quantum*, *7*, 912. https://doi.org/10.22331/q-2023-02-02-912

Zhou, L., Wang, S. T., Choi, S., Pichler, H., & Lukin, M. D. (2020, January). Quantum approximate optimization algorithm: Performance, mechanism, and implementation on near-term devices. *Physical Review. X*, *10*, Article 021067. https://doi.org/10.1103/PhysRevX.10.021067