The Journal of Supercomputing (2019) 75:5019–5039
https://doi.org/10.1007/s11227-019-02778-w

OpenCL-based design of an FPGA accelerator for quantum annealing simulation

Hasitha Muthumala Waidyasooriya[1] · **Masanori Hariyama**[1] · **Masamichi J. Miyama**[1] · **Masayuki Ohzeki**[1]

Published online: 20 February 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract

Quantum annealing (QA) is a method to find the global optimum for a combinatorial optimization problem by using quantum fluctuations. QA can be simulated on a computer using quantum Monte Carlo (QMC) simulation of the Ising model, while sacrificing a huge processing time. It has been shown that the processing time of QMC simulation on a CPU scales similarly to that of QA on the D-wave 2X quantum annealer, although the latter is over 10^8 times faster than the former. However, large problems should be partitioned into sub-problems and solved separately, and this reduces the processing speed of the quantum annealer. Since the access to a quantum annealer is also limited, acceleration of QA simulations using conventional computers is regarded as a very important topic. If we can reduce the huge computational time, it is possible to use QA simulations to solve combinatorial optimization problems. We propose an FPGA accelerator for QA simulations designed using "open computing language." We achieved up to 12.6 times speed-up for single FPGA implementation and 23.8 times speed-up for two-FPGA implementation compared to a CPU. We also achieved over 9-times large energy efficiency compared to a CPU-based system.

Keywords Simulated quantum annealing · OpenCL for FPGA · Quantum Monte Carlo simulation · FPGA accelerator

✉ Hasitha Muthumala Waidyasooriya
hasitha@ecei.tohoku.ac.jp; hasitha@tohoku.ac.jp

Masanori Hariyama
hariyama@tohoku.ac.jp

Masamichi J. Miyama
miyama@smapip.is.tohoku.ac.jp

Masayuki Ohzeki
mohzeki@smapip.is.tohoku.ac.jp

[1] Graduate School of Information Sciences, Tohoku University, 6-3-09, Aramaki-Aza-Aoba, Aoba, Sendai, Miyagi 980-8579, Japan

1 Introduction

Solving "combinatorial optimization problems" [1] is a key technique to find optimal solutions for the problems in many areas such as traffic simulations, circuit simulations and power distributions. A combinatorial optimization problem is an optimization problem to find a minimum or a maximum of a function that has many variables of discrete values. One famous example is the traveling salesman problem [2, 3] that is used to find the shortest route to visit given cities.

However, most of these problems are NP-complete or NP-hard where the processing time increases exponentially with the number of variables. Simulated annealing (SA) [4] is one of the probabilistic approximate algorithms for finding the global optimum. It uses thermal fluctuations to allow the system to escape from local minimum so that the system can reach the global minimum eventually. However, the time to find an optimal solution increases rapidly as the number of variables increases [5]. Quantum annealing (QA) [6] that uses quantum tunneling processes for state transitions is a promising method to perform better than SA [7].

QA simulation has been implemented on ASICs (application-specific integrated circuits) [8, 9] and also FPGAs [10] in recent years and achieved large speed-ups compared to single-core CPU-based simulations. However, the lack of scalability is a problem to perform large simulations. Recently, FPGAs have been evolving rapidly. The number of logic gates is increased to millions, and the floating point units are increased to thousands. Now, FPGAs can compete with CPUs and GPUs in terms of power consumption and also the floating point computation performance. Recently, we can see that FPGAs are entering into the fields of super computing and high-performance computing [11, 12]. Another strong motivation to use FPGAs in super computing is the introduction of the open computing language (OpenCL)-based design environment [13, 14]. OpenCL is not just an FPGA architecture design method, but also a system design method that can configure a system containing multiple FPGAs and CPUs [15]. Complicated tasks such as FPGA-to-FPGA data transfers, FPGA-to-CPU data transfers and DRAM access have become very simple to implement using a few lines of OpenCL coding. However, designing a near-optimal accelerator is not an easy task even using OpenCL. Implementing CPU-oriented codes on FPGAs rarely increases the performance and even reduces the performance compared to CPUs. Therefore, it requires not only the hardware architecture design experience, but also the knowledge of how to write appropriate OpenCL codes to implement the desired architecture on an FPGA.

In this paper, we propose an FPGA accelerator architecture for QA simulations using complete graphs. We discuss how to implement this accelerator using OpenCL. Based on the single FPGA accelerator, we also developed a multi-FPGA system using OpenCL to further accelerate QA simulations. We implemented the proposed system by using two FPGAs and a CPU. According to the experimental results, we achieved 12.6 times speed-up using one FPGA and 23.8 times speed-up using two FPGAs compared to a single-core CPU implementation. The

proposed accelerator is over 9 times energy efficient compared to the CPU implementation. The largest accelerator we designed using two FPGAs has 16,384 nodes and over 134 million connections. We also discuss how to use more FPGAs while reducing the data transfer bottlenecks to increase the speed-up and problem size further.

2 Previous works

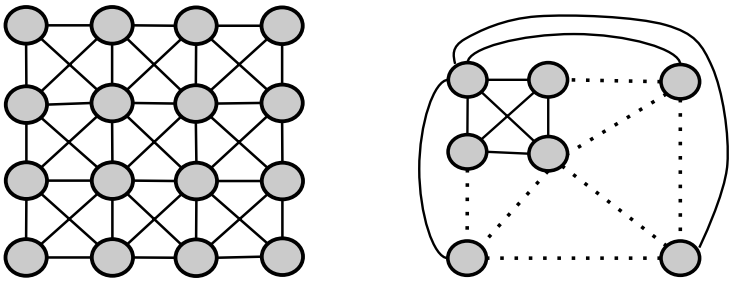
D-wave [16] is a quantum annealer that uses the quantum effects to solve combinatorial optimization problems. There are other quantum-inspired methods to solve combinatorial optimization problems such as "Fujitsu Digital Annealer [17] and "quantum neural network" [18]. The research done by Google in [5] shows that the processing time of quantum Monte Carlo (QMC) simulation on a CPU scales similarly to that of QA on the D-Wave 2X quantum annealer [19] up to 945 variables, although the latter is over 10^8 times faster than the former. The current D-Wave chip DW-2000Q [20] contains 2048 nodes but only 5600 couplers (a coupler represents the connection between two nodes). If a large problem cannot be mapped to the chip, it is divided into several subproblems and solves those separately. After that, the solutions are merged. These problem partitioning and merging are done on conventional computers, and as a result, we may not get a 10^8 times speed-up. Although these limitations will be solved in the future by increasing the number of nodes and couplers, accelerating QA simulations under the current technology using conventional computers is regarded as a very important topic.

In order to accelerate QA simulations, an ASIC was developed in [9], and it has an estimated power efficiency of 1800 times compared to a CPU. However, the fabrication cost is very large and it is not possible to change the problem size or the node topology after fabrication. GPU implementations are also proposed to accelerate QA simulations in [21, 22]. The implementation in [21] is proposed only for "nearest neighbor" graphs. The implementation in [22] is proposed for "max-cut problem," where the data dependency is small. None of these accelerators are suitable for complete graphs where every node has an interaction with all the other nodes. It is very difficult to accelerate QMC algorithm for complete graphs using GPUs due to the lack of SIMD operations and high data dependency. An FPGA-based accelerator has been proposed in [10]. However, this accelerator is implemented only for "King graphs." Since the accelerators proposed in [10, 21, 22] are restricted for a particular problem or a topology, using those to solve different problems is not efficient. In addition, the FPGA accelerator design method used in [10] does not support system designs with multiple FPGAs. Therefore, scalability is not achieved and the problem size depends on the resources available on the FPGA.

3 FPGA accelerator architecture

3.1 Ising model and parallel computation

QA can be simulated on a computer using quantum Monte Carlo (QMC) simulation of the Ising model [7]. In QMC simulation, the n-dimensional quantum system is mapped to an $(n+1)$-dimensional space. The original n-dimensional space is expressed as the "real space," and the additional space is expressed as the "Trotter." Copies of the original space is placed in the Trotter direction. We call those "Trotter slices." Figure 1 shows the Ising model on different graph structures for one Trotter slice. Figure 1a shows the Ising model on a "king graph" where one node is connected to only the nearby nodes. If there are n nodes in a king graph, the number of connections (edges) nearly equals to $4n$ for large values of n. This graph is used by the previous works in [9, 10]. A six-neighbor connected Chimera graph is used in D-wave due to physical restrictions of implementing quantum bits. Since conventional computers do not have such physical restrictions, there is no requirement to use non-complete graphs unless we are designing an accelerator of a one particular application. In this work, we use a complete graph where all nodes are connected to each other. If there are n nodes in a complete graph, the number of connections nearly equals to n^2 for large values of n. As a result, our model has an advantage of mapping any problem that has any type of interaction among nodes. The previous works have severe restrictions on the number of interactions among nodes. However, our model has two severe disadvantages. The first one is the significantly large computation amount compared to that of a king graph. The second one is we cannot use parallel computations for the nodes in a Trotter slice, since a computation of a node depends on the results of all the computations of all the other nodes. Note that [10] uses parallel computations for the nodes that are not connected. Therefore, it is enormously challenging to design an accelerator to implement the computations in a complete graph.



(a) A King graph where one node is connected to only the nearby nodes.

(b) A complete graph where every pair of node is connected.

Fig. 1 Ising model on different graphs (one Trotter slice)

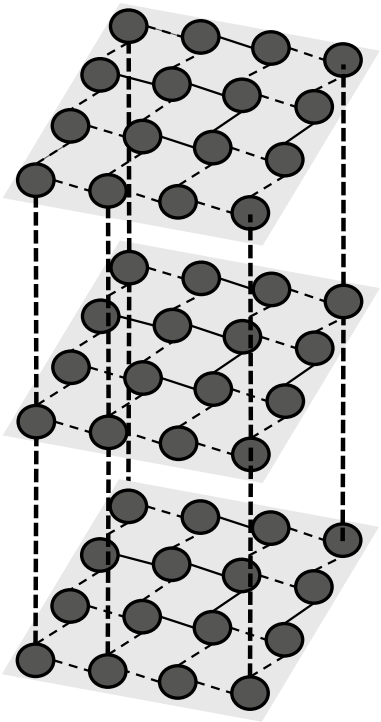


Fig. 2 Multiple Trotter slices and data dependency. Each Trotter slice is a complete graph

Figure 2 shows multiple Trotter slices in the Ising model. A node is only connected to the nodes of its upper and the lower Trotter slices. Therefore, to compute one node, we need the data of the other nodes in the same Trotter and the data of the nodes in the upper and the lower Trotter slices.

In QA simulation, we start from the Trotter number 1 and compute each node in the same Trotter sequentially. Then, we move to the Trotter number 2 and repeat the same process. Similarly, the rest of the Trotter slices are computed one after the other. This computation is shown in the code extract of Listing 1. It has four loops. The three inner loops are computed for T iterations. The two inner loops (k-loop in line 3 and j-loop in line 5) are used to compute the spins of all the nodes in a Trotter slice. The second loop (m-loop in line 2) repeats this computation for M number of Trotter slices. The computation cost nearly equals to $T \times M \times N^2$ for large values of N.

```
for(int t = 0; t<T; t++) {
  for(unsigned int m=0; m<M; m++) {
    for(unsigned int k=0; k<N; k++) {
      lfield = h[k]/M;
      for(unsigned int j=0; j<N-1; j++)
      {
        ...
        lfield = lfield + J[...] * SPIN[m*N+j]/M;
      }
      ...
      up = (m!=0) ? m-1 : M-1;
      down = (m!=M-1) ? m+1 : 0;
      lfield = lfield - Jtrans * (SPIN[up*N+k] + SPIN[down*N+k]);
      ene_diff = 2.0 * SPIN_IN[m*N+k] * lfield;
      p = random_number;
      if( exp(-1.0*ene_diff/T) > p )
          SPIN[m*N + k] = !SPIN[m*N + k];
    }
  }
}
```

Listing 1 Part of the code of quantum annealing simulation.

Figure 3 shows the scheduling of the computation. In the same Trotter slice, the computation of one node uses the data of all the other nodes. Therefore, the computation of the nodes in a Trotter slice must be done sequentially. After computing the fist node of Trotter slice 1, we have all the data required to compute the first node of Trotter slice 2. At the same time, we can also compute the second node of Trotter slice 1. Similarly, we can compute node 1 of slice 3, node 2 of slice 2 and node 3 of slice 1 at the same time in parallel. As a result, we can compute the nodes in multiple Trotter slices in parallel.

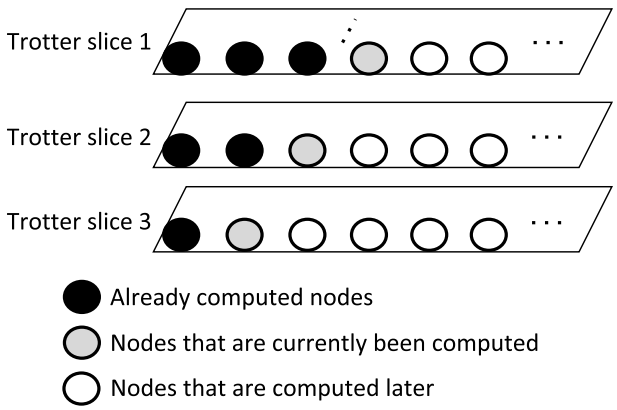


Fig. 3 Scheduling of the computation

3.2 Single FPGA implementation

Figure 4 shows the FPGA architecture for QA simulations. The computation of a Trotter slice is done in a processing element (PE). A PE processes the nodes in a Trotter slice in serial manner, while multiple PEs process nodes in multiple Trotter slices in parallel. As shown in Listing 1, to compute a node, we need spin data (`SPIN`), interaction coefficient among spins (`J`), local magnetic field (`h`) and random numbers. Initially, the spin and local magnetic field data are copied from the global memory of the FPGA board to the internal memory of the FPGA. The interaction coefficient is the same for all Trotter slices, so that we can share those. To do this, we use multiple internal memories for each Trotter slice to store the interaction coefficient data. The Trotter slice 1 accesses those data from the global memory and then stored in the internal memory. This way, we can reduce the global memory access and share the data among multiple Trotter slices. After all the computations of all the Trotter slices are completed, the spin data are copied back to the global memory.

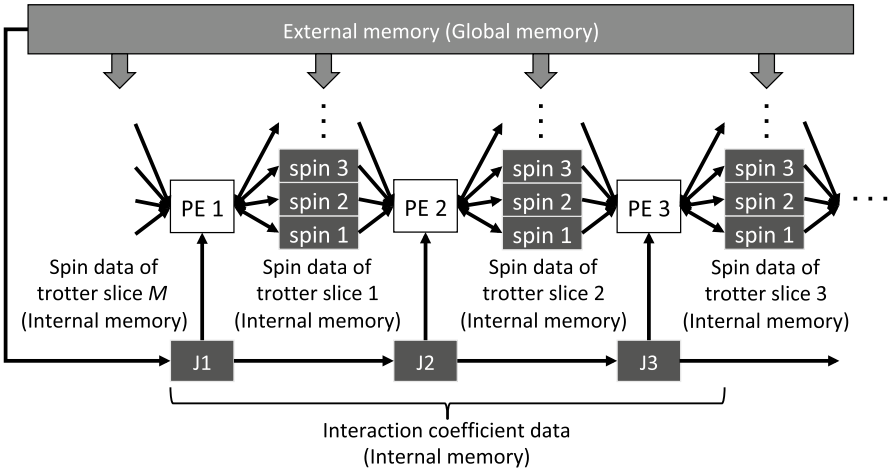


Fig. 4 FPGA architecture for quantum annealing simulation

```
__kernel void sqa(global bool * restrict SPIN_IN, )
{
  unsigned int LOOPCNT = (N+M-1)*(N-1);

  //copy global memory data to the internal memory;
  ...
  unsigned int count = 0;
  float lfield[M];
  while (count != LOOPCNT) {
    #pragma unroll
    for (int i = N-1; i>0; --i) {
      //shifting interaction coefficients
      #pragma unroll
      for (int j=0; j<M; j++ )
        localJ[j][i] = localJ[j][i-1];
    }
    //coping interaction coefficient data
    int index = (int)(count/(N-1))*N+(count%(N-1));
    localJ[0][0] = (index < N*N) ? J[index] : 0.0f;

    //coping interaction coefficient to the next Trotter slice
    #pragma unroll
    for(int m=0; m<M-1; m++)
      localJ[m+1][0] = localJ[m][N-1];

    #pragma unroll
    for(int m=0; m<M; m++) {
      if(count >= m*(N-1) && count < (N+m)*(N-1)) {
        int klocal = (int)((count-m*(N-1)) / (N-1));
        int j = ( count % (N-1) );
        lfield = lfield+localJ[j][m]*lspin[m][j]/M;
        if( ((count+1) % (N-1)) == 0 ) {
          lfield += localh[klocal]/M;
          unsigned int up = (m!=0) ? m-1 : M-1;
          unsigned int down = (m!=M-1) ? m+1 : 0;
          lfield = localfield - Jtrans*(lspin[up][klocal] + lspin[
    down][klocal]);
          enediff = 2.0 * lspin[m][klocal] * lfield;
          p = RandomLog[m*N+klocal];
          if( -1.0*enediff/T > p )
            lspin[m][klocal] = !lspin[m][klocal];
        }
      }
    }
    ++count;
  }

  //copy spin values from the internal memory to the global memory
  ...
}
```

Listing 2 Extract of the OpenCL kernel code for QA simulation.

Listing 2 shows the OpenCL kernel code for FPGA implementation. This code implements the three inner loops of the CPU code in Listing 1. We use single work-item kernel since the quantum annealing simulation is heavily data dependent. Note

that the code is slightly modified for simplicity and clarity by removing some of the FPGA-specific OpenCL modifications. Our objective is to show how to implement the accelerator architecture using OpenCL. At the beginning, spin and local magnetic field data are copied from the global memory to the internal memory. The interaction coefficient is accessed from the global memory and then stored in a shift register array. In each cycle, the data are shifted. The last value of the shift register of Trotter slice m is copied to the first value of the shift register of Trotter slice $m+1$ in line 24. As a result, each coefficient is transferred to the next Trotter slice after a delay of N cycles, and there is a latency of N cycles between the computations of two nearby Trotter slices. Since the architecture is fully pipelined, a computation in a loop iteration is done in every clock cycle. The parallel computation of M Trotter slices is implemented by "#pragma unroll" directive on the m-loop in line 26. In FPGAs, this directive results in implementing parallel computation with the cost of more resource utilization. The exponent computation in the line 18 of the original CPU code in Listing 1 is modified by taking the logarithm from both sides. As a result, we can skip the expensive exponent computation on the FPGA. The logarithm data of the random numbers are pre-calculated in the CPU and transferred to the global memory of the FPGA. After all the computations are completed, the computed spin values are copied back to the global memory. The outermost loop at line 9 is executed for $(N+M-1)*(N-1)$ iterations. Usually, the number of nodes N is much larger than the number of Trotter slices M. Therefore, the computation cost nearly equals to N^2 and independent of M. This is due to the parallel computation of M Trotter slices.

```
1  clSetKernelArg(kernel, 1, sizeof(cl_mem), &mobj_J);
2  clSetKernelArg(kernel, 2, sizeof(cl_mem), &mobj_h);
3  clSetKernelArg(kernel, 3, sizeof(cl_mem), &mobj_RandomLog);
4  for(int t = 0; t<T; t++)
5  {
6    float Gamma = G0*(1-(float)t/Tall);
7    float Jtrans = -T*0.5*log(tanh(Gamma/M/T));
8
9    if((t%2)==0)
10   {
11     clSetKernelArg(kernel,0,sizeof(cl_mem), &mobj_SPIN_IN));
12     clSetKernelArg(kernel,4,sizeof(float), &Jtrans);
13     clSetKernelArg(kernel,5,sizeof(cl_mem), &mobj_SPIN_OUT));
14   }
15   else
16   {
17     clSetKernelArg(kernel,0,sizeof(cl_mem), &mobj_SPIN_OUT);
18     clSetKernelArg(kernel,4,sizeof(TYPE), &Jtrans);
19     clSetKernelArg(kernel,5,sizeof(cl_mem), &mobj_SPIN_IN);
20   }
21
22   clEnqueueTask(queue, kernel, 0, NULL, NULL);
23   clFinish(queue);
24   ...
25 }
```

Listing 3 Part of the host code of quantum annealing simulation.

Listing 3 shows the OpenCL host code that is executed on the host CPU. It executes the time loop of the original CPU code in Listing 1. In each time iteration, the pointers to the spin data are reversed by setting the kernel arguments. Then, the kernel is re-launched. The parameter `Jtrans` is computed on the CPU and transferred to the FPGA as a kernel argument. Since it requires complex computations such as trigonometric and logarithm operations, using CPU for this computation is effective.

3.3 Multi-FPGA implementation

We can increase the processing speed by using multiple FPGAs. A half of the Trotter slices can be processed in one FPGA while the rest in another FPGA in parallel. Figure 5 shows the data partition for two FPGAs. Trotter number 1 to $M/2$ is processed in FPGA 1, and the Trotter number $M/2+1$ to M is processed in FPGA 2. However, to process the Trotter 1, we need the data of the Trotter M from FPGA 2. Similarly, to process Trotter $M/2+1$ in FPGA 2, we need the data of Trotter $M/2$ from FPGA 1.

Data transfer between FPGAs can be done through the host or by connecting two FPGAs directly. The first method requires the kernels to be stopped in order to access the data. This requires a lot of control overhead and is not suitable for low-latency fine-grained data transfers. FPGA boards can be directly connected using high-speed QSFP ports. The Intel Arria 10 FPGA boards contain 40Gbps QSFP ports. Two QSFP ports of two FPGAs can be directly connected using fiber optic cables. Data transfers can be implemented very easily using the OpenCL I/O channel extension.

A channel is a one directional data path between a source and a destination kernel. Channels are used to transfer data between kernels without writing the data into the global memory or the host. The default property of a channel is "blocking." That is, the destination kernel waits until the source kernel writes data to the channel. The source kernel writes the next data just after the destination kernel reads

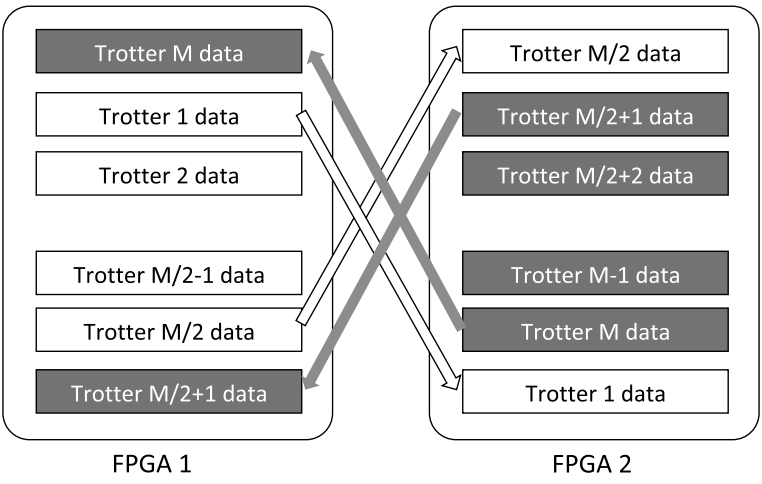


Fig. 5 Data partition for two FPGAs

the previously written data from the channel. Channels can be declared as "non-blocking" also. In this case, the channel provides a flag that shows whether the write operation is succeeded or not. Similarly, it provides a flag to the destination kernel that indicates whether the data are available in the channel or not. Using I/O channels, we can connect two kernels that are executed on two different FPGAs.

```
channel int8 trot_first_f1  __attribute__((io("output_ch0")));
channel int8 trot_last_f1   __attribute__((io("output_ch1")));
channel int8 trot_last_f2   __attribute__((io("input_ch0")));
channel int8 trot_first_f2  __attribute__((io("input_ch1")));

__kernel void FPGA1 ( ... )
{
  ...
  //data read from FPGA 2
  int8 readdata = read_channel_intel(trot_last_f2);
  lspin[M1][j] = readdata.s0 & 0x1;
  ...
  //data write to FPGA 2
  int8 writedata;
  writedata.s0 = lspin[0][klocal];
  write_channel_intel(trot_first_f1, writedata);
  ...
```

Listing 4 I/O channel implementation of FPGA 1 using OpenCL.

Figure 6 shows the accelerator architecture using two FPGAs. The data of the first and the last Trotter slices are transferred using I/O channels. Listings 4 and 5 show extracts of codes that manage the data transfers between FPGAs using I/O channels. There are four channels per FPGA, where two for read and two for write. We directly connect QSFP port 1 of FPGA 1 with the QSFP port 1 of FPGA 2 as shown in Fig. 7. Similarly, the other two QSFP ports of the two board are also directly connected.

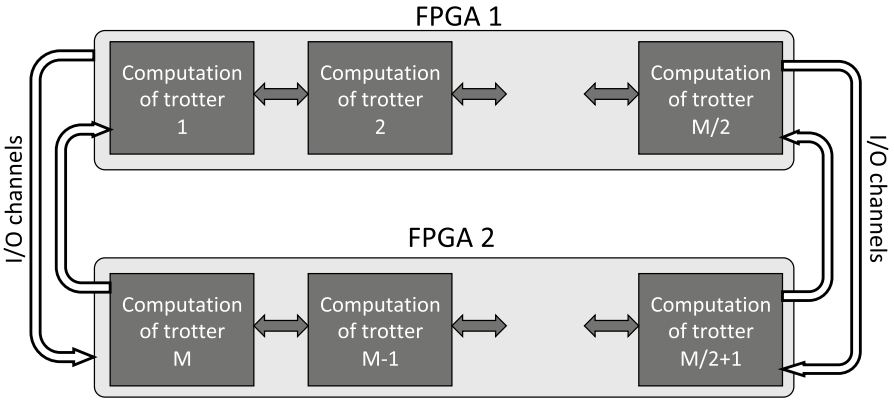


Fig. 6 Accelerator architecture using two FPGAs

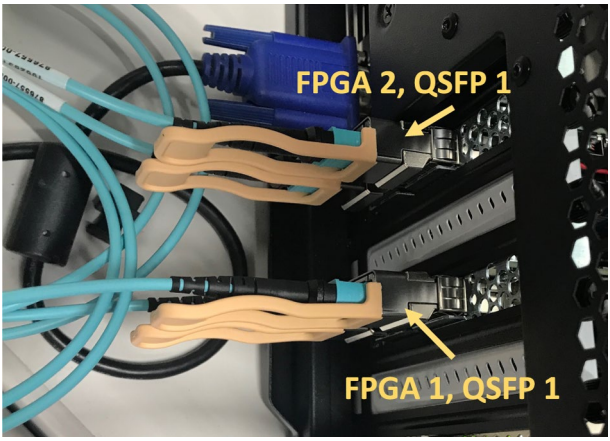


Fig. 7 Two-FPGA system used for the evaluation. FPGAs are directly connected using fiber optic cables

```
channel int8 trot_last_f2  __attribute__((io("output_ch0")));
channel int8 trot_first_f2 __attribute__((io("output_ch1")));
channel int8 trot_first_f1 __attribute__((io("input_ch0")));
channel int8 trot_last_f1  __attribute__((io("input_ch1")));

__kernel void FPGA2 ( ... )
{
  ...
  //data read from FPGA 1
  int8 readdata = read_channel_intel(trot_first_f1);
  lspin[M1][j] = readdata.s0 & 0x1;
  ...
  //data write to FPGA 1
  int8 writedata;
  writedata.s0 = lspin[0][klocal];
  write_channel_intel(trot_last_f2, writedata);
  ...
```

Listing 5 I/O channel implementation of FPGA 2 using OpenCL.

Listing 6 shows a part of the OpenCL host code for two-FPGA implementation. We declare two OpenCL devices, one for each FPGA. We also declare two command queues per each device. Note that both devices are put into the same OpenCL context. The execution of each kernel is queued to different command queues so that we can execute both kernels concurrently.

```
for(int t = 0; t<Tall; t++)
{
  float Gamma = G0*(1-(TYPE)t/Tall);
  float Jtrans = -T*0.5*log(tanh(Gamma/M/T));

  if((t%2)==0)
  {
    clSetKernelArg(kernel[0], 0, sizeof(cl_mem), &mobj_SPIN_IN0);
    ...
    clSetKernelArg(kernel[1], 0, sizeof(cl_mem), &mobj_SPIN_IN1);
    ...
  }
  else
  {
    clSetKernelArg(kernel[0], 0, sizeof(cl_mem), &mobj_SPIN_OUT0);
    ...
    clSetKernelArg(kernel[1], 0, sizeof(cl_mem), &mobj_SPIN_OUT1);
    ...
  }
    ;
  clEnqueueTask(queue[0], kernel[0], 0, NULL, NULL);
  clEnqueueTask(queue[1], kernel[1], 0, NULL, NULL);
  clFinish(queue[0]);
  clFinish(queue[1]);
  ...
}
```

Listing 6 OpenCL host code for two-FPGA implementation.

4 Evaluation

For the evaluation, we use two "Nallatech 385A accelerator boards" [23] that contain Intel Arria 10 10AX115N3F40E2SG FPGAs. We use Quartus 17.1 compiler with Intel FPGA SDK for OpenCL 17.1 to compile the FPGA kernels. The CPU-based QA simulation is done on Intel Xeon Silver 4116 CPU using gcc 4.8.5 compiler. Note that it is difficult to implement parallel computation using CPU due to data dependency in complete graphs. In the two-FPGA implementation, QSFP ports of the FPGAs are connected using QSFP multi-mode fiber optic cables. We use the logarithmic values of the "pre-calculated random data" in the CPU implementation also for a fair comparison. The processing time for random data generation is not included since it can be done offline. Since we use the same QMC simulation algorithm on both CPU and FPGA implementations, the results are exactly the same when using the same input data.

Table 1 shows the processing time comparison for different node sizes while the number of Trotter slices is 16 and 32. When the node size increases by 2×2, the processing time on a CPU increases by nearly 16 times. As we can see in Listing 1, the processing time on a CPU for one iteration (in the time loop) is proportional to $M \times N \times (N-1)$, where M is the number of Trotter slices and N is the node size. When N is increased by four, the processing time is increased by a

Table 1 Processing time versus node size (500 iterations)

Trotter size	Node size	Processing time (s)		Speed-up (times)	FPGA frequency (MHz)
		CPU	FPGA		
16	16×16	2.24	0.29	7.72	172
	32×32	35.91	3.47	10.35	169
	64×64	574.12	62.87	9.13	164
	128×128	9168.81	799.20	11.47	171
32	16×16	4.68	0.69	6.76	125
	32×32	71.20	5.58	12.76	118
	64×64	1151.89	91.43	12.60	120

factor of $\frac{4N \times (4N-1)}{N \times (N-1)}$, which nearly equals to 16. As shown in Listing 2, the processing time on an FPGA is proportional to $(N + M - 1) \times (N - 1)$. Since M is very small compared to N, the processing time must be proportional to 16. However, the processing time of the FPGA implementation increases by a factor of 8 to 18. The following causes can contribute to this difference. As shown in Listing 3, the CPU is used to set the kernel arguments and to execute the kernel. Therefore, the FPGA processing time contains this CPU time also, and this can be a large overhead for the simulations with small node sizes. Moreover, the clock frequency of the FPGA also varies slightly for each implementation.

Table 2 shows the FPGA resource utilization for different node sizes while keeping the Trotter size a constant. As we can see, memory size and the number of RAM blocks are increased by 1∼1.5 times when the node size is increased by 2×2. As explained in Sect. 3.2, increasing the node size does not increase the number of parallel computations, since the nodes in a Trotter slice are computed in serial manner. Because of this, the number of DSPs is unchanged. The other resources are increased very slightly. This shows that we can increase the node size in an FPGA easily without using too many resources.

Table 2 FPGA resource usage versus node size

Trotter size	Node size	Resource usage				
		Logic (%)	Registers	Memory	RAM blocks (%)	DSPs (%)
16	16×16	127,908 (30%)	183,284	1.1 (17%)	967 (36%)	96 (6%)
	32×32	130,485 (31%)	183,782	1.2 (18%)	984 (36%)	96 (6%)
	64×64	133,584 (31%)	185,403	1.5 (22%)	1096 (40%)	96 (6%)
	128×128	136,621 (32%)	187,095	2.4 (36%)	1581 (58%)	96 (6%)
32	16×16	229,147 (54%)	264,256	2.1 (14%)	1807 (67%)	192 (13%)
	32×32	238,468 (56%)	266,394	2.6 (39%)	1930 (71%)	192 (13%)
	64×64	247,987 (58%)	268,896	3.1 (46%)	2147 (79%)	192 (13%)

Table 3 shows the processing time comparison for different Trotter sizes, when the node size is a constant. As shown in Listing 1, the processing time in CPU is proportional to the Trotter size M. We can see that the processing time in CPU increases by a factor of 2 when the Trotter size is doubled. As shown in Listing 2, the impact of the Trotter size to the processing time of the FPGA implementation is very small and often negligible. However, we can see that the FPGA processing time increases by a factor of 1.45∼1.87 when the Trotter size doubles. This is due to the significant reduction in the FPGA clock frequency. As explained in Sect. 3.2, FPGA processes M layers in parallel. For this purpose, FPGA has to access the local memory M times in parallel to get the spin data. This needs a local memory of M ports. Since one memory block has only two ports, the OpenCL compiler replicates the local memory to increase the number of ports [24, 25]. As a result, the RAM block usage increases and the clock frequency decreases. Despite the reduction in the clock frequency, the speed-up increases with the Trotter size. This is because the Trotter size does not affect the processing time of the FPGA. This is a very important observation, since large Trotter sizes are important to increase the accuracy of the simulation. We can implement larger Trotter sizes on FPGAs with a relatively smaller increase in the processing time compared to that of the CPU implementation. As explained in Sect. 3.3, we can also increase the Trotter size by adding more FPGAs.

Table 4 shows the FPGA resource usage for different Trotter sizes while keeping the node size a constant. We can see that the memory size and the number of RAM blocks are increased by nearly two times when the Trotter size is doubled. This is due to the RAM block replication to increase the number of ports. The number of DSPs is also increased by two times since the number of parallel computations is doubled. The other resources are also increased significantly. As shown in Table 3, increasing the Trotter size also increases the speed-up compared to CPU implementation.

Figure 8 shows the processing time comparison using one FPGA and two FPGAs. In this evaluation, Trotter size is 32 for all implementations. In one-FPGA implementation, a single FPGA computes all 32 Trotter slices. In two-FPGA implementation, each FPGA computes 16 Trotter slices in a pipelined manner as explained in Sect. 3.3. Since both implementations process all 32

Table 3 Processing time versus Trotter size (500 iterations)

Node size	Trotter size	Processing time (s)		Speed-up (times)	FPGA clock frequency (MHz)
		CPU	FPGA		
32×32	8	18.05	2.29	7.85	250
	16	35.91	3.47	10.35	169
	32	71.20	5.58	12.76	118
64×64	8	285.83	33.49	8.53	253
	16	574.12	62.87	9.13	164
	32	1151.89	91.43	12.59	120

Trotter slices in parallel (pipelined manner), the processing time must be nearly equal. However, two-FPGA implementation is over 1.5 times faster than one-FPGA implementation. This is because of the higher clock frequency in the two-FPGA implementation due to smaller Trotter size per FPGA.

Figure 9 shows the processing time comparison between CPU and two-FPGA implementations using larger Trotter sizes. In this case, the Trotter size is 64. Note that it is not possible to implement such a large Trotter size using a single FPGA. Two-FPGA implementation gives a large speed-up compared to the CPU implementation. We have achieved 23.8 and 22.8 times speed-ups for large node sizes such as 32×32 and 64×64 nodes, respectively.

In Fig. 10, we evaluated the processing time of multicore CPU and proposed FPGA implementations using "number partitioning" problem [26]. In this problem, a series of numbers are divided into two sets. The solution is optimal when the summations of the numbers in two sets are similar as possible. We consider a problem size of 4096 numbers and evaluate the processing time to find the optimal solution with a probability of over 99%. Evaluation is done for 100 samples. This evaluation method is the same as the one used in [5]. The multicore CPU implementation is done using OpenMP directives with gcc compiler (–O3 optimization). According to the results, we can see that the multicore implementation does not scale with the number of cores. Even though the CPU contains 12 physical cores, the processing time is unchanged from 6 to 12 cores. The maximum speed-up using multicores is only 3.3 times. The reasons for this could be the data dependency, memory access bottlenecks and thread synchronization overhead. Since QA simulation is severely data dependent, it is very difficult to utilize data parallel (or SIMD) operations using multicore CPUs. We can also see that both one-FPGA and two-FPGA implementations are faster than all multicore CPU implementations.

The evaluation on the performance of the accelerator can be summarized as follows. According to the results in Table 3, a larger speed-up can be achieved for larger Trotter sizes using FPGAs. However, this also reduces the clock frequency. The resource usage also increases significantly with the Trotter size as shown in Table 4. It is difficult to use a single FPGA for larger Trotter sizes due to resource constraint. However, we can solve these problems using multiple FPGAs. As shown in Fig. 8,

Table 4 FPGA resource usage versus Trotter size (node size = 32×32, 500 iteration)

Node size	Trotter size	Resource usage				
		Logic (%)	Registers	Memory	RAM blocks (%)	DSPs (%)
32×32	8	80,721 (19%)	138,772	0.8 (13%)	593 (22%)	48 (3%)
	16	130,485 (31%)	183,782	1.2 (18%)	984 (36%)	96 (6%)
	32	238,468 (56%)	266,394	2.6 (39%)	1930 (71%)	192 (13%)
64×64	8	81,095 (19%)	138,961	0.9 (14%)	638 (24%)	48 (3%)
	16	133,584 (31%)	185,403	1.5 (22%)	1096 (40%)	96 (6%)
	32	247,987 (58%)	268,896	3.1 (46%)	2147 (79%)	192 (13%)

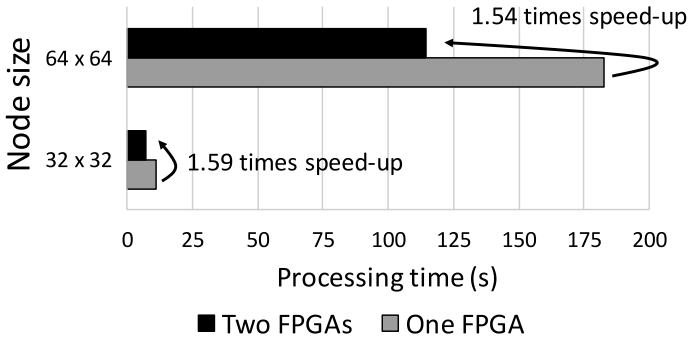


Fig. 8 Processing time comparison using one FPGA and two FPGAs (Trotter size = 32). Evaluation is done for 1000 iterations

two-FPGA implementation gives a smaller processing time compared to one-FPGA implementation for the same node size and the same Trotter size. This is because we implement only half of the Trotter slices per FPGA in the two-FPGA implementation, and that leads to larger clock frequency and speed-up. Moreover, we can implement larger Trotter sizes and larger node sizes using more than two FPGAs.

Table 5 shows the power consumption of the CPU-based system, the CPU-FPGA-based systems and also the FPGA boards. The power consumption of the FPGA boards is measured using the MMD library functions of the Nallatech BSP 17.1. It monitors the current and the voltage of the FPGA board in real time. The power consumption is measured for the whole board including the FPGA, two memory modules, two QSFP ports and all other components on the FPGA board. The power consumption of the systems that contain a CPU-only and a CPU with FPGAs is measured using "Hioki 3334 AC/DC power HiTESTER." We measured the power of the whole system while executing QA simulations. The power consumption of two-FPGA boards is more than twice of that of one-FPGA board. Since FPGAs use high-speed QSFP ports for inter-FPGA data transfers, the power of those ports is also added to the total power consumption. As a result, the performance per Watt for two-FPGA implementation is low compared to that of the one-FPGA implementation. In FPGA implementations, we also use CPU to compute `Jtrans`, to set the kernel arguments and to execute the kernels. The amount of computations done in CPU is negligible compared to those in the FPGAs. However, more than half of the power is consumed by the CPU. We observed that the CPU-based system consumes 41 W of power even at the idle state, and this contributes to the high power consumption. We can expect much smaller power consumption with almost the same processing speed by using more power-efficient CPU in the system.

Table 6 shows the energy consumption of different implementations. Energy is calculated by multiplying power consumption with the processing time. Compared to the CPU implementation, the energy consumption is over 10 times smaller for the system with 1 FPGA and over 9 times smaller for the system with 2 FPGAs. The energy consumption is greatly reduced due to the reduction in the processing time.

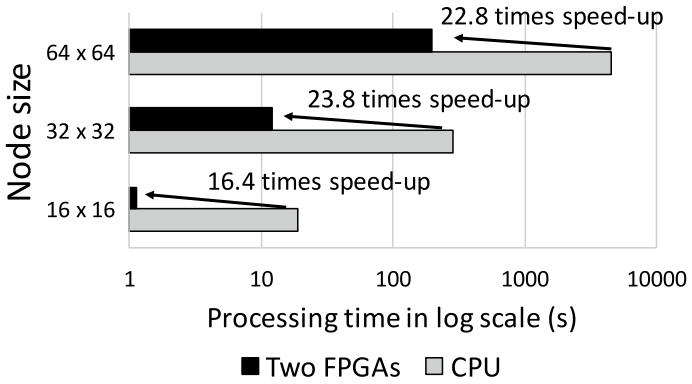


Fig. 9 Processing time for large Trotter size (Trotter size = 64) using a CPU and 2 FPGAs. Evaluation is done for 1000 iterations

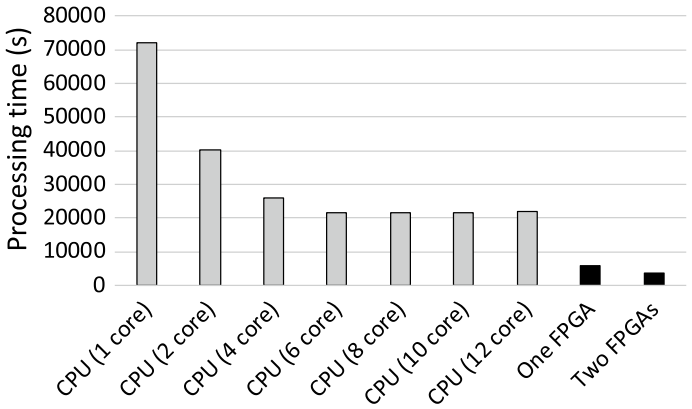


Fig. 10 Processing time to solve the number partitioning problem. The probability of finding the optimal solution is over 99%. The problem contains 4096 nodes, and the processing time is measured for 100 samples

Table 5 Power consumption (power consumption of the FPGA boards only is shown in the brackets)

Totter size	Node size	Power consumption (W)		
		1 CPU	1 CPU + 1 FPGA (1 FPGA only)	1 CPU + 2 FPGAs (2 FPGAs only)
32	32×32	63.9	80.2 (31.27)	134.1 (68.68)
32	64×64	64.1	80.2 (31.44)	141.6 (69.24)
32	128×128	64.1	–	142.8 (70.35)

Table 6 Energy consumption (energy consumption of the FPGA boards only is shown in the brackets)

Totter size	Node size	Energy consumption ($kW \times s$)		
		1 CPU	1 CPU + 1 FPGA (1 FPGA only)	1 CPU + 2 FPGAs (2 FPGAs only)
32	32×32	9.10	0.89 (0.35)	0.97 (0.49)
32	64×64	147.67	14.7 (5.75)	16.21 (7.93)
32	128×128	2337.56	–	246.09 (121.23)

In FPGA-based systems, more than half of the energy is consumed by the CPU, although it performs a very small amount of computations.

The problems that can be mapped to QA simulators or quantum annealers are limited by the number of nodes and the connections between nodes (variables required by the problem). Sparse graphs such as the Chimera graph used in D-wave [16] and the king graph used in [10] have a small number of connections compared to the complete graphs used in ours and [17]. Therefore, minor embedding techniques [27] are used in [10, 16] to map non-native problems. For example, the research on graph partitioning in [28] shows that the largest problem that is fully mappable to the Chimera graph in D-Wave 2X has approximately 45 vertices. Similarly, the research on traffic flow optimization in [29] shows that a problem with 418 cars (assigning 3 routes per car) cannot be mapped to the Chimera graph without partitioning into several sub-problems. Although only 1254 nodes are required in this problem, nodes are highly connected and requires a lot of connections. Since minor embedding increases the number of nodes, It is very difficult to map problems with many connections to sparse graphs without partitioning into several sub-problems [30]. Therefore, the processing times for large problems depend on several factors such as how the problem is formulated (that is, how many variables are required), how the problem is partitioned, how the results are merged, how the results are verified, how those methods are implemented in parallel using conventional computers, etc. To do a fair comparison, we have to evaluate all of those, which is a very difficult task. However, we recognize this as a very important research that should be done in the future works.

It is interesting to evaluate how much processing time is required to solve a well-known problem in the proposed accelerator. The work in [31] discusses a traveling salesman problem (TSP) for 1002 cities using QMC simulation. The number of iterations is nearly 500,000, and the number of Trotter slices is 30. We have designed an accelerator with 32 Trotter slices and 64×64 nodes, which we can use to simulate a TSP of 64 cities. As shown in Table 1, this accelerator executes 500 iterations in 91.43 seconds. Therefore, the processing time for 500,000 iterations is 25.4 hours. If we use two FPGAs, we can reduce the processing time to 15.9 hours. We believe this processing time is acceptable for a relatively large problem such as a TSP of 64 cities, which is in fact larger than the real-world traffic flow estimation problem in [29]. In order to solve more complex real-world problems, the performance of both QA simulators such as ours and also the quantum annealers such as D-wave must be improved

dramatically. Not only the performance of the devices, but also the performance of the problem formulation methods, partitioning and merging algorithms must be improved.

5 Conclusion

We proposed an FPGA accelerator architecture for quantum annealing simulation. We use a complete graph for the implementation, and the parallel computation is achieved by processing multiple nodes in multiple Trotter slices simultaneously. We have achieved over 23 times speed-up compared to single-core CPU implementation and 6 times speed-up compared to multicore CPU implementation. It is possible to increase the processing speed, the number of nodes or the number of Trotter slices by using more FPGAs. The proposed accelerator system is over 9 times energy efficient compared to a CPU-based system. We can further increase the energy efficiency by using a low power CPU in the system. The performance per Watt is decreased slightly for multiple FPGAs due to the high-speed inter-FPGA data transfers.

We can reduce the inter-FPGA data transfers in future works by buffering the intermediate results and combining multiple data transfers into one. Although a QSFP port can transfer 256-bits per step, we use it to transfer only one bit per step. We can reduce the data transfer overhead by 256 times if we buffer 256-bits of spin data. However, this will increase the latency. The CPU overhead can be removed by executing the time loop also in FPGAs. However, using more loops will increase the FPGA resources, so that we have to use more FPGAs. Overall, if we use more FPGAs, there is a huge potential to reduce the overhead and to achieve a huge speed-up for a large number of nodes.

References

1. Schrijver A (2003) Combinatorial optimization: polyhedra and efficiency, vol 24. Springer, Berlin
2. Lawler EL, Lenstra JK, Rinnooy Kan AHG, Shmoys DB (1985) The traveling salesman problem: a guided tour of combinatorial optimization. Wiley, Hoboken
3. Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. Freeman, San Fransisco
4. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680
5. Denchev VS, Boixo S, Isakov SV, Ding N, Babbush R, Smelyanskiy V, Martinis J, Neven H (2016) What is the computational value of finite-range tunneling? Phys Rev X 6(3):031015
6. Kadowaki T, Nishimori H (1998) Quantum annealing in the transverse Ising model. Phys Rev E 58(5):5355
7. Kadowaki T (1998) Study of optimization problems by quantum annealing. PhD thesis, Department of Physics, Tokyo Institute of Technology
8. Lanting T, Przybysz AJ, Smirnov AY, Spedalieri FM, Amin MH, Berkley AJ, Harris R, Altomare F, Boixo S, Bunyk P, Dickson N (2014) Entanglement in a quantum annealing processor. Phys Rev X 4(2):021041

9. Yamaoka M, Yoshimura C, Hayashi M, Okuyama T, Aoki H, Mizuno H (2016) A 20k-spin Ising chip to solve combinatorial optimization problems with CMOS annealing. IEEE J Solid-State Circuits 51(1):303–309
10. Okuyama T, Masato H, Yamaoka M (2017) An Ising computer based on simulated quantum annealing by path integral Monte Carlo method. In: IEEE International Conference on in Rebooting Computing (ICRC), pp 1–6
11. Yang C, Sheng J, Patel R, Sanaullah A, Sachdeva V, Herbordt MC (2017) OpenCL for HPC with FPGAs: case study in molecular electrostatics. In: High Performance Extreme Computing Conference (HPEC), pp 1–8
12. Reconfigurable computing in exascale, SC17. https://sc17.supercomputing.org/SC17%20Archive/bof/bof_pages/bof148.html. Accessed 2019
13. Czajkowski TS, Neto D, Kinsner M, Aydonat U, Wong J, Denisenko D, Yiannacouras P, Freeman J, Singh DP, Brown SD (2012) OpenCL for FPGAs: prototyping a compiler. In: Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA), No. 1
14. Intel FPGA SDK for OpenCL (2016) https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html. Accessed 2018
15. Intel FPGA SDK for OpenCL programming guide. https://www.altera.com/documentation/mwh1391807965224.html. Accessed 2018
16. D-wave systems. https://www.dwavesys.com/home. Accessed 2018
17. Quantum computing and AI start a new era. Fujitsu Journal, 2018. http://journal.jp.fujitsu.com/en/2017/12/13/01/. Accessed 2018
18. Takesue H, Inagaki T, Inaba K, Honjo T (2017) Quantum neural network for solving complex combinatorial optimization problems. NTT Tech. Rev. 15(7):1–5
19. The D-wave 2X quantum computer technology overview. https://www.dwavesys.com/sites/default/files/D-Wave%202X%20Tech%20Collateral_0915F.pdf. Accessed 2018
20. The D-wave 2000Q quantum computer technology overview. https://www.dwavesys.com/sites/default/files/D-Wave%202000Q%20Tech%20Collateral_0117F.pdf. Accessed 2018
21. Weigel M (2012) Performance potential for simulating spin models on GPU. J Comput Phys 231(8):3064–3082
22. Chase C, Zhao H, Sato T, Hiromoto M, Tan SXD (2018) GPU based parallel Ising computing for combinatorial optimization problems in VLSI physical design, pp 1–8. arXiv preprint arXiv:1807.10750
23. Nallatech 385A accelerator card. http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-385a-arria10-1150-fpga/. Accessed 2018
24. Intel FPGA SDK for OpenCL best practices guide. https://www.altera.com/documentation/mwh1391807516407.html. Accessed 2018
25. Waidyasooriya HM, Hariyama M, Uchiyama K (2018) Design of FPGA-based computing systems with OpenCL. Springer, Berlin
26. Lucas A (2014) Ising formulations of many NP problems. Front Phys 2:5
27. Zaribafiyan A, Marchand DJJ, Rezaei SSC (2017) Systematic and deterministic graph minor embedding for Cartesian products of graphs. Quantum Inf Process 16:136
28. Ushijima-Mwesigwa H, Negre CFA, Mniszewski SM (2017) Graph partitioning using quantum annealing on the D-wave system. In: Proceedings of the Second International Workshop on Post Moores Era Supercomputing, pp 22–29
29. Neukart F, Von Dollen D, Compostella G, Seidel C, Yarkoni S, Parney B (2017) Traffic flow optimization using a quantum annealer. Front ICT 4:29
30. Booth M, Reinhardt SP, Roy A (2017) Partitioning optimization problems for hybrid classical/quantum execution. D-wave technical report series
31. Martonak R, Santoro GE, Tosatti E (2004) Quantum annealing of the traveling-salesman problem. Phys Rev E 70(5):057701

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