IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 1, 2022, accepted June 14, 2022, date of publication June 17, 2022, date of current version June 23, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3183993

A Scalable Emulator for Quantum Fourier Transform Using Multiple-FPGAs With High-Bandwidth-Memory

HASITHA MUTHUMALA WAIDYASOORIYA, (Member, IEEE), HIROKI OSHIYAMA, YUYA KUREBAYASHI, MASANORI HARIYAMA, (Associate Member, IEEE), AND MASAYUKI OHZEKI
Graduate School of Information Sciences, Tohoku University, Sendai, Miyagi 980-8579, Japan

Corresponding author: Hasitha Muthumala Waidyasooriya (hasitha@tohoku.ac.jp)

This work was supported in part by Ministry of Education, Culture, Sports, Science and Technology, (Japan) (MEXT) Grants-in-Aid for Scientific Research (KAKENHI) under Grant 19K11998 and Grant 20H04197.

ABSTRACT Quantum computing is regarded as the future of computing that hopefully provides exponentially large processing power compared to the conventional digital computing. However, current quantum computers do not have the capability to correct errors caused by environmental noise, so that it is difficult to run useful algorithms that require deep quantum circuits. Therefore, emulation of quantum circuits in digital computers is essential. However, emulation of large quantum circuits requires enormous amount of computations, and leads to a very large processing time. To reduce the processing time, we propose an FPGA emulator with high-bandwidth-memory to emulate quantum Fourier transform (QFT), which is a major part of many quantum algorithms. The proposed FPGA emulator is scalable in terms of both processing speed and the number of qubits, and extendable to multiple FPGAs. We performed QFT emulations up to 30 qubits using two FPGAs. According to the measured results, we have achieved 23.6 $\sim$ 24.5 times speed-up compared to a fully optimized 24-core CPU emulator.

INDEX TERMS Quantum computing, quantum circuits, high-bandwidth memory, FPGA, quantum Fourier transform.

I. INTRODUCTION

Quantum computing [1] is regarded as the future of computing that hopefully provides exponentially large processing power compared to the conventional digital computing. Some examples of the latest quantum computers are 5000-qubit quantum annealer by D-Wave Systems [2], and 127-qubit quantum processor (Eagle) by IBM [3]. Quantum annealers such as [2] is limited to solve combinatorial optimization problems. Quantum-gate based computers such as [3] are regarded as the potential replacement for general purpose digital computers. However, such quantum computers do not have the capability to correct errors caused by environmental noise, making it difficult to run useful algorithms that require deep quantum circuits. Therefore, emulation of quantum circuits in digital computers is extremely important, since it allows us to invent and evaluate novel algorithms, explore new fields such as quantum machine learning [4], etc. Such quantum circuit emulators provide us a great advantage in future when more powerful quantum computers are available.

Emulation of large quantum circuits requires enormous amount of computations that leads to very large processing time. Therefore, acceleration of quantum circuit emulation is necessary. Usually, quantum computing emulations contain a large amount of parallel operations. FPGAs are an excellent candidate to accelerate such computations. Modern FPGAs contain hardened floating-point modules that provide over 20 TFLOPs of single-precision computation power [5], and extremely capable of data-center oriented applications. FPGAs are highly scalable and can be connected directly using 400 Gbps high-speed connections. However, one weak area of FPGAs is the small external memory bandwidth. To solve this problem, high-bandwidth memory (HBM2) based FPGAs [6]–[8] have been introduced. These FPGAs contain a large number of parallel HBM modules that provide nearly 10 times larger memory bandwidth compared to discrete DDR4-SDRAMs. On the other hand, low-level optimization is required to optimally utilize memory sub-systems containing multiple HBM modules.

The associate editor coordinating the review of this manuscript and approving it for publication was Mario Donato Marino.

In this paper, we propose an HBM-based scalable FPGA emulator for quantum Fourier transform (QFT) [9], [10]. QFT is a major part of many quantum algorithms such as Shor's algorithm [11], quantum eigenvalue estimation [12], etc. We propose a memory allocation method to fully utilize the memory capacity and bandwidth of all HBM memories in multiple FPGAs. The proposed emulator is scalable in terms of both processing speed and the number of qubits, and extendable to multiple FPGAs. We can optimize it according to the resource and bandwidth constraints. We use OpenCL design environment [13] to fully implement the proposed architecture on one-FPGA and two-FPGAs. Using two FPGAs, we can perform QFT emulations up to 30 qubits. According to the measured results, we achieved 23.6 ∼ 24.5 times speed-up against fully optimized 24-core CPU implementation. To the best of our knowledge, this work is the largest and fully functional QFT emulation on FPGAs up-to-date.

II. BACKGROUND AND RELATED WORK

A. FUNDAMENTALS OF QUANTUM CIRCUITS

In classical computing, we use "bits" to represent information. A bit has a well-defined state of either "0" or "1". A "quantum bit" or "qubit" on the other hand does not have a well defined state until a measurement is made. When a measurement is made, we get either 0 or 1. Otherwise, the qubit is in a superposition which is mathematically represented by Eq.(1), where φ is a qubit and α and β are complex numbers.

$$|\varphi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1}$$

In Eq.(1), $|\alpha|^2$ is the probability of φ to be in the state of $|0\rangle$, and $|\beta|^2$ is the probability of φ to be in the state of $|1\rangle$. Since the sum of all probabilities equals to 1, $|\alpha|^2 + |\beta|^2 = 1$. The notations $|0\rangle$ and $|1\rangle$ represent qubit states, and distinguish themselves from the values 0 and 1. These two states are represented by orthonormal vectors in the vector space as shown by Eqs.(2) and (3).

$$|0\rangle = \begin{pmatrix}1\\0\end{pmatrix} \tag{2}$$

$$|1\rangle = \begin{pmatrix}0\\1\end{pmatrix} \tag{3}$$

We can obtain the vector representation of $|\varphi\rangle$ by substituting Eqs.(2) and (3) to Eq.(1). This is called the *state vector* and shown in Eq.(4).

$$|\varphi\rangle = \begin{pmatrix}\alpha\\\beta\end{pmatrix} \tag{4}$$

We use state vectors to explain the quantum Fourier transform in section II-B.

B. QUANTUM FOURIER TRANSFORM

Quantum Fourier Transform (QFT) is the quantum implementation of the Discrete Fourier Transform (DFT) [14]. Figure 1 shows an example of 4-qubit "quantum Fourier transform" circuit using quantum gates. Two types of quantum gates, 1-qubit Hadamard gates and 2-qubit controlled phase-shift gates are used. The k^{th} Hadamard gate is denoted by H_k. The term $R_{k,l}$ denotes a controlled phase-shift gate, where integers k and l are target and control bits respectively. The 1-qubit Hadamard gate and 2-qubit controlled phase-shift gate are given by the following matrix forms.

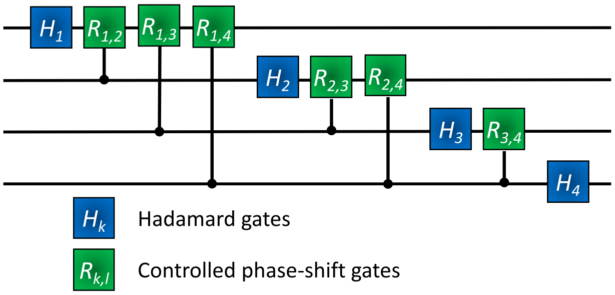


FIGURE 1. Circuit of quantum Fourier transform.

$$H_k = \begin{pmatrix}\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}\end{pmatrix}$$

$$R_{k,l} = \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & e^{2\pi i/2^k}\end{pmatrix}$$

We use a state vector of 2^n states to represent n qubits in a superposition. The input state vector $|\psi\rangle$ is represented as follows.

$$|\psi\rangle = a(0..00)|0..00\rangle + a(0..01)|0..01\rangle \ldots + a(1..11)|1..11\rangle$$

After applying a quantum gate, the output state vector $|\psi'\rangle$ is represented as follows.

$$|\psi'\rangle = a'(0..00)|0..00\rangle + a'(0..01)|0..01\rangle \ldots + a'(1..11)|1..11\rangle$$

For example, coefficients corresponding to j^{th} qubit of the output state vector after applying the Hadamard gate, are calculated as follows.

$$a'(*..*0_j*..*) = \frac{1}{\sqrt{2}}\left(a\left(*..*0_j*..*\right) + a\left(*..*1_j*..*\right)\right)$$

$$a'(*..*1_j*..*) = \frac{1}{\sqrt{2}}\left(a\left(*..*0_j*..*\right) - a\left(*..*1_j*..*\right)\right)$$

The j^{th} bit is represented by 0_j and 1_j. Similarly, we can calculate the coefficients of the state vector after applying controlled phase-shift gate.

Algorithm 1 shows the method used to emulate quantum Fourier transform of n-qubits. It contains two loops. The outer-most loop performs the Hadamard gate computations, while the inner-most loop performs the controlled phase-shift gates computations. Quantum gate computations are done for the whole state vector ψ that has 2^n coefficients.

```
for $l \leftarrow 1$ to $n$ do
    // apply Hadamard gate to $\psi$
    $\psi$ = One-qubit-Hadamard($\psi$, $l$)
    for $k \leftarrow l+1$ to $n$ do
        // apply controlled phase-shift
           gate to $\psi$
        // Target and control bits are l
           and k respectively
        $\psi$ =Two-qubit-controlled-phase-shift($\psi$, $l$, $k$)
    end
end
```

Algorithm 1: An Extract of the Algorithm to Emulate Quantum Fourier Transform of n Qubits

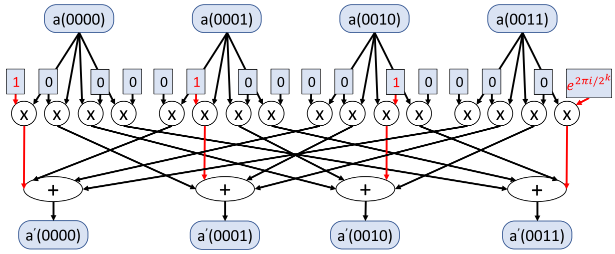


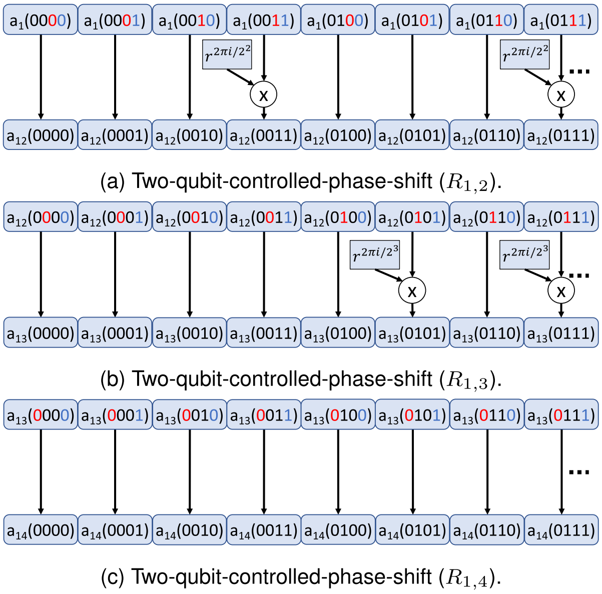
FIGURE 2. A part of the data-flow graph (DFG) corresponding to the controlled phase-shift gate $R_{1,2}$ of the QFT circuit in Figure 1.

Firstly, we discuss the controlled phase-shift gate operations in the inner-most loop of algorithm 1. When k is 1, l changes from 2 to 4 while computing the operations corresponding to controlled phase-shift gates $R_{1,2}$, $R_{1,3}$ and $R_{1,4}$ of the QFT circuit in Figure 1. Let us consider an example of the first controlled phase-shift gate $R_{1,2}$. The computations of the first four elements $a'(0000)$, $a'(0001)$, $a'(0010)$ and $a'(0011)$ of the output state vector, are shown as follows.

$$R_{1,2} \times \begin{pmatrix} a(0000) \\ a(0001) \\ a(0010) \\ a(0011) \end{pmatrix} = \begin{pmatrix} a'(0000) \\ a'(0001) \\ a'(0010) \\ a'(0011) \end{pmatrix}$$

This computation is graphically represented using a data-flow graph (DFG) in Figure 2. As we can see, most of the elements in $R_{1,2}$ are zero. Therefore, each coefficient of the output state vector only depends on a single coefficient in the input state vector. Utilizing this observation, we can optimize the data-flow by removing all unnecessary computations.

Figure 3 shows the DFGs corresponding to the controlled phase-shift gates. The DFGs are simplified based on the observation that a coefficient of the output state vector only depends on a single coefficient in the input state vector. For example, the output $a_{14}(0111)$ of $R_{1,4}$ depends on the output $a_{13}(0111)$ of $R_{1,3}$, that depends on the output $a_{12}(0111)$ of $R_{1,2}$, that is computed from the input $a_1(0111)$. Therefore, if we know the input $a_1(0111)$, we can compute the final output $a_{14}(0111)$, without depending on any other input or intermediate data. As a result, we do not have to store all the intermediate data in the external memory, and this reduces the external memory access drastically. In order to accelerate the computation, all we have to do is to access multiple data values of the input state vector a_1 and process those in parallel. The data of the state vector a_{14} are written back to the external memory to use as the inputs of the next Hadamard gate computation.



(a) Two-qubit-controlled-phase-shift ($R_{1,2}$).

(b) Two-qubit-controlled-phase-shift ($R_{1,3}$).

(c) Two-qubit-controlled-phase-shift ($R_{1,4}$).

FIGURE 3. Relationship between input and output state vector coefficients when applying controlled phase-shift gate as shown in Algorithm 1.

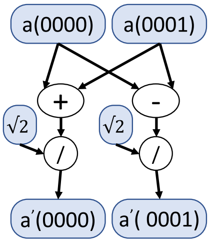


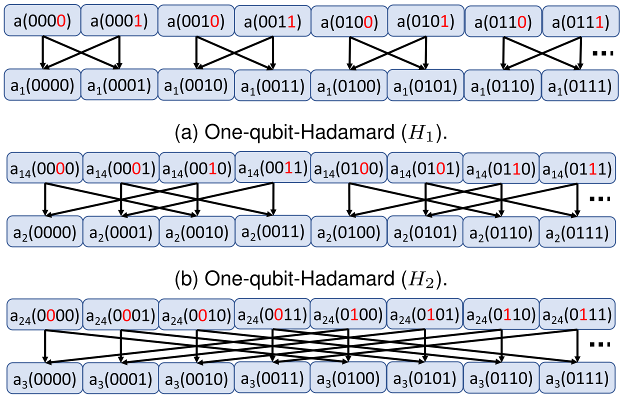
FIGURE 4. Data-flow graph (DFG) of Hadamard gate operation corresponds to the coefficients $a'(0000)$ and $a'(0001)$ of the resulting state vector.

Secondly, we discuss the operations of the outer-most loop of algorithm 1. Hadamard gates are performed in the outer-most loop. Let us consider an example of the first Hadamard gate H_1. The computations of the first two elements $a'(0000)$ and $a'(0001)$ of the output state vector are shown as follows.

$$H \times \begin{pmatrix} a(0000) \\ a(0001) \end{pmatrix} = \begin{pmatrix} a'(0000) \\ a'(0001) \end{pmatrix}$$

This computation is graphically represented using a DFG in Figure 4. As we can see, two inputs are required to calculate two outputs.

Utilizing the observation of every output is calculated using two inputs, we simplified the DFGs corresponding


a(0000) a(0001) a(0010) a(0011) a(0100) a(0101) a(0110) a(0111)
a1(0000) a1(0001) a1(0010) a1(0011) a1(0100) a1(0101) a1(0110) a1(0111)
a14(0000) a14(0001) a14(0010) a14(0011) a14(0100) a14(0101) a14(0110) a14(0111)
a2(0000) a2(0001) a2(0010) a2(0011) a2(0100) a2(0101) a2(0110) a2(0111)
a24(0000) a24(0001) a24(0010) a24(0011) a24(0100) a24(0101) a24(0110) a24(0111)
a3(0000) a3(0001) a3(0010) a3(0011) a3(0100) a3(0101) a3(0110) a3(0111)
…

(a) One-qubit-Hadamard (H_1).

(b) One-qubit-Hadamard (H_2).

(c) One-qubit-Hadamard (H_3).

FIGURE 5. Relationship between input and output state vector coefficients when applying Hadamard gate as shown in Algorithm 1. Dividing by $\sqrt{2}$ operations are not shown for simplicity and to focus only on input and output data.

to H_1, H_2 and H_3 in Figure 5. Note that dividing by $\sqrt{2}$ operations are not shown for simplicity and to focus only on input and output data. Figure 5a shows the DFG corresponding to H_1. Input is the initial state vector a and the output is a_1. The output state vector a_1 is used as the input for the controlled phase-shift gate $R_{1,2}$ as explained in Figure 3a. After the computation of $R_{1,4}$, the output state vector a_{14} is generated. Figure 5b shows the DFG corresponding to H_2. Input and output state vectors are a_{14} and a_2 respectively. Similarly, the output of this computation a_2 is used again as the input of controlled phase-shift gate $R_{2,3}$. Figure 5c shows the computation corresponding to H_3, where input and output state vectors are a_{24} and a_3 respectively. Similar process is applied for the computation of H_4.

Let us further analyze the relationship of inputs and outputs. In order to compute $a_3(0000)$, we need $a_{24}(0000)$ and $a_{24}(0100)$. Outputs $a_{24}(0000)$ and $a_{24}(0100)$ are generated using $a_2(0000)$ and $a_2(0100)$ as the inputs in controlled phase-shift gates. To obtain $a_2(0000)$ and $a_2(0100)$, we need $a_{14}(0000)$, $a_{14}(0010)$, $a_{14}(0100)$ and $a_{14}(0110)$. If we continue this process, we can see that all inputs of the initial state vector are required to generate a single output of the final state vector. Therefore, we have to store the output state vectors of every Hadamard gate in the memory. Moreover, the data access pattern of every Hadamard gate is different. We propose a novel memory allocation to fix the memory access pattern to a single one and also to allow parallel access to multiple memory modules.

C. PREVIOUS WORKS OF QFT EMULATORS

QFT emulators such as [15]–[20] that use FPGAs are already proposed. All those emulators are not compatible with large QFT circuits, while the larget emulation performed is only 16-qubit QFT in [18]. Moreover, emulators in [15]–[18] uses fixed-point computation. According to the results in those works, the error increases with the number of qubits, and may not suitable for emulations with a larger number of qubits. The proposals in [21], [22] can perform large emulations of over 20-qubits. A co-processor based on Xilinx Zynq-7000 SoC is proposed in [21] to find state changes. Although it computes the whole state vector of 4-qubit QFT emulation, it only computes 9 elements for 10-qubit and 3 elements for 32-qubit QFT emulations. Therefore, we can assume that it cannot practically compute all elements of a state vector for more than 10-qubit emulations. The work in [22] proposes a method to accelerate large quantum circuit emulations using tensor networks [23], [24]. However, the evaluation is done based on assumptions and they have not physically implemented the proposal on a real FPGA or measured the results.

There are multicore-CPU and GPU emulators for large quantum circuit emulations. Examples of such emulators are QCGPU [25], Qiskit [26], Qulacs [27], Qibo [28], QuEST [29] and Yao [30]. However, the lack of scalability to multiple processors is the major problem on these emulators. We give a comprehensive evaluation by comparing the proposed method against previous FPGA, CPU and GPU based QFT emulations in section IV.

III. FPGA EMULATOR ARCHITECTURE FOR QFT USING OPENCL

A. USING HBM-BASED MEMORY SUB-SYSTEM IN OPENCL

FPGA boards contain two types of memory sub-systems. One type of memory sub-system contains multiple channels of DDR3 or DDR4 memories. All those memories are usually mapped to a single address space so that we can see a one large memory. The total bandwidth is the additions of the bandwidths of all memories. Data can be stored freely and those data are automatically distributed among multiple memory modules to utilize the bandwidth of all memories. The other type is based on HBM (high-bandwidth memory) modules. In HBM sub-system, there is no common address space and all memories have separate address spaces. Therefore, we have to be specific on which HBM memory we are using. Figure 6 shows the HBM memory sub-system of a Stratix 10 MX FPGA board. It has 16 HBM2 memories, where each has two channels. This provides 32 pseudo memories that can be accessed independently in parallel. Data should be accessed from multiple HBM memory modules in parallel to get a large bandwidth.

We use OpenCL-based design environment to implement the FPGA emulator. OpenCL is a framework to write programs to execute across heterogeneous parallel platforms [31]. Currently it is used extensively as a high level design environment for FPGA-based system designs [13]. CPU is the host and the FPGA is the device, while kernels are the modules executed on FPGA. OpenCL channel is a data transfer method between a source and a destination kernels based on hand-shake protocol. I/O channels are an extension that is used to transfer data between two kernels belonging to two FPGAs. In OpenCL, memory access is done by accessing arrays. Memory objects are bind to a specific memory module at the beginning of the host program. Memory modules

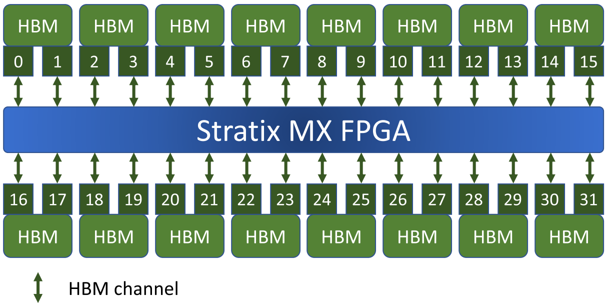
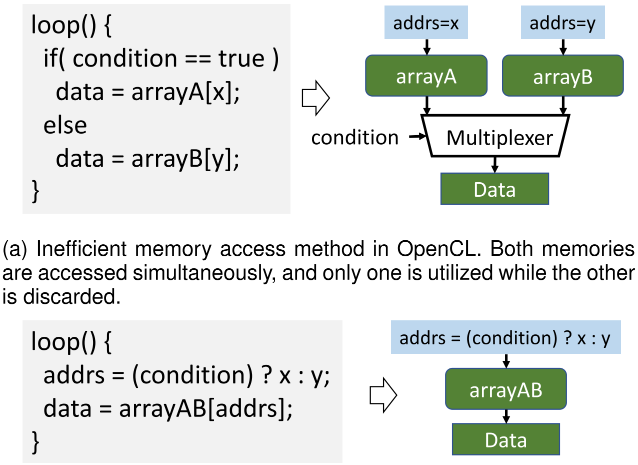


FIGURE 6. High-bandwidth memory sub-system of Stratix 10 MX FPGA.

connected to FPGA kernels are specifically defined at the compilation time. A kernel cannot access the data in HBM modules that are not connected to it. In order to access the data in other HBM memories, those data have to be transferred into an HBM module that the kernel is connected with, either explicitly by implementing the data-transfer in the kernel, or implicitly by changing the binding from the host. This results in a large data transfer time and data duplication. Another method is to connect multiple memory modules to every kernel. However, this increases the resource utilization and decreases the clock frequency significantly.

Memory access optimization is a critical factor to increase the processing speed. Therefore, when using HBM-based memory sub-systems, it is extremely important to localize the memory in such a way that every kernel always read from the same memory and write to the same memory. This will eliminate unnecessary data transfers, reduce the resource utilization for memory access, and increase the usable memory bandwidth. Another important point is to optimize conditional memory accesses. In Figure 7a, "arrayA" or "arrayB" is accessed depending on the "condition". Note that arrayA and array B can be assigned to one or two memory modules. Since memory access can take many clock cycles, accessing data after evaluating the condition can reduce the throughput. Therefore OpenCL compiler generates a circuit that has parallel access to both arrays. Both arrays are accessed simultaneously and one data is selected while the other is discarded, depending on the state of the condition. This implementation maximizes the processing speed, but reduces the memory access efficiency by wasting half of the bandwidth. The solution for this problem is to combine both arrays into a single one and allocate it to one memory module. Then select the memory addresses conditionally as shown in Figure 7b. In this case, there is only one memory address, so that only one address is accessed in each clock cycle. In summary, the following points must be considered to optimize memory access.

- Avoid conditional access to multiple memories
- Evenly distribute data on all memories
- Data that are accessed in parallel must be allocated to multiple memory modules
- Reduce/eliminate data duplication
- Avoid explicit/implicit data transfers among memories.



```
loop() {
  if( condition == true )
    data = arrayA[x];
  else
    data = arrayB[y];
}
```

(a) Inefficient memory access method in OpenCL. Both memories are accessed simultaneously, and only one is utilized while the other is discarded.

```
loop() {
  addrs = (condition) ? x : y;
  data = arrayAB[addrs];
}
```

(b) Efficient memory access method in OpenCL. Different addresses of a single memory is accessed.

FIGURE 7. Memory access optimization for HBM-based memory sub-systems.

The proposed memory allocation in this paper satisfies all of the above conditions to fully utilize all HBM modules.

B. SINGLE FPGA IMPLEMENTATION

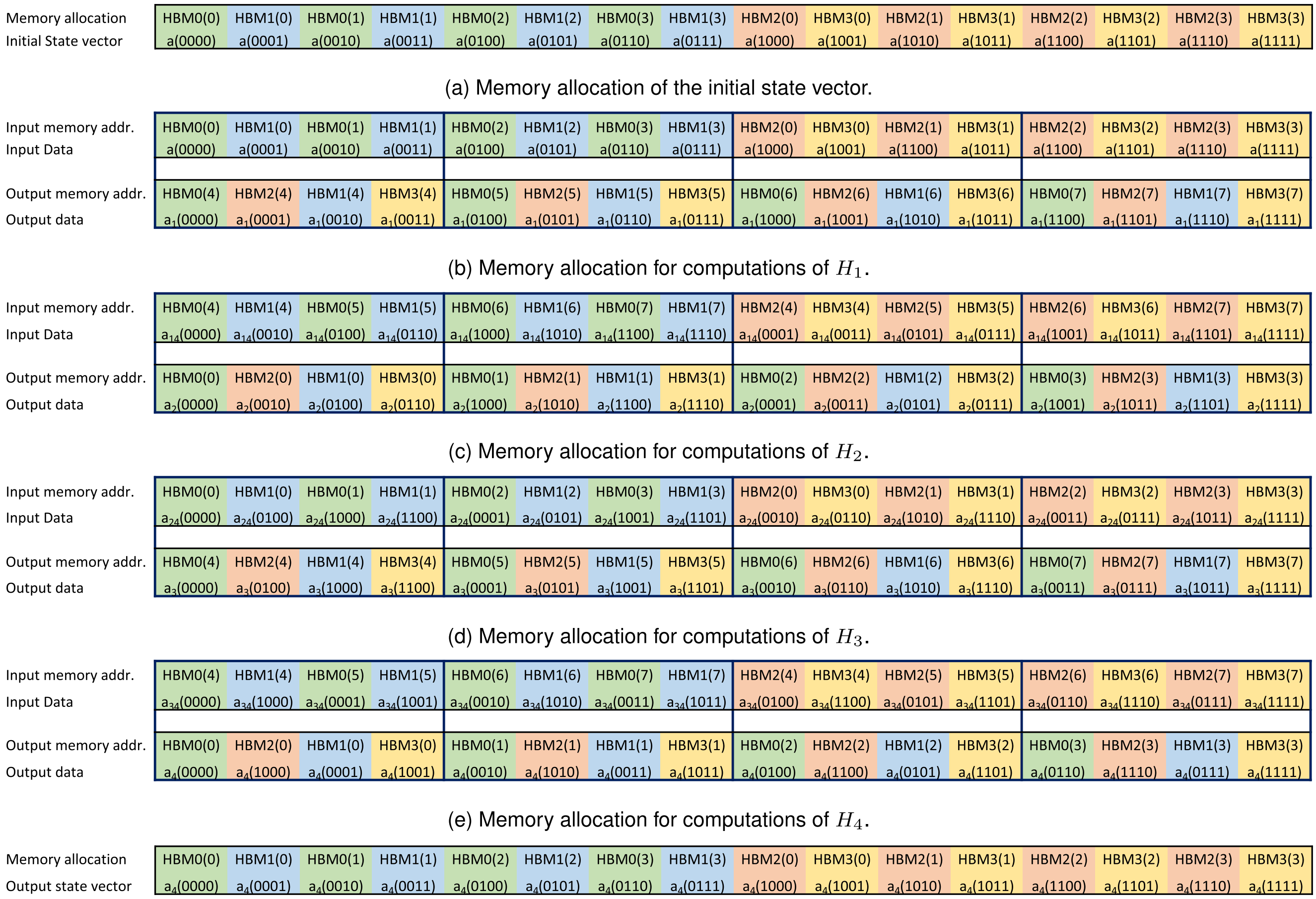
In this section, we explain the proposed memory allocation. Firstly, we explain the memory allocation for controlled phase-shift gate computations. As explained in Figure 3 in section II-B, inputs and outputs have one-to-one relationship where each output is calculated using only a single input, and each input is used to calculate a single output. We allocate the outputs to the same location of the memory of their corresponding inputs. Therefore, the input data are replaced by the output data.

Secondly, we explain the memory allocation for Hadamard gate computation. Equation (5) shows how the data of an input state vector is distributed to HBM modules. The number of HBM modules is M, the number of qubits is N, the state vector is a, HBM module number is $[m]$, and memory address is (n).

$$\mathrm{HBM}[m](n) = \begin{cases} a(n \times \dfrac{M}{2} + m) \\ \text{where, } m < \dfrac{M}{2}, 0 \leq n < 2^N/M \\ \\ a(2^{N-1} + n \times \dfrac{M}{2} + m - \dfrac{M}{2}) \\ \text{where, } m \geq \dfrac{M}{2}, 0 \leq n < 2^N/M \end{cases} \tag{5}$$

As we can see, half of the state vector data are distributed evenly to $M/2$ HBMs, while the other half is distributed to the rest of $M/2$ HBMs.

Figure 8 shows a concrete example of the memory allocation of 4-qubit QFT emulation, assuming that we have 4 HBM modules. Figure 8a shows the initial memory allocation which is done according to Eq.(5). The first half of the state



Memory allocation	HBM0(0)	HBM1(0)	HBM0(1)	HBM1(1)	HBM0(2)	HBM1(2)	HBM0(3)	HBM1(3)	HBM2(0)	HBM3(0)	HBM2(1)	HBM3(1)	HBM2(2)	HBM3(2)	HBM2(3)	HBM3(3)
Initial State vector	a(0000)	a(0001)	a(0010)	a(0011)	a(0100)	a(0101)	a(0110)	a(0111)	a(1000)	a(1001)	a(1010)	a(1011)	a(1100)	a(1101)	a(1110)	a(1111)

(a) Memory allocation of the initial state vector.

Input memory addr.	HBM0(0)	HBM1(0)	HBM0(1)	HBM1(1)	HBM0(2)	HBM1(2)	HBM0(3)	HBM1(3)	HBM2(0)	HBM3(0)	HBM2(1)	HBM3(1)	HBM2(2)	HBM3(2)	HBM2(3)	HBM3(3)
Input Data	a(0000)	a(0001)	a(0010)	a(0011)	a(0100)	a(0101)	a(0110)	a(0111)	a(1000)	a(1001)	a(1100)	a(1011)	a(1100)	a(1101)	a(1110)	a(1111)
Output memory addr.	HBM0(4)	HBM2(4)	HBM1(4)	HBM3(4)	HBM0(5)	HBM2(5)	HBM1(5)	HBM3(5)	HBM0(6)	HBM2(6)	HBM1(6)	HBM3(6)	HBM0(7)	HBM2(7)	HBM1(7)	HBM3(7)
Output data	a_1(0000)	a_1(0001)	a_1(0010)	a_1(0011)	a_1(0100)	a_1(0101)	a_1(0110)	a_1(0111)	a_1(1000)	a_1(1001)	a_1(1010)	a_1(1011)	a_1(1100)	a_1(1101)	a_1(1110)	a_1(1111)

(b) Memory allocation for computations of H_1.

Input memory addr.	HBM0(4)	HBM1(4)	HBM0(5)	HBM1(5)	HBM0(6)	HBM1(6)	HBM0(7)	HBM1(7)	HBM2(4)	HBM3(4)	HBM2(5)	HBM3(5)	HBM2(6)	HBM3(6)	HBM2(7)	HBM3(7)
Input Data	a_{14}(0000)	a_{14}(0010)	a_{14}(0100)	a_{14}(0110)	a_{14}(1000)	a_{14}(1010)	a_{14}(1100)	a_{14}(1110)	a_{14}(0001)	a_{14}(0011)	a_{14}(0101)	a_{14}(0111)	a_{14}(1001)	a_{14}(1011)	a_{14}(1101)	a_{14}(1111)
Output memory addr.	HBM0(0)	HBM2(0)	HBM1(0)	HBM3(0)	HBM0(1)	HBM2(1)	HBM1(1)	HBM3(1)	HBM0(2)	HBM2(2)	HBM1(2)	HBM3(2)	HBM0(3)	HBM2(3)	HBM1(3)	HBM3(3)
Output data	a_2(0000)	a_2(0010)	a_2(0100)	a_2(0110)	a_2(1000)	a_2(1010)	a_2(1100)	a_2(1110)	a_2(0001)	a_2(0011)	a_2(0101)	a_2(0111)	a_2(1001)	a_2(1011)	a_2(1101)	a_2(1111)

(c) Memory allocation for computations of H_2.

Input memory addr.	HBM0(0)	HBM1(0)	HBM0(1)	HBM1(1)	HBM0(2)	HBM1(2)	HBM0(3)	HBM1(3)	HBM2(0)	HBM3(0)	HBM2(1)	HBM3(1)	HBM2(2)	HBM3(2)	HBM2(3)	HBM3(3)
Input Data	a_{24}(0000)	a_{24}(0100)	a_{24}(1000)	a_{24}(1100)	a_{24}(0001)	a_{24}(0101)	a_{24}(1001)	a_{24}(1101)	a_{24}(0010)	a_{24}(0110)	a_{24}(1010)	a_{24}(1110)	a_{24}(0011)	a_{24}(0111)	a_{24}(1011)	a_{24}(1111)
Output memory addr.	HBM0(4)	HBM2(4)	HBM1(4)	HBM3(4)	HBM0(5)	HBM2(5)	HBM1(5)	HBM3(5)	HBM0(6)	HBM2(6)	HBM1(6)	HBM3(6)	HBM0(7)	HBM2(7)	HBM1(7)	HBM3(7)
Output data	a_3(0000)	a_3(0100)	a_3(1000)	a_3(1100)	a_3(0001)	a_3(0101)	a_3(1001)	a_3(1101)	a_3(0010)	a_3(0110)	a_3(1010)	a_3(1110)	a_3(0011)	a_3(0111)	a_3(1011)	a_3(1111)

(d) Memory allocation for computations of H_3.

Input memory addr.	HBM0(4)	HBM1(4)	HBM0(5)	HBM1(5)	HBM0(6)	HBM1(6)	HBM0(7)	HBM1(7)	HBM2(4)	HBM3(4)	HBM2(5)	HBM3(5)	HBM2(6)	HBM3(6)	HBM2(7)	HBM3(7)
Input Data	a_{34}(0000)	a_{34}(1000)	a_{34}(0001)	a_{34}(1001)	a_{34}(0010)	a_{34}(1010)	a_{34}(0011)	a_{34}(1011)	a_{34}(0100)	a_{34}(1100)	a_{34}(0101)	a_{34}(1101)	a_{34}(0110)	a_{34}(1110)	a_{34}(0111)	a_{34}(1111)
Output memory addr.	HBM0(0)	HBM2(0)	HBM1(0)	HBM3(0)	HBM0(1)	HBM2(1)	HBM1(1)	HBM3(1)	HBM0(2)	HBM2(2)	HBM1(2)	HBM3(2)	HBM0(3)	HBM2(3)	HBM1(3)	HBM3(3)
Output data	a_4(0000)	a_4(1000)	a_4(0001)	a_4(1001)	a_4(0010)	a_4(1010)	a_4(0011)	a_4(1011)	a_4(0100)	a_4(1100)	a_4(0101)	a_4(1101)	a_4(0110)	a_4(1110)	a_4(0111)	a_4(1111)

(e) Memory allocation for computations of H_4.

Memory allocation	HBM0(0)	HBM1(0)	HBM0(1)	HBM1(1)	HBM0(2)	HBM1(2)	HBM0(3)	HBM1(3)	HBM2(0)	HBM3(0)	HBM2(1)	HBM3(1)	HBM2(2)	HBM3(2)	HBM2(3)	HBM3(3)
Output state vector	a_4(0000)	a_4(0001)	a_4(0010)	a_4(0011)	a_4(0100)	a_4(0101)	a_4(0110)	a_4(0111)	a_4(1000)	a_4(1001)	a_4(1010)	a_4(1011)	a_4(1100)	a_4(1101)	a_4(1110)	a_4(1111)

(f) Memory allocation of the output state vector.

FIGURE 8. Proposed memory allocation of Hadamard gates. After each Hadarmard gate computation, the output data are used as the inputs of the controlled phase-shift gate computation. Since each output of controlled phase-shift gate replaces its inputs, the memory allocation remains the same. The inputs of H2, H3 and H4 are the data after the controlled phase-shift gate computation.

vector is assigned to HBM0 and HBM1, while the second half is assigned to HBM2 and HBM3. Note that each color represents a different memory module. We can see that the data are equally distributed to all memory modules, while neighboring data are allocated to different memory modules.

Figure 8b shows the memory allocation related to the first Hadamard gate H_1. In each step, minimum of four data values are accessed in parallel. For example, inputs $a(0000), a(0001), a(0010)$ and $a(0011)$ are read by accessing HBM0(0), HBM1(0), HBM0(1) and HBM1(1) simultaneously. The outputs $a_1(0000), a_1(0001), a_1(0010)$ and $a_1(0011)$ are written to HBM0(4), HBM2(4), HBM1(4) and HBM3(4) respectively. This process continues for the rest of the data in the first half of the input vector. When all the data are read from HBM0 and HBM1, we read the second half of the state vector from HBM2 and HBM3. The input and output addresses are different, where inputs are read from the addresses 0 to 3, while the outputs are written to the addresses 4 to 7, in each HBM module. These output data are then used as the inputs for the controlled phase-shift gate operations. As already explained using Figure 3, the inputs are over-written by the outputs of the controlled phase-shift gate operations. Therefore, the data in the state vector a_1 are replaced by the corresponding data in the output state vector a_{14}. For example, $a_1(0101)$ in HBM2(5) is replaced by $a_{14}(0101)$ belonging to the state vector a_{14}, that is generated after the computation corresponding to the controlled phase-shift gate $R_{1,4}$.

Figure 8c shows the memory allocation corresponding to H_2 computation. The input and output state vectors are a_{14} and a_2 respectively. We read inputs $a_{14}(0000)$, $a_{14}(0010)$, $a_{14}(0100)$ and $a_{14}(0110)$ by accessing HBM0(4), HBM1(4), HBM0(5) and HBM1(5) simultaneously. The outputs $a_2(0000), a_2(0010), a_2(0100)$ and $a_2(0110)$ are written to HBM0(0), HBM2(0), HBM1(0) and HBM3(0) respectively. In this case, the same memory modules used in H_1 computation are accessed after reversing the input and output addresses. Other than the addresses, the memory allocation is similar to that of H_1 in Figure 8b. Figure 8d shows the memory allocation corresponding to H_3 computation. We read

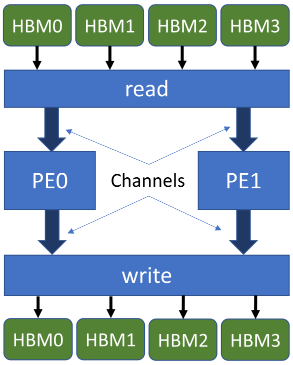


FIGURE 9. Accelerator architecture using 2 processing elements (PEs) and four memory modules.

inputs $a_{24}(0000), a_{24}(0100), a_{24}(1000)$ and $a_{24}(1100)$ by accessing HBM0(0), HBM1(0), HBM0(1) and HBM1(1) simultaneously. The outputs $a_3(0000), a_3(0100), a_3(1000)$ and $a_3(1100)$ are written to HBM0(4), HBM2(4), HBM1(4) and HBM3(4) respectively. As we can see, the memory access is similar to that of H_1 in Figure 8b, where the same memory modules and the same memory addresses are accessed for both inputs and outputs. Memory access of H_4 is shown in Figure 8e. It is similar to that of H_2 in Figure 8c. Figure 8f shows the memory allocation of the output data after rearranging for better visibility. (Note that the memory allocations of the output data in Figure 8e and Figure 8f are exactly the same). We can see that the memory allocation of the output data is similar to that of the input data in Figure 8a.

```
__kernel void read(HBM0, HBM1, HBM2, HBM3)
{
  base = (odd gate) ? 0 : size;

  for(i=0; i<size/2; i++)
  {
    addr = base + 2*i;
    dtmp0 = {HBM0[addr+0], HBM1[addr+0]};
    dtmp1 = {HBM0[addr+1], HBM1[addr+1]};

    write_channel_intel(ch_din0, dtmp0);
    write_channel_intel(ch_din1, dtmp1);
  }

  for(i=0; i<size/2; i++)
  {
    addr = base + 2*i;
    dtmp0 = {HBM2[addr+0], HBM3[addr+0]};
    dtmp1 = {HBM2[addr+1], HBM3[addr+1]};

    write_channel_intel(ch_din0, dtmp0);
    write_channel_intel(ch_din1, dtmp1);
  }
}
```

Listing 1. Extract of the read kernel. Coding is simplified and details are removed for the explanation purpose.

Overall, the memory allocations of all odd number of Hadamard gates are the same. Similarly, the memory allocations of all even number of Hadamard gates are also the same. The difference between the memory allocations of odd and even is only the memory addresses. Despite the fact that different data are required for different computations as discussed in Figure 5, the same memory modules are accessed in the same order for all computations belonging to all gates. The required data are already allocated to the appropriate location in the previous computation. Therefore, the structure between the memory and kernels is the same and we do not have to conditionally select memory modules. All the inputs and output data are equally distributed among all memory modules. The data can be read and write in parallel from and to multiple memory modules. If we want to increase the degree of parallelism further, we can do it by accessing more data in parallel.

```
__attribute__((autorun))
__kernel void PE0( )
{
  for(i=0; i<size; i++)
  {
    dat = read_channel_intel(ch_din0);

    //Hadamard computation. k is a constant
    dtmp0 = k*(dat.s0 + dat.s1);
    dtmp2 = k*(dat.s0 - dat.s1);

    //Controlled phase-shift computation
    #pragma unroll
    for(all phase shift gates) {
      dtmp0 = phase_shift(dtmp0);
      dtmp2 = phase_shift(dtmp2);
    }

    //send to write kernel
    write_channel_intel(ch_dout0, dtmp0);
    write_channel_intel(ch_dout2, dtmp2);
  }
}
```

Listing 2. Computation kernel PE0. Coding is simplified and details are removed for the explanation purpose.

Figure 9 shows the emulator architecture to use the memory allocation shown in Figure 8. It consists of a read kernel, a write kernel and two computation kernels PE0 and PE1. All kernels execute concurrently, and the data transfer among kernels are done through OpenCL channels. Listing 1 shows an extract of the OpenCL code used for the "read kernel". It contains two loops executed one after the other in serial manner. In the first loop, two neighboring data values each from $HBM0$ and $HBM1$ are accessed in parallel. In the second loop, two neighboring data values each from $HBM2$ and $HBM3$ are accessed in parallel. The *size* is the number of data values per memory. When the number of qubits are N, and the number of memory modules are M, $size = 2^N/M$. The memory address is dependent on whether the gate is an odd number or even number. This gate number parameter is transferred from the host. Note that all data are in complex number format and we use a structure of two single-precision floating-point numbers to represents the real and imaginary parts.

Listings 2 and 3 show an extract of computation kernels PE0 and PE1 corresponding to two PEs (processing elements). These two kernels receive the data from the "read kernel" through OpenCL channels. After each computation, the results are written back to the "write kernel" through channels. Since computation kernels do not access external memory directly, we can design those as "autorun" kernels. As a result, those kernels are automatically executed without any host intervention, and this reduces the control overhead of the CPU. The constants are stored internally in the FPGA on a ROM (read only memory) and accessed by kernels. Since all computations of controlled phase-shift gates are mutually independent, we compute all phase-shift gates in parallel, by using "unroll" directive.

```
__attribute__((autorun))
__kernel void PE1( )
{
  for(i=0; i<size; i++)
  {
    dat = read_channel_intel(ch_din1);

    //Hadamard computation
    dtmp1 = k*(dat.s0 + dat.s1);
    dtmp3 = k*(dat.s0 - dat.s1);

    //Controlled phase-shift computation
    #pragma unroll
    for(all phase shift gates) {
      dtmp1 = phase_shift(dtmp1);
      dtmp3 = phase_shift(dtmp3);
    }

    //send to write kernel
    write_channel_intel(ch_dout1, dtmp1);
    write_channel_intel(ch_dout3, dtmp3);
  }
}
```

Listing 3. Computation kernel PE1. Coding is simplified and details are removed for the explanation purpose.

```
__kernel void write(HBM0, HBM1, HBM2, HBM3)
{
  base = (odd gate) ? size : 0;
  for(unsigned i=0; i<size; i++)
  {
    unsigned addr = base + i;

    HBM0[addr] = read_channel_intel(ch_dout0);
    HBM1[addr] = read_channel_intel(ch_dout1);
    HBM2[addr] = read_channel_intel(ch_dout2);
    HBM3[addr] = read_channel_intel(ch_dout3);
  }
}
```

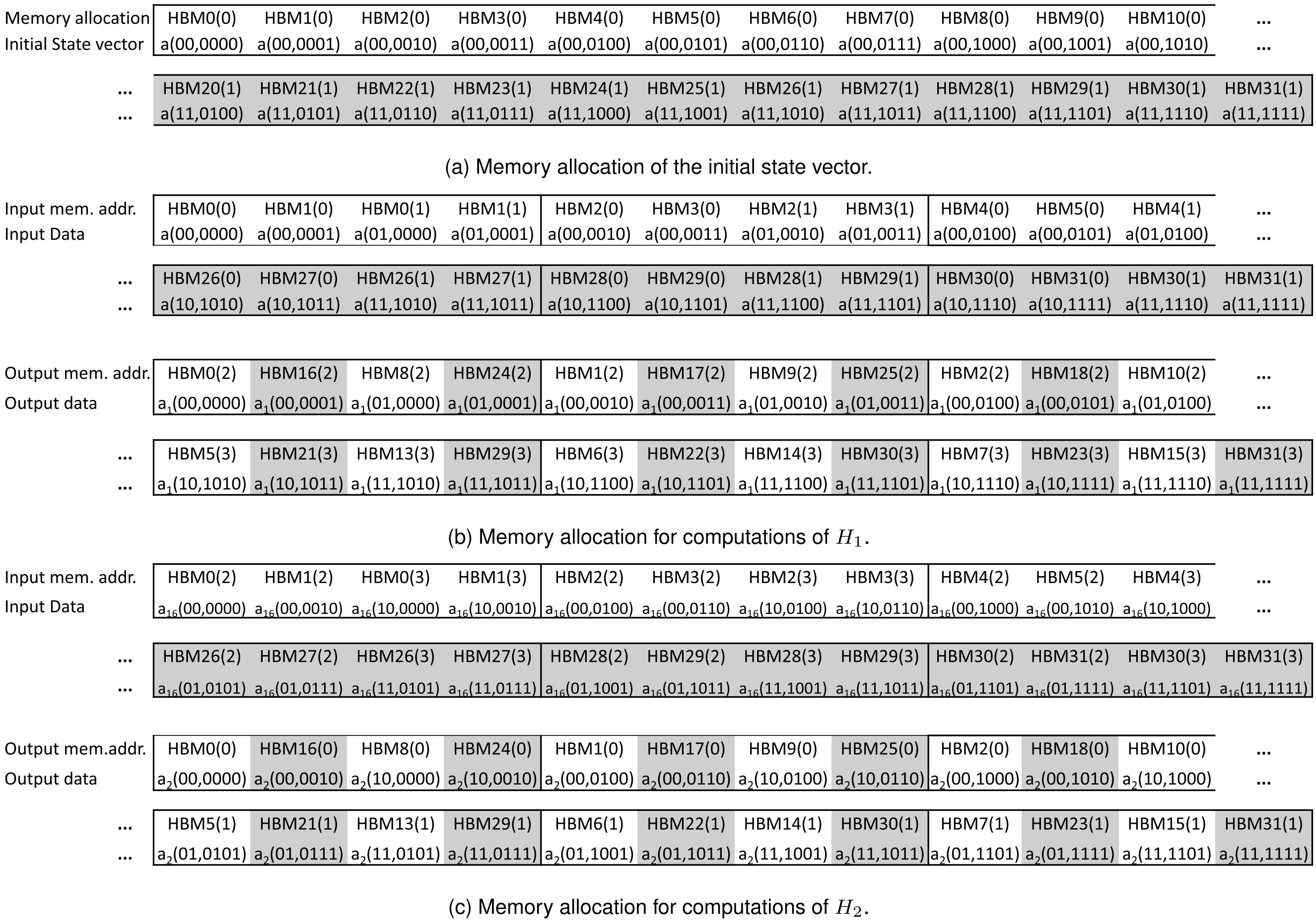
Listing 4. Write kernel. Coding is simplified and details are removed for the explanation purpose.

Listing 4 shows an extract of the "write kernel". It receives all the results from both PEs through channels. Those data are then written to 4 HBM memories in parallel. All kernels have loops that iterates for all data in the memory. Both read and write kernels are executed by the host for each Hadamard gate, while the autorun kernels are executed automatically. We have a continuous data stream from read, computation and write. Read, computation and write kernels are executed concurrently in pipelined manner. Channels are implemented in blocking mode, so that the next execution is blocked if the data are not received from the source kernel, or data are not accepted by the destination kernel. Each kernel is in standby state until it receives data from the previous kernel through channels. In order to tolerate small differences of reading and writing speeds, we use buffered channels. Therefore, a channel can write to a buffered channel until the buffer is completely filled.

Generalized memory allocation for M HBM modules and N qubits is shown in Eqs.(6), (7) and (8). Memory access from the input HBMs connected to PE[$2m$] and PE[$2m+1$] are given by Eq.(6). The HBM module number and PE number are given by [m] and memory address is given by (n). In the first half, data are accessed from HBM[$2m$] and HBM[$2m+1$]. In the second half, data are accessed from HBM[$\frac{M}{2}+2m$] and HBM[$\frac{M}{2}+2m+1$]. Total of four data values, two from each HBM, are required by two PEs in each step. All $M/2$ PEs operate in parallel, while reading two data values per HBM from $M/2$ HBMs per step. Since there are 2^N data in the state vector, and $2 \times M/2$ data are read in each step, a total of $2^N/M$ steps are required for the whole computation. The output HBMs connected to PE[$2m$] and PE[$2m+1$] are shown by Eqs. (7) and (8) respectively. Input and output data use different address spaces, where inputs are read from the addresses $0 \sim 2^N/M$ and outputs are written to the addresses $2^N/M \sim 2^{N+1}/M$ in each HBM. These address spaces are interchanged for each Hadamard gate computation.

$$\begin{array}{l}\text{Inputs of}\\ \text{PE}[2m],\\ \text{PE}[2m+1]\end{array}\begin{cases}\text{HBM}[2m](2n),\\ \text{HBM}[2m+1](2n),\\ \text{HBM}[2m](2n+1),\\ \text{HBM}[2m+1](2n+1)\\ \quad\text{where, } 0 \le n < \dfrac{2^{N-1}}{M}, 0 \le m < \dfrac{M}{4}\\ \\ \text{HBM}[\dfrac{M}{2}+2m](2n-\dfrac{2^N}{M}),\\ \text{HBM}[\dfrac{M}{2}+2m+1](2n-\dfrac{2^N}{M}),\\ \text{HBM}[\dfrac{M}{2}+2m](2n+1-\dfrac{2^N}{M}),\\ \text{HBM}[\dfrac{M}{2}+2m+1](2n+1-\dfrac{2^N}{M})\\ \quad\text{where, } \dfrac{2^{N-1}}{M} \le n < \dfrac{2^N}{M}, 0 \le m < \dfrac{M}{4}\end{cases} \tag{6}$$

$$\text{Outputs of PE}[2m]\begin{cases}\text{HBM}[m](\dfrac{2^N}{M}+n),\\ \text{HBM}[\dfrac{M}{2}+m](\dfrac{2^N}{M}+n)\\ \quad\text{where, } 0 \le n < \dfrac{2^N}{M},\\ 0 \le m < \dfrac{M}{4}\end{cases} \tag{7}$$



Memory allocation	HBM0(0)	HBM1(0)	HBM2(0)	HBM3(0)	HBM4(0)	HBM5(0)	HBM6(0)	HBM7(0)	HBM8(0)	HBM9(0)	HBM10(0)	...
Initial State vector	a(00,0000)	a(00,0001)	a(00,0010)	a(00,0011)	a(00,0100)	a(00,0101)	a(00,0110)	a(00,0111)	a(00,1000)	a(00,1001)	a(00,1010)	...

...	HBM20(1)	HBM21(1)	HBM22(1)	HBM23(1)	HBM24(1)	HBM25(1)	HBM26(1)	HBM27(1)	HBM28(1)	HBM29(1)	HBM30(1)	HBM31(1)
...	a(11,0100)	a(11,0101)	a(11,0110)	a(11,0111)	a(11,1000)	a(11,1001)	a(11,1010)	a(11,1011)	a(11,1100)	a(11,1101)	a(11,1110)	a(11,1111)

(a) Memory allocation of the initial state vector.

Input mem. addr.	HBM0(0)	HBM1(0)	HBM0(1)	HBM1(1)	HBM2(0)	HBM3(0)	HBM2(1)	HBM3(1)	HBM4(0)	HBM5(0)	HBM4(1)	...
Input Data	a(00,0000)	a(00,0001)	a(01,0000)	a(01,0001)	a(00,0010)	a(00,0011)	a(01,0010)	a(01,0011)	a(00,0100)	a(00,0101)	a(01,0100)	...

...	HBM26(0)	HBM27(0)	HBM26(1)	HBM27(1)	HBM28(0)	HBM29(0)	HBM28(1)	HBM29(1)	HBM30(0)	HBM31(0)	HBM30(1)	HBM31(1)
...	a(10,1010)	a(10,1011)	a(11,1010)	a(11,1011)	a(10,1100)	a(10,1101)	a(11,1100)	a(11,1101)	a(10,1110)	a(10,1111)	a(11,1110)	a(11,1111)

Output mem. addr.	HBM0(2)	HBM16(2)	HBM8(2)	HBM24(2)	HBM1(2)	HBM17(2)	HBM9(2)	HBM25(2)	HBM2(2)	HBM18(2)	HBM10(2)	...
Output data	a_1(00,0000)	a_1(00,0001)	a_1(01,0000)	a_1(01,0001)	a_1(00,0010)	a_1(00,0011)	a_1(01,0010)	a_1(01,0011)	a_1(00,0100)	a_1(00,0101)	a_1(01,0100)	...

...	HBM5(3)	HBM21(3)	HBM13(3)	HBM29(3)	HBM6(3)	HBM22(3)	HBM14(3)	HBM30(3)	HBM7(3)	HBM23(3)	HBM15(3)	HBM31(3)
...	a_1(10,1010)	a_1(10,1011)	a_1(11,1010)	a_1(11,1011)	a_1(10,1100)	a_1(10,1101)	a_1(11,1100)	a_1(11,1101)	a_1(10,1110)	a_1(10,1111)	a_1(11,1110)	a_1(11,1111)

(b) Memory allocation for computations of H_1.

Input mem. addr.	HBM0(2)	HBM1(2)	HBM0(3)	HBM1(3)	HBM2(2)	HBM3(2)	HBM2(3)	HBM3(3)	HBM4(2)	HBM5(2)	HBM4(3)	...
Input Data	a_{16}(00,0000)	a_{16}(00,0010)	a_{16}(10,0000)	a_{16}(10,0010)	a_{16}(00,0100)	a_{16}(00,0110)	a_{16}(10,0100)	a_{16}(10,0110)	a_{16}(00,1000)	a_{16}(00,1010)	a_{16}(10,1000)	...

...	HBM26(2)	HBM27(2)	HBM26(3)	HBM27(3)	HBM28(2)	HBM29(2)	HBM28(3)	HBM29(3)	HBM30(2)	HBM31(2)	HBM30(3)	HBM31(3)
...	a_{16}(01,0101)	a_{16}(01,0111)	a_{16}(11,0101)	a_{16}(11,0111)	a_{16}(01,1001)	a_{16}(01,1011)	a_{16}(11,1001)	a_{16}(11,1011)	a_{16}(01,1101)	a_{16}(01,1111)	a_{16}(11,1101)	a_{16}(11,1111)

Output mem.addr.	HBM0(0)	HBM16(0)	HBM8(0)	HBM24(0)	HBM1(0)	HBM17(0)	HBM9(0)	HBM25(0)	HBM2(0)	HBM18(0)	HBM10(0)	...
Output data	a_2(00,0000)	a_2(00,0010)	a_2(10,0000)	a_2(10,0010)	a_2(00,0100)	a_2(00,0110)	a_2(10,0100)	a_2(10,0110)	a_2(00,1000)	a_2(00,1010)	a_2(10,1000)	...

...	HBM5(1)	HBM21(1)	HBM13(1)	HBM29(1)	HBM6(1)	HBM22(1)	HBM14(1)	HBM30(1)	HBM7(1)	HBM23(1)	HBM15(1)	HBM31(1)
...	a_2(01,0101)	a_2(01,0111)	a_2(11,0101)	a_2(11,0111)	a_2(01,1001)	a_2(01,1011)	a_2(11,1001)	a_2(11,1011)	a_2(01,1101)	a_2(01,1111)	a_2(11,1101)	a_2(11,1111)

(c) Memory allocation for computations of H_2.

FIGURE 10. An extract of the proposed memory allocation for 6-qubit QFT emulation using 32 HBMs.

$$\text{Outputs of PE}[2m+1]\begin{cases}\text{HBM}[\frac{M}{4}+m](\frac{2^N}{M}+n),\\ \text{HBM}[\frac{3M}{4}+m](\frac{2^N}{M}+n)\\ \text{where, } 0\leq n<\frac{2^N}{M},\\ 0\leq m<\frac{M}{4}\end{cases} \qquad (8)$$

An example of the memory allocation for 6-qubit QFT emulation using 32 HBMs is shown in Figure 10. The initial memory allocation is done according to Eq.(5) and shown in Figure 10a. We can see that the data are evenly distributed to all 32 HBMs. The memory allocations for the first and the second Hadamard gates are shown in Figures 10b and 10c respectively. For each Hadamard gate computation, the same memory modules are accessed, while interchanging the address spaces. Exploiting this behavior, we implement the architecture using 16 PEs as shown in Figure 11. As we can see, the data path among PEs and memories is fixed for all computations. Although each PE writes to only two memory modules, one write kernel is shared among two PEs to reduce the number of kernels and the kernel control overhead by the CPU. We can scale the number of PEs by accessing more HBM modules (or different data in the same modules) in parallel. When more memory modules are used, data are evenly distributed among all memory modules. Therefore, we can simulate larger quantum circuits with more qubits. Moreover, the total required bandwidth is the sum of all bandwidths of all HBM modules.

C. MULTI-FPGA IMPLEMENTATION

This section shows how to scale both computation and data storage across multiple FPGAs. Figure 12 shows the memory allocation using two FPGAs. We use the same example used for single FPGA in Figure 8, and assume that each FPGA contains 4 HBM modules. Two FPGAs are identified by FPGA0 and FPGA1. Figure 12a shows the initial memory allocation. The first-half of the state vector is stored in FPGA 0, and the second-half is stored in FPGA 1. Half of the first-half (or the first-quarter) is stored in HBM0 and HBM1 of FPGA0, while the rest of the first-half (or the second quarter) is stored in HBM2 and HBM3 of FPGA0. The second-half of the state vector is stored similarly in FPGA1. For each half, memory allocation is similar to that of a single FPGA as explained in section III-B.

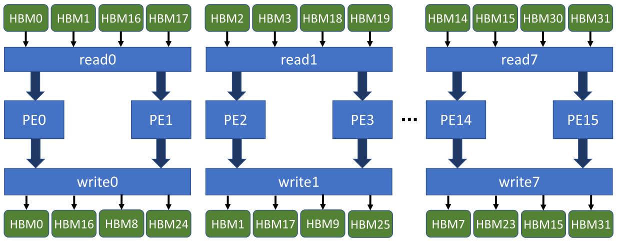


FIGURE 11. **Accelerator architecture extended to 16 PEs.**

Figure 12b shows the memory allocation corresponding to the computation of H_1. Half of the output data generated by each FPGA are retained, while the other half is sent to the other FPGA. Therefore, both FPGAs exchange half of their results after each Hadamard gate computation. The data received by the each FPGA are then allocated to the appropriate memory module as shown in Figure 12b. Figure 12c shows the memory allocation corresponding to the computation of H_2. The memory allocation is similar to the previous one, while the input and output addresses are reversed. Note that, compared to the single FPGA implementation in 8, two FPGA implementation use only half of the capacity of each HBM module. This is because, data are evenly distributed among the HBM modules of both FPGAs, without any data duplication.

Figure 13 shows the emulator architecture using two FPGAs. In each FPGA, half of the output produced by each PE is sent to the other FPGA. This data transfer is done using fiber optic cables connected to the QSFP28 ports in each FPGA. We use I/O channels to implement inter-FPGA data transfers. I/O channels connect source and destination kernels belonging to two different FPGAs. Each channel contains two separate paths for both read and write. The FPGA that is used for the evaluation contains 4 I/O channel per FPGA, while each channel is capable of 100 Gbps transfer speed in either direction. When there are more PEs than the number of I/O channels, outputs of multiple PEs are combined and then sent through a single channel. We have to consider the maximum I/O channel bandwidth when we increase the degree of parallelism. If the required bandwidth exceed the I/O channel bandwidth, the computation is automatically paused until the data transfer is completed. This is automatically done due to the blocking behavior of I/O channels.

As explained in this section, two-FPGA implementation is done by computing half of the state vector in each FPGA. We can repeat this process again and again for each portion of the allocated data per FPGA. For example, the portion of data allocated to FPGA0 can be divided again to two FPGAs. Similarly, data allocated to FPGA1 can also be divided again to two FPGAs. We can use four FPGAs, where each FPGA computes a quarter of the state vector. As a result, the degree of parallelism and the processing speed are doubled. Since total memory capacity is also doubled, we can increase the emulation size by one qubit. If we apply the same process repeatedly, we can use multiple FPGAs to do larger emulations or to increase the processing speed. Therefore, the proposed emulator can be scaled across multiple FPGAs to increase the speed-up and the emulation size.

IV. EVALUATION

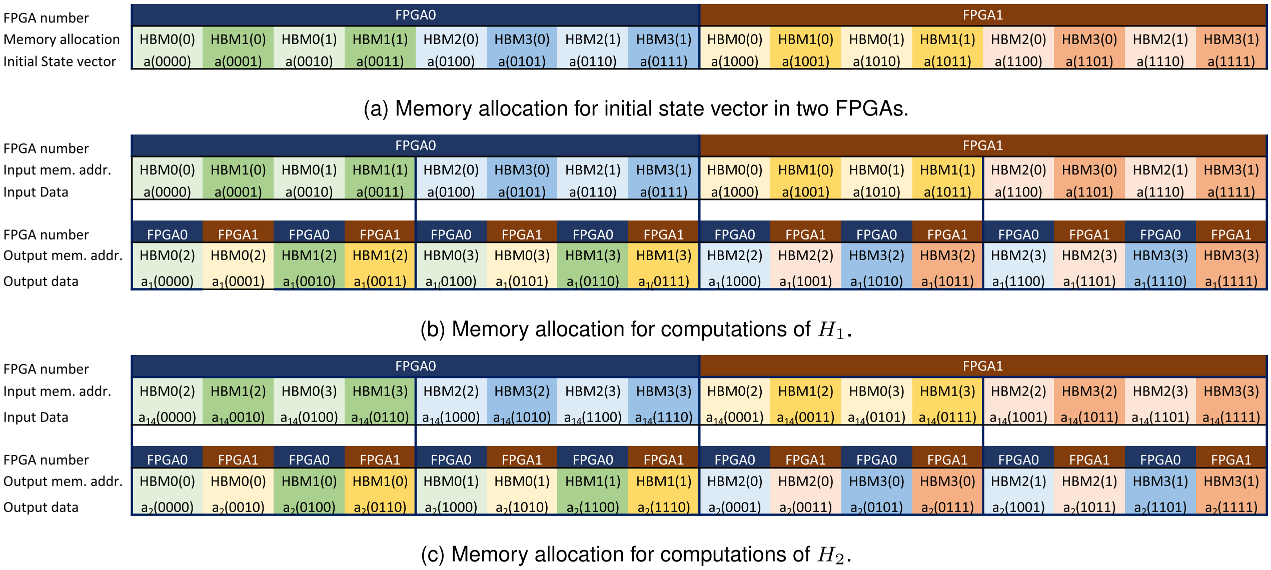
For the evaluation, we used a workstation with 16-core Intel Xeon Gold 6226R processor, 192GB DDR4 memory and two BittWare 520N-MX FPGA boards [32]. An FPGA board contains Intel Stratix 10 MX2100 FPGA with 16GB HBM2 memory that have a theoritical bandwidth of 512 GB/s. Each FPGA has four QSFP28 ports. QSFP28 ports of one FPGA are connected directly to the QSFP28 ports of the other FPGA through fiber-optic cables as shown in Figure 14. This provides a bi-directional data-paths between two FPGAs, where all four paths collectively provide 400 Gbps data transfer speed per direction. We use CentOS 7.9 operating system, Intel C compiler version 2019 and Quartus pro version 19.4.0.

A. EVALUATION OF PROCESSING TIME AND RESOURCE UTILIZATION

Figure 15 shows the computation time against the number of qubits. Computation time is measured from the host CPU by running emulations for 100 iterations and then taking the average. The kernel execution time, inter-FPGA data transfer time, and control overhead by the host are included in the computation time. The control overhead includes kernel-execution control time and synchronization time. One-FPGA implementation is faster up to 25 qubits, and two-FPGA implementation is faster for over 25 qubits. When the number of qubits is small, the control time is relatively larger compared to the computation time. Moreover, the control time is larger for two-FPGA implementation, since CPU has to control more kernels compared to that of one-FPGA implementation. As a result, two-FPGA implementation becomes slower for smaller number of qubits. When the number of qubits is large, the control time is negligible compared to the computation time. Therefore the processing speed nearly doubles when using two FPGAs. For example, two-FPGA 29-qubit implementation is two times faster compared to that of one-FPGA implementation.

Figure 16 shows the total processing time including data transfers between host CPU and FPGAs. Two-FPGA implementation is faster for larger number of qubits. Since, we evenly distribute data on to two FPGAs, the amount of data stored per FPGAs is reduced by half. Moreover, data transfers to two FPGAs are done in parallel using two PCIe buses. Therefore, both computation time and data transfer time are reduced by half. As a result, two-FPGA 29-qubit implementation is 1.9 times faster compared to that of one-FPGA implementation.

Table 1 shows the resource usage of one-FPGA implementations. When the number of qubits increases, there is a small increase in logic, registers and DSP blocks, while RAM blocks are nearly a constant. The clock frequencies are nearly the same since there is no significant increase



FPGA number	FPGA0								FPGA1							
Memory allocation	HBM0(0)	HBM1(0)	HBM0(1)	HBM1(1)	HBM2(0)	HBM3(0)	HBM2(1)	HBM3(1)	HBM0(0)	HBM1(0)	HBM0(1)	HBM1(1)	HBM2(0)	HBM3(0)	HBM2(1)	HBM3(1)
Initial State vector	a(0000)	a(0001)	a(0010)	a(0011)	a(0100)	a(0101)	a(0110)	a(0111)	a(1000)	a(1001)	a(1010)	a(1011)	a(1100)	a(1101)	a(1110)	a(1111)

(a) Memory allocation for initial state vector in two FPGAs.

FPGA number	FPGA0								FPGA1							
Input mem. addr.	HBM0(0)	HBM1(0)	HBM0(1)	HBM1(1)	HBM2(0)	HBM3(0)	HBM2(1)	HBM3(1)	HBM0(0)	HBM1(0)	HBM0(1)	HBM1(1)	HBM2(0)	HBM3(0)	HBM2(1)	HBM3(1)
Input Data	a(0000)	a(0001)	a(0010)	a(0011)	a(0100)	a(0101)	a(0110)	a(0111)	a(1000)	a(1001)	a(1010)	a(1011)	a(1100)	a(1101)	a(1110)	a(1111)
FPGA number	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1
Output mem. addr.	HBM0(2)	HBM0(2)	HBM1(2)	HBM1(2)	HBM0(3)	HBM0(3)	HBM1(3)	HBM1(3)	HBM2(2)	HBM2(2)	HBM3(2)	HBM3(2)	HBM2(3)	HBM2(3)	HBM3(3)	HBM3(3)
Output data	a_1(0000)	a_1(0001)	a_1(0010)	a_1(0011)	a_1(0100)	a_1(0101)	a_1(0110)	a_1(0111)	a_1(1000)	a_1(1001)	a_1(1010)	a_1(1011)	a_1(1100)	a_1(1101)	a_1(1110)	a_1(1111)

(b) Memory allocation for computations of H_1.

FPGA number	FPGA0								FPGA1							
Input mem. addr.	HBM0(2)	HBM1(2)	HBM0(3)	HBM1(3)	HBM2(2)	HBM3(2)	HBM2(3)	HBM3(3)	HBM0(2)	HBM1(2)	HBM0(3)	HBM1(3)	HBM2(2)	HBM3(2)	HBM2(3)	HBM3(3)
Input Data	a_{14}(0000)	a_{14}0010)	a_{14}(0100)	a_{14}(0110)	a_{14}(1000)	a_{14}(1010)	a_{14}(1100)	a_{14}(1110)	a_{14}(0001)	a_{14}(0011)	a_{14}(0101)	a_{14}(0111)	a_{14}(1001)	a_{14}(1011)	a_{14}(1101)	a_{14}(1111)
FPGA number	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1	FPGA0	FPGA1
Output mem. addr.	HBM0(0)	HBM0(0)	HBM1(0)	HBM1(0)	HBM0(1)	HBM0(1)	HBM1(1)	HBM1(1)	HBM2(0)	HBM2(0)	HBM3(0)	HBM3(0)	HBM2(1)	HBM2(1)	HBM3(1)	HBM3(1)
Output data	a_2(0000)	a_2(0010)	a_2(0100)	a_2(0110)	a_2(1000)	a_2(1010)	a_2(1100)	a_2(1110)	a_2(0001)	a_2(0011)	a_2(0101)	a_2(0111)	a_2(1001)	a_2(1011)	a_2(1101)	a_2(1111)

(c) Memory allocation for computations of H_2.

FIGURE 12. Proposed memory allocation for two FPGAs for Hadamard gates H_1 and H_2 computations.

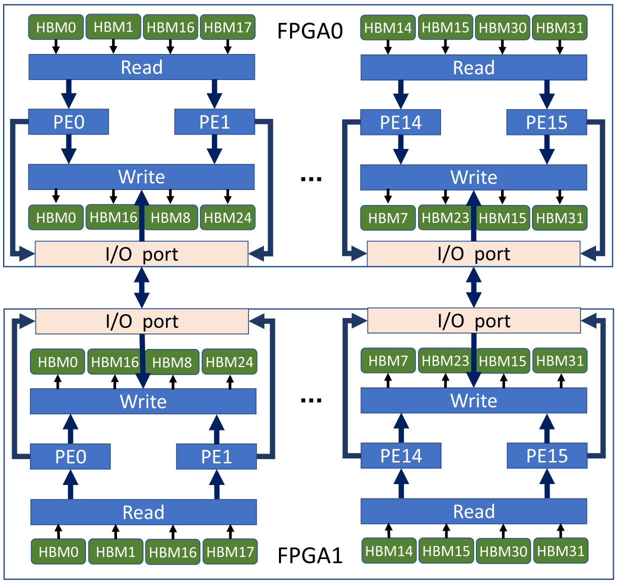


FIGURE 13. Accelerator architecture using two FPGAs.

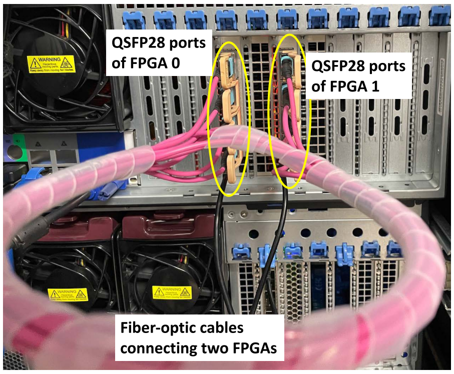


FIGURE 14. Multi-FPGA evaluation environment with two Intel Stratix 10 MX FPGA boards connected directly using fiber-optic cables.

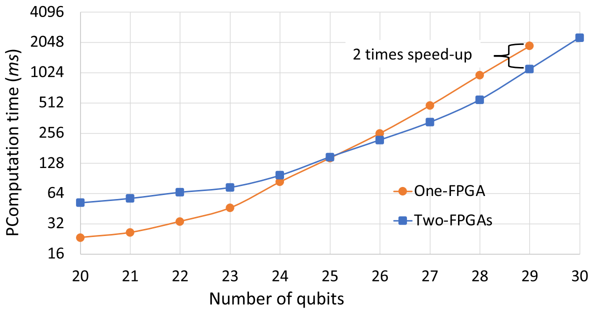


FIGURE 15. Computation time of one and two FPGA implementations for different number of qubits.

in on-chip resource utilization. However, the DRAM usage doubles with every additional qubit. DRAM is used to store the input and output state vectors, and each vector has 2^n complex floating-point elements for n qubits. Due to the limited DRAM capacity, we can implement emulations up to 29 qubits in one FPGA.

Table 2 shows the resource utilization of two-FPGA implementations. The resource utilization except for DRAMs is similar to those of one-FPGA implementations. Since we distribute the state vector data on to two FPGAs evenly, the DRAM usage per FPGA is reduced by 50% compared to one-FPGA implementation. The clock frequencies are nearly the same across two FPGAs for different number of qubits.

As shown in Figures 15 and 16, the processing time can be scaled using more FPGAs. In addition, we can increase the emulation size by adding more FPGAs. This shows the scalability of the proposed emulator. Note that, although the total logic usage is nearly 50% per FPGA, we cannot increase the degree of parallelism further. For example, if we doubles

TABLE 1. Resource utilization of one FPGA implementations.

Number of qubits	Logic	Registers	DSP blocks	RAM blocks	SRAM (MB)	DRAM (MB)	Frequency (MHz)
20	306,782 (44%)	672,713	1,008 (25%)	1,513 (22%)	2.03 (12%)	16	299
21	309,186 (44%)	680,786	1,040 (26%)	1,513 (22%)	2.04 (12%)	32	295
22	316,823 (45%)	684,676	1,072 (27%)	1,513 (22%)	2.05 (12%)	64	294
23	322,266 (46%)	693,077	1,104 (28%)	1,513 (22%)	2.05 (12%)	128	290
24	325,364 (46%)	703,402	1,136 (29%)	1,513 (22%)	2.05 (12%)	256	286
25	327,185 (47%)	710,937	1,168 (29%)	1,513 (22%)	2.05 (12%)	512	294
26	328,310 (47%)	717,952	1,200 (30%)	1,513 (22%)	2.05 (12%)	1,024	295
27	325,005 (46%)	722,814	1,232 (31%)	1,513 (22%)	2.05 (12%)	2,048	292
28	331,912 (47%)	733,189	1,264 (32%)	1,513 (22%)	2.05 (12%)	4,096	300
29	331,238 (47%)	742,779	1,296 (33%)	1,513 (22%)	2.05 (12%)	8,192	300

TABLE 2. Resource utilization of two FPGA implementations.

Number of qubits	Logic	Registers	DSP blocks	RAM blocks	SRAM (MB)	DRAM (MB)	Frequency (MHz)
20	322,697 (46%)	685,872	1,008 (25%)	1,565 (23%)	2.10 (12%)	8	293
	318,223 (45%)	689,638	1,008 (25%)	1,565 (23%)	2.10 (12%)	8	297
21	315,480 (45%)	695,755	1,040 (26%)	1,565 (23%)	2.10 (12%)	16	306
	323,168 (46%)	692,962	1,040 (26%)	1,565 (23%)	2.10 (12%)	16	307
22	319,262 (45%)	700,475	1,072 (27%)	1,565 (23%)	2.07 (12%)	32	286
	324,806 (46%)	705,734	1,072 (27%)	1,565 (23%)	2.07 (12%)	32	294
23	325,741 (46%)	710,450	1,104 (28%)	1,565 (23%)	2.07 (12%)	64	300
	326,516 (46%)	712,470	1,104 (28%)	1,565 (23%)	2.07 (12%)	64	308
24	330,546 (47%)	723,767	1,168 (30%)	1,565 (23%)	2.07 (12%)	128	320
	334,973 (48%)	720,280	1,168 (30%)	1,565 (23%)	2.07 (12%)	128	291
25	333,577 (47%)	720,264	1,168 (30%)	1,565 (23%)	2.07 (12%)	256	320
	336,722 (48%)	728,715	1,168 (29%)	1,565 (23%)	2.07 (12%)	256	320
26	336,538 (48%)	730,720	1,200 (30%)	1,565 (23%)	2.07 (12%)	512	304
	339,058 (48%)	735,390	1,200 (30%)	1,565 (23%)	2.07 (12%)	512	300
27	340,610 (48%)	742,478	1,232 (31%)	1,565 (23%)	2.07 (12%)	1,024	313
	332,636 (47%)	743,433	1,232 (32%)	1,565 (23%)	2.07 (12%)	1,024	317
28	344,482 (49%)	753,155	1,264 (32%)	1,565 (23%)	2.07 (12%)	2,048	313
	343,468 (49%)	758,784	1,264 (32%)	1,565 (23%)	2.07 (12%)	2,048	326
29	345,042 (49%)	755,915	1,296 (33%)	1,565 (23%)	2.07 (12%)	4,096	300
	345,922 (49%)	752,401	1,296 (33%)	1,565 (23%)	2.07 (12%)	4,096	300
30	349,886 (50%)	763,939	1,328 (34%)	1,565 (23%)	2.07 (12%)	8,192	276
	346,457 (49%)	763,193	1,328 (34%)	1,565 (23%)	2.07 (12%)	8,192	287

the degree of parallelism, the required logic usage is over 90% and the compilation fails. However, if we use a slightly larger FPGA, we can expect doubling the processing speed.

In quantum mechanics, ''fidelity'' [33], the absolute square of the inner product between two state vectors, is a measure of ''closeness'' of two quantum states. When two state vectors are getting closer, fidelity gets closer to 1. If the two state vectors are the same, fidelity equals to 1. We calculate fidelity using the output state vector of the proposed method and the output state vector obtained by CPU emulation. We use double-precision floating-point computation on CPU, and assume that the resulting state vector is the most accurate one we can obtain. According to the results, fidelity of all QFT emulations are larger than 0.9999986, which is extremely close to 1. Therefore, we can say that the proposed emulator provides accurate emulation results. Such high accuracy is obtained by using hardened floating-point units in FPGA that are fully compatible with IEEE 754 standard [34].

B. COMPARISON AGAINST PREVIOUS WORKS

The work in [35] is one of the most recent and comprehensive evaluation of large QFT emulations on modern multicore processors. It exploits multiple techniques such as cache access optimization, SIMD vectorization and OpenMP based

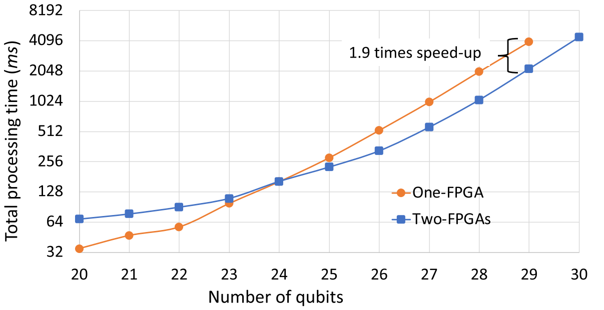


FIGURE 16. Total processing time of one and two FPGA implementations for different number of qubits. Total processing time includes both computation and data transfer times.

multicore processing on CPUs to maximize performance. It used a system with a 2.3 GHz 24-core Haswell architecture CPU and 128 GB memory for the evaluation. In [35], performance is compared against another state-of-art quantum computing emulator "QuEST" (Quantum exact simulation toolkit) [29] that also utilizes parallel computation. Both implementations are optimized using GNU and Intel compilers with OpenMP multi-threading, and the comparisons are done for up to 30 qubits. We compare the three largest QFT emulations in Table 3. Note that the fastest implementation of each work is selected for the comparison. Due to the unavailability of numerical values of processing times, we read the values directly from the graphs in [35] as accurately as possible. Compared to the fastest CPU emulation of 28 $\sim$ 30 qubits, we achieved 12.3 $\sim$ 13.2 and 23.6 $\sim$ 24.5 times speed-ups for one-FPGA and two-FPGA implementations respectively, while considering all control and data transfer overheads. If we compare only the computation time, two-FPGA implementation is over 46 times faster compared to the fastest CPU implementation.

TABLE 3. Processing time comparison against multicore CPU emulations.

	Processing time (s)		
	28-qubit	29-qubit	30-qubit
QuEST [29]	43	92	190
HpQC [35]	25	53	107
Proposed one-FPGA (computation only) (Total time)	 0.95 2.02	 1.90 4.00	 - -
Proposed two-FPGA (computation only) (Total time)	 0.53 1.06	 1.08 2.16	 2.29 4.47

The work in [36] has done a comprehensive evaluation on GPU-based quantum circuit emulators. It proposes a GPU-based emulator called HyQuas, and compared the performance against other state-of-the-art quantum circuit emulators such as QCGPU [25], Qiskit [26], Qulacs [27], Qibo [28], QuEST [29] and Yao [30]. The comparisons are done for 28-qubit QFT on high-end GPUs such as V100 and A100. Note that we cannot compare the results of A100 GPU since only the normalized processing time values are given and the absolute processing time values are not available. However, we were able to extract the processing time values of V100-SXM2 GPU by reading data directly from the figures, and then multiplying those by the absolute processing time of the normalized unit, which is available in [36]. Figure 17 shows the comparison against the GPU-based 28-qubit QFT emulations. The comparison is done only for the computation time since host-device data transfers are not available in [36]. According to the results, both of the proposed one-FPGA and two-FPGA implementations are faster than all GPU-based emulators except HyQuas [36]. According to the method in HyQuas, a quantum circuit is divided into multiple groups of gates. A hybrid of the following two methods is used by applying the most suitable one for each group of gates. One method is to use shared memory by analyzing data dependency of the computations among multiple gates. This method is used for sparse circuits with low data dependency. The other method is to combine the operations of multiply gates into a large matrix multiplication task, that is performed using highly optimized cuBLAS library [37]. This method is used for dense circuits. Since GPUs are very efficient in matrix multiplication, have more computation resources and a larger external memory bandwidth compared to FPGAs, HyQuas provides better performance. Both proposed and HyQuas are capable of using multi-FPGAs and multi-GPUs respectively. However, the processing time "scaling factor" using 2 GPUs in HyQuas is only 1.56, while the scaling factor using 2 FPGAs in the proposed method is 2.0. Since it is possible to connect any number of FPGAs in the proposed architecture, there is a great potential to increase the processing speed for large emulations. Moreover, significantly larger processing speed can be expected by using the latest Intel Agilex M series FPGAs [38], due to the availability of more resources and larger clock frequency.

The BSP provided with Stratix 10 MX FPGA board does not allow real-time on-board power monitoring, and HyQuas [36] also does not provide power consumption details. According to the FPGA board manual, the maximum power consumption of the FPGA including core resources, RAMs, HBM memories and transceivers is 140W. After adding the power consumption of the board interface including PSUs, the maximum power consumption is expected to be 175W. Maximum power consumption of V100-SXM2 GPU is 300W [39], which is 1.7 times larger compared to that of FPGA. Considering the fact that FPGA uses less than 50% of the logic resources, 34% of the DSPs and 23% of the RAM blocks, there is a greater possibility that the power consumption of the FPGA is significantly lower than the maximum value. Therefore, we have measured the system power consumption at the operational state of the FPGA, and also the system power consumption without the FPGA board. The difference of these measurements is 88W, and we can assume that this is mostly caused by the power consumption of the FPGA board. As a result, we can assume that the power consumption of the FPGA is significantly lower than that of

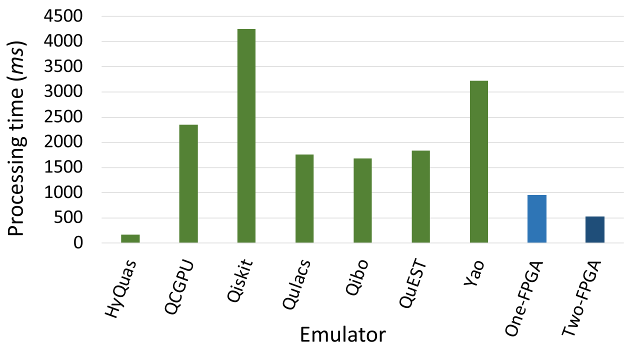


FIGURE 17. Comparison against GPU-based emulation methods for 28-qubit QFT. The GPU emulations are done on V100 GPU in [36].

the GPU implementation in HyQuas. However, real-time on-board power monitoring of both FPGA and GPU is required to compare the power consumption accurately.

Table 4 shows the performance of previous FPGA emulators. All of those methods are only available for small emulations of up to 16 qubits, while the proposed method can process 30 qubits. Methods such as [15]–[18] uses fixed-point computation, so that the results may inaccurate for larger number of qubits. Moreover, none of these methods are compatible with multiple FPGAs. Therefore, we cannot expect these methods to be usable for large emulations, even if those are implemented on latest FPGAs.

TABLE 4. Performance of previous FPGA emulators.

Previous Work	FPGA	Number of qubits	Processing time (ms)
[15]	Stratix IV	8	1.2×10^{-3}
[16]	Stratix IV	5	not given
[17]	Xilinx Artix-7	3	not given
[18]	Stratix II	16	not given
[19]	Xilinx ZYNQ	6	115.47×10^{-3}
[20]	Arria 10	4	not given

V. CONCLUSION

In this paper, we proposed an HBM-based scalable multi-FPGA emulator for QFT. We discuss a memory allocation method that efficiently utilize HBM-based memory subsystem and inter-FPGA channels to evenly distribute data while allowing parallel access to HBM modules on multiple FPGAs. We implemented the proposed emulator up to 30 qubits using two FPGAs, and achieved 23.6 $\sim$ 24.5 times speed-up compared to a fully optimized 24-core CPU emulator. The proposed emulator is faster than most of the GPU emulators except HyQuas [36]. Since we use OpenCL software for design, it is easy to optimize the proposed emulator architecture for FPGA boards with different resource constraints, while potentially compatible with FPGAs in future generations. Therefore, we can increase the processing speed further by using the latest Intel Agilex M series FPGAs [38], and could expect even larger improvements with future FPGAs.

HyQuas [36] that uses GPUs is the fastest QFT emulator. One of the significant difference in HyQuas [36] compared to all other emulators is the usage of combined quantum gates. This eliminates intermediate data transfers to the external memory and also fully exploits highly efficient matrix multiplication in GPUs. Usage of combined quantum gates should be considered in future for FPGA-based emulators also to increase the speed-up further. On the other hand, proposed emulator has a better scalability compared to HyQuas. Since we use direct data transfers among FPGAs using fiber-optic cables, the inter-FPGA data transfer overhead is negligible. Therefore, we can expect high scalability for more than two FPGAs. FPGA-based super computers such as Cygnus [40] can be used to increase both the speed-up and the number of qubits further. Recently, FPGAs are available in cloud environments such as Amazon web services [41] and Intel DevCloud [42]. Therefore, the proposed software-based design can be optimized and distribute easily in such cloud environments.

REFERENCES

[1] M. A. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*. Cambridge, U.K.: Cambridge Univ. Press. [Online]. Available: https://www.amazon.com/Quantum-Computation-Information-10th-Anniversary/dp/1107002176/ref=sr_1_1?crid=2HWZAAHFL7C1&keywords=Quantum +Computation+and+Quantum+Information&qid=1655413394&s=books&sprefix= quantum+computation+and+quantum+information%2Cstripbooks-intl-ship%2C209&sr=1-1

[2] (2022). *The Advantage Quantum Computer—D-Wave Systems*. [Online]. Available: https://www.dwavesys.com/solutions-and products/systems/

[3] J. Chow, O. Dial, and J. Gambetta, "IBM quantum breaks the 100-qubit processor barrier," IBM, Armonk, NY, USA, Tech. Rep., 2021. [Online]. Available: https://research.ibm.com/blog/127-qubit-quantum-processor-eagle

[4] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, "Quantum machine learning," *Nature*, vol. 549, pp. 195–202, Sep. 2017.

[5] M. S. Won, "Intel Agilex FPGAs deliver a game-changing combination of flexibility and agility for the data-centric world," Intel, Santa Clara, CA, USA, Tech. Rep., 2019. [Online]. Available: https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/wp/agilex -fpgas-deliver-game-changing-combination-wp.pdf

[6] *Intel Stratix 10 MX FPGA*, Intel, Santa Clara, CA, USA, 2021.

[7] Y.-k. Choi, Y. Chi, J. Wang, L. Guo, and J. Cong, "When HLS meets FPGA HBM: Benchmarking and bandwidth optimization," 2020, *arXiv:2010.06075*.

[8] K. Mizutani, H. Yamaguchi, Y. Urino, and M. Koibuchi, "OPTWEB: A lightweight fully connected inter-FPGA network for efficient collectives," *IEEE Trans. Comput.*, vol. 70, no. 6, pp. 849–862, Jun. 2021.

[9] D. Coppersmith, "An approximate Fourier transform useful in quantum factoring," IBM, Armonk, NY, USA, Tech. Rep. RC 19642, 1994. [Online]. Available: https://arxiv.org/abs/quant-ph/0201067

[10] Y. S. Weinstein, M. A. Pravia, E. M. Fortunato, S. Lloyd, and D. G. Cory, "Implementation of the quantum Fourier transform," *Phys. Rev. Lett.*, vol. 86, no. 9, p. 1889, 2001.

[11] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, Nov. 1994, pp. 124–134.

[12] A. Yu. Kitaev, "Quantum measurements and the abelian stabilizer problem," 1995, *arXiv:quant-ph/9511026*.

[13] H. M. Waidyasooriya, M. Hariyama, and K. Uchiyama, *Design of FPGA-Based Computing Systems With OpenCL*. Berlin, Germany: Springer, 2018.

[14] S. Winograd, "On computing the discrete Fourier transform," *Math. Comput.*, vol. 32, no. 141, pp. 175–199, 1978.

[15] Y. Qian, M. Wang, J. Chen, L. Wang, and Z. Feng, "Efficient FPGA emulation of quantum Fourier transform," in *Proc. China Semicond. Technol. Int. Conf. (CSTIC)*, Mar. 2019, pp. 1–3.

[16] Y. H. Lee, M. Khalil-Hani, and M. N. Marsono, "FPGA-based quantum circuit emulation: A case study on quantum Fourier transform," in *Proc. Int. Symp. Integr. Circuits (ISIC)*, Dec. 2014, pp. 512–515.

[17] T. M. Aye and M. Iwahashi, "Implementation and analysis of quantum Fourier transform gates over FPGA framework," in *Proc. 8th Medit. Conf. Embedded Comput. (MECO)*, Jun. 2019, pp. 1–5.

[18] J. F. Rivera-Miranda, A. J. Caicedo-Beltran, J. D. Valencia-Payan, J. M. Espinosa-Duran, and J. Velasco-Medina, "Hardware emulation of quantum Fourier transform," in *Proc. IEEE 2nd Latin Amer. Symp. Circuits Syst. (LASCAS)*, Feb. 2011, pp. 1–4.

[19] A. Silva and O. G. Zabaleta, "FPGA quantum computing emulator using high level design tools," in *Proc. Eight Argentine Symp. Conf. Embedded Syst. (CASE)*, Aug. 2017, pp. 1–6.

[20] N. Mahmud and E. El-Araby, "A scalable high-precision and high-throughput architecture for emulation of quantum algorithms," in *Proc. 31st IEEE Int. Syst.-Chip Conf. (SOCC)*, Sep. 2018, pp. 206–212.

[21] V. Hlukhov, "FPGA based digital quantum computer verification," in *Proc. IEEE 11th Int. Conf. Dependable Syst., Services Technol. (DESSERT)*, May 2020, pp. 178–182.

[22] M. Levental, "Tensor networks for simulating quantum circuits on FPGAs," 2021, *arXiv:2108.06831*.

[23] E. S. Fried, N. P. D. Sawaya, Y. Cao, I. D. Kivlichan, J. Romero, and A. Aspuru-Guzik, "QTorch: The quantum tensor contraction handler," *PLoS One*, vol. 13, no. 12, Dec. 2018, Art. no. e0208510.

[24] I. L. Markov and Y. Shi, "Simulating quantum computation by contracting tensor networks," *SIAM J. Comput.*, vol. 38, no. 3, pp. 963–981, Jan. 2008.

[25] A. Kelly, "Simulating quantum computers using OpenCL," 2018, *arXiv:1805.00988*.

[26] M. S. Anis, H. Abraham, A. Offei-Danso, and E. A. R. Agarwal, "Qiskit: Open-source quantum development," IBM, Armonk, NY, USA, Tech. Rep., 2022. [Online]. Available: https://qiskit.org

[27] Y. Suzuki, Y. Kawase, Y. Masumura, Y. Hiraga, M. Nakadai, J. Chen, K. M. Nakanishi, K. Mitarai, R. Imai, S. Tamiya, T. Yamamoto, T. Yan, T. Kawakubo, Y. O. Nakagawa, Y. Ibe, Y. Zhang, H. Yamashita, H. Yoshimura, A. Hayashi, and K. Fujii, "Qulacs: A fast and versatile quantum circuit simulator for research purpose," *Quantum*, vol. 5, p. 559, Oct. 2021.

[28] S. Efthymiou, S. Ramos-Calderer, C. Bravo-Prieto, A. Pérez-Salinas, D. García-Martín, A. Garcia-Saez, J. I. Latorre, and S. Carrazza, "Qibo: A framework for quantum simulation with hardware acceleration," *Quantum Sci. Technol.*, vol. 7, no. 1, Jan. 2022, Art. no. 015018.

[29] T. Jones, A. Brown, I. Bush, and S. C. Benjamin, "QuEST and high performance simulation of quantum computers," *Sci. Rep.*, vol. 9, no. 1, pp. 1–11, Jul. 2019, doi: 10.1038/s41598-019-47174-9.

[30] X.-Z. Luo, J.-G. Liu, P. Zhang, and L. Wang, "Yao.Jl: Extensible, efficient framework for quantum algorithm design," *Quantum*, vol. 4, p. 341, Oct. 2020.

[31] Khronos Group. (2020). *OpenCL Overview*. [Online]. Available: https://www.khronos.org/opencl/

[32] BittWare. (2022). *520N-MX*. [Online]. Available: https://www.bittware.com/fpga/520n-mx/

[33] R. Jozsa, "Fidelity for mixed quantum states," *J. Mod. Opt.*, vol. 41, no. 12, pp. 2315–2323, 1994.

[34] Intel. *Enabling High-Performance Floating-Point Designs*. Accessed: 2022. [Online]. Available: https://www.intel.com/content/dam/support/us/en/programmable/support-resources/bulk-container/pdfs/literature/wp/wp-01267-fpgas-enable-high-performance-floating-point.pdf

[35] H. Bian, J. Huang, R. Dong, Y. Guo, and X. Wang, "HpQC: A new efficient quantum computing simulator," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.*, M. Qiu, Ed. Cham, Switzerland: Springer, 2020, pp. 111–125.

[36] C. Zhang, Z. Song, H. Wang, K. Rong, and J. Zhai, "HyQuas: Hybrid partitioner based quantum circuit simulation system on GPU," in *Proc. ACM Int. Conf. Supercomputing*, Jun. 2021, pp. 443–454.

[37] Nvidia. (2022). *CuBLAS*. [Online]. Available: https://developer.nvidia.com/cublas

[38] Intel. (2022). *FPGA Solves Memory Bandwidth Bottleneck*. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/programmable/agilex-m-series-memory-bandwidth-solution-brief.html

[39] Nvidia. (2022). *NVIDIA V100 Tensor Core GPU*. [Online]. Available: https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf

[40] (2022). *Overview of Cygnus: A New Supercomputer at CCS*. [Online]. Available: https://www.ccs.tsukuba.ac.jp/wp-content/uploads/sites/14/2018/12/About-Cygnus.pdf

[41] (2022). *Amazon Web Services*. [Online]. Available: https://aws.amazon.com/

[42] (2022). *Overview of Cygnus: A New Supercomputer at CCS*. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html

HASITHA MUTHUMALA WAIDYASOORIYA received the B.E. degree in information engineering and the M.S. and Ph.D. degrees in information sciences from Tohoku University, Japan, in 2006, 2008, and 2010, respectively. He is currently an Associate Professor with the Graduate School of Information Sciences, Tohoku University. His research interests include reconfigurable computing, high-performance computing, processor architectures, and high-level design methodology for VLSIs.

HIROKI OSHIYAMA received the B.S., M.S., and Ph.D. degrees in physics from Tohoku University, Japan, in 2015, 2017, and 2020, respectively. He is currently a Specially Appointed Assistant Professor with the Graduate School of Information Sciences, Tohoku University. His research interests include quantum many-body physics, tensor networks algorithms, and quantum computing.

YUYA KUREBAYASHI received the B.S. and M.S. degrees in physics from Tohoku University, Japan, in 2017 and 2019, respectively, where he is currently pursuing the Ph.D. degree with the Graduate School of Science. His research interests include quantum many-body physics, tensor networks algorithms, and quantum software development.

MASANORI HARIYAMA (Associate Member, IEEE) received the B.E. degree in electronic engineering and the M.S. and Ph.D. degrees in information sciences from Tohoku University, Sendai, Japan, in 1992, 1994, and 1997, respectively. He is currently a Professor with the Graduate School of Information Sciences, Tohoku University. His research interests include real-world applications such as robotics and medical applications, big data applications such as bio-informatics, high-performance computing, VLSI computing for real-world application, high-level design methodology for VLSIs, and reconfigurable computing.

MASAYUKI OHZEKI (Member, IEEE) received the Ph.D. degree in physics from the Tokyo Institute of Technology, in 2008. He has subsequently spent one and a half years as a Postdoctoral Fellow. He has been working as an Assistant Professor with Kyoto University, since 2010; and an Associate Professor with the Graduate School of Information Sciences, Tohoku University, since 2016. Since 2020, he has been a Professor with Tohoku University. His research interests include broad, including machine learning and its potential from a perspective of theoretical physics and itself. He was awarded the 6th Young Scientists' Award of the Physical Society of Japan and the Young Scientists' Prize by The Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology, in 2016.