Journal of Signal Processing Systems (2020) 92:1017–1037
https://doi.org/10.1007/s11265-020-01569-4

Efficient Computation Techniques and Hardware Architectures for Unitary Transformations in Support of Quantum Algorithm Emulation

Naveed Mahmud[1] · Bennett Haase-Divine[1] · Annika Kuhnke[1] · Apurva Rai[1] · Andrew MacGillivray[1] · Esam El-Araby[1]

Received: 2 December 2019 / Revised: 25 May 2020 / Accepted: 11 June 2020 / Published online: 14 July 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract

As the development of quantum computers progresses rapidly, continuous research efforts are ongoing for simulation and emulation of quantum algorithms on classical platforms. Software simulations require use of large-scale, costly, and resource-hungry supercomputers, while hardware emulators make use of fast Field-Programmable-Gate-Array (FPGA) accelerators, but are limited in accuracy and scalability. This work presents a cost-effective FPGA-based emulation platform that demonstrates improved scalability, accuracy, and throughput compared to existing FPGA-based emulators. In this work, speed and area trade-offs between different proposed emulation architectures and computation techniques are investigated. For example, stream-based computation is proposed that greatly reduces resource utilization, improves system scalability in terms of the number of emulated quantum bits, and allows for dynamically changing algorithm inputs. The proposed techniques assume that the unitary transformation of the quantum algorithm is known, and the matrix values can be pre-computed or generated dynamically. 32-bit floating-point precision is used for high accuracy and the architectures are fully pipelined to ensure high throughput. As case studies for emulation, the quantum Fourier transform and Grover's search algorithms are investigated and quantum circuits for multi-pattern Grover's search are also proposed. Experimental evaluation and analysis of the emulation architectures and computation techniques are provided for the investigated quantum algorithms. The emulation framework is prototyped on a high-performance reconfigurable computing (HPRC) system and the results show quantitative improvement over existing FPGA-based emulators.

Keywords Quantum computing · Quantum algorithm emulation · Reconfigurable computing · Field-Programmable-Gate-Arrays

1 Introduction

Quantum computers represent a promising new computing paradigm with the potential to solve complex problems [1–3] at speeds superior to modern state-of-the-art classical supercomputers. Quantum computers are expected to demonstrate their superior performance in a class of problems called Bounded-Error-Quantum-Polynomial (BQP)[4], e.g., factoring composite integers, computing discrete logarithms, sampling from a Fourier transform, and estimating eigenvalues. Properties of quantum physics such as entanglement and superposition [5] enable quantum computers to run quantum algorithms and to converge to solutions much faster than a conventional von-Neumann machine. The first steps towards quantum supremacy have already been taken [6], sparking further debate and research thrust in the quantum computing community [7]. Research estimates that a quantum computer should be capable of processing fully entangled qubits in the range of thousands to millions before they surpass the capabilities of classical computing machines and achieve quantum supremacy [8]. Current technology, including systems under development by companies such as D-Wave, IBM, Google, Intel, Rigetti, and IonQ [9–11], is still far away from achieving supremacy.

These hardware systems have limitations, for instance, it is difficult to conduct experiments using fully entangled qubit operations. The highest number of qubits physically entangled so far at 18 qubits has been claimed by Wang et al.

Extended author information available on the last page of the article.

[12]. There are also many challenges in the maintenance of a physical quantum computer [14]. For example, quantum computers need to correct errors and tolerate faults caused by quantum decoherence [13]. Physical quantum computers also require cryogenic operating temperatures and advanced equipment, making them expensive to acquire or access. There have been efforts by IBM, D-Wave and Rigetti [15] to make quantum computing systems accessible to the public via the cloud, but the access time and the available hardware capabilities remain highly limited. These factors are encouraging research into efficient simulation and emulation of quantum algorithms on classical high-performance computer systems using FPGAs. The motivation is to provide alternative and less costly methods of testing and improving quantum algorithms and to provide reliable support as quantum technology matures. Until large-scale quantum systems are complete, real-world quantum applications can be benchmarked and researched by simulating quantum algorithms on classical platforms. They can also be used as reference models for validation of quantum experiments. Therefore, there is an intrinsic value in the research and development of quantum simulations on classical systems such as FPGAs.

Among related efforts, we have reviewed quantum simulators developed for parallel architectures [16–20, 22] and works of quantum emulation on FPGA hardware [23–30]. FPGA emulators demonstrate parallelism by taking advantage of the concurrent nature of many quantum algorithms. They have demonstrated lower latency and greater speedup over quantum simulators running on sequential machines [27, 28]. Moreover, FPGAs have the distinguishing feature of dynamic reconfigurability over other parallel architectures, e.g., GPUs. One of the largest challenges of quantum hardware emulation is addressing resource constraints. FPGA emulation of quantum circuits is inevitably bound by the available system resources as the algorithmic resource requirements increase exponentially with the number of qubits. To alleviate the resource issue, most FPGA emulator architectures use fixed-point arithmetic as a trade-off between resource utilization and accuracy. Compared to a software simulator, the accuracy of the hardware emulator is reduced when emulating larger-scale systems, as quantization errors increase with an increase in the circuit size. Thus, scalability and accuracy are problem areas for FPGA-based quantum emulators.

In this paper, we propose using efficient computation techniques and hardware architectures for performing unitary transformations, i.e., unitary matrix multiplication, in support of emulation of quantum algorithms. The proposed techniques and architectures are space-efficient, providing scalable, high-precision, and high-throughput FPGA-based implementations of quantum algorithms. The objective of our work is to improve emulation by minimizing hardware utilization and improving system scalability. Previous works in the literature have generally used a quantum gate-based circuit modeling approach that leads to low scalability and low throughput. In our work, we propose a space-efficient emulation data-flow architecture that computes the output quantum state of a given quantum algorithm using a single unitary operation on the input state. Based on this model, we derive complex multiply-and-accumulate (CMAC)-based emulation architectures that leverage efficient computation techniques such as *lookup*, *dynamic generation*, and *streaming*. Our hardware architectures are generalized and can be used to emulate any quantum algorithm/circuit with the assumption that the algorithm can be reduced to a single unitary operation and the algorithm matrix can be precomputed or generated dynamically. As case studies for this work, we investigate the quantum Fourier transform (QFT) algorithm and Grover's quantum search algorithm. We present different hardware architectures exploring design and memory space, and provide analysis of time and space complexities for each architecture. Our experiments are prototyped on a high-performance reconfigurable computing (HPRC) platform and the obtained results show that the proposed emulation model improves system scalability. Finally, we provide a quantitative comparison of our results with existing work.

The rest of the paper is organized as follows. In Section 2, we provide a qualitative analysis of the work related to parallel quantum simulators and FPGA-based quantum emulators. In Section 3, we discuss fundamental concepts in quantum computing. In Section 4, we present the case study algorithms, i.e., QFT and Grover's search, and propose their corresponding quantum circuits. In Section 5, we present the emulation architectures and techniques for the proposed quantum emulation platform. In Section 6, we elaborate on the experimental setup and results, and provide quantitative comparison with existing work. Section 7 contains our concluding comments and some discussions of future work.

2 Related Work

2.1 Parallel Quantum Simulators

A variety of work has been done on quantum simulators that are based on parallel platforms. For example, in [16] the authors demonstrate a massively parallel quantum simulator implemented on different supercomputing platforms. However, their simulator while implementing up to 36 qubits, consumes a high amount of resources (1 terabyte memory) with no software optimizations reported. The authors in [17] propose a GPU-based simulator, showing implementation of up to 25 qubits, while a simple quantum gate was used in the simulation case study. The work in [18] also

uses GPUs and achieves simulation of entangled Hadamard gates up to 21 qubits. In [19], the authors demonstrate simulation of up to 38 qubits using a GPU accelerated platform. However, cost-prohibitive amounts of computing resources (2048 nodes and 24 cores/node) were dedicated in the simulation. One of the newer works on quantum simulation [20] uses a cluster supercomputing platform supported by the Alibaba group. In that work, the authors demonstrated simulation of up to 144 qubits with circuit depth of 27 gate levels using 131,072 processors and 1 petabyte memory. However, they have not investigated any quantum algorithms and the circuits consist of random gates. Furthermore, their simulator is shown to evaluate only one out of all possible output states. Existing parallel quantum simulators are highly costly since they consume large amount of resources in terms of required number of processors and system memory. Our proposed solution in this work is much less resource intensive and is therefore highly cost-effective, as will be demonstrated by the experimental results.

2.2 FPGA Quantum Emulators

An assortment of work has also been done on hardware emulation of quantum circuits using FPGAs. In [23] the authors presented a quantum processor that abstracted quantum circuit operations into binary logic. The proposed system was shown to emulate up to 75 qubits. However, the modeling methodology of the quantum operations was highly inaccurate due to the use of low precision (1 bit) for the representation of state coefficients. Moreover, hardware cost in terms of resource utilization was not reported. In [24] the authors implement an emulator based on a library of quantum gates. The gate operations were implemented using fixed-point arithmetic, and a low operating frequency of 82.4 MHz was reported for the emulation of 3-qubit QFT and Grover's search algorithm. In [25] the authors proposed a similar fixed-point emulator, reporting up to 3-qubit QFT, but details regarding both their approach and the mapping of the quantum algorithm to the proposed architecture are missing. Moreover, quantum entanglement was also missing in their model. The authors in [29] demonstrated a modular emulation framework based on a library of quantum gates and proposed space scheduling and space-time scheduling methods. Quantum algorithms such as QFT and Grover's search were emulated for up to 5 qubits. In [26] and [27] the authors present hardware architectures emulating QFT and Grover's search circuits. In their work, a maximum fixed-point precision of 24-bits was used to emulate up to 5-qubit QFT and 7-qubit Grover's search on a single FPGA. Scalability of their design is limited and there is no proposed solution to the problem of scalability. In [28] the authors propose a high-level synthesis (HLS) based emulation framework for QFT, but here also, the scalability of their design is limited and the authors did not address that limitation. In a related work, ProjectQ [21] has compared simulation and emulation results trying to showcase the superiority of quantum computer emulators in terms of performance. An extensive list of quantum simulators can be found in [22].

While modular and hierarchical modeling approaches in previous works improved re-usability, the modeling of each quantum gate as an individual component consumes greater resources, reduces accuracy, and limits scalability. Our proposed approaches in this paper significantly reduce the resource utilization and emulation times, thus improving scalability and allows us to use floating-point precision improving accuracy. In this work, we report the highest number of fully entangled qubits on a single FPGA among related work. Lastly, a fully pipelined design of the hardware architectures result in higher operating frequency and throughput compared to existing emulators.

3 Background

3.1 Quantum Computing

In a circuit-based model of quantum computing [5], the process of computation begins with the system being in a specific quantum state determined by entanglement of qubits. A quantum state $|\psi\rangle$ can be expressed as a superposition of 2^n=N basis states, as shown in Eq. 1, where n = number of qubits and N = number of states. The state coefficients $\alpha_0, \alpha_1, ..., \alpha_{N-1}$ are complex-valued and the magnitude of a given coefficient represents the probability of finding the qubit in the corresponding basis state. The quantum state goes through unitary transformations, or 'gates', depending on the steps and operations of a quantum algorithm to reach a final quantum state, see Eq. 2. These operations involve transformations on the complex coefficients of the basis states. Measurement of the final quantum state is a probabilistic sample across the basis states according to their coefficients.

$$|\psi\rangle = \sum_{i=0}^{N-1} \alpha_i |i\rangle \tag{1}$$

$$|\psi_{out}\rangle = U_m \cdot U_{m-1} \cdot \ ... \cdot U_2 \cdot U_1 \cdot |\psi_{in}\rangle \tag{2}$$

3.2 Qubits, Superposition, and Entanglement

The qubit is defined as the smallest unit of quantum information and can be represented by a two-level quantum mechanical system. Physical representations of the qubit can be photon polarization, superconducting Josephson junction, etc. [32]. A theoretical representation of the single

qubit is the Bloch sphere [5], as shown in Fig. 1. The poles of the Bloch sphere represent the two basis states of the qubit, i.e., $|0\rangle$ and $|1\rangle$. The qubit can be in a mixed state, which is any other point on the surface of the sphere. This means that the qubit can exist in a superposition of the two basis states at any point in time. The overall state of the qubit is satisfied by the linear superposition equation $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, where α and β are complex numbers whose values depend on φ and θ as shown in Fig. 1. Entanglement is another distinguishing property of qubits [5]. Two or more qubits may become entangled meaning that each entangled qubit becomes strongly correlated to the other and the quantum state cannot be factored into a tensor product of the individual qubits. When entangled, measurement across qubits are correlated, but isolated measurement of a single qubit demonstrates completely random behavior.

3.3 Quantum Gates

In quantum computing, quantum gates are the set of unitary transformations on qubits and are analogous to classical logic gates [14]. Quantum gates are used to manipulate the states of qubits and are represented as $N \times N$ matrices where $N = 2^n$ and n is the number of input qubits. In other words, a one-qubit gate is represented as a 2×2 matrix, a two-qubit gate is represented as a 4×4 matrix and so forth. A general representation of a 1-qubit gate, U, is shown in Eq. 3 where U is a unitary matrix, i.e., the inverse of U is equal to its transposed conjugate. Commonly used quantum gates such as H, X, cX, etc., are discussed in the next sections.

$$U = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \begin{array}{c} \text{where } a, b, c, d \in \mathbb{C}\,, \\ UU^{\dagger} = U(U^*)^T = I\,, \text{ and} \\ (U^*)^T \text{ is the transposed} \\ \text{conjugate of } U \end{array} \tag{3}$$

3.3.1 Hadamard Gate

The Hadamard, or H gate, is an important single-qubit gate because it creates an equal superposition of the basis states [14]. When an H gate is applied to the ground or $|0\rangle$ state, the resulting state will be an equal probability superposition between the $|0\rangle$ and $|1\rangle$ states, i.e., $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. The matrix representation of an H gate is given in Eq. 4.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{4}$$

3.3.2 X and cX Gate

The X gate is a single-qubit gate that is analogous to the NOT gate in classical computing and applies a swap on the α and β coefficients of a qubit [14]. A control qubit can be added to an X gate creating a two-qubit gate called the cX or Controlled NOT gate. If the control qubit is equal to $|1\rangle$, an X gate will be applied to the target qubit. The matrix representation of the X and cX gate is given in Eq. 5.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad cX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{5}$$

3.3.3 Z and cZ Gate

The Z gate, or Phase Inversion gate, inverts the phase of the input qubit, i.e., inverts the $|1\rangle$ basis state while leaving the $|0\rangle$ basis state unchanged [14]. Like the cX gate a control qubit can be added creating the cZ gate. The cZ gate will apply a Z gate to the target qubit when the control qubit is equal to $|1\rangle$. The matrix representation of the Z and cZ gate is given in Eq. 6.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad cZ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \tag{6}$$

3.3.4 R_k Gate

The R_k gate, or Phase Shift gate, shifts the phase of the input qubit by $2\pi i/2^k$ radians based on k, where k is an integer and i is $\sqrt{-1}$ [14]. The corresponding matrix for an R_k gate is shown in Eq. 7.

$$R_k = \begin{bmatrix} 1 & 0 \\ 0 & \exp(\frac{2\pi i}{2^k}) \end{bmatrix} \tag{7}$$

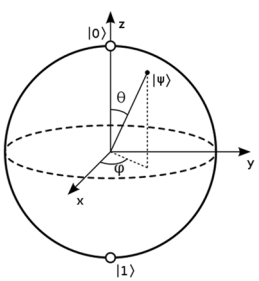

Figure 1 Bloch sphere representation of a single qubit.

$$\begin{aligned} |\psi\rangle &= cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle \\ &= \cos\left(\frac{\theta}{2}\right)|0\rangle + (cos\varphi + isin\varphi)\sin\left(\frac{\theta}{2}\right)|1\rangle \\ &= \alpha|0\rangle + \beta|1\rangle \\ where, 0 \le \theta \le \pi \; and \; 0 \le \varphi \le 2\pi \end{aligned}$$

4 Investigated Algorithms and Proposed Circuits

4.1 Quantum Fourier Transform

Quantum Fourier transform (QFT) is a fundamental part of many well-known quantum algorithms [1, 3]. QFT is the equivalent to the classical discrete Fourier transform (DFT). The mathematical model and quantum circuit for QFT can be determined from the classical DFT as demonstrated in [33]. The classical DFT transformation is given by

$$F_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} f_j w_n^{jk} \tag{8}$$

where $k = 0, 1, 2, ..., N-1$ and $w_n = e^{\frac{2\pi i}{N}}$. The QFT transformation for n qubits is shown in Eq. 9, where $|\psi\rangle$ is the input quantum state, n is the number of qubits, and $N = 2^n$ is the number of states. The input signal samples are encoded as a normalized amplitude sequence given by Eq. 10.

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{q=0}^{N-1} f(q\Delta t)|q\rangle$$
$$|\psi\rangle \xrightarrow{QFT} \frac{1}{\sqrt{N}} \sum_{q=0}^{N-1} f(q\Delta t)\omega_n^{qk}|q\rangle \tag{9}$$

$$\sum_{q=0}^{N-1} |f(q\Delta t)|^2 = 1 \tag{10}$$

QFT can be modeled as a circuit consisting of Hadamard and Controlled Phase Shift [14] quantum gates and the corresponding quantum circuit is shown in Fig. 2. The QFT transformation can be represented using a single unitary matrix, U_{QFT}, of size $N \times N$, as shown in Eq. 11.

$$U_{QFT} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w_n & w_n^2 & \cdots & w_n^{N-1} \\ 1 & w_n^2 & w_n^4 & \cdots & w_n^{2(N-1)} \\ 1 & w_n^3 & w_n^6 & \cdots & w_n^{3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_n^{N-1} & w_n^{2(N-1)} & \cdots & w_n^{(N-1)(N-1)} \end{bmatrix} \tag{11}$$

4.2 Quantum Grover's Search

4.2.1 Overview

Quantum Grover's search or Grover's algorithm is a quantum algorithm used to search over an unsorted list of N elements in $O(\sqrt{N})$ time [14]. The objective of Grover's search algorithm is to find the element s^* that makes $f(s^*) = 1$ and holds Eq. 12 true, where s^* belongs to the set $S = \{s_1, s_2, s_3 \ldots, s_N\}$, N is the cardinality of S, and f is a boolean function such that $f(x) \rightarrow \{0, 1\}$.

$$f(x) = \begin{cases} 1, & if\ x = s^* \\ 0, & if\ x \neq s^* \end{cases} \tag{12}$$

For a set of cardinality N, a classical computer takes on average $\frac{N}{2}$ queries to find s^*, whereas Grover's algorithm on a quantum computer accomplishes the same search in $\sqrt{N}$ queries. This results in a quadratic speedup on quantum machines in comparison to classical ones [14]. Grover's algorithm can also be used to search for multiple patterns. To do this effectively, the total number of patterns/elements must be known ahead of running the algorithm [14]. When searching for multiple patterns, Grover's algorithm will find any of the target patterns with equal probability. The

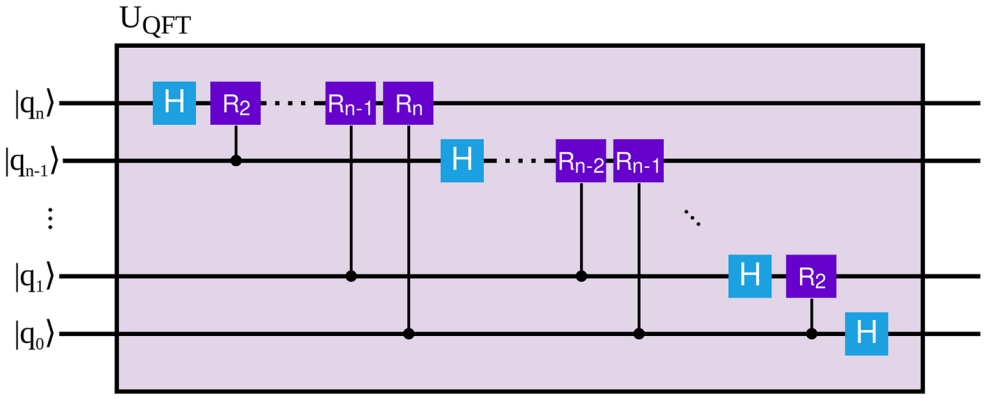

Figure 2 Quantum circuit for QFT.

input to Grover's algorithm are the items encoded as the basis states of a superimposed quantum state. Initially, the input state is in equal superposition, where their coefficients (amplitudes) are equal, and therefore the probabilities of locating any item in the list are also equal. To setup the input state, an H gate is applied to the ground or zero state. Then, in traditional Grover's algorithm, two operations are performed in multiple iterations on this input state: *phase inversion*, and *diffusion* [14]. However, as traditional Grover's algorithm requires a new quantum circuit every time the search pattern changes, we proposed a modified Grover's algorithm in [34] that dynamically modifies the search pattern with a single circuit. This modified Grover's algorithm alters the *phase inversion* step and requires an additional step called the *permutation step*. The general overview of this process is shown in Fig. 3.

4.2.2 Phase Inversion and Diffusion

The *phase inversion* operation is often thought of as a black box called the oracle [14], which will take the input set, then identify and invert the coefficient on the pattern(s) that are being searched for. To see how this works functionally, let our oracle be denoted as U_{oracle}. In Eq. 13 if $x \neq s^*$, then $f(x) = 0$ and $|x\rangle$ will have no change. Otherwise, $|x\rangle$ will be multiplied by -1 resulting in a phase inversion for $|x\rangle$.

$$U_{oracle}|x\rangle = (-1)^{f(x)}|x\rangle \tag{13}$$

Modification of the oracle circuit allows us to extend and generalize the algorithm to dynamically search for any pattern with a single quantum circuit. This modified oracle model, U_{oracle}, uses cX instead of the traditional X to dynamically modify the target pattern as seen in Fig. 4 [34]. In the traditional oracle circuit, an X gate should be placed if the basis state for the target qubit is $|0\rangle$ and no gate should be placed if the basis state for the target qubit is $|1\rangle$. In the modified oracle, ancilla qubits will hold the pattern that is being searched for and by being the control qubits for the cX gates, the X gate will be applied as needed. For single-pattern search, only the amplitude of the first state, $|0...0\rangle$, will be inverted. For multi-pattern search, cascaded incremental single-pattern oracle quantum circuits are performed to invert the first $N_{patterns}$ amplitudes as seen in Fig. 5. As each oracle circuit only inverts a single state and does not affect any other state, multiple oracle operations can be performed sequentially. The output from the oracle, $|\psi_1\rangle$, will subsequently be provided to the diffusion circuit for amplification.

In the *diffusion* operation, the inverted coefficient amplitudes will be amplified and the other coefficient amplitudes will be attenuated [14]. The general quantum circuit implementation of the *diffusion* operation is shown in Fig. 6. The quantum circuit only depends on the number of qubits and always follows the same pattern therefore no modification is needed in the modified Grover's algorithm. The *diffusion* operation is also sometimes referred to as *inversion about the mean*, as the quantum circuit detects negative amplitudes, amplifies them, and negates them resulting in a positive amplitude with a higher probability than the non-inverted states[14].

Repeating these two steps, i.e., *oracle*, and *diffusion*, m times will make the coefficient amplitudes of the sought patterns close to 1, thus making the probability of the entangled qubits collapsing to the target state also close to 1. The optimal number of iterations [35] m is shown in Eq. 14, where $N_{patterns}$ equals the number of solutions/patterns being searched for. This process can be represented using a single unitary matrix, $U_G = (U_{diffusion} \cdot U_{oracle})^m$. An

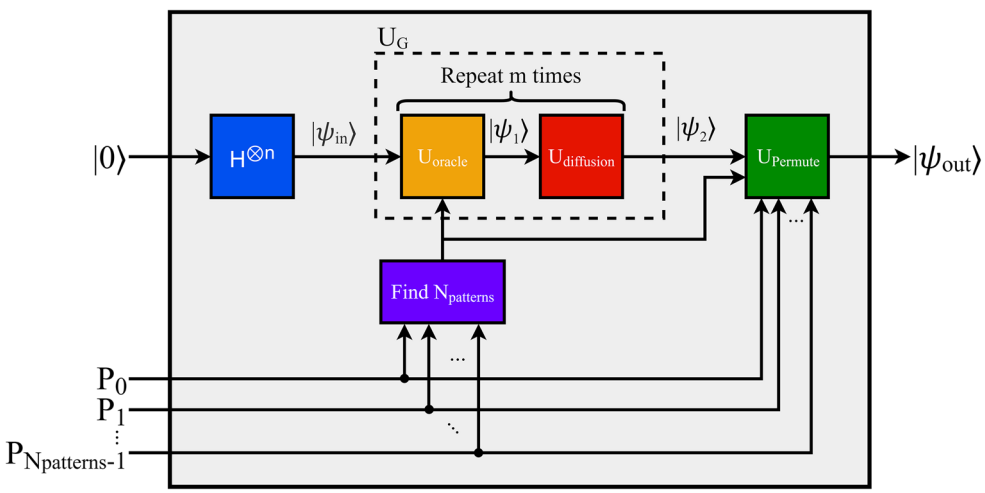

Figure 3 Modified multi-pattern Grover's algorithm.

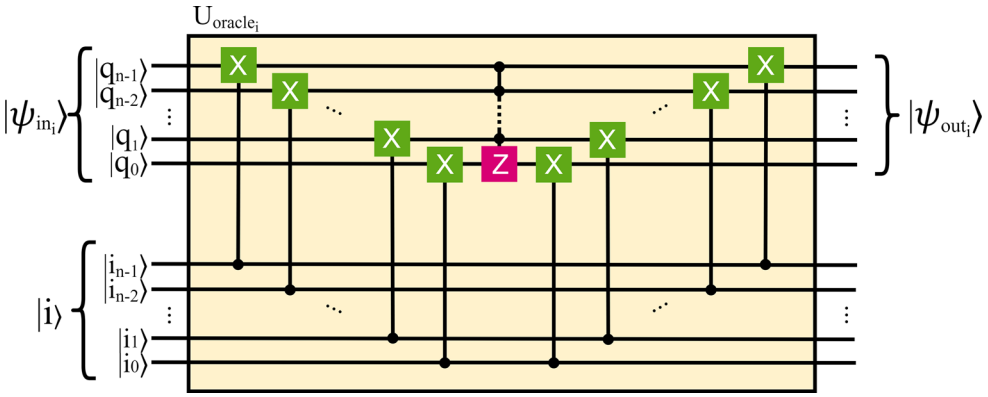

Figure 4 Modified oracle for Grover's algorithm for a single solution/pattern.

example U_G matrix for $n = 3$, $N_{patterns} = 3$, and $m = 1$ is shown in Eq. 15.

$$m = \left\lfloor \frac{\pi}{4 \sin^{-1}(\sqrt{\frac{N_{patterns}}{N}})} \right\rfloor \tag{14}$$

$$U_G = \begin{bmatrix} -\frac{3}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & -\frac{3}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & -\frac{3}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{3}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & \frac{3}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{3}{4} & -\frac{1}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{3}{4} & -\frac{1}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & \frac{3}{4} \end{bmatrix} \tag{15}$$

4.2.3 Quantum State Permutation

As our modification of Grover's algorithm only amplifies the first $N_{patterns}$ states, a permutation step $U_{permute}$, see Fig. 3, is required to shift the target patterns to the target states in the output superimposed quantum state $|\psi_{out}\rangle$ [34]. The quantum circuit for the permutation using cX gates is shown in Fig. 7 with the pattern ancilla qubits acting as the control qubits. The use of cX gates allows for inverting the selected qubits and move the amplitude to a selected state based on any given pattern P_i, where $i = 0, 1, 2, ..., n - 1$, see Fig. 7 . The permutation step allows for a more generalized and flexible design as the oracle/diffusion circuit only needs to amplify the first $N_{patterns}$ and the permutation circuit can dynamically swap the necessary states.

5 Emulation Architectures and Computation Techniques

Previous implementations of FPGA emulation of quantum circuits use circuit modeling approaches, where each stage or quantum gate of the quantum circuit is modeled for hardware using the corresponding classical operation [5]. This approach results in poor scalability, as the hardware resource utilization increases exponentially with circuit size, i.e., number of qubits and number of stages (cascaded gates). We propose a highly scalable, generalized emulation model that is optimized in terms of resource utilization and emulation time. A quantum algorithm is a series of

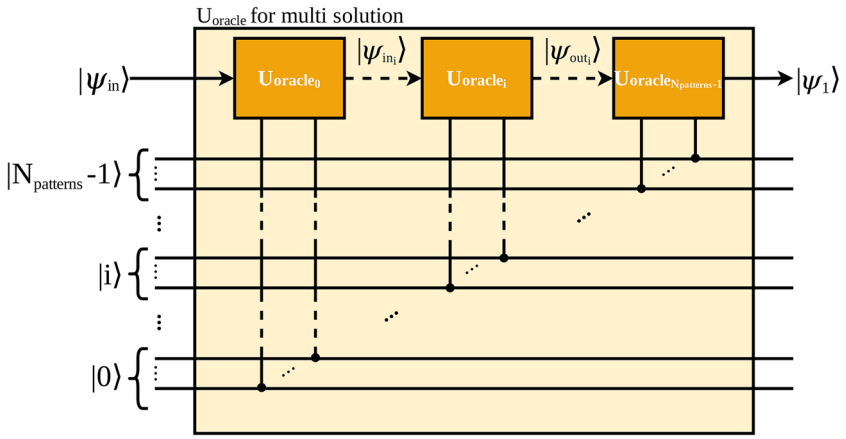

Figure 5 Modified oracle for Grover's algorithm for multiple solutions/patterns.

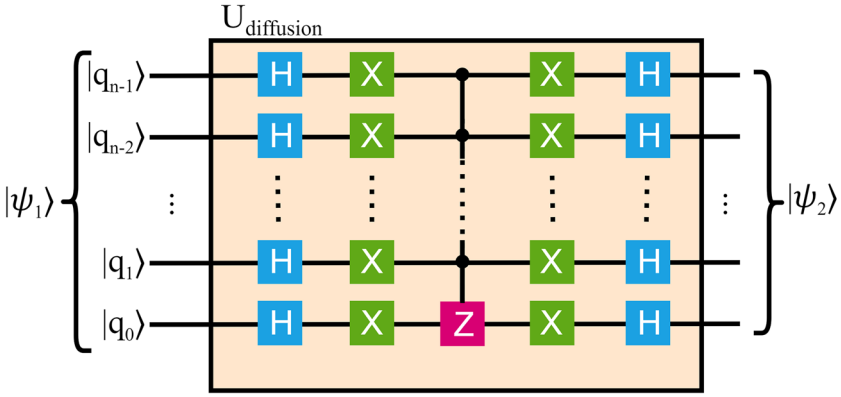

Figure 6 General inversion about the mean for Grover's algorithm.

transformations on the entangled quantum state of the qubits. The series of transformations can be represented as a single unitary complex-valued matrix, U_{ALG} [4]. An input quantum state, $|\psi_{in}\rangle$, can be represented by a vector comprising of the complex coefficients of the basis states of the quantum state. A complex vector-matrix multiplication of the input vector with the algorithm matrix produces the output quantum state vector, $|\psi_{out}\rangle$, whose coefficients represent the basis states of the output quantum state. We use this approach, illustrated in Fig. 8, as a model for designing hardware architectures for the proposed quantum emulation framework. This model is generalized and can be used to emulate any quantum algorithm/circuit that can be reduced to a single unitary operation, i.e., the transformation matrix can be pre-computed or generated dynamically.

By reducing the algorithm/circuit to a single transformation and performing the necessary vector-matrix product, the corresponding hardware implementation becomes independent of the circuit depth, resulting in a space- and time-efficient emulation architecture. This methodology assumes that the algorithm matrix is known and pre-computed, or can be dynamically generated. A limitation of this methodology is that for some algorithms, pre-computing and storing the algorithm matrix may not be feasible as the matrix dynamically changes with the algorithm input, for example, Shor's algorithm [1]. Dynamically generating the matrix is also difficult for algorithms with no pattern in the matrix elements, but it is certainly doable. To mitigate the limitations of pre-computing or dynamically generating the matrix and account for dynamically changing algorithm inputs and matrices, we incorporate data *streaming* techniques for emulation as elaborated in Section 5.2.3.

5.1 CMAC Architectures

To implement complex-valued vector-matrix multiplications on hardware (FPGA), we use a generic complex multiply-and-accumulate (CMAC) unit, as shown in Fig. 9. The inputs of the unit are complex values, i.e., elements of

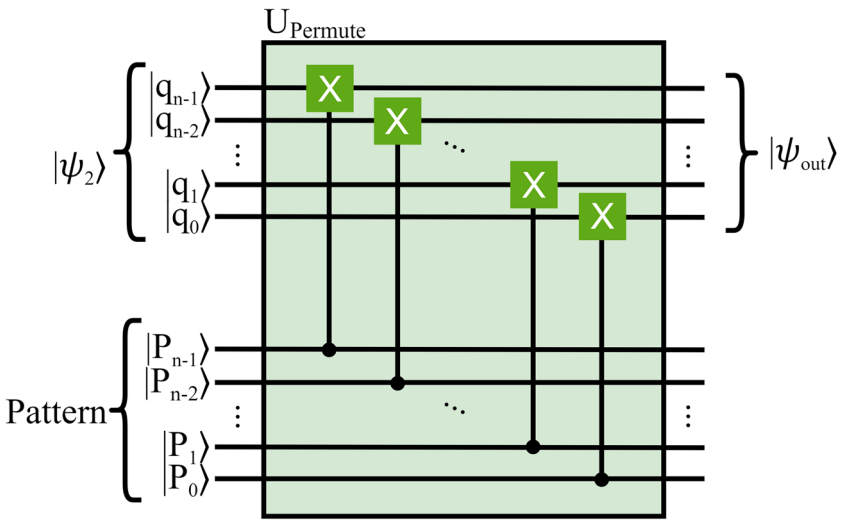

Figure 7 Quantum permutation circuit.

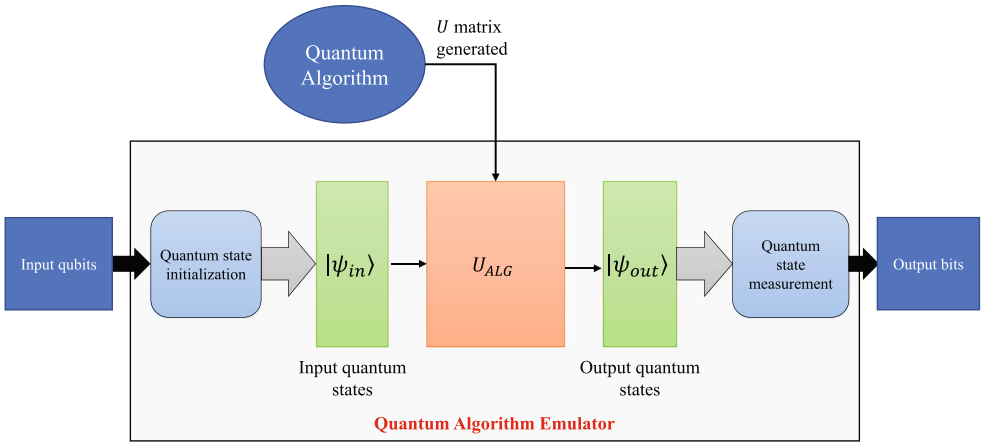

Figure 8 Model for quantum algorithm emulation.

the input state vector, $|\psi_{in}(j)\rangle$, and of the algorithm matrix, $U(i, j)$. The complex values are represented using 64 bits, with 32 floating-point bits for each of the real and imaginary parts. The benefit of using a CMAC is that different computation techniques, each with space and time trade-offs, can be applied during computation. To operate on complex values, the internal components of the CMAC (such as the multiplier and adder) have been designed for complex operations. The CMAC operations are described in Eq. 16. One CMAC unit performs, in total, four multiplications and four additions, see Fig. 9.

$$\begin{aligned} \psi_{out}^{real}(i) &= \sum_{j=0}^{N-1} R^{real}(i, j) \\ \psi_{out}^{imag}(i) &= \sum_{j=0}^{N-1} R^{imag}(i, j) \end{aligned} \tag{16}$$

where,

$$\begin{aligned} i &= 0, 1, 2, \ldots, (N-1), \\ R^{real}(i, j) &= (\psi_{in}^{real}(j) \times U^{real}(i, j)) \\ &\quad -(\psi_{in}^{imag}(j) \times U^{imag}(i, j)), \text{ and} \\ R^{imag}(i, j) &= (\psi_{in}^{imag}(j) \times U^{real}(i, j)) \\ &\quad +(\psi_{in}^{real}(j) \times U^{imag}(i, j)) \end{aligned}$$

We explored different hardware architectures, as listed in Table 1, by varying the number of CMAC instances. The purpose of this design space exploration was to implement either fully resource-optimized or fully latency-optimized designs to find an optimized CMAC configuration for developing a scalable hardware emulation framework. Space and time complexities for these architectures are summarized in Table 1.

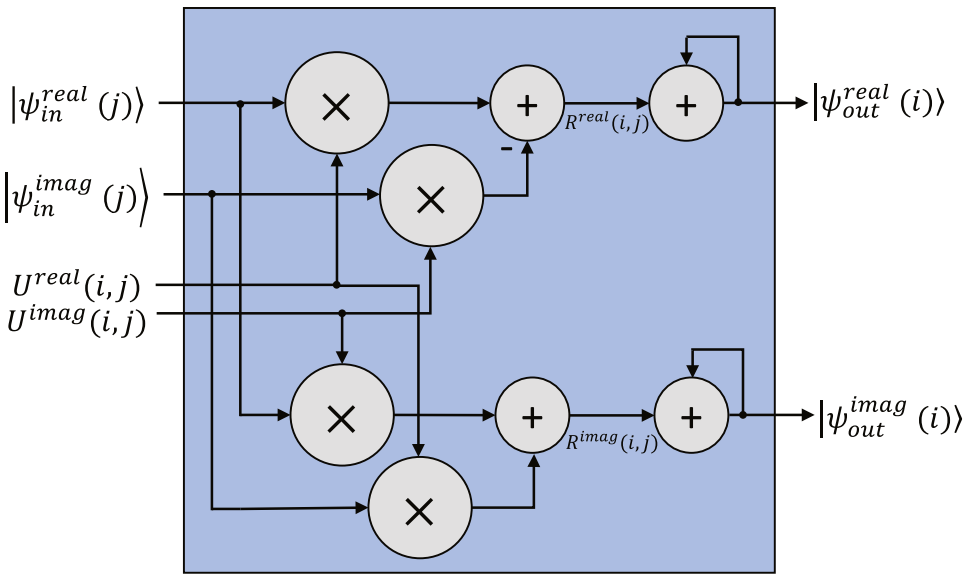

Figure 9 Complex multiply-and-accumulate unit.

Table 1 Space and time complexities of CMAC architectures.

CMAC Architecture	Complexity	
	Space (O_s)	Time (O_t)
Single	$O(1)$	$O(N^2)$
N-concurrent	$O(N)$	$O(N)$
Dual-sequential	$O(1)$	$O(N^2)$

5.1.1 Single-CMAC Architecture

For a fully resource-optimized design, we instantiate only one CMAC unit and feed it with one algorithm matrix element and one input quantum state vector element for each clock cycle. This is repeated for all N^2 items in the U_{ALG} matrix. For this architecture, the time complexity is $O(N^2)$, as shown in Eq. 17, where T_{clock} is the clock period. The hardware takes N cycles to store each input coefficient and N^2 cycles to process each element of the algorithm matrix, in addition to some initial latency, L_1. The space complexity is $O(1)$, as shown in Eq. 18, since a single CMAC instance is being used.

$$O_{time} = (L_1 + N + N^2) \times T_{clock} = O(N^2) \tag{17}$$

$$O_{space} = 1 \times \text{CMAC} = O(1) \tag{18}$$

5.1.2 *N*-concurrent-CMAC Architecture

In a fully parallel implementation, N CMAC instances are used to operate concurrently, for processing each row of the U_{ALG} matrix. The time complexity of this design, as shown in Eq. 19, is effectively $O(N)$ as it takes N cycles to store the input states, and N more cycles to concurrently process all N rows of the algorithm matrix, along with initial latency L_2. The space complexity now becomes $O(N)$, due to the N instances of CMAC units, as shown in Eq. 20.

$$O_{time} = (L_2 + 2N) \times T_{clock} = O(N) \tag{19}$$

$$O_{space} = N \times \text{CMACs} = O(N) \tag{20}$$

5.1.3 Dual-sequential-CMAC Architecture

In this implementation, two CMAC instances are utilized sequentially. After the initial latency L_3, the first CMAC processes the first row of the matrix while the input vector is being stored, and the second CMAC instance continues the subsequent processing of the remaining rows using the stored inputs. This implementation has double the resource requirements of the first architecture but has the benefit of improvement in execution time. The time complexity is determined as in Eq. 21 and the space complexity is given by Eq. 22.

$$O_{time} = (L_3 + N^2 - 1) \times T_{clock} = O(N^2) \tag{21}$$

$$O_{space} = 2 \times \text{CMACs} = O(1) \tag{22}$$

5.2 CMAC Computation Techniques

For the CMAC architectures, in our previous work we have explored different computation techniques such as *lookup*, and *dynamic generation* [36], to investigate trade-offs in area and speed of the emulator. In this work, we leverage data *streaming* techniques and apply them for the CMAC architecture, and show that it is better optimized in both area and speed, compared to the previous techniques. For completeness, we discuss the previously used *lookup*, and *dynamic generation* techniques along with the proposed *streaming* technique in the following sections.

5.2.1 Lookup-based CMAC

Look-up-tables (LUTs) simplify hardware design by replacing complex parts of computation with simple array-indexed operations. It is generally implemented as an array in memory that stores pre-calculated values which results in low resource requirements. In the CMAC architecture, we use the process of *lookup* to fetch pre-computed algorithm matrix values from memory during complex computation operations. A limitation of this technique is that for some algorithms, the algorithm matrix changes dynamically with the inputs, hence this method will not be feasible in those cases. We combine this *lookup* approach with the *dual-sequential-CMAC architecture* to optimize the design in terms of speed. The total memory, M_L, required using this combination is derived in Eq. 23, assuming 32-bit floating point numbers are used for each real and imaginary component of the complex matrix and vector elements.

$$M_L = M_{vec} + M_{mat} = 8N + 8N^2 = 2^{n+3} + 2^{2n+3} \tag{23}$$

5.2.2 Dynamic Generation-based CMAC

The *lookup* approach is optimized for speed, but storing the algorithm matrix consumes resources that increase exponentially with circuit size. For resource utilization optimization and improved scalability, we propose integrating the *dual-sequential-CMAC* architecture with a dynamic approach that involves generating the algorithm matrix values at runtime, storing only input vectors in memory. The advantage of this method is that it significantly reduces the total memory utilization, M_{DG}, of the simulation, as shown in Eq. 24. The algorithm matrix U_{QFT}, see Eq. 11, is generated as part of the architecture using dedicated hardware

units as shown in Fig. 10. The operations of this architecture are summarized in Eq. 25.

$$M_{DG} = M_{vec} = 8N = 2^{n+3} \tag{24}$$

$$\begin{aligned} R(i) &= R(i-1).\omega \\ U(i,j) &= U(i,j-1).R(i) \end{aligned} \tag{25}$$

$$\begin{aligned} where, \quad \omega &= e^{\hat{i}\frac{2\pi}{N}} = cos\left(\frac{2\pi}{N}\right) + \hat{i}.sin\left(\frac{2\pi}{N}\right), \\ i,j &= 0 \rightarrow N-1, \\ and, \quad R(0) &= 1,\ U(0,0) = 1 \end{aligned}$$

The drawback of this technique is that the dynamic generation hardware can introduce pipeline latencies depending on the complexity of the algorithm, degrading the speed of the overall emulation. Furthermore, designing a dedicated hardware generation unit also requires us to find and exploit some pattern in the algorithm matrix, which might not be possible in every case. Generally, it is hard to efficiently generate the matrix values if the algorithm matrix doesn't have a special structure, therefore the generation hardware would be complex, and the approach may not be feasible for particular algorithms.

5.2.3 Stream-based CMAC

While the *lookup* approach improves speed, it sacrifices area, and similarly, while *dynamic generation* improves area, it sacrifices speed. We investigate a more optimal approach that sustains both speed and area improvements and improves scalability and latency of the emulator. Instead of being stored into on-chip resources (OCR) or on-board memory (OBM), or dynamically generated during computation, the algorithm matrix elements are streamed in during computation as an input stream from an external control processor. The cost of streaming is typically the I/O channel latency between the control processor and the FPGA, which is negligible relative to the compute time necessary for processing the algorithm matrix. The contribution of this technique is that it greatly reduces the constraint on memory requirement compared to the *lookup* based technique, while also avoiding the hardware cost and bottleneck of using the *dynamic generation* technique. As a result, a significantly higher number of qubits can be emulated on the same FPGA area. The total memory requirement, M_S, using this method is equivalent to M_{DG} and shown in Eq. 26. The top-level view of the emulator design using the data streaming technique is shown in Fig. 11.

$$M_S = M_{DG} = M_{vec} = 8N = 2^{n+3} \tag{26}$$

6 Experimental Results and Analysis

Our experimental work was carried out on DS8, a state-of-the-art high-performance reconfigurable computing (HPRC) system provided by DirectStream [37]. A single C2 compute node of the DS8 system is equipped with high-end Intel-Altera Arria 10 FPGA and on-board memory (OBM) SRAM and SDRAM modules, as shown in Fig. 12. The FPGA on-chip resources (OCR) consists of 427,200 Adaptive Logic Modules (ALMs), 2,713 Block RAMs (BRAMs), and 1,518 Digital Signal Processing (DSP) blocks. The DS8 hardware system is integrated with DirectStream's programming environment, which succeeds the previous Carte-C compiler [38]. DirectStream's environment was selected for our experimental work because it uses a High-Level Language (HLL) which facilitates the development of complex, parallel, and reconfigurable codes in an efficient manner. Moreover, an HLL design flow provides more control over floating-point resources, compared to conventional Hardware Description Language (HDL) design flow. The study in [39] showed that Carte-C had highly productive environment, short acquisition time, short learning time, and short development time.

We performed several experiments and implementations on the DS8 system. All results are collected from hardware deployments on FPGAs with complete system and memory interface implementations on the DS platform. The hardware architectures were implemented in High-Level Synthesis (HLS) using C++. The high-level C++ codes were built for hardware using Quartus Prime version 17.0.2 on an

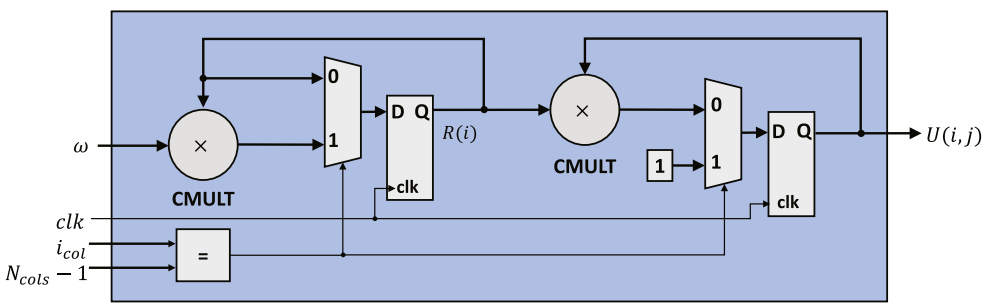

Figure 10 Hardware architecture for dynamic generation of the QFT algorithm matrix.

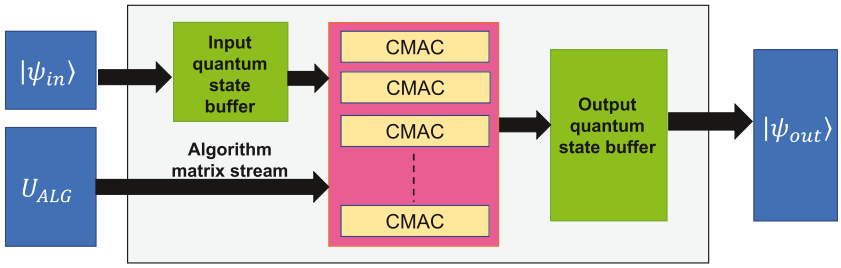

Figure 11 Architecture of the quantum algorithm emulator using data streaming.

Arria 10 10AX115N4F45E3SG FPGA, and the resource utilizations and latencies were obtained from compiler reports. As a result of fully pipelining the designs, a high operating frequency of 233 MHz was reported, resulting in high system throughput. For functional verification of the DS hardware designs, reference circuit models were developed in MATLAB. The models that we developed in MATLAB and our DS hardware designs both use single-precision floating-point arithmetic, and the emulation results on the hardware and circuit simulations in MATLAB were identical in accuracy. The algorithm matrix and input vectors were generated from the MATLAB models and imported into the DS environment. In the experiments, when using the lookup technique, the matrix/vector elements were stored into the on-chip resources (OCR) or the on-board memory (OBM), and when using the streaming technique the elements were streamed in from the DS host control processor.

6.1 Implementation Results using *Lookup* Technique

We first implemented the single, N-concurrent, and dual-sequential-CMAC architectures using the *lookup* technique and both on-chip and on-board memory configurations. Emulation of the QFT algorithm was performed using these implementations. Table 2 reports the on-chip implementation results for the single-CMAC architecture. Figure 13 presents the resource utilizations as a function of the number of qubits. From this experiment, the ALM and DSP resource utilizations reported were constant, which was a result of using one CMAC hardware unit. The BRAM units are used for on-chip storage and lookup of the algorithm matrix/vector elements, and the BRAM resource utilization increases exponentially with the number of qubits. An increase in emulation time with circuit size is also observed, as expected, due to the increasing number of temporal iterations of the single CMAC unit.

The implementation results of the N-concurrent-CMAC architecture are reported in Table 3 and Fig. 14. There is a consistent increase in ALMs as the number of CMAC hardware units in this architecture increases with the number of qubits. The Intel Quartus Prime hardware compiler applies optimizations to maintain the constant utilizations for scarce DSP units. For example, the compiler automatically searches for functions using common inputs or completely independent functions to be placed in one ALM to make efficient use of device resources [40].

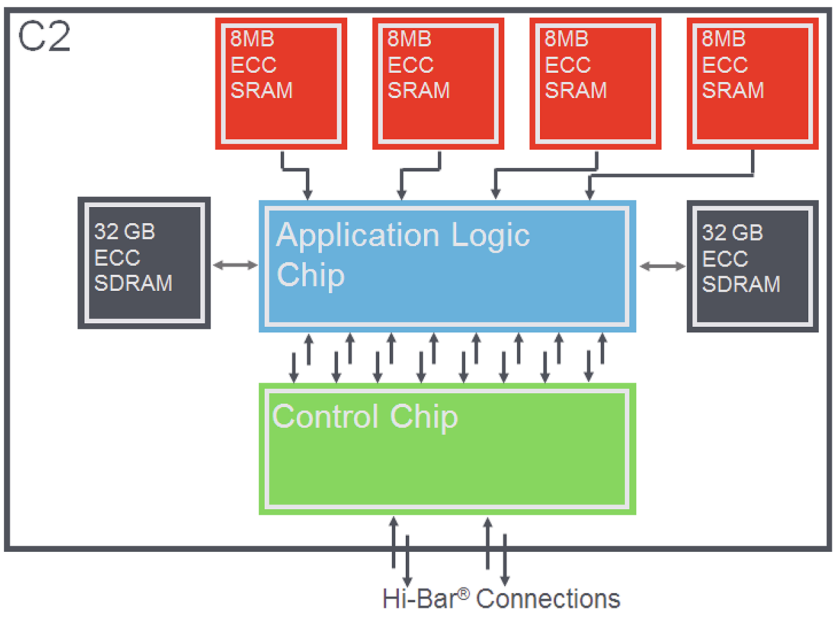

Figure 12 DS8 system architectures.

Table 2 QFT implementation results using single-CMAC architecture, on-chip resources, and *lookup*.

Number of Qubits	OCR[a] Utilization (%)			Emulation time (sec)
	ALMs	BRAMs	DSPs	
2	10.30	8.04	1.05	1.4E-6
3	10.24	8.12	1.05	1.15E-6
4	10.24	8.11	1.05	2.01E-6
5	10.27	8.18	1.05	5.37E-6
6	10.26	8.55	1.05	1.87E-5
7	10.26	10.25	1.05	7.17E-5
8	10.29	16.73	1.05	3.19e-4
9	10.31	41.28	1.05	0.0013

[a]Total on-chip resources: $N_{ALM} = 427,200, N_{BRAM} = 2,713, N_{DSP} = 1,518$

Table 3 QFT implementation results using N-concurrent-CMAC architecture, on-chip resources, and *lookup*.

Number of Qubits	OCR[a] Utilization (%)			Emulation time (sec)
	ALMs	BRAMs	DSPs	
2	10.70	7.08	1.05	6.78E-7
3	10.74	7.08	1.05	7.64E-7
4	11.53	7.08	1.05	9.36E-7
5	17.10	7.08	1.05	1.28E-6
6	24.50	7.08	1.05	1.97E-6
7	39.50	7.08	1.05	3.34E-6
8	74.88	7.08	1.05	6.09E-6

[a]Total on-chip resources: $N_{ALM} = 427,200, N_{BRAM} = 2,713, N_{DSP} = 1,518$

We implemented the third proposed architecture, i.e., dual-sequential-CMAC, in which two sequentially operating CMAC computations are overlapped with data write operations. Table 4 and Fig. 15 show the obtained results. The results are similar to the first architecture implementation in which the ALM utilization increases exponentially while the remaining resource utilization is fixed. In Fig. 16, we compare the emulation time of all three implementations and we observe that the N-concurrent implementation has the highest performance. This is due to the parallel operation of the CMAC units. The trade-off for the N-concurrent implementation is area since we were only able to emulate up to 8 qubits, while using the single-CMAC and dual-sequential-CMAC architectures we were able to emulate up to 9 qubits. Any larger circuit exceeds the FPGA on-chip resources allocated for storing the computation vectors and algorithm matrix.

On-board memory (OBM) configurations of the proposed architectures were also implemented to scale the emulation to a higher number of qubits. The storage of state vectors and algorithm matrix is performed using on-board SRAM and on-board SDRAM memories respectively. We implemented this for the single-CMAC and dual-sequential-CMAC architectures running QFT. For the N-concurrent-CMAC architecture, an OBM configuration leads to SDRAM read/write contention issues, which significantly degrades the performance, and it was not considered for implementation. Table 5 shows the results from implementation of the single-CMAC architecture with an OBM configuration. The obtained results demonstrate that the on-chip resources are constant with increasing qubits because they are only used for the fixed number of adders/multipliers of the single CMAC unit. Therefore, the scalability limit is determined by the size of the on-board memory, which is

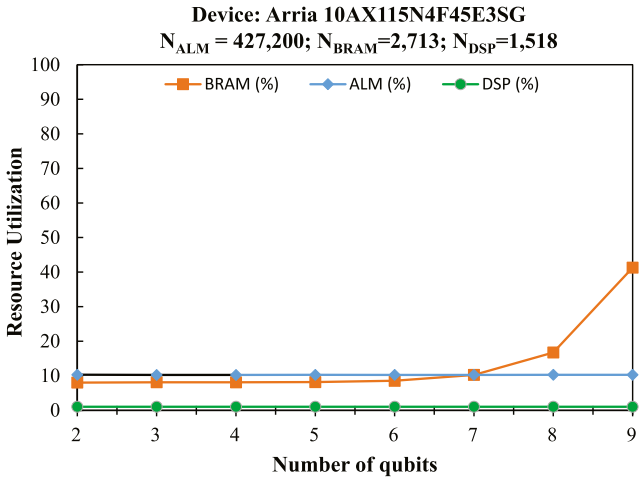

Figure 13 QFT on-chip resource utilizations using single-CMAC architecture and *lookup*.

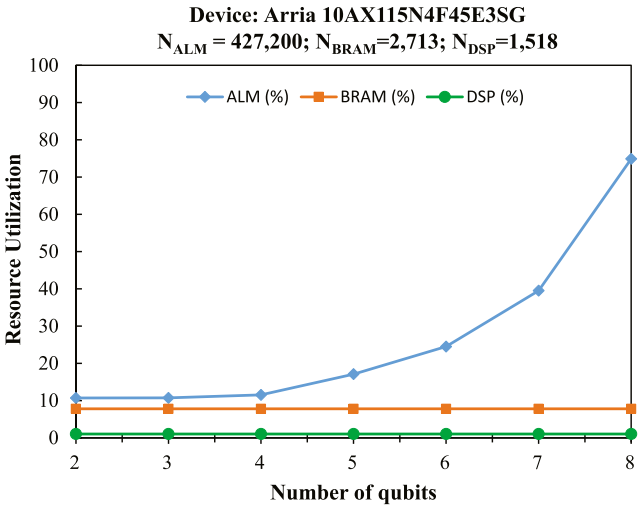

Figure 14 QFT on-chip resource utilization using N-concurrent-CMAC architecture and *lookup*.

Table 4 QFT implementation results using dual-sequential-CMAC architecture, on-chip resources and *lookup*.

Number of Qubits	OCR[a] Utilization (%)			Emulation time (sec)
	ALMs	BRAMs	DSPs	
2	12.39	8.55	2.11	7.55E-7
3	12.34	8.55	2.11	9.61E-7
4	12.36	8.63	2.11	1.79E-6
5	12.43	8.70	2.11	5.08E-6
6	12.38	8.99	2.11	1.83E-5
7	12.39	10.69	2.11	7.1E-5
8	12.37	17.18	2.11	0.0003
9	12.37	43.54	2.11	0.0011

[a]Total on-chip resources: $N_{ALM} = 427,200$, $N_{BRAM} = 2,713$, $N_{DSP} = 1,518$

being used to store the state vectors and algorithm matrix. Using 1 × 32 GB SDRAM bank of a single C2 compute node, it was possible to emulate up to 16-qubit QFT, compared to 9-qubit QFT using on-chip resources.

We also implemented the dual-sequential-CMAC architecture with OBM configuration and the results are shown in Table 6. For both OBM configurations, we observe, as expected, that the on-chip resources (OCR) on the FPGA are fixed for emulation of a particular algorithm due to the fixed architecture of the CMAC. Figure 17 shows the comparison of the emulation times between the two configurations. It can be observed that the dual-sequential-CMAC architecture performs better in terms of emulation time. The time complexity of $O(N^2)$ for single-CMAC and dual-sequential-CMAC, see Table 1, is also reflected in these results. From our experiments, we conclude that the

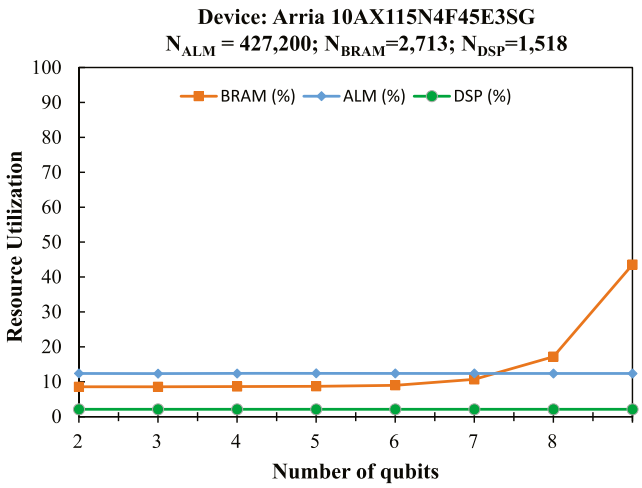

Figure 15 QFT on-chip resource utilizations using dual-sequential-CMAC architecture and *lookup*.

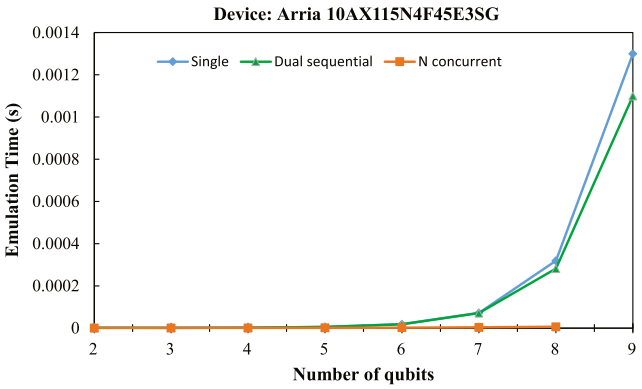

Figure 16 Comparison of QFT emulation times using CMAC architectures, on-chip resources and *lookup*.

proposed dual-sequential CMAC architecture provides the highest performance in terms of emulation time when compared to other configurations. Integrating on-board memory with that architecture enables us to emulate QFT using 16 fully-entangled qubits on a single Arria 10 FPGA node with 32 GB memory, with an emulation time of 18 seconds.

6.2 Implementation Results using *Dynamic Generation* Technique

To emulate larger QFT circuits, we perform implementation of the dual-sequential-CMAC architecture with OBM and using the *dynamic generation* technique. QFT results are shown in Table 7. Using the *dynamic generation* technique, the algorithm matrix elements are generated in hardware dynamically, and the SDRAM stores only the input/output state vectors. Therefore, up to 32-qubit emulation of QFT was possible on a single FPGA with 32 GB on-board memory, compared to the maximum of 16 qubits using *lookup*. On-chip resources are slightly higher because of the additional generation hardware units. Although QFT circuits for up to 32 qubits were successfully built on hardware, emulation times were unrealistically large for circuits larger than 20 qubits, and the runtimes for these circuits were estimated using an accurate model derived from Eqs. 17, 19, and 21 for the proposed pipelined architectures.

6.3 Implementation Results using *Streaming* Technique

Finally, we implement the dual-sequential-CMAC architecture with OBM and use the data *streaming* technique. The algorithm matrix elements are streamed in during computation and only the state vectors require storage. As a case study for this technique, we emulated Grover's search algorithm. The target patterns were set to {1 11 2 13 4 15 6

Table 5 QFT implementation results using single-CMAC architecture, on-board memory, and *lookup*.

Qubits	OCR[a] Utilization (%)			OBM[b] Utilization (bytes)		Emulation time (sec)
ALMs	BRAMs	DSPs	SRAM	SDRAM		
2	10.71	8.44	1.05	32	128	1.7E-6
3	10.71	8.44	1.05	64	512	2.0E-6
4	10.71	8.44	1.05	128	2K	3.9E-6
5	10.71	8.44	1.05	256	8K	1.1E-5
6	10.71	8.44	1.05	512	32K	3.9E-5
7	10.71	8.44	1.05	1K	128K	0.00015
8	10.71	8.44	1.05	2K	512K	0.00061
9	10.71	8.44	1.05	4K	2M	0.00241
10	10.71	8.44	1.05	8K	8M	0.00963
11	10.71	8.44	1.05	16K	32M	0.03851
12	10.71	8.44	1.05	32K	128M	0.15399
13	10.71	8.44	1.05	64K	512M	0.61586
14	10.71	8.44	1.05	128K	2G	2.36324
15	10.71	8.44	1.05	256K	8G	9.853
16	10.71	8.44	1.05	512K	32G	39.4209

[a]Total on-chip resources: $N_{ALM} = 427,200$, $N_{BRAM} = 2,713$, $N_{DSP} = 1,518$

[b]Total on-board memory: 4 parallel SRAM banks of 8MB each and 2 parallel SDRAM banks of 32GB each

7}, where each number corresponds to the index of a target state we are searching for. Output results demonstrated high probability amplitudes identifying the target states, and these were verified against results obtained from software simulations in MATLAB. The hardware implementation results are shown in Table 8. The space complexity of this architecture is $O(1)$, as there are only two operating CMACs. The time complexity is $O(N^2)$ due to the computation of N^2 elements of the algorithm matrix, see Table 1. Hardware builds of up to 32-qubit circuits for Grover's algorithm were performed on a single FPGA with 32 GB SDRAM memory. Emulation times for circuits larger

Table 6 QFT implementation results using dual-sequential-CMAC architecture, on-board memory, and *lookup*.

Qubits	OCR[a] Utilization (%)			OBM[b] Utilization (bytes)		Emulation time (sec)
	ALMs	BRAMs	DSPs	SRAM	SDRAM	
2	12	8.63	2.11	32	128	7.55E-7
3	12	8.63	2.11	64	512	9.61E-7
4	12	8.63	2.11	128	2K	1.79E-6
5	12	8.63	2.11	256	8K	5.08E-6
6	12	8.63	2.11	512	32K	1.83E-5
7	12	8.63	2.11	1K	128K	7.10E-5
8	12	8.63	2.11	2K	512K	0.00028
9	12	8.63	2.11	4K	2M	0.00113
10	12	8.63	2.11	8K	8M	0.00451
11	12	8.63	2.11	16K	32M	0.018002
12	12	8.63	2.11	32K	128M	0.072006
13	12	8.63	2.11	64K	512M	0.2888021
14	12	8.63	2.11	128K	2G	1.152083
15	12	8.63	2.11	256K	8G	4.608329
16	12	8.63	2.11	512K	32G	18.4331

[a]Total on-chip resources: $N_{ALM} = 427,200$, $N_{BRAM} = 2,713$, $N_{DSP} = 1,518$

[b]Total on-board memory: 4 parallel SRAM banks of 8MB each and 2 parallel SDRAM banks of 32GB each

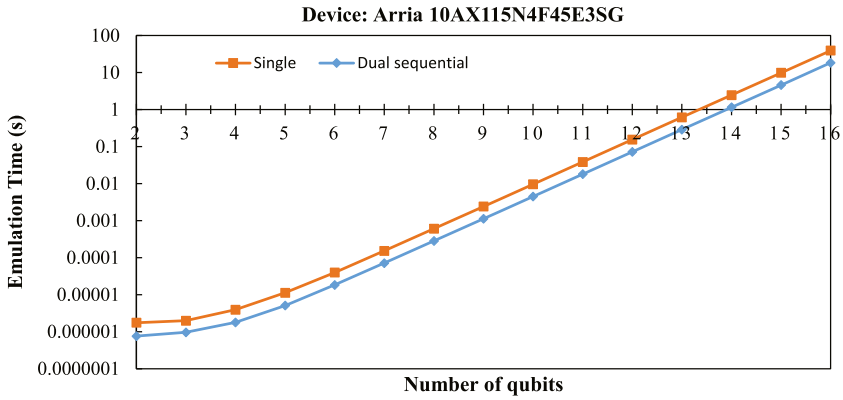

Figure 17 Comparison of QFT emulation times using CMAC architectures with on-board memory.

Table 7 QFT implementation results using dual-sequential-CMAC architecture, on-board memory and *dynamic generation*.

Number of qubits	OCR utilization[a] (%)			OBM[b] utilization (bytes)	Emulation time (sec)
	ALMs	BRAMs	DSPs	SDRAM	
2	13.16	9.58	3.23	32	1.99E-6
3	13.16	9.58	3.23	64	2.20E-6
4	13.16	9.58	3.23	128	3.03E-6
5	13.16	9.58	3.23	256	6.32E-6
6	13.16	9.58	3.23	512	1.95E-5
7	13.16	9.58	3.23	1K	7.22E-5
8	13.16	9.58	3.23	2K	2.83E-4
9	13.16	9.58	3.23	4K	1.13E-3
10	13.16	9.58	3.23	8K	4.50E-3
11	13.16	9.58	3.23	16K	1.80E-2
12	13.16	9.58	3.23	32K	7.20E-2
13	13.16	9.58	3.23	64K	2.88E-1
14	13.16	9.58	3.23	128K	1.15E0
15	13.16	9.58	3.23	256K	4.61E0
16	13.16	9.58	3.23	512K	1.84E1
17	13.16	9.58	3.23	1M	7.37E1
18	13.16	9.58	3.23	2M	2.95E2
19	13.16	9.58	3.23	4M	1.18E3
20	13.16	9.58	3.23	8M	4.72E3
21	13.16	9.58	3.23	16M	1.89E4
22	13.16	9.58	3.23	32M	7.55E4
23	13.16	9.58	3.23	64M	3.02E5[c]
24	13.16	9.58	3.23	128M	1.21E6[c]
25	13.16	9.58	3.23	256M	4.83E6[c]
26	13.16	9.58	3.23	512M	1.93E7[c]
27	13.16	9.58	3.23	1G	7.73E7[c]
28	13.16	9.58	3.23	2G	3.09E8[c]
29	13.16	9.58	3.23	4G	1.24E9[c]
30	13.16	9.58	3.23	8G	4.95E9[c]
31	13.16	9.58	3.23	16G	1.98E10[c]
32	13.16	9.58	3.23	32G	7.92E10[c]

[a]Total on-chip resources: $N_{ALM} = 427,200$, $N_{BRAM} = 2,713$, $N_{DSP} = 1,518$

[b]Total on-board memory: 4 parallel SRAM banks of 8MB each and 2 parallel SDRAM banks of 32GB each

[c]Results are projected using a performance estimation model

Table 8 Grover's algorithm implementation results using dual-sequential-CMAC architecture, on-board memory and *streaming*.

Number of qubits	OCR[a] utilization (%)			OBM[b] utilization (bytes)	Emulation time (sec)
	ALMs	BRAMs	DSPs	SDRAM	
2	11	8	1	32	2.3E-6
3	11	8	1	64	2.54E-6
4	11	8	1	128	3.4E-6
5	11	8	1	256	7.22E-6
6	11	8	1	512	2.0E-5
7	11	8	1	1K	7.36E-5
8	11	8	1	2K	2.8E-4
9	11	8	1	4K	1.13E-3
10	11	8	1	8K	4.5E-3
11	11	8	1	16K	1.8E-2
12	11	8	1	32K	7.2E-2
13	11	8	1	64K	2.88E-1
14	11	8	1	128K	1.15E0
15	11	8	1	256K	4.61E0
16	11	8	1	512K	1.84E1
17	11	8	1	1M	7.37E1
18	11	8	1	2M	2.95E2
19	11	8	1	4M	1.18E3
20	11	8	1	8M	4.72E3
21	11	8	1	16M	1.89E4
22	11	8	1	32M	7.5E4
23	11	8	1	64M	3.02E5[c]
24	11	8	1	128M	1.2E6[c]
25	11	8	1	256M	4.83E6[c]
26	11	8	1	512M	1.93E7[c]
27	11	8	1	1G	7.73E7[c]
28	11	8	1	2G	3.09E8[c]
29	11	8	1	4G	1.24E9[c]
30	11	8	1	8G	4.95E9[c]
31	11	8	1	16G	1.98E10[c]
32	11	8	1	32G	7.92E10[c]

[a]Total on-chip resources: $N_{ALM} = 427,200$, $N_{BRAM} = 2,713$, $N_{DSP} = 1,518$

[b]Total on-board memory: 4 parallel SRAM banks of 8MB each and 2 parallel SDRAM banks of 32GB each

[c]Results are projected using a performance estimation model

than 22 qubits were estimated using the performance model derived from Eqs. 17, 19, and 21. The emulation time as a function of the number of qubits is shown in Fig. 18. The developed architectures for this technique were fully pipelined and an operating frequency of 233 MHz was reported by the compiler.

The *streaming* technique does not require any generation hardware and therefore can be used to emulate any quantum algorithm that is reducible to a single unitary transformation. The complexity of the algorithm does not affect the performance of emulation. Therefore, emulation of other algorithms would yield the same results in terms of hardware utilization and emulation time. The reconfigurable architecture of our emulator allows improvement of the time complexity to $O(N)$ by instantiating N parallel instances of CMAC units for vector matrix multiplications. To emulate a larger number of qubits using the single CMAC approach, the amount of on-chip resources, and/or on-board memory would need to be increased. Other approaches include adopting multi-CMAC architectures, and/or using a multi-node architecture where the design is partitioned among the nodes [29]. In other words, scaling to higher quantum circuit

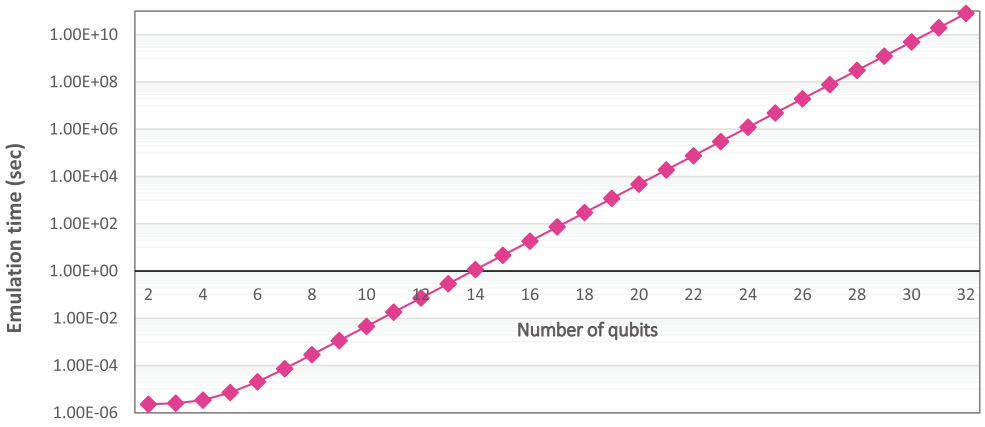

Figure 18 Grover's search algorithm emulation using dual-sequential-CMAC architecture, on-board memory, and *streaming*.

sizes would require using more hardware resources such as on-chip resources (OCR), on-board memory (OBM), number of CMACs, or the number of FPGA nodes.

6.4 Comparison with Related Work

We here quantitatively compare our obtained results with current and existing work on FPGA-based emulation, as shown in Table 9. Among the related work on FPGA emulation of quantum circuits [23–30], our proposed emulation framework has the capability of emulating the largest quantum circuit (31-qubit quantum sorting [31]), with the highest operating frequency (233 MHz) at the highest accuracy (64-bit floating-point precision). Current FPGA hardware emulators have many discrepancies such as incomplete reporting of their results on resource utilization, operating frequency, and emulation time, which makes a comprehensive comparison difficult. In our comparison, we included only hardware emulators, as most parallel-software-simulators are extremely costly and resource-hungry, e.g., 131,072 processors and 1 petabyte memory [20], and cannot be compared fairly with single node FPGA-based emulators. Also, they have mostly simulated random quantum circuits and not full algorithms, e.g., [19], while in our work we emulated various important algorithms in quantum computing, e.g., QFT and Grover's search.

7 Conclusions and Future Work

A major limitation in FPGA emulation of quantum circuits is scalability. In this paper, we proposed space-efficient computation techniques and hardware architectures for performing unitary transformations, for improved and scalable emulation of quantum algorithms. High accuracy is maintained through the use of floating-point operations,

Table 9 Comparison of proposed work with existing work on FPGA emulation.

Reported Work	Algorithm	Number of qubits emulated	Precision frequency (MHz)	Operating	Emulation time(sec)
Fujishima [23], 2003	Shor's factoring	–	–	80	10
Khalid et al. [24], 2004	QFT*	3	16-bit fixed pt.	82.1	61E-9
	Grover's search	3	16-bit fixed pt.	82.1	84E-9
Aminian et al. [25], 2008	QFT	3	16-bit fixed pt	131.3	46E-9
Lee et al. [27], 2016	QFT	5	24-bit fixed pt.	90	219E-9
	Grover's search	7	24-bit fixed pt.	85	96.8E-9
Silva et al. [28], 2017	QFT	4	32-bit floating pt.	–	4E-6
Pilch et al. [30], 2018	Deutsch	2	–	–	–
Proposed work	Quantum Fourier Transform [27]	22	32-bit floating pt.	233	7.55E4[a]
	Quantum Haar Transform [36]	30	32-bit floating pt.	233	13.8
	Grover's search	22	32-bit floating pt.	233	7.5E4[a]
	Quantum sorting[31]	31	64-bit floating pt.	233	1.14E11[a]

[a]Results are projected using a performance estimation model

and high throughput is achieved by fully pipelining the hardware architectures. We also provided experimental analysis to compare the proposed approaches in terms of area and speed. Our results include fully deployable synthesized hardware builds of up to 32-qubit QFT and 32-qubit Grover's search circuits. While emulation runs for up to 22-qubit circuits could be fully performed/executed on hardware, emulation runtimes for larger circuits were estimated using an accurate derived performance model. The FPGA used was Intel Arria 10 with 32 GB on-board memory.

In future work, we will investigate more complex quantum algorithms such as Shor's algorithm for integer factoring, and applications such as quantum machine learning and quantum cryptography. We will conduct accuracy trade-off studies between fixed-point and floating-point implementations of our emulation framework targeting embedded systems. We plan to investigate techniques inherent to FPGAs such as full run-time reconfiguration (FRTR) and partial run-time reconfiguration (PRTR) for dynamic adaptation of quantum algorithm simulations. Also, more efficient I/O data conversion techniques between classical and quantum systems will be investigated.

References

1. Shor, P.W. (1994). Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of the 35^{t}h IEEE Annual Symposium on Foundations of Computer Science (SFCS '94), Santa Fe, NM, USA* (pp. 124–134).
2. Grover, L.K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing (STOC '96), Philadelphia, PA USA* (pp. 212–219).
3. Deutsch, D., & Jozsa, R. (1992). Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A:, Mathematical and Physical Sciences*, *439*(1907), 553–558.
4. Williams, C.P. (2011). *Explorations in Quantum Computing*. New York: Springer.
5. Nielsen, M.A., & Chuang, I.L. (2010). *Quantum Computation and Quantum Information*. Cambridge: Cambridge university press.
6. Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J.C., Barends, R., Biswas, R., Boixo, S., Brandao, F.G.S.L., Buell, D.A., Burkett, B., Chen, Y., Chen, Z., Chiaro, B., Collins, R., Courtney, W., Dunsworth, A., Farhi, E., Foxen, B., Fowler, A., Gidney, C., Giustina, M., Graff, R., Guerin, K., Habegger, S., Harrigan, MP., Hartmann, M.J., Ho, A., Hoffmann, M., Huang, T., Humble, T.S., Isakov, S.V., Jeffrey, E., Jiang, Z., Kafri, D., Kechedzhi, K., Kelly, J., Klimov, P.V., Knysh, S., Korotkov, A., Kostritsa, F., Landhuis, D., Lindmark, M., Lucero, E., Lyakh, D., Mandrà, S., McClean, J.R., McEwen, M., Megrant, A., Mi, X., Michielsen, K., Mohseni, M., Mutus, J., Naaman, O., Neeley, M., Neill, C., Niu, M.Y., Ostby, E., Petukhov, A., Platt, J.C., Quintana, C., Rieffel, E.G., Roushan, P., Rubin, N.C., Sank, D., Satzinger, K.J., Smelyanskiy, V., Sung, K.J., Trevithick, M.D., Vainsencher, A., Villalonga, B., White, T., Yao, Z.J., Yeh, P., Zalcman, A., Neven, H., Martinis, J.M. (2019). Quantum supremacy using a programmable superconducting processor. *Nature*, *574*, 505–510.
7. Pednault, E., Gunnels, J., Nannicini, G., Horesh, L., Wisnieff, R. (2019). Leveraging Secondary Storage to Simulate Deep 54-qubit Sycamore Circuits. arXiv:910.09534v2 [quant-ph]. https://arxiv.org/pdf/1910.09534.pdf. Last Accessed: 2020.
8. Boixo, S., Isakov, S.V., Smelyanskiy, V.N., Babbush, R., Ding, N., Jiang, Z., Bremner, M.J., Martinis, J.M., Neven, H. (2018). Characterizing quantum supremacy in near-term devices. *Nature Physics*, *14*(6), 595.
9. D-Wave Systems Inc (2020). The D-Wave 2000Q^{TM} Quantum computer Technology Overview. https://www.dwavesys.com/sites/default/files/D-Wave%202000Q%20Tech%20Collateral_0117F2.pdf. Last Accessed:.
10. Gomes, L. (2018). Quantum computing: both here and not here. *IEEE Spectrum*, *55*(4), 42–47.
11. Wang, B. (2020). IonQ Has the Most Powerful Quantum Computers With 79 Trapped Ion Qubits and 160 Stored Qubits. https://www.nextbigfuture.com/2018/12/ionq-has-the-most-powerful-quantum-computers-with-79-trapped-ion-qubits-and-160-stored-qubits.html. Last Accessed:.
12. Wang, X., Luo, Y.H., Huang, H., Chen, M.C., Su, Z., Chen, C., Li, W., Fang, Y., Jiang, X., Zhang, J., Li, L., Liu, N., Lu, C., Pan, J. (2018). 18-Qubit Entanglement with Six Photons' Three Degrees of Freedom. *Physical review letters*, *120*(26), 260502.
13. Shor, P.W. (1995). Scheme for reducing decoherence in quantum computer memory. *Physical Review*, *52*(4), 737–741.
14. Yanofsky, N.S., & Mannucci, M.A. (2008). *Quantum Computing for Computer Scientists* Vol. 20. Cambridge University Press: Cambridge.
15. Harris, M. (2018). D-Wave Launches Free Quantum Cloud Service IEEE Spectrum. Available from: https://spectrum.ieee.org/tech-talk/computing/hardware/dwave-launches-free-quantum-cloud-service. Last Accessed: March 2020.
16. De Raedt, K., Michielsen, K., DeRaedt, 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.
17. Amariutei, A., & Caraiman, S. (2011). Parallel quantum computer simulation on the GPU. In *15^{t}h International Conference on System Theory, Control and computing, Sinaia, Romania* (pp. 1–6).
18. Avila, A., Maron, A., Reiser, R., Pilla, M., Yamin, A. (2014). GPU-Aware distributed quantum simulation. In *proceedings of the 29^{t}h Annual ACM Symposium on Applied Computing (SAC'14), New York, NY, USA* (pp. 860–865).
19. Jones, T., Brown, A., Bush, I., Benjamin, S. (2020). QuEST and High Performance Simulation of Quantum Computers. arXiv:1802.08032v6 [quant-ph]. https://arxiv.org/pdf/1802.08032.pdf. Last Accessed:.
20. Chen, J., Zhang, F., Huang, C., Newman, M., Shi Y. (2020). Classical simulation of intermediate-size quantum circuits. arXiv:1805.01450v2 [quant-ph]. https://arxiv.org/pdf/1805.01450.pdf. Last Accessed:.
21. Haner, T., Steiger, D., Smelyanskiy, M., Troyer, M. (2020). High Performance Emulation of Quantum Circuits. arXiv:1604.06460 [quant-ph]. https://arxiv.org/pdf/1604.06460.pdf. Last Accessed:.
22. Quantiki (2019). List of QC Simulators. https://quantiki.org/wiki/list-qc-simulators. Last Accessed:.
23. Fujishima, M. (2003). FPGA-based high-speed emulator of quantum computing. In *IEEE International Conference on Field-Programmable Technology (FPT 2003), Tokyo, Japan, 21-26*.
24. Khalid, A.U., Zilic, Z., Radecka, K. (2004). FPGA Emulation of quantum circuits. In *IEEE International Conference on Computer*

design: VLSI in Computers and Processors (ICCD 04), San Jose, USA (pp. 310–315).
25. Aminian, M., Saeedi, M., Zamani, M.S., Sedighi, M. (2008). FPGA-Based circuit model emulation of quantum algorithms. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI '08), Montpellier, France* (pp. 399–404).
26. Khalil-Hani, M., Lee, Y.H., Marsono, M.N. (2015). An accurate FPGA-based hardware emulation on quantum fourier transform. In *Proceedings of the Australasian Symposium on Parallel and Distributed Computing (ausPDC'15), Sydney, Australia* (pp. 23–30).
27. Lee, Y.H., Khalil-Hani, M., Marsono, M.N. (2016). An FPGA-based quantum computing emulation framework based on serial-parallel architecture. International Journal of Reconfigurable Computing.
28. Silva, A., & Zabaleta, O.G. (2017). FPGA Quantum computing emulator using high level design tools. In *Eight Argentine Symposium and Conference on Embedded Systems (CASE'17), Buenos Aires, Argentina* (pp. 1–6).
29. Mahmud, N., & El-Araby, E. (2018). Towards Higher Scalability of Quantum Hardware Emulation using Efficient Resource Scheduling. In *The 3rd IEEE International Conference on Rebooting Computing (ICRC 2018), Washington, DC, USA*.
30. Pilch, J., & Długopolski, J. (2018). An FPGA-based real quantum computer emulator. *Journal of Computational Electronics*, *18*(1), 239–342.
31. Mahmud, N., Srimoungchanh, B., Hasse-Divine, B., Blankenau, N., Kuhnke, A., El-Araby, E. (2019). Combing Perfect Shuffle and Bitonic Networks for Efficient Quantum Sorting. 2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing.
32. Quantum Computing Report (2019). Qubit Technology. https://quantumcomputingreport.com/scorecards/qubit-technology/ Last Accessed: 2020.
33. Marquezino, F.L., Portugal, R., Sasse, F.D. (2010). Obtaining the Quantum Fourier Transform from the classical FFT with QR decomposition. *Journal of Computational and Applied Mathematics*, *235*, 74–81.
34. Mahmud, N., Haase-Divine, B., Srimoungchanh, B.K., Blankenau, N., Kuhnke, A., El-Araby, E. (2019). Emulating Multipattern Quantum Grover's Search on a High-Performance Reconfigurable Computer. Poster session in The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC19) Denver, CO,USA.
35. Boyer, M., Brassard, G., Høyer, P., Tapp, A. (1998). Tight Bounds on Quantum Searching. *Fortschritte der Physik*, *46*(4-5), 493–505.
36. Mahmud, N., El-Araby, E., Caliga, D. (2019). Scaling reconfigurable emulation of quantum algorithms at High-Precision and High-Throughput. *Quantum Engineering*, *1*, e19.
37. DirectStream, LLC https://directstream.com. Last Accessed: (2020).
38. El-Araby, E., El-Ghazawi, T., Le Moigne, J., Irish, R. (2009). Reconfigurable processing for satellite On-Board automatic cloud cover assessment (ACCA). *Journal of Real-Time Image Processing (JRTIP)*, *4*(3), 245–259.
39. El-Araby, E., Merchant, S.G., El-Ghazawi, T. (2013). *Assessing Productivity of High-Level Design Methodologies for High-Performance Reconfigurable Computers. High-performance Computing using FPGAs*, (pp. 719–745). New York: Springer.
40. Altera. Intel Stratix 10 Logic Array Blocks and Adaptive Logic Modules User guide. https://www.altera.com/en_S/pdfs/literature/hb/stratix-10/ug-s10-lab.pdf Last accessed: (2020).

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Naveed Mahmud received his B.Sc. degree in Electrical and Electronic Engineering from American International University, Bangladesh, in 2013. He is currently a Ph.D. candidate in the department of Electrical Engineering and Computer Science (EECS) at the University of Kansas (KU), USA. He is also working as a Graduate Teaching Assistant and Graduate Research Assistant at KU. He is teaching the Digital Systems Design and Digital Logic Design Labs. Prior to joining KU, he worked as a Design Verification Engineer for UlkaSemi, developing automated, self-checking testbenches and test platforms for ASICs. His research is mainly focused on quantum computing, emulation of quantum algorithms on reconfigurable architectures, and high-level synthesis techniques for FPGAs. His publication record includes four journal articles and nine conference papers.

Bennett Haase-Divine received his B.Sc. degree in Computer Engineering from the department of Electrical Engineering and Computer Science (EECS) at the University of Kansas (KU), USA, in Spring 2020. During his time at KU, Bennett worked as a grader for Computer Architecture as well as an undergraduate Research Assistant focusing on quantum computing and emulation of quantum algorithms on reconfigurable architectures. Bennett currently works as an Embedded Software Engineer at Ainstein AI Inc., Kansas, USA. His publication record includes two journal articles and three conference papers.

Annika Kuhnke received her B.Sc. degree in Computer Science from the department of Electrical Engineering and Computer Science (EECS) at the University of Kansas(KU), USA, in Spring 2020. As an honor undergraduate student at KU, Annika worked as a grader for Computer Architecture as well as an undergraduate Research Assistant focusing on quantum computing and emulation of quantum algorithms on reconfigurable architectures. She currently works as an Application Developer at Honeywell FM&T and is working towards her M.Sc. in Information Technology. Her publication record includes two journal articles and three conference papers.

Apurva Rai is an honor undergraduate student with double majors in Computer Science and Mathematics at the University of Kansas (KU), USA. He works as a grader for Applied Statistics as well as an undergraduate Research Assistant. His research interests focus on Quantum Computing, and on Elliptic Curve Factorization Methods and their implementation using Tensor Cores. His publication record includes two journal articles.

Andrew MacGillivray is an undergraduate student in Computer Engineering in the department of Electrical Engineering and Computer Science (EECS) at the University of Kansas (KU), USA. Andrew has acquired industry experience through his ongoing employment as a programmer in the aerospace sector, where he focuses on the development of business-administration tools. Andrew's current research is focused on the emulation of quantum algorithms using FPGAs. His publication record includes two journal articles.

Esam El-Araby is an assistant professor in the department of Electrical Engineering and Computer Science (EECS) at the University of Kansas (KU), USA. He received his M.Sc. and Ph.D. degrees in Computer Engineering from the George Washington University (GWU), USA, in 2005, and 2010 respectively. Dr. El-Araby worked at the High-Performance Computing Laboratory (HPCL) at GWU as well as the National Science Foundation (NSF) Center for High-Performance Reconfigurable Computing (NSF-CHREC). His research work was mainly funded by organizations such as DoD, DARPA, NSF, and NASA. Dr. El-Araby is the Principal Investigator (PI) of the most prestigious award of Faculty Early Career Development Program (NSF-CAREER) from the National Science Foundation (NSF). His primary research interests are in computer architecture, reconfigurable computing, quantum computing, quantum communications, reversible computing, heterogeneous computing, biologically-inspired and neuromorphic architectures, and evolvable hardware.

Affiliations

Naveed Mahmud[1] · Bennett Haase-Divine[1] · Annika Kuhnke[1] · Apurva Rai[1] · Andrew MacGillivray[1] · Esam El-Araby[1]

✉ Naveed Mahmud
naveed_923@ku.edu

Bennett Haase-Divine
b.haase-divine@ku.edu

Annika Kuhnke
akkuhnke@ku.edu

Apurva Rai
apurva@ku.edu

Andrew MacGillivray
amacgillivray@ku.edu

Esam El-Araby
esam@ku.edu

[1] University of Kansas, Lawrence, Kansas, 66045, USA