Highly-Parallel FPGA Accelerator for Simulated Quantum Annealing

Hasitha Muthumala Waidyasooriya, *Member, IEEE,* and Masanori Hariyama, *Member, IEEE*

Abstract—Quantum annealing (QA) is a method to find the global optimum of a combinatorial optimization problem by using quantum fluctuations. Quantum annealers such as D-wave are efficient to solve small problems with less than 2048 variables. Simulated quantum annealing on digital computers allows us to solve large real-world problems. However, the processing time increases exponentially with the number of variables. This paper proposes a highly-parallel accelerator for simulated quantum annealing exploiting spatial and temporal parallelism. The accelerator is implemented using "open computing language (OpenCL)" on FPGA. For 8,192 spin models, we achieve 145 times speed using 32 Trotters in one FPGA and 290 times speed-up using 64 Trotters in two FPGAs, compared to single-core CPU implementation.

Index Terms—Simulated quantum annealing, optimization problems, OpenCL for FPGA, high performance computing, multi-FPGA acceleration.

1 Introduction

The concept of quantum computing is regarded as a future computing method with massive degree of parallelism. Quantum gate based computing is implemented by IBM [1] and quantum annealing [2], [3] is implented by D-wave Systems [4]. Quantum annealing potentially holds the promise of solving combinatorial optimization problems [5] extremely faster than digital computers. One famous example of combinatorial optimization problems is the traveling salesman problem [6], [7] that is used to find the shortest route to visit given cities. Solving "combinatorial optimization problems" is important to find optimal solutions for many real-world applications such as traffic-flow analysis [8], workforce management, financial analysis [9], [10], graph problems [11], [12], etc. However, the number of quantum bits in quantum annealers such as [13], [14] is not enough to solve large-scale real-world problems. Therefore, this paper focus on the acceleration of simulated quantum annealing (SQA) in conventional digital computers, which is regarded as an important topic.

In recent years, SQA has been implemented on ASICs (application specific integrated circuits) [15], on FPGAs [16], [17], and also on GPUs [18], [19]. Many such accelerators implement parallel computations by restricting the number of interactions between spins. It is difficult to map real-world problems with dense spin-models to such accelerators without employing minor embedding [20]. Previous work such as [8], [21] have shown that minor embedding reduces the number of usable spins and allows only small problems to be mapped. In addition, the lack of scalability, small dynamic range in computation, lack of support for large simulations, large design time and high cost are some of the other problems of those works.

SQA is based on quantum Monte Carlo (QMC) algorithm which is severely data dependent, and its execution is naturally sequential. Parallel implementation of SQA for fully connected spin models is regarded as extremely difficult task. This paper proposes a highly-parallel FPGA-based SQA accelerator for fully connected spin models. Parallel implementation is done by exploiting spatial and temporal parallelism together at clock-cycle level. According to the evaluation using fully connected spin models, we have observed that the proposed FPGA implementation produces exactly the same output of the sequential implementation done on a single-core CPU, while giving a massive speed-up upto 290 times. The proposed accelerator is implemented using a "C-like" programming method called OpenCL [22], [23]. OpenCL is a heterogeneous system design method that is used to design FPGA accelerators, manages data transfers among FPGAs and the CPU, and controls FPGA execution [24]. Since the accelerator is implemented using software, the design time is dramatically reduced, and modifications can be done easily in software level. We expand our method to multiple FPGAs so that the computations are performed in many FPGAs in parallel. We have implemented upto 32,768 spins, and 1 billion spin-to-spin interactions in the proposed FPGA accelerator. It is larger than previous work [15], [16] and commercially available SQA accelerators such as [25], [26].

2 Previous work

D-wave [4] is a quantum annealer that uses the quantum effects to solve combinatorial optimization problems. D-Wave 2000Q QPU (quantum processing unit) has 2048 qubits and 5600 couplers [14]. A coupler represents the interaction between two qubits. We can change the coupler strength in such a way that the two qubits to be end up in the same state or the opposite states. Due to the limited

• *Hasitha Muthumala Waidyasooriya and Masanori Hariyama are with Graduate School of Information Sciences, Tohoku University, 6-3-09, Aramaki-Aza-Aoba, Aoba, Sendai, Miyagi 980-8579, Japan. E-mail: {hasitha, hariyama}@tohoku.ac.jp*

Manuscript received June 25, 2018; revised August 26, 2018.

number of couplers, we need minor embedding [20] to map problems where each qubit has interactions among many qubits. Therefore, only the small problems such as a TSP of 7 cities can be mapped directly onto D-wave [4]. In addition, quantum annealers are operated on extremely low temperature and extremely low noise. Such physically challenging conditions make it difficult to expand the number of qubits and couplers.

The research done by Google [27] shows that the processing time of SQA on a CPU scales similarly to that of QA on D-Wave 2X quantum annealer [13]. Since the number of qubits and couplers in a QPU is limited, acceleration of SQA under the current technology using conventional computers is regarded as a very important topic. In order to solve larger problems, ASICs are proposed in [15], [25]. However, the development cost is very large and it is not possible to change the problem size or the number of interactions among spins after fabrication. As a result, more flexible designs using GPUs and FPGAs have been proposed [16], [17], [18], [19]. Studies in [16], [17], [18] are based on sparse spin models where one spin interact with only a few other spins. It is relatively easy to implement parallel processing on such sparse spin models due to small data dependency. The method in [19] ignores the data dependencies to implement parallel computation on fully connected spin models. Since the modified algorithm in [19] does not follow the mathematical model that the QMC is based on, the output of the simulation could deviate from the optimum.

Our previous work in [28] also proposes an FPGA accelerator for SQA using temporal parallelism. Compared to [28], this work dramatically increases the degree of parallelism by employing both spatial and temporal parallelism together. We propose a clock-cycle level scheduling scheme for parallel processing while preserving the data dependency. The processing speed of the proposed architecture has increased by 32 times and the number of spins are doubled, compared to [28]. The proposed accelerator implemented using OpenCL-based design method. We can add more FPGA and connect those using fiber optic cables to increase the degree of parallelism further.

3 Parallel Execution of Simulated Quantum Annealing

3.1 Quantum Monte-Carlo Simulation

The Hamiltonian of the Ising model with a transverse field is expressed by Eq.(1) [29], [30], [31]. Ising model is a mathematical model of a system consists of spins and their interactions.

$$\hat{H} = -\sum_{ij} J_{i,j}\hat{\sigma}_i^z\hat{\sigma}_j^z - \Gamma\sum_i \hat{\sigma}_i^x \quad (1)$$

A spin is denoted by σ, the interaction coefficient between spin i and j is denoted by $J_{i,j}$ and the transverse field is denoted by Γ. The transverse field controls the rate of transition between states and plays a similar role that the temperature does in simulated annealing [32]. By decreasing transverse field from a very large value to zero, we hopefully drive the system into the optimal state that has the lowest energy. The m-dimensional transverse-field Ising model can be mapped to a (m+1)-dimensional classical Ising model [33]. It uses multiple replicas called "Trotters" as shown in Fig.1 to represent the superposition of spins. Each spin have either the value -1 (spin-down) or the value +1 (spin-up).

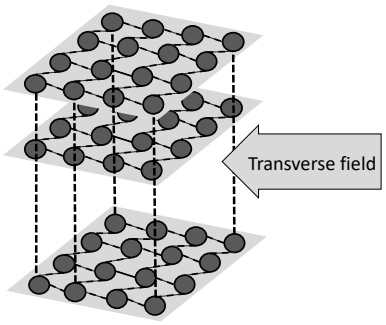

Fig. 1. Ising model with a transverse field.

```
1  for(int t = 0; t<T; t++) {
2    for(int m=0; m<M; m++) {
3      for(i=0; i<N; i++) {
4          local_field[i] = 0;
5        for(j=0; j<N; j++) {
6          //Compute energy
7          local_field[i] += spin(m,j) * J(i,j);
8        }
9        //Compute energy considering the two
       neighboring Trotter slices;
10       local_field[i] -= spin(m-1,i) * K;
11       local_field[i] += spin(m+1,i) * K;
12       if( exp(-1.0*local_field[i]/T) > rand_num )
13           spin[i] = !spin[i];
14       }
15     }
16   }
17 }
```

Listing 1. An extract of the algorithm used for SQA.

Quantum Monte Carlo simulation [34] is used to simulate the quantum tunneling phenomena of an Ising model with a transverse field. Listing 1 shows an extract of the algorithm used for SQA. It consists of four loops. The outermost loop is the time-loop that executes for many iterations. Each iteration is called a "Monte Carlo step". When there are a lot of Monte Carlo steps, it implies that the transverse field decreases slowly, which usually produces better result that are closer to the global optimum. The next loop in line 2 is executed for all Trotters. The next loop in line 3 is executed for all spins in a Trotter. The inner-most loop (in line 5) executes for all interactions among spins. In each Monte-Carlo step, we start the computation from Trotter 1 and compute each spin belonging to the same Trotter sequentially. Then, we move to Trotter 2 and repeat the same process. Similarly, the rest of the Trotters are computed one after the other. The computation time equals to $T \times M \times N \times N$, where T is the number of Monte Carlo steps, M is the number of Trotters, and N is the number of spins.

3.2 Inter-Trotter data access and temporal parallelism

The data flow graph (DFG) in Fig.2 shows the data-flow among Trotters in two consecutive Monte Carlo steps. A node corresponds to the computations of the two inner-loops in Listing 1. The computation of Trotter m requires the

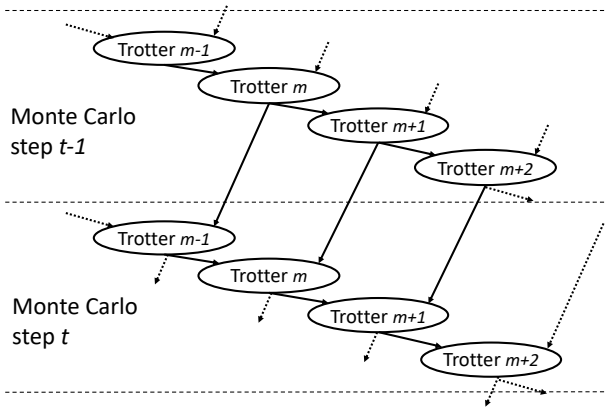

Fig. 2. DFG of SQA that shows the data-flow among Trotters in different Monte Carlo steps.

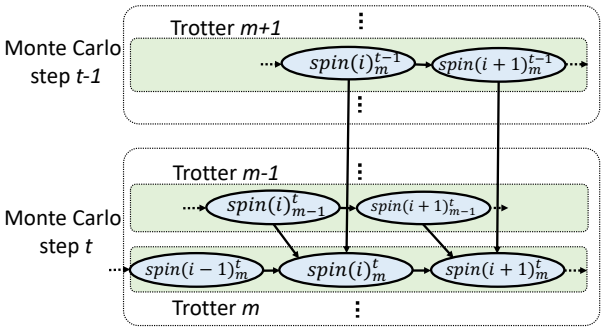

Fig. 3. DFG of SQA that shows the data-flow inside Trotters and among Trotters. For the simplicity, only a part of the data dependencies among spins belonging to same Trotter is shown.

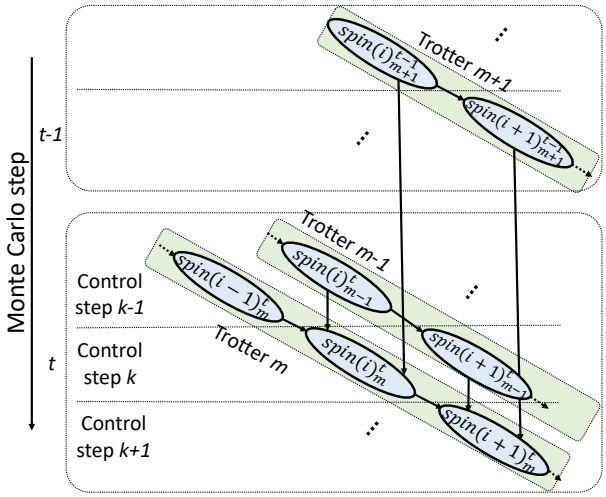

Fig. 4. Scheduled Data-flow graph of SQA. The computations corresponding to the spins belonging to the same Trotter are done in serial to preserve the data dependency.

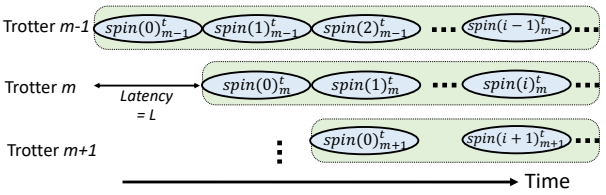

Fig. 5. Time chart of the computations in Monte Carlo step t. Nodes in different Trotters are processed in parallel.

data of the Trotter $m-1$ of the same Monte Carlo step, and the data of Trotter $m+1$ of the previous Monte Carlo step. Due to this data dependency, Trotters of the same Monte Carlo step have to be processed in serial.

The DFG in Fig.3 shows the data flow inside Trotters and among Trotters. A node corresponds to the computations of a spin shown from lines 4 to 13 in Listing 1. For the simplicity, we have shown only a part of the data dependencies among spins belonging to the same Trotter. We discuses the data dependency in the inner-most loop in section 3.3. The computations corresponding to the spin i of Trotter m belonging to Monte Carlo step t is denoted by the node $spin(i)^t_m$. The computations of $spin(i)^t_m$ requires $spin(i)^t_{m-1}$ and $spin(i)^{t-1}_{m+1}$ that are belonging to two neighboring Trotters. It also requires $spin(i-1)^t_m$ which is belonging to the same Trotter m. Therefore, a data dependency exists between the spins of the same Trotter, and $spin(i)$ of the neighboring Trotters. There is no data dependency between any other spins. Therefore, the data dependency among Trotters is partial, and some computations can be done in parallel, without violating the data dependency constraint.

Fig.4 shows the scheduled DFG of SQA. The data dependent nodes are assigned to different control steps, while the nodes with no data dependencies are assigned to the same control step. For example, $\mathrm{spin}(i)^t_m$ and in its predecessor $\mathrm{spin}(i)^t_{m-1}$ are done in control steps k and $k-1$ respectively so that $\mathrm{spin}(i)^t_m$ can use the results of its predecessor. On the other hand, nodes such as $spin(i)^t_m$ and $spin(i+1)^t_{m-1}$ are assigned to the same control step since they do not have any data dependency. Different Monte Carlo steps are processed in serial to preserve the data dependencies between nodes belonging to Trotters m and $m+1$.

Fig.5 shows the time-chart of the computations in Monte Carlo step t, derived from the scheduled DFG in Fig.4. The nodes belonging to the same Trotter are processed in serial to preserve the data dependency. The computations of each Trotter begins after a delay of L clock cycles of its previous Trotter. The delay L is the latency of the computation corresponding to a spin. That is, the computation of $spin(0)^t_m$ begins after the computation of $spin(0)^t_{m-1}$ is completed. The computations of $spin(0)^t_m$ and $spin(1)^t_{m-1}$ are done in parallel. The computation of $spin(0)^t_m$ requires the data of $spin(0)^t_{m-1}$ and $spin(0)^{t-1}_{m+1}$. Those data are available since the computation of $spin(0)^t_{m-1}$ is just completed, and the computation of $spin(i)^t_{m+1}$ is already completed during the previous Monte Carlo step. As a results, we can process multiple nodes belonging to multiple Trotters in parallel. This is called temporal parallelism.

3.3 Intra-Trotter data access and spatial parallelism

Fig.6 shows the DFG of the computations corresponding to the inner-most loop in Listing 1. Nodes $spin(i)^t_m$ and

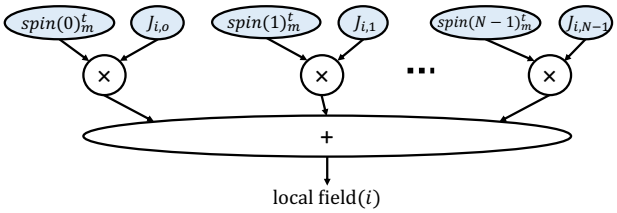

Fig. 6. DFG of the computations in the inner-most loop of Listing 1.

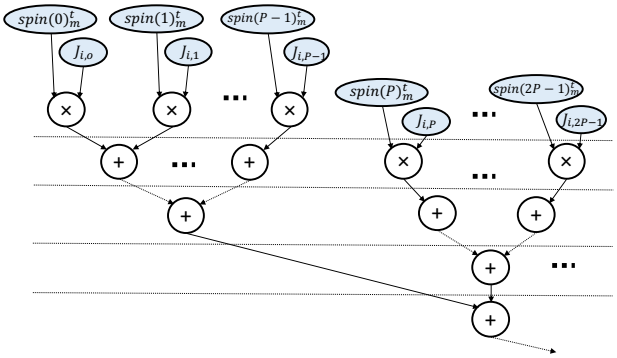

Fig. 7. Scheduled DFG of the computations in the inner-most loop of Listing 1.

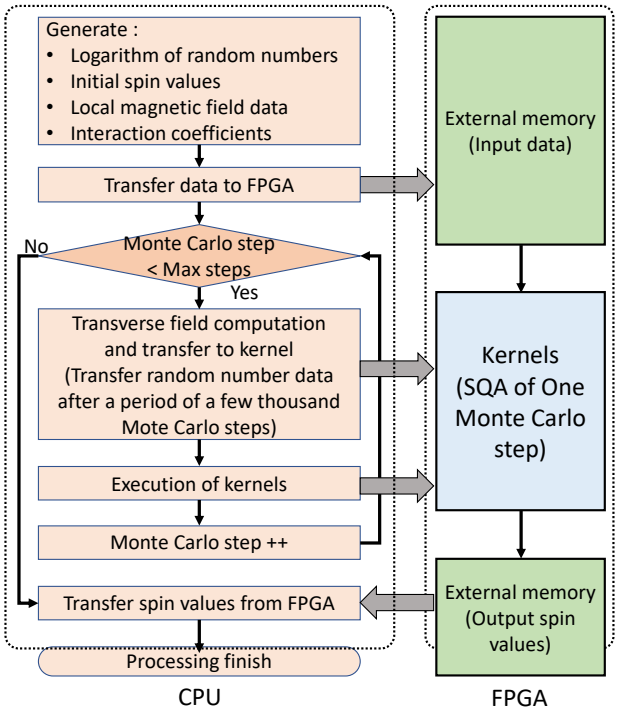

Fig. 8. Flow-chart of SQA computation using the host CPU and an FPGA.

$J_{i,j}$ represents the computations corresponding to the spin i of Trotter m belonging to Monte Carlo step t, and the interaction coefficient between spins i and j, respectively. This computation can be done in parallel since there is no data dependency among nodes.

Fig.7 shows the scheduled DFG. In each control step, P multiplications are performed in parallel. This is called spatial parallelism. Employing spatial parallelism, we reduce the number of loop-iterations to N/P, and increase the processing speed by P times. When P increases, more data should be accessed in parallel. As a result, the required memory access bandwidth increases.

In section 3.2, we have shown how to achieve a M degree of parallelism by processing all M Trotters in parallel. In this section, we have shown how to achieve a P degree of parallelism in each Trotter by processing the inner-most loop in parallel. Therefore, we can achieve a $P \times M$ degree of parallelism in total.

4 FPGA ACCELERATOR ARCHITECTURE

4.1 Overall architecture

In this section, we explains the proposed FPGA accelerator for SQA. As shown in Listing 1, the `exp` computation in line 12 is executed only once per every N iterations of the inner-most loop. Therefore, it is inefficient to allocate hardware resources for a computation that has a low utilization ratio. In order to avoid this computation on FPGA, we change line 12 into logarithm computation of random numbers. Note that it is also inefficient to allocate hardware resources for logarithm computation due to low utilization ratio. However, we can pre-calculate the logarithm of random numbers and transfer those data to the external memory of FPGA periodically as shown in the flow chart in Fig.8. The local magnetic field and interaction coefficient data of an optimization problem are computed in CPU and transferred to the FPGA at the beginning of the computation. Transverse field data are also computed in CPU and transfer to the FPGA before the computation of each Monte Carlo step. The SQA of one Monte Carlo step is computed in FPGA. The computations are done using single precision floating-points. After all the Monte Carlo steps are computed, the output spin values are transferred to the CPU.

Fig.9 shows the FPGA architecture of SQA. It consists of an external memory and multiple kernels. The external memory is connected to the host CPU through a PCIe bus. Spin data, interaction coefficient among spins, local magnetic field and random numbers are transferred from the CPU and stored in the external memory. The transverse field data are transferred from the CPU in each Monte Carlo step, just before the execution of kernels. When the kernels are executed, spin and the local magnetic field data are copied from the external memory and distributed to the local memories of multiple kernels. Each kernel in the accelerator computes several Trotters in parallel. Multiple kernels also executed in parallel, so that all Trotters in all kernels are computed in parallel, by exploiting temporal parallelism. Each Trotter contains multiple processing elements (PEs). Multiple PEs computes the local field of a spin in parallel, by exploiting spatial parallelism. Note that different spins belonging to the same Trotter are computed in serial to preserve the data dependency.

The proposed SQA accelerator architecture is implemented using high-level OpenCL-based design method [23]. The CPU source code is written in "C language" while

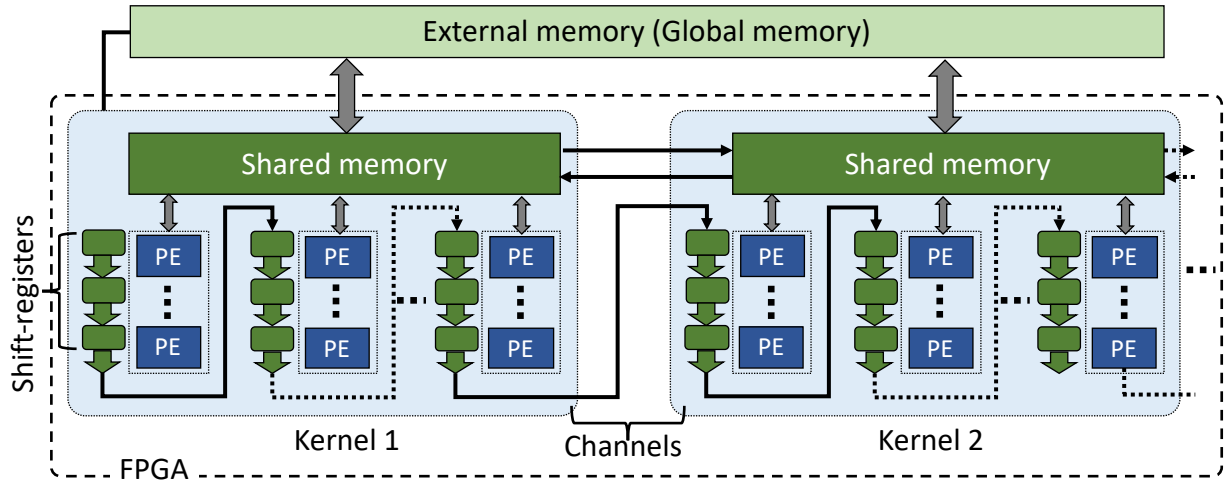

Fig. 9. FPGA architecture for SQA.

the FPGA source code (kernel code) is written in OpenCL as single work-item kernels. The accelerator architecture is designed to realize the following three objectives.

1) Implementation of spatial and temporal parallelism
2) Reduction of the external memory access
3) Scalability

4.2 Implementation of spatial and temporal parallelism

As shown in Fig.9, all kernels execute M Trotters and each Trotter is executed by P PEs. Since the accelerator is fully pipelined, $M \times P$ local filed computations are done in every clock cycle. Fig.10 shows the time-chart of the computation of one Monte Carlo step. Initially, we transfer the data of N spins of all M Trotters from the external memory to the internal memory in $N \times M$ clock cycles. Then we access P interaction coefficients in parallel from the external memory and compute the first spin of Trotter 0. The data access and the computation is pipelined, and "$N/P + latency$" clock cycles are required to compute the first spin. For larger N values, we can neglect the latency and approximate the number of clock cycles to N/P. Therefore, the execution of the first Trotter is completed in $N \times N/P$ clock cycles, and the execution of the other $M-1$ Trotters are completed in $N/P \times (M-1)$ clock cycles. After all computations are computed, the resulting spin data are transferred back to the external memory in $N \times M$ clock cycles. The total number of clock cycles per Monte Carlo step is given by Eq.(2).

$$\text{Total clock cycles} = (N + M - 1)N/P + 2N \times M \quad (2)$$

Assuming that the local filed computation is done in every clock cycle, the serial execution of one Monte Carlo step of SQA in Listing 1 requires $N \times N \times M$ clock cycles. As a result, we have achieved a speed-up of $\frac{N \times N \times M}{(N+M-1)N/P+2N \times M}$. When $N >> M$, the speed-up nearly equals to the total degree of parallelism $M \times P$. In our implementation, the proposed clock-cycle level scheduling scheme is implemented using high-level OpenCL coding. As we can see in the evaluation in section 5, the measured processing time is very close to the theoretical one obtained by Eq.(2).

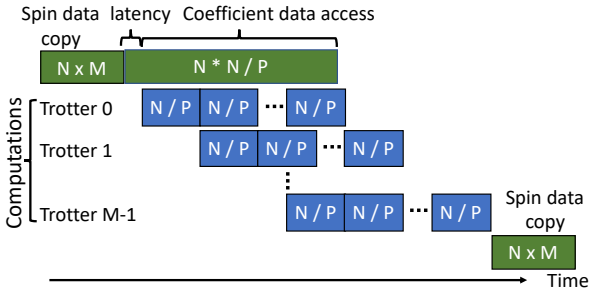

Fig. 10. Time chart of the computation of one Monte Carlo step.

4.3 Reduction of external memory access

Since the external memory bandwidth of FPGAs is relatively small compared to CPUs and GPUs, reducing the external memory access is critical for high performance. We use shared and distributed memory to store spin data locally in the FPGA. Since the computation of a Trotter slice requires the spins of neighboring Trotters, the spin data are stored in a shared memory in each kernel. Since there are many kernels, spin data are distributed among multiple kernels. This allows us to localized the memory access and to reduce the access complexity, while reducing the storage capacity by sharing data. We use OpenCL channels to transfer spin data among neighboring kernels. A channel is a hand-shake based data transfer method between source and destination kernels.

Since all Trotters use the same interaction coefficients (J), we can share those by storing in internal memory of the FPGA. However, the required memory capacity of J is $N \times N \times sizeof(float)$ which can be several gigabytes large. It is not possible to store such a large amount of data in the internal memory of an FPGA. Therefore, we exploit the temporal locality of J to reduce the storage capacity. As shown in Fig.4, $J(i,0) \sim J(i,N-1)$ are required to compute $spin(i)_m^t$. The same $J(i,0) \sim J(i,N-1)$ are required to

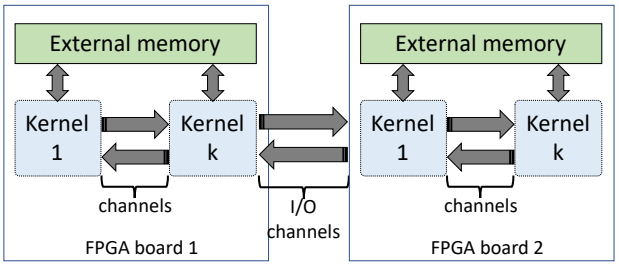

Fig. 11. Multi-FPGA architecture for SQA.

compute $spin(i)^t_{m+1}$. As shown in Fig.5, the computation of $spin(i)^t_{m+1}$ begins immediately after the completion of the computation of $spin(i)^t_m$. Therefore, we use shift-registers in each Trotter to temporally store J as shown in Fig.9. Since the computation of a spin is completed in N/P clock cycles, the length and width of shift-registers are N/P and P, respectively. In each clock cycle, the oldest data of the shift-registers in one Trotter is copied to the beginning of the shift-registers of the next Trotter. At the same time, new data are copied from the external memory to the shift-registers of Trotter 1. We need only $N \times M \times sizeof(float)$ storage to implement the shift-registers. Therefore, we can implement shift registers using the internal memory of FPGA. When there are M number of Trotters, we can reduce the external memory access by M times.

4.4 Scalability

The proposed accelerator architecture consists of multiple kernels. We can scale in Trotter dimension by adding or removing kernels. The number of Trotters per kernel is also adjustable by changing a single parameter in the OpenCL code. Since J data are reused internally and the spin data are already in the local memory, increasing the number of Trotters does not increase the external memory access. However, it increases the internal memory utilization since more data should be stored internally. The number of PEs (P) in a Trotter is also adjustable by changing a single parameter. Using a large value for P increases the required external memory access bandwidth, since more data have to be accessed in parallel. Increasing either M or P increases the DSP due to the increase in parallel operations.

We can increase the number of Trotters by adding more FPGAs, as shown in Fig.11. We use I/O channels to transfer spin data between kernels that are belonging to different FPGAs. I/O channels have the same behavior of channels but are implemented using high-speed QSFP (quad small form-factor pluggable) ports that are connected by fiber-optic cables. Therefore, multiple FPGAs work as a one large FPGA [35], and all the kernels in all the FPGAs are executed in parallel. Since the accelerator is composed of multiple kernels, the clock frequency is nearly a constant when we add more kernels to the same FPGA or when we add multiple FPGAs. This has been shown in evaluation in section 5.

The scalabiltiy of the processing time against the number of spins of the proposed accelerator is similar to that of CPU-based SQA for a fixed Trotter size. However, for the same number of spins, the processing time of the proposed accelerator is nearly a constant for different Trotter sizes. Although, there is no specific rule to define the Trotter size of an Ising model, a larger Trotter size is usually required to simulate larger spin models. Therefore, the proposed method is efficient for larger simulations since increase in processing time due to larger Trotter size is negligible.

5 Evaluation

The evaluation is done using a system that has two FPGA boards and a CPU. The FPGA boards are "Nallatech 385A accelerator boards" [36] that contain Intel Arria 10 10AX115N3F40E2SG FPGAs. Compilation of the OpenCL codes is done using Quartus 17.1 with Intel FPGA SDK for OpenCL 17.1 [37]. The CPU is Intel Xeon Silver 4116 CPU that has 12 physical cores. The operating system is CentOS Linux 7.6 and compilation of the CPU code is done using gcc 9.2. We have implemented the proposed SQA accelerator using one and two FPGAs for different number of spins and Trotters.

5.1 Performance evaluation of single and multi-FPGA implementations

Table 1 shows the processing time per Monte Carlo step and resource utilization of one-FPGA implementation. We measure the processing time of 4,096 Monte Carlo steps using a fully connected spin model and calculate the average time for one Mote Carlo step. The data transfer time from CPU to FPGA is included in the evaluation, and it is accounts for less than 3% of the total processing time. When the number of spins is a constant, the processing time is very similar for different number of Trotters. When the number of spins doubles, the processing time increases by nearly four times. As discussed in section 4.2, the total number of clock cycles per Monte Carlo step using the FPGA accelerator is given by Eq.(2). Since $N >> M$, the computation cost nearly equals to $N \times N/P$, where the number of spins, the number of Trotters and the degree of spatial parallelism are denoted by N, M and P respectively. Therefore, theoretically the processing time is independent of the number of Trotters and relative to N^2, since P is a constant. We have observed this theoretical relationship in the experimental results in Table 1. We can obtain the theoretical processing time by dividing the number of clock cycles in Eq.(2) from the clock frequency. The measured processing times of all implementations are within the limits of $92\% \sim 99\%$ of the theoretical processing time. This shows that the clock cycle level scheduling discussed in section 4 has been accurately implemented using our high-level OpenCL-based design method.

As shown in Table 1, RAM blocks is the most used resource. When either of the number of spins or Trotters increases, we have to store more data in local memory and shift-registers. There the RAM block utilization increases with the number of Trotters and spins. Note that, it is usually difficult to design an accelerator when the RAM block usage is more than 80%. The DSP usage and logic resources are only proportional to the number of Trotters. Since increasing the number of Trotters requires more parallel operations, we need more DSP blocks and logic resources

TABLE 1
Processing time and resource utilization of one-FPGA implementation.

Number of spins	Trotters	Kernels	Processing time per Monte Carlo step [ms]	FPGA clock frequency [MHz]	Resource utilization: Logic resources	Registers	DSP blocks	Memory bits [MB]	RAM blocks
4096	4	1	4.61	255	78,779 (18%)	149,884	100 (7%)	0.64 (10%)	476 (18%)
	8	2	4.71	256	112,593 (26%)	231,561	200 (13%)	0.86 (13%)	677 (25%)
	16	4	5.11	249	142,740 (33%)	305,139	400 (26%)	1.22 (18%)	886 (33%)
	32	8	5.81	244	205,215 (48%)	454,442	800 (53%)	1.90 (29%)	1300 (48%)
8192	4	1	17.45	252	79,225 (19%)	151,061	100 (7%)	0.81 (12%)	542 (20%)
	8	2	17.79	249	113,349 (27%)	232,438	200 (13%)	1.20 (18%)	809 (30%)
	16	4	19.87	246	143,768 (34%)	306,956	400 (26%)	1.90 (29%)	1154 (43%)
	32	8	20.732	245	206,517 (48%)	457,753	800 (53%)	3.26 (49%)	1850 (68%)
16384	4	1	65.77	260	79,512 (19%)	151,802	100 (7%)	1.15 (17%)	680 (25%)
	8	2	68.97	249	113,987 (27%)	233,562	200 (13%)	1.89 (28%)	1085 (40%)
	16	4	70.33	249	143,625 (34%)	308,268	400 (26%)	3.26 (49%)	1720 (63%)
32768	4	1	264.93	256	79,471 (19%)	152,012	100 (7%)	1.82 (28%)	1085 (40%)
	8	2	270.08	251	113,935 (27%)	235,340	200 (13%)	3.25 (49%)	1702 (63%)

for computation. The number of spins has no relationship with the degree of parallelism. Therefore, despite increasing the number of spins, the DSP and logic utilization remains nearly the same. The DRAM usage is also critical when the number of spins increases. In order to store the interaction coefficients of 32,768 spins, 4 GB of external memory is required. Since we also have to store other data such as spin values, local magnetic filed and random numbers, and only 8 GB of external memory is available on the FPGA board [36], it is difficult to increase the number of spins further.

Despite the resource usage varies considerably for different number of spins and Trotters, clock frequency is nearly a constant. Since the proposed architecture is highly scalable as explained in section 4.4, their is no performance degradation when the resource utilization is large. According to these results, we can expect to scale the accelerator for larger number of spins and Trotters using high-end FPGAs such as Stratix 10 [38].

Fig.12 shows the processing time comparison of one and two FPGA implementations. The processing time is shown for one Monte Carlo step. The processing times of 4,096, 8,192, 16,384 and 32,768 spins are shown in Figs.12(a)∼12(d), respectively. Compared to 1-FPGA implementation, the number of Trotters are doubled in 2-FPGA implementation. However, the processing time is nearly a constant. Note that the clock frequencies of all implementations are not exactly the same, and this is one reason for the small differences in of the processing times. The other reason is N/P increase of clock cycles per Trotter as shown by Eq.(2). Despite such small differences, the processing time is nearly independent of the number of Trotters even more than one FPGA is used. Therefore, we can scale the accelerator for even larger number of Trotters using multiple FPGAs.

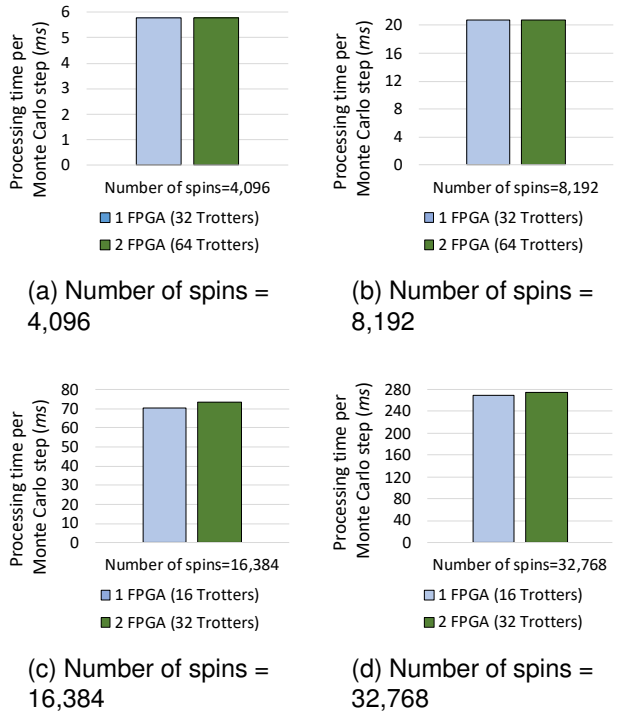

Fig. 12. Processing time comparison of one and two FPGA implementations for one Monte Carlo step.

5.2 Comparison against CPU implementation

We have compared the proposed FPGA accelerator against CPU implementation of SQA using single and multiple CPU cores. For a fair comparison, we have performed the following optimization steps to the CPU implementation. The exponential computations are replaced by logarithm computations of random numbers, similar to the FPGA implementation. We use pre-calculated logarithm values of random numbers, and this computation time is not included in the CPU processing time. The inner-most loop of Listing 1 is performed as a parallel reduction operation using multiple cores. The parallel implementation is done using the OpenMP [39] directive *#pragma omp parallel for shared(...) reduction(+: ..)*. The compilation of the CPU code is done using the optimization options *-O3*, *-march=native* and *-fopenmp*.

Figs.13(a), 13(b), 13(c) and 13(d) show the processing time comparison between CPU and FPGA, when the number of spins are 4,096, 8,192, 16,384 and 32,768 respectively. We implement upto 64 Trotters for 4,096 and 8,192 spins using two FPGAs. However, we can implement upto 32 and 16 Trotters only for 16,384 and 32,768 spins respectively due

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2019.2957177, IEEE Transactions on Emerging Topics in Computing

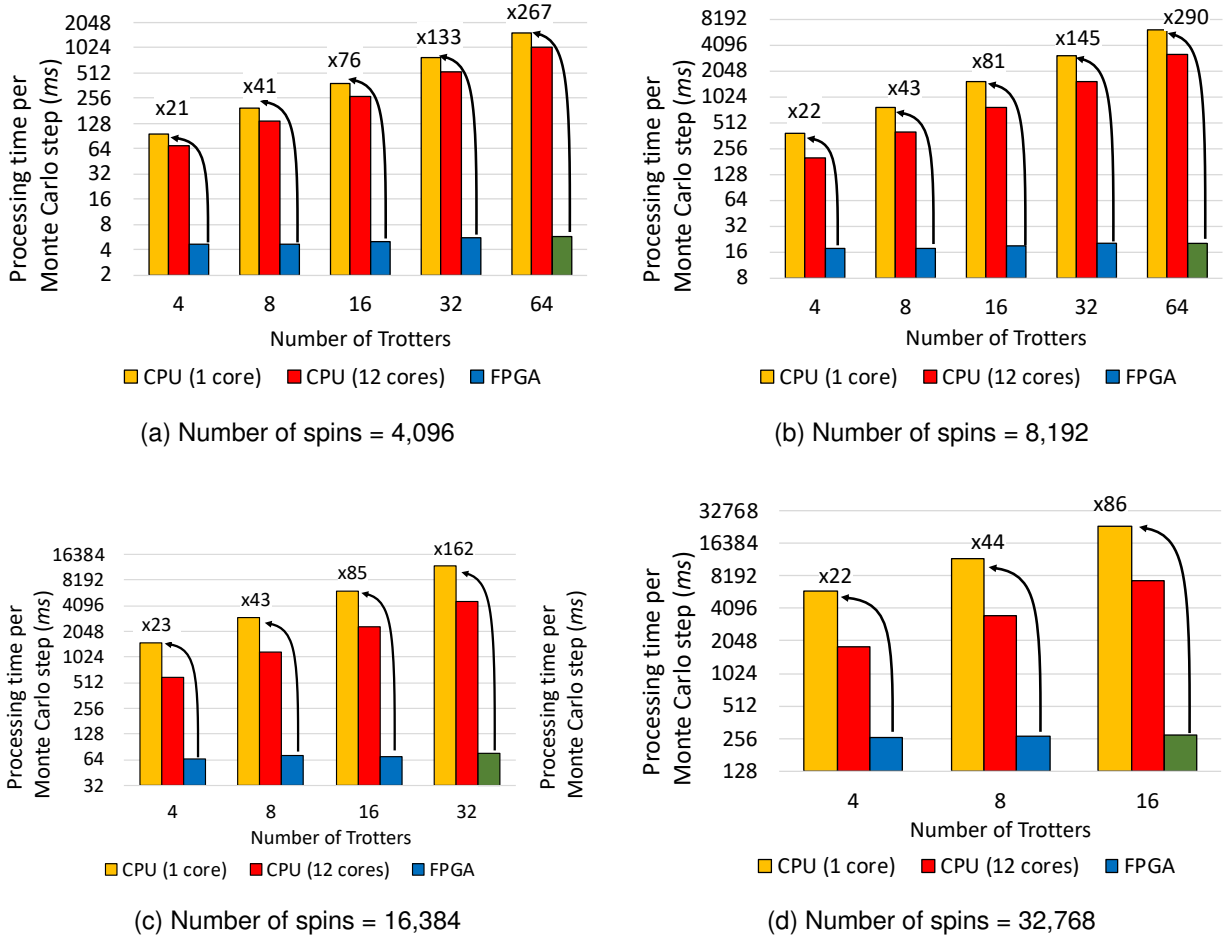

Fig. 13. Processing time comparison of CPU and FPGA for one Monte Carlo step. Processing time is shown in log scale.

to FPGA resource constraints. FPGA implementations use Trotter-level temporal parallelism, so that the processing time is nearly a constant irrespective of the number of Trotters. CPU implementations do not use temporal parallelism, and the processing time increases with the number of Trotters. Note that, It is not easy to efficiently implement Trotter-level temporal parallelism on CPUs since it requires clock-cycle level scheduling. As a result, we achieved a large speed-up for the largest Trotter sizes in 4,096 and 8,192 spins. The maximum speed-ups of 1-FPGA implementation compared to single-core and 12-core CPU implementations are 145 and 100 times respectively. The maximum speed-ups of 2-FPGA implementation compared to single-core and 12-core CPU implementations are 290 and 199 times respectively. Note that the speed-up of 8,192-spin implementation is slightly larger than that of 4,096-spin implementation. One possible reason for this small difference is the better cache utilization in CPU for smaller number of spins that require small memory capacity.

Fig.14 shows the processing time per Monte Carlo step against the number of CPU cores. We have achieved 4.7 times speed-up using 6 CPU cores, compared to the single-core implementation. However, we have achieved only 6.5 times speed-up using all 12 cores. The processing time initially decreases when the number of cores increases, and then saturates. Therefore, it is difficult to expect a larger speed-up using more CPU cores.

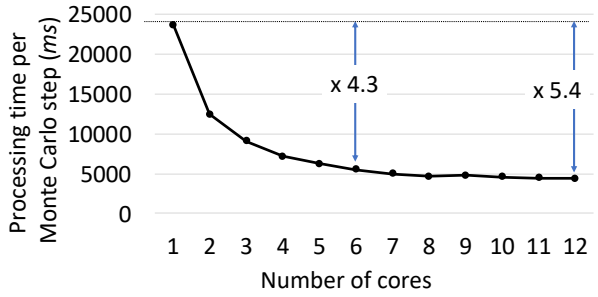

Fig. 14. Processing time per Monte Carlo step against the number of CPU cores. The processing time is saturated when the number of cores increased.

Increasing the number of Trotters can lead to find the solution in smaller number of Mote Carlo steps, and smaller processing time in FPGAs. In order to show this, we evaluate the processing time required to solve the "number partitioning problem" [40] using different number of Trotters. In the number partitioning problem, a series of numbers is divided in to two sets where the summations of the numbers in each set are similar as possible. The solution is optimized

when the difference between the summation of two sets is minimized. In order to map the number partitioning problem onto Ising model, we represent each number as n_i and assign an Ising variable σ_i to n_i. The value of σ_i is either "+1" or "-1", depending on which set that the number n_i is belonging to. Therefore $\sum_{i=1}^{N} n_i\sigma_i$ equals to the difference of the "summation of set 1" and the "summation of set 2". We can optimize the solution by minimizing the difference. The Hamiltonian of the number partitioning problem is written as Eq.(3).

$$H = -\left(\sum_{i=1}^{N} n_i\sigma_i\right)^2 = -\sum_{i,j=1}^{N} n_i n_j \sigma_i \sigma_j \qquad (3)$$

According to Eq.(3), we can determine the interaction coefficients $J_{ij} = -n_i n_j$ and solve the problem using SQA. The Ising formulation of number partitioning problem is discussed in detail in [41].

We have created a number partitioning problem of 8,192 random integers between 0 to 24,575. The created problem has a guaranteed optimal solution where the numbers can be divided into two sets with equal sums. We have run 100 simulations, where each simulation is executed until the probability of finding the optimal solution is more than 95%. For this purpose, we executed each simulation for 100 trials, where the optimal solution is obtained in more than 95% of those. Initially, each trial is done in one Monte Carlo step. If the optimal solution is not found in 95% of the trials, we increase the number of Monte Carlo steps by one and execute the 100 trials again. The number of Monte Carlo steps are increased until the probability of finding the optimal solution is over 95%. This evaluation method is similar to the one used in [27].

Fig.15 shows the average processing time to solve the number partitioning problem with a probability of 95%. We can see that the processing time of FPGA continuously decreases with the number of Trotters. When we increase the number of Trotters, a solution can be found in smaller number of Monte Carlo steps. Since multiple Trotters are computed in parallel in FPGA, the processing time per Monte Carlo step remains the same even the number of Trotters is increased. On the other hand, the processing time of CPU initially decreases from 16 to 32 Trotters, and then increased for 32 to 64 Trotters. Since multiple Trotters are processed in serial in CPU, the processing time per Monte Carlo step increases with the number of Trotters. The number of Monte Carlo steps required to find the optimal solution hugely depends on the given optimization problem. As a result, despite the reduction of Mote Carlo steps, the total processing time of CPU can either increase or decrease, depending on the optimization problem. This shows that the proposed FPGA implementation can be used to decrease the processing time in two ways, one is by increasing the degree of parallelism, and the other is by reducing the number of Monte Carlo steps using larger number of Trotters.

5.3 Comparison with previous work

Fig.16 shows the processing time comparison against our previous work in [28]. The proposed accelerator is $14 \sim 32$ times faster compared to [28]. A major reason for this speed-up is the usage of spatial and temporal parallelism together, as explained in section 3.

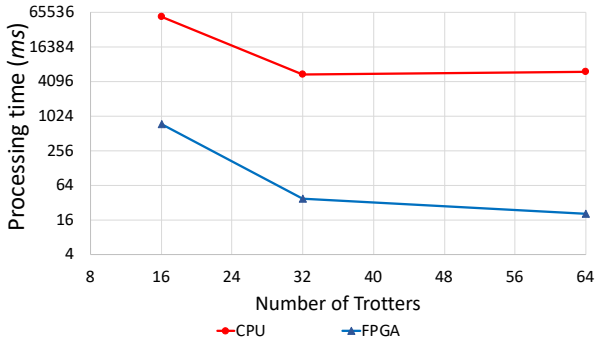

Fig. 15. Average processing time to solve the number partitioning problem of 8192 integers for different number of Trotters.

Fig. 16. Processing time comparison against [28].

Table 2 shows the comparison against the commercially available SQA accelerators. The proposed accelerator has the most number of couplings, and the largest dynamic range with sufficient precision. It has 4 times more spins compared to the Fujitsu Digital Annealer [25] that also use total coupling. The proposed accelerator is highly scalable from spin-level to FPGA-level, easy to design using OpenCL software and cheaper to implement using FPGAs. Note that the processing time comparison is not done since those details of the previous work are not available.

In the proposed FPGA accelerator, we transfer pre-calculated logarithm values of random numbers to FPGA to avoid `exp` or `log` computation in FPGA. Although the data transfer time is small compared to the total processing time, using pre-calculated data can be inconvenient. Therefore, it is necessary to explore methods to implement complex computations such as `exp` or `log` efficiently in FPGA utilizing small amount of hardware resources. The random number generation can be done efficiently in FPGA as shown in many previous studies such as [42], [43].

TABLE 2
Comparison against commercially available SQA accelerators.

	Fujitsu Digital Annealer [25]	Hitachi CMOS Annealer [15], [26]	Hitachi CMOS Annealer [16], [26]	This work
Maximum number of spins	8192	61,952	6,400	32,768
Type of coupling	Total coupling	King graph	King graph	Total Coupling
Number of couplings	67 million	0.37 million	0.4 million	1 billion
Computation	64-bit fixed-point	not mentioned	8-bit fixed point	32-bit floating-point
Implementation	ASIC	ASIC	FPGA	2-FPGA connected via fiber

6 Conclusion

We have proposed a highly-parallel FPGA accelerator for simulated quantum annealing. It is designed using OpenCL software, so the design time is reduced considerably. For 8,192 spin models, we have achieved 145 times speed using 32 Trotters in one FPGA and 290 times speed-up using 64 Trotters in two FPGAs, compared to single-core CPU implementation. The number of spins are large enough to compete with commercially available accelerators that are designed as ASICs or FPGAs. We have shown that it is possible to increase the number of Trotters by using multiple FPGAs, with negligible processing time increase. The number of spins can be increased by increasing the DRAM capacity and using an FPGA with a large internal memory. There is a huge potential to build a large accelerator using multiple high-end Stratix 10 FPGAs [38] for ultra-scale quantum annealing simulations to solve real-world problems.

Acknowledgment

This research is partly supported by MEXT KAKENHI, grant number 19K11998.

References

[1] "IBM Q," https://www.ibm.com/quantum-computing/, 2019.
[2] T. Kadowaki and H. Nishimori, "Quantum annealing in the transverse ising model," *Physical Review E*, vol. 58, no. 5, p. 5355, 1998.
[3] K. Tadashi and N. Hidetoshi, "Study of optimization problems by quantum annealing [ph. d. thesis]," *Tokyo: Department of Physics, Tokyo Institute of Technology*, 1998.
[4] "D-wave," https://www.dwavesys.com, 2019.
[5] A. Schrijver, *Combinatorial optimization: polyhedra and efficiency*. Springer Science & Business Media, 2003, vol. 24.
[6] E. L. Lawler, J. K. Lenstra, A. R. Kan, D. B. Shmoys *et al.*, *The traveling salesman problem: a guided tour of combinatorial optimization*. Wiley New York, 1985, vol. 3.
[7] M. R. Gary and D. S. Johnson, "Computers and intractability: A guide to the theory of np-completeness," 1979.
[8] F. Neukart, G. Compostella, C. Seidel, D. von Dollen, S. Yarkoni, and B. Parney, "Traffic flow optimization using a quantum annealer," *Frontiers in ICT*, vol. 4, p. 29, 2017.
[9] R. Orús, S. Mugel, and E. Lizaso, "Quantum computing for finance: overview and prospects," *Reviews in Physics*, p. 100028, 2019.
[10] N. Elsokkary, F. S. Khan, D. La Torre, T. S. Humble, and J. Gottlieb, "Financial portfolio management using d-wave quantum optimizer: The case of abu dhabi securities exchange," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), Tech. Rep., 2017.
[11] O. Titiloye and A. Crispin, "Quantum annealing of the graph coloring problem," *Discrete Optimization*, vol. 8, no. 2, pp. 376–384, 2011.
[12] H. Ushijima-Mwesigwa, C. F. Negre, and S. M. Mniszewski, "Graph partitioning using quantum annealing on the d-wave system," in *Proceedings of the Second International Workshop on Post Moores Era Supercomputing*. ACM, 2017, pp. 22–29.
[13] "The D-Wave 2X Quantum Computer Technology Overview," https://docs.dwavesys.com/docs/latest/_downloads/09-1076A-T_GettingStarted.pdf.
[14] "The D-Wave 2000Q Quantum Computer Technology Overview," https://www.dwavesys.com/sites/default/files/D-Wave%202000Q%20Tech%20Collateral_0117F.pdf, 2019.
[15] M. Yamaoka, C. Yoshimura, M. Hayashi, T. Okuyama, H. Aoki, and H. Mizuno, "A 20k-spin ising chip to solve combinatorial optimization problems with cmos annealing," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 303–309, 2016.
[16] T. Okuyama, M. Hayashi, and M. Yamaoka, "An ising computer based on simulated quantum annealing by path integral monte carlo method," in *Rebooting Computing (ICRC), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–6.
[17] H. M. Waidyasooriya, Y. Araki, and M. Hariyama, "Accelerator architecture for simulated quantum annealing based on resource-utilization-aware scheduling and its implementation using opencl," in *International Symposium on Intelligent Signal Processing and Communication Systems (ISPAC)*, 2018, pp. 336–340.
[18] M. Weigel, "Performance potential for simulating spin models on gpu," *Journal of Computational Physics*, vol. 231, no. 8, pp. 3064–3082, 2012.
[19] C. Cook, H. Zhao, T. Sato, M. Hiromoto, and S. X.-D. Tan, "Gpu based parallel ising computing for combinatorial optimization problems in vlsi physical design," *arXiv preprint arXiv:1807.10750*, 2018.
[20] A. Zaribafiyan, D. J. Marchand, and S. S. C. Rezaei, "Systematic and deterministic graph minor embedding for cartesian products of graphs," *Quantum Information Processing*, vol. 16, no. 5, p. 136, 2017.
[21] M. Booth, S. P. Reinhardt, and A. Roy, "Partitioning optimization problems for hybrid classical/quantum execution," *D-wave technical report series*, pp. 01–09, 2017.
[22] T. S. Czajkowski, D. Neto, M. Kinsner, U. Aydonat, J. Wong, D. Denisenko, P. Yiannacouras, J. Freeman, D. P. Singh, and S. D. Brown, "Opencl for fpgas: Prototyping a compiler," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012, p. 1.
[23] "Intel FPGA SDK for OpenCL," https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html, 2016.
[24] "Intel FPGA SDK for OpenCL Programming Guide," https://www.altera.com/documentation/mwh1391807965224.html, 2018.
[25] "Digital Annealer," https://www.fujitsu.com/global/digitalannealer/superiority/, 2019.
[26] "Anneling Cloud Web," https://annealing-cloud.com, 2019.
[27] V. S. Denchev, S. Boixo, S. V. Isakov, N. Ding, R. Babbush, V. Smelyanskiy, J. Martinis, and H. Neven, "What is the computational value of finite-range tunneling?" *Physical Review X*, vol. 6, no. 3, p. 031015, 2016.
[28] H. M. Waidyasooriya, M. Hariyama, M. J. Miyama, and M. Ohzeki, "Opencl-based design of an fpga accelerator for quantum annealing simulation," *The Journal of Supercomputing*, pp. 1–21, 2019.
[29] R. Stinchcombe, "Ising model in a transverse field. i. basic theory," *Journal of Physics C: Solid State Physics*, vol. 6, no. 15, p. 2459, 1973.
[30] P. Pfeuty and R. Elliott, "The ising model with a transverse field. ii. ground state properties," *Journal of Physics C: Solid State Physics*, vol. 4, no. 15, p. 2370, 1971.
[31] T. Kadowaki and H. Nishimori, "Quantum annealing in the transverse ising model," *Physical Review E*, vol. 58, no. 5, p. 5355, 1998.
[32] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[33] M. Suzuki, "Relationship between d-dimensional quantal spin systems and (d+1)-dimensional ising systems: Equivalence, critical exponents and systematic approximants of the partition function and spin correlations," *Progress of theoretical physics*, vol. 56, no. 5, pp. 1454–1469, 1976.
[34] M. Suzuki, S. Miyashita, and A. Kuroda, "Monte carlo simulation of quantum spin systems. i," *Progress of Theoretical Physics*, vol. 58, no. 5, pp. 1377–1387, 1977.
[35] H. M. Waidyasooriya and M. Hariyama, "Multi-fpga accelerator architecture for stencil computation exploiting spacial and temporal scalability," *IEEE Access*, vol. 7, pp. 1–1, 2019.
[36] "Nallatech 385A accelerator card," http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-385a-arria10-1150-fpga/, 2018.
[37] "Intel FPGA SDK for OpenCL," https://www.altera.com/products/design-software/embedded-software-developers/opencl/overview.html, 2018.
[38] "INTEL STRATIX 10 FPGAS," https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html, 2019.
[39] "OpenMP," https://www.openmp.org, 2019.
[40] S. Mertens, "The easiest hard problem: Number partitioning," *Computational Complexity and Statistical Physics*, vol. 125, no. 2, pp. 125–139, 2006.
[41] A. Lucas, "Ising formulations of many np problems," *Frontiers in Physics*, vol. 2, p. 5, 2014.
[42] F. Belletti, M. Cotallo, A. Cruz, L. A. Fernandez, A. Gordillo, A. Maiorano, F. Mantovani, E. Marinari, V. Martin-Mayor, A. Muñoz-Sudupe *et al.*, "Simulating spin systems on ianus, an fpga-based computer," *Computer Physics Communications*, vol. 178, no. 3, pp. 208–216, 2008.
[43] S. Tatsumi, M. Hariyama, K. Ito, and T. Aoki, "An fpga accelerator for patchmatch multi-view stereo using opencl," *Journal of Real-Time Image Processing*, pp. 1–13, 2018.

Hasitha Muthumala Waidyasooriya received the B.E degree in Information Engineering, M.S degree in Information Sciences and Ph.D. in Information Sciences from Tohoku University, Japan, in 2006, 2008 and 2010 respectively. He is currently an Assistant Professor with the Graduate School of Information Sciences, Tohoku University. His research interests include reconfigurable computing, processor architectures for big-data processing and high-level design methodology for VLSIs.

Masanori Hariyama received the B.E. degree in electronic engineering, the M.S. degree in information sciences, and the Ph.D. degree 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 bioinformatics, high-performance computing, VLSI computing for real-world application, high-level design methodology for VLSIs, and reconfigurable computing.