Received: 13 December 2022 | Revised: 25 January 2023 | Accepted: 12 March 2023

IET Quantum Communication

DOI: 10.1049/qtc2.12058

The Institution of Engineering and Technology WILEY

ORIGINAL RESEARCH

QuDiet: A classical simulation platform for qubit-qudit hybrid quantum systems

Turbasu Chatterjee[1] | **Arnav Das**[1] | **Subhayu Kumar Bala**[1] | **Amit Saha**[1,2] | **Anupam Chattopadhyay**[3] | **Amlan Chakrabarti**[1]

[1]A K Choudhury School of Information Technology, University of Calcutta, Kolkata, India

[2]Atos, Pune, India

[3]School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

Correspondence

Amit Saha, T-7, Cluster-X, Purbachal, Sector-3, Salt Lake, Kolkata 700097, India.
Email: abamitsaha@gmail.com

Abstract

In recent years, numerous research advancements have extended the limit of classical simulation of quantum algorithms. Although, most of the state-of-the-art classical simulators are only limited to binary quantum systems, which restrict the classical simulation of higher-dimensional quantum computing systems. Through recent developments in higher-dimensional quantum computing systems, it is realised that implementing qudits improves the overall performance of a quantum algorithm by increasing memory space and reducing the asymptotic complexity of a quantum circuit. Hence, in this article, **QuDiet**, a state-of-the-art user-friendly python-based higher-dimensional quantum computing simulator is introduced. **QuDiet** offers multi-valued logic operations by utilising generalised quantum gates with an abstraction so that any naive user can simulate qudit systems with ease as compared to the existing ones. Various benchmark quantum circuits is simulated in **QuDiet** and show the considerable speedup in simulation time as compared to the other simulators without loss in precision. Finally, **QuDiet** provides a full qubit-qudit hybrid quantum simulator package with quantum circuit templates of well-known quantum algorithms for fast prototyping and simulation. Comprehensive simulation up to 20 qutrits circuit on depth 80 on **QuDiet** was successfully achieved. The complete code and packages of **QuDiet** is available at https://github.com/LegacYFTw/QuDiet.

KEYWORDS

quantum computing, quantum computing techniques

1 | INTRODUCTION

A significant progress has been made in quantum computing recently due to its asymptotic advantage over classical computing [1–5]. Moreover, with the introduction of the mathematical notion of a qudit in [6], the boundaries of quantum computing are extended beyond the binary state space. Qudits are states in a d-dimensional Hilbert space where $d > 2$, thus allowing a much larger state space to store and process information as well as simultaneous control operations [7–13]. It has been shown that usage of qudits leads to circuit complexity reduction as well as enhanced efficiency of some quantum algorithms. For practical demonstrations, qudits have been realised on several different hardware, including photonic quantum systems [14], ion-trap systems [15], topological quantum systems [16–18], superconducting systems [19], nuclear magnetic resonance systems [20, 21], continuous spin systems [22, 23], and molecular magnets [24].

1.1 | Problem statement

In order to continue innovation in the multi-valued logic space for quantum computing, an efficient, easy to use quantum computing simulator with the support of qudits is the need of the hour. Given the size of the unitaries in the qudit space, one also needs to consider the computational costs incurred while simulating such large complex systems, which presents a significant challenge even in the binary state-space simulators. The inception of this simulator was in the wake (or lack thereof) of accessible simulators that were capable of simulating multi-valued logic in a user-friendly manner. This meant that a lot of research time and effort was previously spent in the construction of matrices and checking their compatibility, dimensions and kronecker products manually before their output could be deciphered from a huge 1D-array [25–27]. As an example, let us say a generalised gate is imposed on five qudits in a 4-dimensional quantum system. For simulation purpose, the matrix of that generalised gate of $4^5 \times 4^5$ *i.e.*, 1024×1024 needs to be prepared manually, which apparently makes the simulation time-consuming and error-prone.

1.2 | Aims and objectives

Our proposed simulator, named **QuDiet**, aims to solve all that by providing suitable abstractions that shy the user away from behemoth calculations and focus on purely the logic building and quantum phenomenology in the higher dimensional space. **QuDiet** does this thanks to its simple, yet effective, lean architecture that could be used to debug implementations quickly and provide outputs without unnecessary computational or memory overhead. It also aims to provide the users with the flexibility of adding gates accordingly to a quantum circuit with only a few commands.

The main contributions of this work are as follows:

- The first of its kind proposal for a simulator based on higher-dimensional state space, multi-valued logic, utilising generalised quantum gates.
- Using sparse matrices and related algorithms at the core of all quantum operations to unlock potential speed-up.
- Using GPU acceleration and efficient memory maps to process large matrices with considerable speedup.
- Benchmarking multiple quantum circuits in qudit systems and showing overall simulation time for the different backends for the first time to the best of our knowledge.
- A full package with quantum circuit templates for fast prototyping and simulation.

The structure of this article is as follows. Section 2 describes the higher-dimensional quantum circuit and its classical simulation. Section 3 proposes the higher-dimensional quantum simulator, **QuDiet**. Section 4 analyses the efficiency of the proposed simulator with the help of benchmark circuits. Future scope of the proposed simulator is outlined in Section 5. Section 6 captures our conclusions.

2 | PRELIMINARIES

In this section, firstly, we discuss about qudits and generalised quantum gates. Later, we put some light on classical simulation of a higher-dimensional quantum circuit.

2.1 | Higher-dimensional quantum circuits

Any quantum algorithm can be expressed or visualised in the form of a quantum circuit. Commonly for binary quantum systems, logical qubits and quantum gates comprise these quantum circuits [28]. The number of gates present in a circuit is called gate count and the number of qubits present in a circuit is known as qubit cost. In this work, we mainly deal with qudits and generalised quantum gates since our simulator is based on higher-dimensional quantum computing.

2.1.1 | Qudits

A logical qudit that encodes a quantum algorithm's input/output in d-ary or multi-valued quantum systems is often termed as data qudit. Another sort of qudit used to store temporary findings is the ancilla qudit. The unit of quantum information in d-dimensional quantum systems is *qudit*. In the d dimensional Hilbert space $\mathcal{H}_d$, qudit states can be substantiated as a vector.

The vector space is defined by the span of orthonormal basis vectors $\{|0\rangle, |1\rangle, |2\rangle, \ldots |d-1\rangle\}$. In qudit systems, the general form of quantum state can be stated as

$$|\psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle + \alpha_2|2\rangle + \cdots + \alpha_{d-1}|d-1\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{d-1} \end{pmatrix} \tag{1}$$

where $|\alpha_0|^2 + |\alpha_1|^2 + |\alpha_2|^2 + \cdots + |\alpha_{d-1}|^2 = 1$ and $\alpha_0, \alpha_1, \ldots, \alpha_{d-1} \in \mathbb{C}^d$.

2.1.2 | Generalised quantum gates

In this section, a brief discussion on generalised qudit gates is exhibited. The generalisation can be described as discrete quantum states of any arity in this way. Unitary qudit gates are applied to the qudits to evolve the quantum states in a quantum algorithm. It is required to take into account one-qudit generalised gates such as NOT gate (X_d), Phase-shift gate (Z_d), Hadamard gate (F_d), two-qudit generalised CNOT gate ($C_{X,d}$) and generalised multi-controlled Toffoli gate ($C^n_{X,d}$) for logic synthesis of quantum algorithms in d-dimensional quantum systems. For better understanding, these gates are described in detail:

Generalised NOT gate

X^d_{+a}, the generalised NOT can be defined as $X^d_{+a}|x\rangle = |(x+a) \bmod d\rangle$, where $1 \le a \le d-1$. For visualisation of the X^d_{+a} gate, we have used a 'rectangle' (□). 'X^d_{+a}' in the 'rectangle' box represents the generalised NOT.

Generalised phase-shift gate

Z_d is the generalised phase-shift gate represented by a $(d \times d)$ matrix is as follows, with $\omega = e^{\frac{2\pi i}{d}}$ henceforth:

$$Z_d = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & \omega & 0 & \dots & 0 \\ 0 & 0 & \omega^2 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & \omega^{d-1} \end{pmatrix}$$

We have used Z_d in the 'rectangle' (□) box to represent the generalised phase-shift gate.

Generalised Hadamard Gate

The superposition of the input basis states is produced via the generalised quantum Fourier transform, also known as the generalised Hadamard gate, F_d. The generalised quantum Fourier transform or generalised Hadamard gate, produces the superposition of the input basis states. We have used F_d in the 'rectangle' (□) box to represent the generalised Hadamard gate. The $(d \times d)$ matrix representation of it is as shown below:

$$F_d = \frac{1}{\sqrt{d}} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{d-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(d-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{d-1} & \omega^{2(d-1)} & \dots & \omega^{(d-1)(d-1)} \end{pmatrix}$$

Generalised CNOT gate

In a binary quantum system, a controlled NOT (CNOT) gate can achieve quantum entanglement, which is an unrivalled property of quantum mechanics. For d-dimensional quantum systems, the binary two-qubit CNOT gate is generalised to the *INCREMENT* gate:

INCREMENT$|x\rangle|y\rangle = |x\rangle|(x+a) \bmod d\rangle$, if $x = d-1$, and $= |x\rangle|y\rangle$, otherwise, where $1 \le a \le d-1$. In schematic design of the generalised CNOT gate, $C_{X,d}$, we have used a 'Black dot' (•) to represent the control, and a 'rectangle' (□) to represent the target. 'X^d_{+a}' in the target box represents the increment operator.

The $(d^2 \times d^2)$ matrix representation of the generalised CNOT $C_{X,d}$ gate is as follows:

$$C_{X,d} = \begin{pmatrix} I_d & 0_d & 0_d & \dots & 0_d \\ 0_d & I_d & 0_d & \dots & 0_d \\ 0_d & 0_d & I_d & \dots & 0_d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0_d & 0_d & 0_d & \dots & X^d_{+a} \end{pmatrix}$$

where I_d and 0_d are both $d \times d$ matrices as shown below:

$$I_d = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix} \quad \text{and,}$$

$$0_d = \begin{pmatrix} 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix}$$

Generalised multi-controlled Toffoli gate

We expand the generalised CNOT or INCREMENT further to work over n qudits as a generalised Multi-controlled Toffoli Gate or n-qudit Toffoli gate $C^n_{X,d}$. For $C^n_{X,d}$, the target qudit is increased by a (mod d) only when all $n-1$ control qudits have the value $d-1$, where $1 \le a \le d-1$. In schematic design of the generalised Multi-controlled Toffoli Gate, $C^n_{X,d}$, we have used 'Black dots' (•) to represent all the control qudits, and a 'rectangle' (□) to represent the target. 'X^d_{+a}' in the target box represents the increment operator. The $(d^n \times d^n)$ matrix representation of generalised Multi-controlled Toffoli (MCT) gate is as follows:

$$C^n_{X,d} = \begin{pmatrix} I_d & 0_d & 0_d & \dots & 0_d \\ 0_d & I_d & 0_d & \dots & 0_d \\ 0_d & 0_d & I_d & \dots & 0_d \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0_d & 0_d & 0_d & \dots & X^d_{+a} \end{pmatrix}$$

For the sake of simplicity, we decompose generalised Multi-controlled Toffoli gate into set of generalised CNOT gates in rest of the article [29, 30]. It is also exhibited that this decomposition of Toffoli gate can be logarithmic in depth as compared to linear depth while using conventional approach of using generalised T gate. This depth reduction is also useful for implementing different algorithms in quantum computing. A generalised Toffoli decomposition in a d-ary system using $|d\rangle$ state is shown in Figure 1.

2.2 | Classical simulation of a higher-dimensional quantum circuit

This section highlights how a program that runs on a classical computer can resemble the evolution of a quantum computer. Before jumping on that let us discuss about the challenges that the current qubit-only classical simulators are facing to simulate qudit systems or higher-dimensional quantum circuits.

- For the state-of-the-art qubit-only simulators [25, 26, 31–33], the simulators need to act on an n qubit state with a $2^n \times 2^n$ matrix. The dimension of the matrices of the unitary gates are also quite straight-forward as it is only based on qubit systems [34]. Due to the engineering challenge of maintaining the dimension of the matrices automatically for the qubit–qudit hybrid systems, the current simulators are unable to provide a solution to simulate the higher-dimensional quantum circuits efficiently, which is addressed in this paper.
- The other challenge is that it requires a significant amount of memory to store the higher-dimensional quantum state vectors and to perform matrix multiplication to simulate the generalised quantum gates. Hence, the current simulators reach long simulation times and memory limitations very quickly due to its conventional memory management. In this paper, we also address this issue by designing unitary matrix simulator with various backends to simulate qudit systems effectively.

Before explaining our proposed simulator more elaborately, we would like to discuss about the technicalities of the classical simulation of a higher-dimensional quantum circuit.

To simulate a higher-dimensional quantum circuit, we first need to specify the dimension of each qudit so that generalised quantum gates or quantum operations can act on a sequence of qudits effectively [26]. This can be done through a method, which returns a tuple of integers corresponding to the required dimension of each qudit it operates on, as an instance (2, 3, 4) means an object that acts on a qubit, a qutrit, and a ququad. To apply a generalised gate to some qudits, the dimensions of the qudits must match the dimensions it works on. For example, for a single qubit gate, its unitary is a 2×2 matrix, whereas for a single qutrit gate its unitary is a 3×3 matrix. A two qutrit gate will have a unitary that is a 9×9 matrix ($3 \times 3 = 9$) and a qubit-ququad gate will have a unitary that is an 8×8 matrix ($2 \times 4 = 8$). The size of the matrices involved in defining mixtures and channels follow the same pattern.

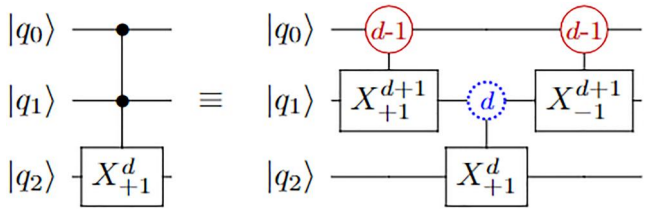

FIGURE 1 Generalised Toffoli in d-ary quantum systems – the control qudits in red circles activate on $|d-1\rangle$ and those in the blue dotted-circles activate on $|d\rangle$.

After simulating higher-dimensional quantum circuit by considering the dimension of qudits and generalised gates appropriately, the size of the resultant state is determined by the product of the dimensions of the qudits being simulated. For example, the state vector output after simulating a circuit on a qubit, a qutrit, and a ququad will have $2 \times 3 \times 4 = 24$ elements, Since, circuits on qudits are always assumed to start in the computational basis state $|0\rangle$, and all the computational basis states of a qudit are assumed to be $|0\rangle$, $|1\rangle$, …, $|d-1\rangle$. Measurements of qudits are assumed to be in the computational basis and for each qudit return an integer corresponding to these basis states. Thus measurement results for each qudit are assumed to run from $|0\rangle$ to $|d\rangle$ like $|0\rangle$ to $|1\rangle$ for qubit systems.

3 | QuDIET: A QUBIT–QUDIT HYBRID QUANTUM SIMULATOR

The lean architecture of **QuDiet** has been laid out briefly in the subsequent subsections. Before that the flow on the user end can be summed in Figure 2. This figure shows the high-level description of **QuDiet** to understand the general features of the proposed simulator. First, the quantum algorithm is expressed as a quantum circuit with the help of either QuDiet's QASM specification or simple python console. Next, this needs to be compiled to a specific quantum gate set of **QuDiet**. Finally, the quantum circuit is simulated with **QuDiet** to get the final outcome of the given quantum algorithm.

Now that we have a high level understanding of the compiler, let us take a deep dive and look at the internals in the following subsections.

3.1 | High level architecture

At its core, **QuDiet** is managed by two integral parts: The `Moment` and the `OperatorFlow` objects. These two; however, are simple vector objects that behave like a stack, during the compiler's operation. This is portrayed in Figure 3. A circuit, in essence, is an `OperatorFlow` object at its heart, running on a `Backend`. Once a `QuantumCircuit` object has been instantiated, in the background an `OperatorFlow` object is also created, which consists of a single `Moment` object. This `Moment` object can carry two types of objects: A `InitState`, which is the representation of an initial state of a quantum circuit

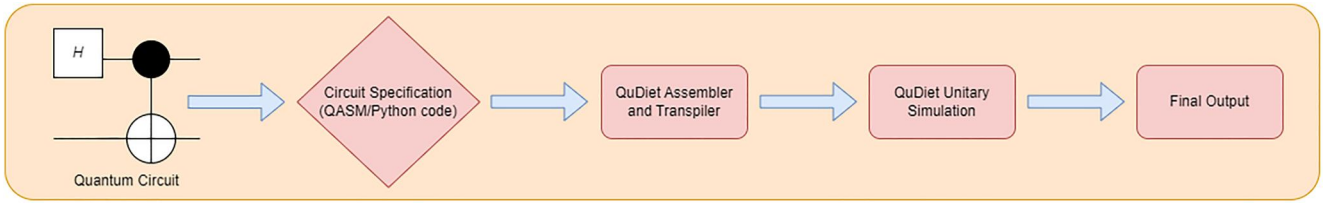

FIGURE 2 A user-level compilation flow of the **QuDiet**, where the input is a quantum circuit and the output is the probable quantum states of the input circuit.

or an `QuantumGate`, an abstract class, inherited by all quantum gates that are implementable by the simulator. The algorithm to create a quantum circuit is given by Algorithm 1. Once the user has implemented a quantum circuit, using the available commands, the `measure_all()` function is invoked, thereby pushing a `Moment` carrying measurement gates to all quantum registers. The measurement gate is a symbolic gate that tells the compiler that a program routine has ended and can be executed. The execution occurs when the `QuantumCircuit.run()` method is invoked, using the circuit's specified `Backend` object which has several interfaces for plug-and-play operability. This is characterised by Algorithm 2.

Algorithm 1 Building a Quantum Circuit

Input: A linear algebra backend B, an array containing the dimensions of the quantum registers `qregs`, quantum gates G and `init_states` I_i

Set quantum register length = N(`qregs`)
Initialize `OperatorFlow` object with `OperatorFlow` instance list L'
while L' *does not contain a* `Moment` *object with* `measure_all` **do**
  if I_k *is specified and* $N(I_k) = N($`qregs`$)$ **then**
    Initialize `Moment` object B_0 with `Moment` instance list L
    $qregs_i = I_i$
    $L_i = qregs_i$
    Push `Moment` into L'
  else
    Initialize `Moment` B_0 object with `Moment` instance list L
    $qregs_i = L_i$
    $L_i = qregs_i$
    Push `Moment` into L'
  foreach *in user invoked gate* G *acting on quantum register* **do**
    if *Gate* G_i *is preceded with* `IdentityGate` *in the preceding Moment list* **then**
      Invoke instance of user gate G_i acting on quantum register with dimension $qregs_i$
      Replace `IdentityGate` with G_i in the preceding Moment list
    else
      Initialize `Moment` B_i object with `Moment` instance list L
      Invoke instance of user gate G_i acting on quantum register with dimension $qregs_i$
      Invoke instance of `IdentityGate` acting on all other quantum registers
      Set L to be the list of gates acting on each register in order
      Push `Moment` into L'
  Initialize `Moment` B_n object with `Moment` instance list L
  Invoke instance of `measure_all`
  Push `Moment` into L'

Algorithm 2 Execute a quantum circuit

Input: `OperatorFlow` object with `OperatorFlow` instance list L', a linear algebra backend B

Pop L'
Initialize matrix M_1 of Kronecker product of IdentityGate objects as per `qregs`
foreach `Moment` *object with* `Moment` *instance list* L **do**
  foreach *object in* L **do**
    Calculate Kronecker product and store matrix M_i as per B
  Calculate $M = M_1 \cdot M_i$
  Delete M_i from memory

3.2 | The quantum circuit

The quantum circuit is represented by the `QuantumCircuit` class in the simulator. Whenever a new quantum circuit is invoked, a QuantumCircuit object is instantiated. This `QuantumCircuit` object takes the following arguments for instantiation.

- `qregs`: The dimensions of the quantum registers is represented as a heterogeneous list of integer dimensions. In other words, the dimension of the quantum 'wire', in order, or as a tuple represents a homogeneous register of fixed length, with the first qudit acting as the Least Significant Bit (LSB) and the last qudit in the register acting as the Most Significant Bit (MSB).
- `cregs`: The length of the classical register (optional)
- `name`: A string that represents the name of the quantum circuit (optional)
- `init_states`: Represents the configuration of the initial states of the register. This is represented by an array of the same length as that of `qregs`. If none is provided, the registers gets automatically initialised to $|0\rangle$'s.
- `backend`: This represents the `Backend` on which the quantum circuit is to be executed. There are four `Backend` objects to choose from, and are elaborated in subsection 3.7. The default backend to be used is the `SparseBackend`
- `debug`: A flag argument that forms the base of a debugger engine, implemented in a simple manner in this release cycle and shall be expanded upon in future versions for easy debugging and callback functions.

For example, in order to make a circuit with three qudits of dimensions 4,5,3 respectively, with initial states being $|0\rangle$, $|3\rangle$ and $|2\rangle$, that calculates using the `CUDASparseBackend` we just need the following lines of `python3` code:

```
qreg_dims = [4,5,3]
init_states = [0,3,2]
backend = CUDASparseBackend
qc = QuantumCircuit(qregs=qreg_dims,
init_states=init_states, backend=backend)
```

This shows how easily we can simulate different dimensional qudits with this simulator. Let us now take a closer look at these initial states, or more generally, the quantum states that they represent and how does **QuDiet** handle them.

3.3 | Representation of quantum states

Any quantum simulator is incomplete without their interpretation of quantum states. This preliminary version of **QuDiet** assumes that quantum states as state vectors. This comes with a caveat that **QuDiet** can only deal with states, as represented by an array or a vector list. This will of course be improved upon in future releases where we hope to incorporate density matrices, tensor-network and ZX/ZH calculus based representations. But we shall stick to the notion that these state vectors would be represented by:

$$|\psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle + \alpha_2|2\rangle + \cdots + \alpha_{d-1}|d-1\rangle = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_{d-1} \end{pmatrix} \tag{2}$$

Therefore, a register of qudits $|\psi_1\psi_2\ldots\psi_n\rangle$ would now need to be represented as:

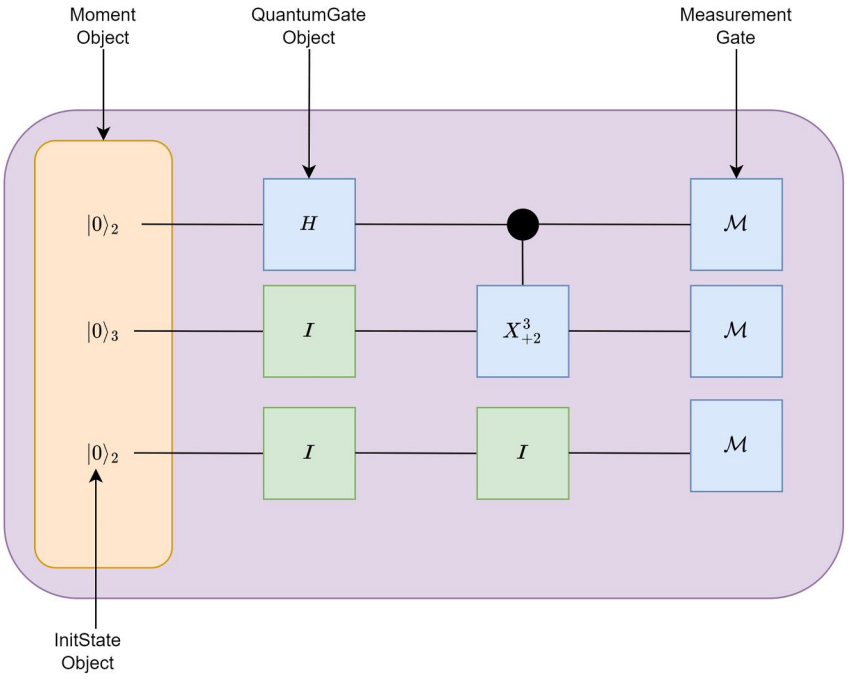

FIGURE 3 High level overview of the **QuDiet** quantum simulator. This figure shows the different parts of the quantum simulator, namely the `InitState` objects, with the subscripts representing the dimensions of the respective qudits, and the `QuantumGate` objects with the CX^3_{+2} gate being a generalised version of the CNOT gate with the target acting on a qutrit whereas the control being a qubit. The `QuantumGate` objects are enclosed using the `Moment` object and the `Moment` objects are enclosed within the `OperatorFlow` object.

$$|\psi_1\rangle \otimes \cdots \otimes |\psi_n\rangle = \begin{pmatrix} \alpha_{10} \\ \alpha_{11} \\ \alpha_{12} \\ \vdots \\ \alpha_{1d-1} \end{pmatrix} \otimes \cdots \otimes \begin{pmatrix} \alpha_{n0} \\ \alpha_{n1} \\ \alpha_{n2} \\ \vdots \\ \alpha_{nd-1} \end{pmatrix} \quad (3)$$

Unlike qubit-only quantum simulators, this presents an engineering challenge: An efficient method of storing these matrices and their computations in memory. On observation, we note that the number these quantum states are, in essence very sparse matrices with non-zero elements scattered around them. Therefore, it seemed natural to use the sparse array format for storing these state vectors.

In order to achieve this, **QuDiet** makes use of `scipy`'s Compressed Sparse Column array implementation. We shall be improving on this format to reduce latency of Sparse Matrix Vector multiplication (SpMV) and General Matrix Vector multiplication (GeMV). This is elaborated in the Section 5. However we have added support for `numpy` matrices for small circuits.

This technique is also used when representing quantum operators or quantum gates as we shall see in the following subsection.

3.4 | Representation of quantum gates

One of the key features of **QuDiet** is its ability to construct generalised gates and operators for quantum computing using multi-valued logic automatically. This is a standout feature and there does not exist a simulator in known literature that allows the construction of single and multi-qudit quantum gates and operators with the ease as that of **QuDiet**.

The **QuDiet** contains a limited gate set. These are as follows, and their descriptions are available in the Section 2.

(1) NOT Gate (XGate)
(2) Phase-Shift Gate (ZGate)
(3) Hadamard Gate (HGate)
(4) CNOT Gate (CXGate)
(5) User Defined Gate (QuantumGate)
(6) Measurement Gate
(7) Identity Gate (IGate)

Each of these quantum gates take into account two things: The qudit register it is acting on, and the dimension of the qudit register. Using a user-defined acting register, these quantum gates are able to take into account the dimension of the acting register, and dynamically construct a gate unitary at runtime. The XGate and the CXGate have an added functionality, that an arbitrary shift can be induced, as long as the shift has a value less than the dimension of the qudit register. The Measurement Gate in **QuDiet** is a special gate in that, it has no unitary associated with it. Up until this version, the Measurement Gate merely acts as flag that signifies the end of a quantum circuit. Any gate post measurement will be ignored by the simulator.

Once the unitary of a generic quantum gate has been created, it then utilises the same sparse matrix format to store the data. Contrary to quantum states, however, quantum gates utilise `scipy's` Compressed Sparsed Row matrix implementation. This format will of course, be improved upon to not only facilitate SpMV and GeMV but also SpGeMM or Sparse Matrix-Matrix multiplication and GeMM, or General Matrix-Matrix multiplication [35]. However, we have added support for `numpy` matrices for small circuits.

The backends have been engineered to provide minimal speedup using naive algorithms, and has CUDA support for GPU executions. These backends are elaborated in Section 3.7.

Therefore for demonstration, if we were to use the same quantum circuit as previously, in order to add a Hadamard gate acting on the first qudit and a CNOT gate with a shift of plus 2, acting on the first and the third qudit, we simply invoke the following lines of python3 code:

```
qc.h(0)
qc.cx((0,2), plus=2)
```

QuDiet also decomposes Toffoli gates (Let us say, `qc.Toffoli((0,1,2), plus=1)` into suitably mapped higher order `CXGate` objects. This is given as follows:

```
qc.cx((0,1), plus=1)
qc.cx((1,2), plus=1)
qc.cx((0,1), plus=2)
```

As stated before, these form the basis of a `Moment` object, which we shall elaborate on next.

3.5 | The moment

The `Moment` object is a forms an abstraction between the `QuantumGate` and the `OperatorFlow` objects, in that it maintains the orientation of the quantum gates that are acting on the respective qudits and the `OperatorFlow` maintains the sequence of execution of the quantum operations. The `Moment` object is inherently an array of length equal to the breadth of the quantum circuit. The primary job of the `Moment` object is to maintain the position of an acting register on the intended qudit so that errors in evaluating Kronecker products are avoided when the quantum circuit is executed.

There can be a single `Moment` object containing a list of `InitState` objects spanning across all the qudits in the quantum register and is initialised and pushed into the `OperatorFlow` object's stack whenever a quantum circuit is initialised. The `OperatorFlow` object can hold an arbitrary number of `Moment` objects, containing quantum gates across the breadth of the quantum register.

As per the **QuDiet**, whenever the user invokes a quantum gate onto a quantum circuit, the `QuantumGate` is pushed into the specific register corresponding to the index in the `Moment` object. All other corresponding registers or indices in the `Moment` object shall contain the `IdentityGate` object.

Something to note here is that, the `OperatorFlow` and the `Moment` data structures only store the data when it is pushed. No kronecker product or matrix multiplication operation would be done until the Measurement gates would be pushed and the user invokes the `run()` method from the quantum circuit object.

QuDiet also performs preliminary optimisations at the logic level whenever a gate is pushed. Whenever a new gate is pushed into the `OperatorFlow` stack, the gate is enclosed in a `Moment` object, while ensuring that an index in the `Moment` array corresponds to the acting qudit as specified by the user. All the other registers have an Identity gate acting on them. When pushing a `Moment` object containing a quantum gate into the `OperatorFlow` stack, the simulator checks if the any other immediate predecessor `Moment` has an Identity gate in them at the same position, if so, the Identity Gate is swapped out for the currently incoming quantum gate. This is done until it reaches an `InitState` object or it finds a `QuantumGate` object in any of its immediate predecessors.

3.6 | The operator flow stack

The OperatorFlow object is at the heart of the **QuDiet**: it maintains the order of execution of `QuantumGate` objects nestled inside the `Moment` objects. The `OperatorFlow` inherently maintains a vector list which acts like a stack during execution.

In order to run a quantum circuit, a `measure_all()` is called. This places `MeasurementGate` objects, across all quantum registers, thereby raising flag variables inside a `Moment` object. Any `Moment` containing quantum gates pushed after the `Moment` object shall be discarded, post the `measure_all()` command. In order to execute the said circuit, the `run()` is invoked, thereby outputting the state-vectors of the quantum states that have non-negative probabilities.

Under the hood, during this time, all circuits prior to the `Moment` containing the `MeasurementGate` objects are called to be executed in reverse order. This means that the `OperatorFlow` object will first 'pop' out that `Moment` and evaluate the Kronecker product depending on the type of `Backend`, inside the Moment and store it in a variable. Then it will advance onto the second-to-last `Moment`, evaluate the Kronecker product, and then store it in a separate variable. Now once these two Kronecker products have been evaluated, **QuDiet** will perform a SpGEMM or a GEMM operation, depending on the Backend selected for the circuit. When the matrix products have been evaluated, the two variable storing the Kronecker product is freed from memory. This continues until the Moment containing the InitState objects are reached where the last operation is and SpMV or a GEMV, depending on the Backend chosen.

3.7 | Acceleration and the backends

In **QuDiet**, acceleration can be achieved in two different ways. One is through a GPU, that is, hardware acceleration, and the other is by using sparse matrices, that is, algorithmic or software acceleration. These accelerations are delivered through different Backends, like **CudaBackend**. To access the GPU as a host for hardware acceleration, we use **CuPy**, which is a GPU equivalent for **NumPy** and **SciPy**. In terms of software acceleration, **SciPy** is used instead of **NumPy**, because of its useful interface for Sparse matrices. The very basis of **QuDiet**, lies in the availability and choice of Backends. For example, nearly dense matrices have showed no acceleration when run on the **SparseBackend**. On the contrary, it is a better choice to use **CudaBackend** instead and ignore the **sparse like**

Backends when dealing with nearly dense matrices. A more detail discussion on Backends is carried out in next.

QuDiet is a model-level library, providing high-level building blocks for developing quantum circuit algorithms. The low-level operations such as dot product, kronecker product etc. These low-level operations are interfaced through the class `Backend`. This enables one to use the best backend option based on the type of the circuit and the hardware accessible.

Right now, the backends accessible for use are **NumpyBackend**, **SparseBackend**, **CudaBackend** and **CudaSparseBackend**.

3.7.1 | NumpyBackend

NumpyBackend is the default backend used when no backend type is explicitly defined. This interfaces to the basic numpy operations, without any external optimisation.

3.7.2 | SparseBackend

SparseBackend interfaces to scipy's sparse module instead of interfacing to numpy's ndarray. It stores only the non-zero elements of the matrix and reduce the computation time by eliminating operations on zero elements. In cases, this can reduce the matrix size exponentially, showing an overall speedup in execution runtime and memory compression.

This speedup directly depends on the nature of the circuit, where the worst scenario is the arrays being mostly dense. In that case, the SparseBackend will perform nearly like a NumpyBackend.

3.7.3 | CUDABackend

CudaBackend interfaces with **CuPy's Numpy Routine**, using numpy-like dense matrices, accessing the GPUs with the purpose of runtime speedup only.

3.7.4 | CUDASparseBackend

CUDASparseBackend interfaces with **CuPy's Scipy Routine**, using sparse representation of the matrices and then accessing the GPUs with the purpose of runtime speedup.

The CUDASparseBackend optimises the operations using sparse matrices, followed by hardware(**GPU**) optimisation.

3.8 | Output and interpretability of results

Output Representation of a quantum circuit is still a less explored domain, especially when dealing with large circuits. **QuDiet** additionally provides some small contributions in terms of Output Representation and Interpretability for the sake of better research experience. **QuDiet** comes with two types of output representation, OUTPUTTYPE and two types of output method, OUTPUTMETHOD. OUTPUTTYPE is the way of output representation, which has two types, **print** and **state**. The OUTPUTTYPE.STATE provides the raw output state as a binary array. Whereas the OUTPUTTYPE.PRINT provides the output state as a *ket* string. On the contrary, OUTPUTMETHOD dictates whether the output would provide the probability distribution or the amplitude of the quantum states. By default, a quantum circuit returns the final *ket* representation along with the distribution probability, which looks like {|1,110,100,000,000⟩: 1.0 } for an instance.

3.9 | Example workflow

Let us now, take an example to understand the flow of work in **QuDiet**. The circuit to be executed is shown in Figure 4.

In order to do this, we must first create the quantum circuit by specifying the dimensions of the circuit lines in the quantum circuit, this is done by the following lines of python code:

```
qreg_dims = [2,3,3]
init_states = [0,0,0]
backend = SparseBackend
qc = QuantumCircuit(qregs=qreg_dims,
init_states=init_states, backend=backend)
```

These lines of code do the following.

1. The `InitState` objects are created for each of the circuit lines. Each of these `InitState` objects have the following matrices:

$$\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}1\\0\\0\end{pmatrix}, \text{ and } \begin{pmatrix}1\\0\\0\end{pmatrix}$$

   for the initiated circuit lines respectively.

2. The `Moment` object is initialised and then the `InitState` objects are pushed into the `Moment`. This object is then pushed into the `OperatorFlow` object's stack.

Now that the circuit has been initialised with the desired states, we can now add in the required gates. These gates are invoked by calling the respective methods of the circuit, which then creates the respective objects of the quantum gates. These quantum gates, at the time of their creation, take into account the dimensions of the acting register, among other factors, to construct the correct unitary automatically and push it into the `Moment` object, which is then pushed into the `OperatorFlow` object. This is done by the following lines of code:

```
qc.h(0)
qc.cx((0,1), plus=2)
```

These lines of code does the following.

1. The first line of the above snippet detects the dimension of the acting register. Since the dimension of the acting register is 2, it shall construct a 2×2 unitary suitably as follows:

$$\begin{pmatrix} 1/\sqrt{2} & 1/\sqrt{2} \\ 1/\sqrt{2} & -1/\sqrt{2} \end{pmatrix}$$

Once done, it will assign it to the instance variable of the `HGate` object.

1. Before pushing the `HGate` object to the `Moment` list, **QuDiet** will create Identity gates tailored to the dimensions of the qudit register and push it into the `Moment` object.
2. Next, **QuDiet** will create the `CXGate` object using a generated unitary and then place the Identity gates before pushing it into the `Moment` object.

The state of the `OperatorFlow` object and its internals are given as follows in Figure 5.

Note that the tensor and inner products are not evaluated at the time of creation. In order to begin the process of execution, we invoke the following lines of code:

```
qc.measure_all()
qc.run()
```

These will add measurement operators, interpreted here as flag variables, to signify the end of all operations within a circuit. The circuit is then executed with the `run()` command, which begins calculating the tensor and the inner products using the backend specified. The final output is as follows:

```
Build elapsed: 0.0001919269561767578s
Execution elapsed: 0.0010845661163330
078s
[ {'|000¿': 0.7071067811865475}, {'|120¿':
0.7071067811865475} ]
```

The final simulation result comes with loading-time and execution-time of the given circuit as shown in the above example. We also obtain the final output quantum states as $|000\rangle$ and $|120\rangle$ with amplitude 0.7071067811865475 for the example circuit.

4 | EXPERIMENTS AND DISCUSSION

Apart from python console, **QuDiet** offers another form of circuit specification *i.e.*, QuDiet's QASM. With the increasing usage of quantum circuit description, QASM (Quantum Assembly Language) [36] was introduced. Through QuDiet's QASM, one can declare the qubits or qudits and can describe the operations (gates) on those qubits or qudits to be run on **QuDiet**. For ease of understanding, a sample QASM program on **QuDiet** is presented as following:

```
.qudit 3
qudit x0 (2)
qudit x1 (3)
qudit x2 (3)
.begin
X x0
H x0
Z x0
X x1
X X2 2
CX x0 x1
CX x1 x1 2
.end
```

In this QASM program, we declare three qudits, one is qubit (`x0`), one is qutrit (`x1`) and last one is qutrit (`x2`). NOT, Hadamard and phase gate are applied on qubit `x0`. Then a generalised NOT gate with +1 is applied on qutrit `x1` followed by a generalised NOT gate with +2 is applied on qutrit `x2`. A generalised CNOT with +1 is on `x0` and `x1` and a generalised CNOT with +2 is on `x1` and `x2`. The most widely used QASM, *i.e.*, OpenQASM [37] can be converted to the QuDiet's QASM form with the help of inbuilt lexer that is available in **QuDiet** to make it more user-friendly.

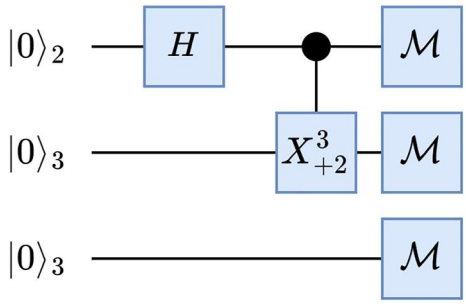

FIGURE 4 A quantum circuit to be simulated using **QuDiet**.

4.1 | Benchmarking with state-of-the-art simulators.

We have taken 21 benchmark circuits as an initial benchmarking, ranging from 3 qubit-qutrit to 7 qubit-qutrit in the form of QASM from [38] to verify our proposed simulator. To simulate all the 21 circuits, Toffoli gate is decomposed with intermediate qutrits as discussed earlier to get the algorithmic advantage. The simulation results are shown in Table 1 and the results are visually portrayed in Figure 6. The complete simulation time is based on three different parameters, (a) prepossessing-time; (b) loading-time; and (c) execution-time. We run these circuits with two backends, Numpy and Sparse. The maximum run-time (loading-time + execution-time) of these circuits is 0.3 s, which is akin to ref. [26], albeit the total simulation-time is much lower since the prepossessing-time is much higher for ref. [26] as gates are needed to be defined manually based on dimensions. The exact prepossessing-time of ref. [26] can never be determined since it is manual and very complicated to be defined mathematically. In our case, **QuDiet** being fully automatic, the prepossessing-time is negligible.

We further simulate more larger circuits on our proposed simulator. The results are shown in Table 2 and Figure 7. These 17 medium-sized circuits are also taken from ref. [38]. It is exhibited through numerical simulation that these circuits are well executable with Sparse-cuda backend due to its dense nature as compared to other backends. It can also be noted

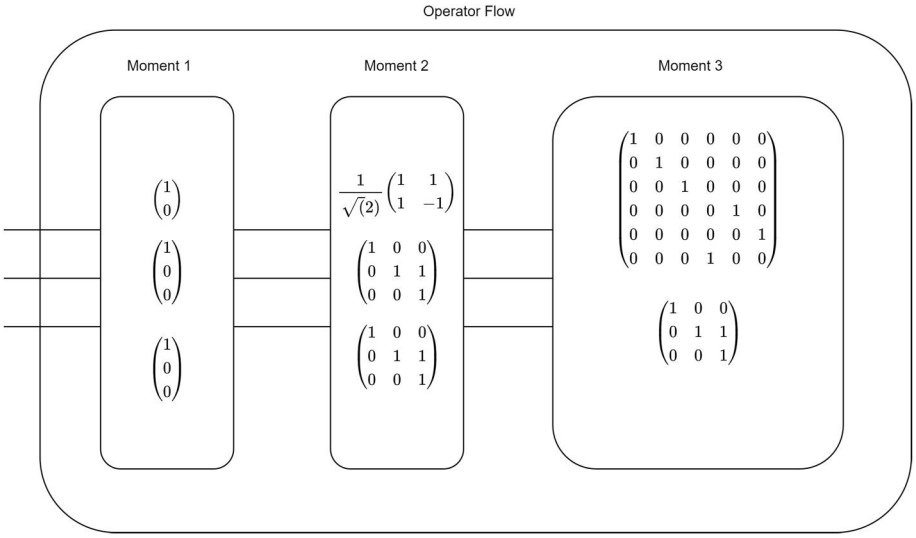

FIGURE 5 The `OperatorFlow` and its internals before the circuit is compiled.

TABLE 1 Loading and Execution Times (*in milliseconds*) of benchmark circuits on the Numpy and Sparse Backend.

		Numpy backend		Sparse backend	
Circuit	**(Width, depth)**	**Loading-time**	**Execution-time**	**Loading-time**	**Execution-time**
toffoli_2_tof	(3, 3)	0.289	1.219	1.193	3.062
ex-1_166_tof	(3, 6)	0.347	2.254	1.21	5.753
3_17_14_tof	(3, 10)	0.937	4.43	3.33	14.656
3_17_13_tof	(3, 10)	1.363	33.272	3.33	14.656
miller_11_tof	(3, 11)	0.571	5.379	1.154	10.428
decod24-v0_38_tof	(4, 12)	1.055	8.435	1.509	16.738
4_49_17_tof	(4, 22)	0.732	32.765	2.324	11.9348
mod5d1_63_tof	(5, 9)	0.435	11.948	1.339	17.745
mod5mils_65_tof	(5, 9)	0.463	7.46	1.86	15.955
4gt11_82_tof	(5, 14)	0.579	17.099	1.458	26.949
4mod5-v0_18_tof	(5, 16)	0.68	27.89	2.505	34.822
rd32_270_tof	(5, 17)	0.78	22.968	2.1836	36.203
alu-v0_26_tof	(5, 19)	0.732	64.351	1.538	69.933
4gt5_76_tof	(5, 26)	0.951	41.93	2.311	54.336
aj-e11_165_tof	(5, 33)	1.076	103.887	2.531	102.521
4_49_16_tof	(5, 48)	1.675	125.8	2.324	119.348
decod24-enable_125_tof	(6, 15)	0.61	69.66	2.274	66.186
decod24-bdd_294_tof	(6, 17)	0.734	38.439	1.957	52.381
4gt4-v0_72_tof	(6, 49)	1.577	315.283	2.555	129.431
alu-bdd_288_tof	(7, 18)	0.755	93.461	2.592	92.534
4mod5-bdd_287_tof	(7, 22)	0.701	112.74	2.61	88.45

that if these 38 qubit-only circuits without decomposition from Tables 1 and 2 are simulated on **QuDiet**, the performance time is same as refs. [25, 26].

We have employed the multiplication of 3 × 2 as an example to illustrate our simulator's efficiency in designing a quantum multiplier with intermediate qutrit. In Figure 8a, in

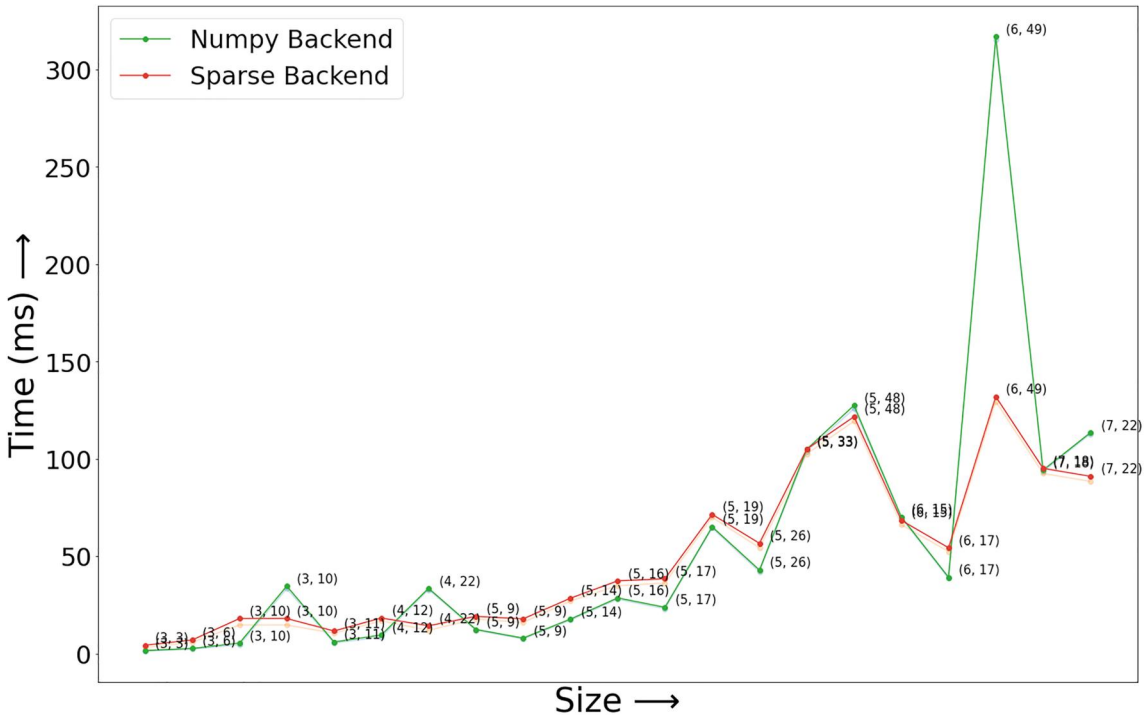

FIGURE 6 Circuit Size versus Total Time. As per Table 1, circuits execution time are plotted for both **Numpy** and **Sparse** Backend, with respect to the size of the circuits. As we go from left to right, the circuit size (both width and depth) increases. We can observe, for some circuits, the Sparse backend works *better* than conventional Numpy Backend since the operational matrices are sparse in nature.

TABLE 2 Loading and execution times *(in milliseconds)* of some benchmark circuits on Sparse-cuda Backend.

Circuit	(Width, depth)	Loading-time	Execution-time
hwb4_49_tof	(5, 51)	2.58	760.01
one-two-three-v0_97_tof	(5, 56)	575.1	963.46
mod8-10_177_tof	(6, 89)	568.71	1656.48
mod5adder_127_tof	(6, 106)	462.74	696.1
sf_274_tof	(6, 148)	5.91	1717.76
ham7_104_tof	(7, 73)	3.18	2196.92
C17_204_tof	(7, 106)	4.4	5135.62
majority_239_tof	(7, 149)	5.9	5966.78
sym6_145_tof	(7, 945)	24.09	43,371.93
f2_232_tof	(8, 293)	9.59	19,534.77
con1_216_tof	(9, 227)	10.2	25,159.1
mini_alu_305_tof	(10, 39)	784.24	884,254.49
sys6-v0_111_tof	(10, 46)	7.4	2060.27
wim_266_tof	(11, 219)	7.87	997,068.12
dc1_220_tof	(11, 435)	13.96	2,329,661.45
0410184_169_tof	(14, 60)	6.58	337.46

light of the preceding example, a multiplier circuit has been provided in accordance with ref. [38], in which all the qubits are initialised with $|0\rangle$. In this circuit, the first four qubits (q_0–q_3) are the input qubits, where the first two qubits (q_0 and q_1) represent the number 3 by applying two NOT gates on them and the other two qubits (q_2 and q_3) represent the number 2 by applying NOT gate on qubit q_2. Subsequently, using Toffoli gates, we conduct a multiply operation on these qubits and store the result in ancilla qubits (q_4–q_7). Now, using CNOT gates on an ancilla qubit q_8, we execute addition. Lastly, to

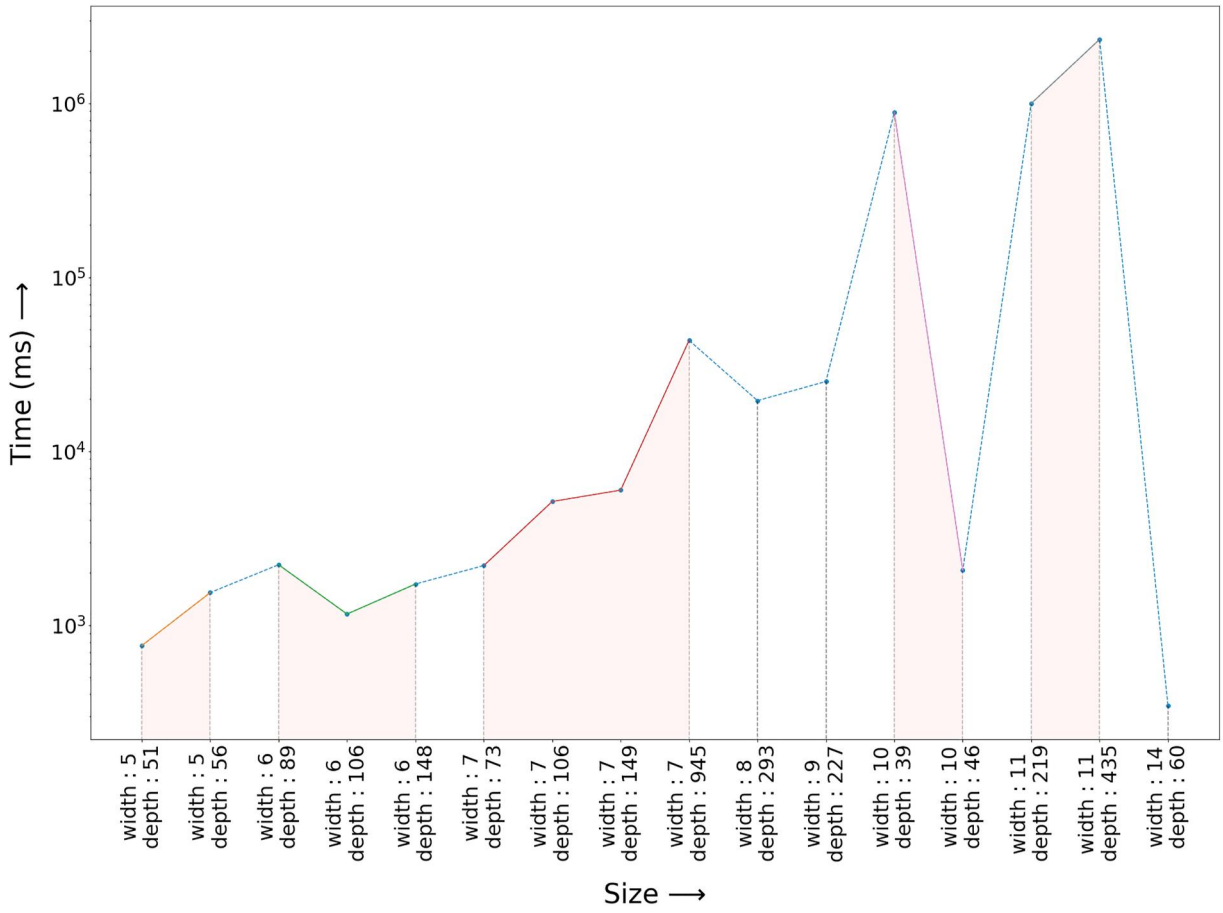

FIGURE 7 Circuit size versus Time for Sparse-CUDA Backend for different circuits as shown in Table 2. The *Y*-Axis represents the total time taken to execute a circuit (ms) in a logarithmic scale. The *X*-Axis represents the circuit size (both width and depth), in ascending order, as we go from left to right.

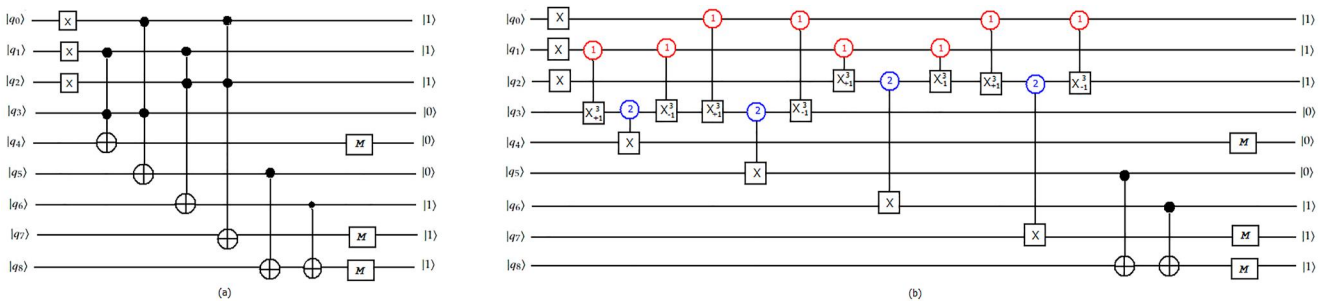

FIGURE 8 (a) Quantum Multiplier Circuit for the multiplication of 3 × 2; (b) Quantum Multiplier with Intermediate Qutrit for the multiplication of 3 × 2.

obtain the resultant output of 3 × 2, we need to measure the qubits (q_4, q_7 and q_8). Additionally, each of the Toffoli gates shown in Figure 8a are realised with the help of the intermediate qutrit method as shown in Figure 8b to achieve asymptotic advancement of the circuit. Our numerical simulation on **QuDiet** also yields 3 × 2 = 6 appropriately. Our simulation results further show that if we use Numpy backend, then the loading-time is 34.631 (ms) and execution-time is 136.532 (ms). For Sparse beckend, the loading-time is 17.157 (ms) and the execution-time is 1.865 (s) for the multiplier of 3 × 2 circuit, whose width is 9 and depth is 15.

4.2 | Simulation of some well-known quantum algorithms

We simulate some well-known quantum algorithms like, Grover's algorithm, Simon algorithm, Berstein-Vazirani algorithm etc on **QuDiet** considering different backends and amalgamate them as a package for faster prototyping. The result of the simulation is shown in Table 3. The trade-off of loading-time and execution-time between different backends for various quantum algorithms is exhibited in Table 3. As a concluding remark, using the **QuDiet** simulator, we also simulate a

TABLE 3 Loading and execution times *(in milliseconds)* of some supremacy circuits on various backend.

Circuit (backend)	(Width, depth)	Loading-time	Execution-time
Grover_n2 (sparse)	(2, 9)	2.55	25.76
Grover_n2 (cuda)	(2, 9)	1.16	248.16
Grover_n2 (sparse-cuda)	(2, 9)	1.19	44.05
Simon_n6 (sparse)	(6, 13)	5.39	92.94
Simon_n6 (cuda)	(6, 13)	1.05	30.7
Simon_n6 (sparse-cuda)	(6, 13)	2.11	183.22
Seca_n11 (cuda)	(11, 69)	180.8	13,536.19
Seca_n11 (sparse-cuda)	(11, 69)	524.42	12,008.1
SAT_n11 (cuda)	(11, 139)	5.9	16,861.99
SAT_n11 (sparse-cuda)	(11, 139)	8.62	12,229.81
BV_n14 (cuda)	(14, 16)	2.25	75,112.75
BV_n14 (sparse-cuda)	(14, 16)	4.65	932,159.35

comparatively larger circuit of 20 qutrits on depth 80 quite comprehensibly. The simulation has been performed on a local computer with processor Intel(R) Core(TM) i5-6300U CPU 2.40 GHz 2.50 GHz, RAM 8.00 GB, and 64-bit windows operating system.

5 | LIMITATIONS AND FUTURE SCOPE

In this section, we provide a glimpse of the future version of **QuDiet**, since there are some limitations in this version of **QuDiet**. The present form of **QuDiet** is based on python. In future, we can use a multi-threaded C++ core that uses AVX instructions, which can achieve state-of-the-art performance in matrix-vector multiplication of quantum states. The C++ core, which uses a similar syntax of numpy.dot can be integrated into our simulator whenever a matrix-vector multiplication is needed. Formation of a DAG interfaces for better optimisation workflows can be incorporated in the future. Large-scale simulations on heterogeneous HPC clusters via MPI can also be looked into. Memorisation and lookup tables for fast simulation can be a good case-study for future version of **QuDiet**. Simulation with noise models shall give a more broader picture of qudit simulation. Interfaces for different quantum hardware topologies shall need to be taken care in the future version of **QuDiet** to reduce the gap between physical and logical qudit simulation.

6 | CONCLUSION

In this paper, we introduced **QuDiet**, a hybrid qubit-qudit simulator that can classically simulate any finite-dimensional quantum system. It is exhibited that the **QuDiet** offers user-friendly environment to simulate qudit systems with an abstraction. **QuDiet** is efficient since generalised gates can be easily used without spending much time to defining them manually during simulation. We also showed considerable speed-up in simulation time for benchmark circuits. Finally, we simulated some well-known quantum algorithms in qudit setting to analysis the performance of our proposed simulator **QuDiet**. Furthermore, other available platforms can integrate **QuDiet** as a classical simulation option for higher-dimensional quantum systems to their platforms, since **QuDiet** is an open-source python-based simulator.

AUTHOR CONTRIBUTIONS

Turbasu Chatterjee: Conceptualisation, Data curation, Formal analysis, Investigation, Methodology, Resources, Software, Validation, Visualisation, Writing – original draft. **Arnav Das**: Conceptualisation, Data curation, Formal analysis, Investigation, Methodology, Resources, Software, Validation, Visualisation, Writing – original draft. **Subhayu Kumar Bala**: Conceptualisation, Data curation, Investigation, Software, Validation, Visualisation, Writing – original draft. **Amit Saha**: Conceptualisation, Data curation, Formal analysis, Investigation, Methodology, Project administration, Resources, Software, Supervision, Validation, Visualisation, Writing – original draft, Writing – review & editing. **Anupam Chattopadhyay**: Conceptualisation, Formal analysis, Investigation, Methodology, Project administration, Resources, Supervision, Validation, Visualisation, Writing – review & editing. **Amlan Chakrabarti**: Conceptualisation, Formal analysis, Investigation, Methodology, Project administration, Resources, Supervision, Validation, Visualisation, Writing – review & editing.

ACKNOWLEDGEMENTS

The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

CONFLICT OF INTEREST STATEMENT

There is no conflict of interest.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in https://github.com/LegacYFTw/QuDiet.

ORCID

Subhayu Kumar Bala https://orcid.org/0000-0001-8195-3118

Amit Saha https://orcid.org/0000-0003-2583-9825

REFERENCES

1. Farhi, E., Gutmann, S.: Quantum computation and decision trees. Phys. Rev. 58(2), 915–928 (1998). https://doi.org/10.1103/physreva.58.915
2. Mumtaz, S., Guizani, M.: An overview of quantum computing and quantum communication systems. IET Quan. Commun. 2(3), 136–138 (2021). https://doi.org/10.1049/qtc2.12021
3. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information, 10th Anniversary Edition. Cambridge University Press (2010). https://doi.org/10.1017/CBO9780511976667
4. Preskill, J.: Quantum computing in the NISQ era and beyond. Quantum 2, 79 (2018). https://doi.org/10.22331/q-2018-08-06-79

5. Sanyal Bhaduri, A., et al.: Circuit design for clique problem and its implementation on quantum computer. IET Quan. Commun. 3(1), 30–49 (2022). https://doi.org/10.1049/qtc2.12029
6. Muthukrishnan, A., Stroud, C.R.: Multivalued logic gates for quantum computation. Phys. Rev. 62, 5 (2000). https://doi.org/10.1103/physreva.62.052309
7. Bocharov, A., Roetteler, M., KrystaSvore, M.: Factoring with qutrits: Shor's algorithm on ternary and metaplectic quantum architectures. Phys. Rev. 96, 1 (2017). https://doi.org/10.1103/physreva.96.012306
8. Cao, Ye, et al.: Quantum Fourier transform and phase estimation in qudit system. Commun. Theor. Phys. 55, 790–794 (2011). https://doi.org/10.1088/0253-6102/55/5/11
9. Yao-Min, Di, Wei, H.R.: Synthesis of multivalued quantum logic circuits by elementary gates. Phys. Rev. 87, 1 (2013). https://doi.org/10.1103/physreva.87.012325
10. Fan, Y.: A generalization of the Deutsch-Jozsa algorithm to multi-valued quantum logic. In: 37th International Symposium on Multiple-Valued Logic (ISMVL'07), vol. 12. IEEE Computer Society, Los Alamitos, CA, USA (2007). https://doi.org/10.1109/ISMVL.2007.3
11. Khan, F.S., Perkowski, M.: Synthesis of multi-qudit hybrid and d-valued quantum logic circuits by decomposition. Theor. Comput. Sci. 367(3), 336–346 (2006). https://doi.org/10.1016/j.tcs.2006.09.006
12. Amit, S., et al.: One-dimensional lazy quantum walk in ternary system. IEEE Trans. Quan. Eng. 2(2021), 1–12 (2021). https://doi.org/10.1109/TQE.2021.3074707
13. Wang, Y., et al.: Qudits and high-dimensional quantum computing. Front. Phys. 8 (2020). https://doi.org/10.3389/fphy.2020.589504
14. Gao, X., et al.: Computer-inspired concept for high-dimensional multipartite quantum gates. Phys. Rev. Lett. 125, 5 (2020). https://doi.org/10.1103/physrevlett.125.050501
15. Klimov, A.B., et al.: Qutrit quantum computer with trapped ions. Phys. Rev. 67(6), 062313 (2003). https://doi.org/10.1103/PhysRevA.67.062313
16. Bocharov, A., et al.: Improved Quantum Ternary Arithmetics (2015). arXiv:1512.03824 [quant-ph]
17. Cui, S.X., Hong, S.M., Wang, Z.: Universal quantum computation with weakly integral anyons. Quant. Inf. Process. 14(8), 2687–2727 (2015). https://doi.org/10.1007/s11128-015-1016-y
18. Cui, S.X., Wang, Z.: Universal quantum computation with metaplectic anyons. J. Math. Phys. 56(3), 032202 (2015). https://doi.org/10.1063/1.4914941
19. Koch, J., et al.: Charge-insensitive qubit design derived from the Cooper pair box. Phys. Rev. A. 76(4), 042319 (2007). https://doi.org/10.1103/PhysRevA.76.042319
20. Dogra, S., Arvind, Dorai, K.: Determining the parity of a permutation using an experimental NMR qutrit. Phys. Lett. 378(46), 3452–3456 (2014). https://doi.org/10.1016/j.physleta.2014.10.003
21. Gedik, Z., et al.: Computational speed-up with a single qudit. Sci. Rep. 5, 1 (2015). https://doi.org/10.1038/srep14671
22. Adcock, M.R.A., Høyer, P., Sanders, B.C.: Quantum computation with coherent spin states and the close Hadamard problem. Quant. Inf. Process. 15(4), 1361–1386 (2016). https://doi.org/10.1007/s11128-015-1229-0
23. Bartlett, S.D., de Guise, H., Sanders, B.C.: Quantum encodings in spin systems and harmonic oscillators. Phys. Rev. 65, 5 (2002). https://doi.org/10.1103/physreva.65.052316
24. Leuenberger, M.N., Loss, D.: Quantum computing in molecular magnets. Nature 410(6830), 789–793 (2001). https://doi.org/10.1038/35071024
25. Bello, L., et al.: Abhari, Paco Martin, Diego Moreda, Jesus Perez, Erick Winston, and Chris Wood. Qiskit: Open-source Framework Quan. Comput. (2021). https://doi.org/10.5281/zenodo.2573505
26. Cirq Developers: Cirq (2022). See Full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors. https://doi.org/10.5281/zenodo.6599601
27. Giraldo-Carvajal, A., et al.: QuantumSkynet: A High-Dimensional Quantum Computing Simulator (2021). https://doi.org/10.48550/ARXIV.2106.15833
28. Barenco, A., et al.: Elementary gates for quantum computation. Phys. Rev. 52(5), 3457–3467 (1995). https://doi.org/10.1103/PhysRevA.52.3457
29. Gokhale, P., et al.: Asymptotic improvements to quantum circuits via qutrits. In: Proceedings of the 46th International Symposium on Computer Architecture (2019). https://doi.org/10.1145/3307650.3322253
30. Amit, S., et al.: Asymptotically improved circuit for a *d*-ary Grover's algorithm with advanced decomposition of the *n*-qudit Toffoli gate. Phys. Rev. A. 105(6), 062453 (2022). https://doi.org/10.1103/PhysRevA.105.062453
31. Smith, R.S., Curtis, M.J., Zeng, W.J.: A Practical Quantum Instruction Set Architecture (2017). arXiv:1608.03355 [quant-ph]
32. Steiger, D.S., Häner, T., Troyer, M.: ProjectQ: an open source software framework for quantum computing. Quantum 2, 49 (2018). https://doi.org/10.22331/q-2018-01-31-49
33. Wecker, D., Svore, K.M.: LIQUi|⟩: A Software Design Architecture and Domain-specific Language for Quantum Computing (2014). https://doi.org/10.48550/ARXIV.1402.4467
34. Ryan, L.R.: Overview and comparison of gate level quantum software platforms. Quantum 3, 130 (2019). https://doi.org/10.22331/q-2019-03-25-130
35. Gao, J., et al.: A Systematic Survey of General Sparse Matrix-Matrix Multiplication (2020). https://doi.org/10.48550/ARXIV.2002.11273
36. Svore, K., et al.: Toward a software architecture for quantum computing design tools. In: Proceedings of the 2nd International Workshop on Quantum Programming Languages (QPL), pp. 145–162 (2004)
37. Cross, A.W., et al.: Open Quantum Assembly Language (2017). https://doi.org/10.48550/ARXIV.1707.03429
38. Wille, R., et al.: RevLib: an online resource for reversible functions and reversible circuits. In: 38th International Symposium on Multiple Valued Logic (ISMVL), pp. 220–225 (2008). https://doi.org/10.1109/ISMVL.2008.43

How to cite this article: Chatterjee, T., et al.: QuDiet: A classical simulation platform for qubit-qudit hybrid quantum systems. IET Quant. Comm. 4(4), 167–180 (2023). https://doi.org/10.1049/qtc2.12058