IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 28, 2020, accepted March 29, 2020, date of publication April 6, 2020, date of current version April 22, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2985699

A GPU-Based Quantum Annealing Simulator for Fully-Connected Ising Models Utilizing Spatial and Temporal Parallelism

HASITHA MUTHUMALA WAIDYASOORIYA AND MASANORI HARIYAMA, (Member, IEEE)
Graduate School of Information Sciences, Tohoku University, Sendai 980-8579, Japan

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

This work was supported in part by the MEXT KAKENHI, under Grant 19K11998.

ABSTRACT Simulated quantum annealing (SQA) is a probabilistic approximation method to find a solution for a combinatorial optimization problem using digital computers. The processing time of SQA increases exponentially with the number of variables. Therefore, acceleration of SQA is regarded as a very important topic. However, parallel implementation is difficult due to the serial nature of the quantum Monte Carlo algorithm used in SQA. In this paper, we propose a method to implement SQA in parallel on a GPU while preserving the data dependency. According to the experimental results, we have achieved over 97 times speed-up while maintaining the same accuracy-level compared to a single-core CPU implementation.

INDEX TERMS Simulated quantum annealing, optimization problems, high performance computing, GPU acceleration.

I. INTRODUCTION

Quantum annealing (QA) [1], [2] is a probabilistic approximation method to find the global optimum of a "combinatorial optimization problem" [3]. Solving combinatorial optimization problems is important in real world applications such as traffic-flow simulation [4], financial analysis [5], [6], graph problems [7], [8], etc. A quantum annealer such as D-wave [9] uses quantum properties to find solutions. However, the number of bits available in a quantum annealer is too small to solve many real world problems. As a result, simulated quantum annealing (SQA) on digital computers is important.

Quantum annealing can be simulated on a digital computer using quantum Monte Carlo algorithm [10]. However, the processing time of SQA increases exponentially with the number of variables. As a result, extremely large processing time is required to solve real world problems. ASICs (application specific integrated circuits) [11], [12] and FPGAs (feild programmable gate arrays) [13]–[17] are already employed to accelerate SQA. Using a highly-parallel processor such as a GPU that has thousands of cores is a promising way to accelerate SQA. Currently, GPUs are commonly available and already included in many supercomputers [18], [19] and computing clusters [20], [21]. However, it is very difficult to accelerate SQA for fully-connected Ising models due to severe data dependency. Existing GPU accelerators such as [22] are proposed for sparse Ising models with small data dependency. GPU accelerators such as [23] completely ignore the data dependency so that the quality of the solutions of fully-connected Ising models is low.

This paper proposes a highly-parallel GPU-based SQA accelerator for fully-connected Ising models. We schedule mutually independent operations in parallel, while the rest of the operations serially. Parallelism is achieved within the computation of a single spin, and also within the computations among multiple spins belonging to different Trotters. Note that a Trotter is a replica of spins, and using more Trotters often increases the quality of the solution. We propose a method to execute spins belonging to multiple Trotters in parallel using "concurrent kernel execution". According to the evaluation, the proposed accelerator provides over 97 times speed-up with a similar accuracy-level, compared to single-core CPU implementation. We also compare the performance, power and power-efficiency against recent FPGA-based accelerators and discus the advantages and disadvantages of each method.

The associate editor coordinating the review of this manuscript and approving it for publication was Muhamamd Aleem.

II. PREVIOUS WORKS

A. SIMULATED QUANTUM ANNEALING (SQA)

Ising model is a mathematical model of a system consisting of spins and their interactions. The energy or the Hamiltonian

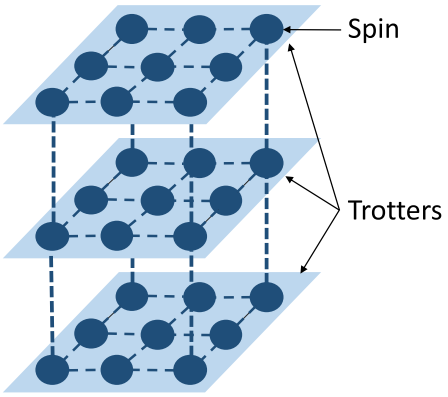


FIGURE 1. Transverse-field Ising model.

of an Ising model is expressed by Eq.(1).

$$H(\sigma) = -\sum_{ij} J_{i,j}\sigma_i\sigma_j - \sum_{i} h_i\sigma_i \tag{1}$$

A spin is denoted by σ, the interaction coefficient between spin i and j is denoted by $J_{i,j}$ and the local magnetic field is denoted by h_i. A spin has either the value -1 (spin-down) or the value $+1$ (spin-up). Transverse-field Ising model [1], [24], [25] is used to solve optimization problems. The transverse-field controls the rate of transition between states and plays a similar role that the temperature does in simulated annealing [26]. 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 [27]. It uses multiple replicas called "Trotters" as shown in Fig.1. When there are interactions between every spin-pair in a Trotter, we call it a "fully-connected Ising model".

Quantum Monte Carlo simulation [10] is used on digital computers to simulate quantum annealing. Algorithm 1 shows an extract of the quantum Monte Carlo simulation. It consists of four loops. The outer-most loop is executed for multiple iterations, and each iteration is called a "Monte Carlo steps" (MC step). The transverse-field decreases with each MC step. Having a lot of MC steps implies that the transverse filed is decreasing slowly, and it 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 computation of a spin is called "*one spin flip*". The inner-most loop (in line 5) is executed for all interactions among spins. In each MC step, we start the computation from Trotter 1 and compute the *local-field* energy of each spin sequentially. A spin flips its value in probabilistic manner considering the energy difference. After all the spins of a Trotter are computed, the same process is repeated in the next Trotter.

B. PREVIOUS SQA ACCELERATORS

Many previously proposed accelerators such as [12]–[14], [22], [23] are suitable only for "sparse Ising models" such as king-graphs or near-neighbor connections, where most of the spins do not have interactions with each other. When two spins have no interactions, there is no data dependency. Therefore, previous studies employ parallel computations among such spins. However, we have to use minor embedding [28] to simulate "dense Ising models" or "fully-connected Ising models" on those accelerators. Studies in [4], [29] show that minor embedding reduces the number of usable spins and allows only small problems to be mapped.

Previous studies such as [15]–[17] employs Trotter-level temporal parallelism on FPGAs to accelerate SQA. This method can be used on any Ising model including fully-connected ones. These studies use shift-registers in FPGAs to precisely schedule each computation at clock-cycle-level to preserve the data dependency while employing parallel computation. However, it is not possible to use the same implementation method on GPUs, since we cannot control the timing of the operations at clock-cycle-level.

In contrast to previous GPU accelerators [22], [23] which are suitable for sparse Ising models, this paper proposes a GPU accelerator for fully-connected Ising models. The accelerator uses both temporal and spatial parallelism similar to [17]. However, the implementation is completely different from the shift-register based clock-cycle-level scheduling proposed in [17]. We use a concurrent kernel execution approach with parallel reduction to accelerate SQA.

Algorithm 1 An Extract of the SQA Algorithm

```
1  for t ← 1 to MC_steps do
2      for m ← 1 to M do
3          for i ← 1 to N do
               // one spin flip computation
4              local_field[i] ← 0
               // compute local-field
5              for j ← 1 to N do
6                  local_field[i]+ = spin(m, j) × J(i, j)
7              end
               // compute transverse-field
8              local_field[i] + =
               J_tran × (spin(m + 1, i) − spin(m − 1, i))
9              energy_diff = local_field[i]× ...
               // spin update
10             if exp(−energy_diff /..) > rand_num then
11                 spin[i] = ¬spin[i]
12             end
13         end
14     end
15 end
```

III. GPU ACCELERATION OF SQA

A. SPATIAL PARALLEL PROCESSING FOR LOCAL-FIELD COMPUTATION USING PARALLEL REDUCTION

As shown in Algorithm 1, the computation of the *local-field* is a reduction operation. We can use "parallel reduction" on a GPU to decrease the processing time. Fig.2 shows how

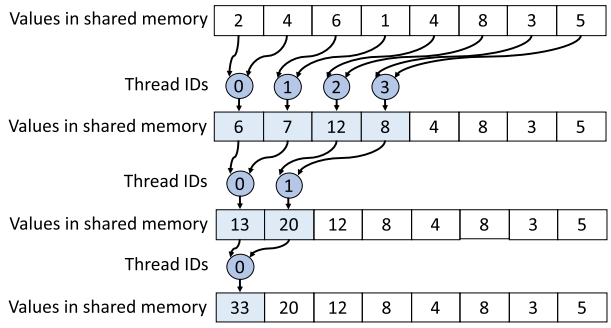


FIGURE 2. Parallel reduction computation in a GPU.

to perform the reduction operation in parallel using multiple threads of a thread-block. First, the required data are copied to the shared memory. Then each thread in the thread-block access two data values and do the computation. The results are written back to the shared memory. After all threads have written the results, a half of the threads access two values again from the shared memory. In each step, the number of threads become half. At the final step, the *thread 0* writes the summation to the shared memory.

We can increase the degree of parallelism by using multiple thread-blocks for reduction operation. When there are multiple blocks, each block computes a partial sum. Each block writes the partial sum to the global memory. After all the partial sums are written, we access those from the global memory and perform another parallel reduction operation. Previous works such as [30], [31] explain the implementation of parallel reduction operation on a GPU.

Algorithm 22 shows how to implement parallel reduction on *local-field* computation. The kernel program computes the *local-field* of $spin(i)$ using multiple threads. This is called the "spatial parallelism". After the *local-field* is obtained, the transverse-field computation and spin update is performed. The transverse-field computation and update is done using a single thread, since there is no parallelism. The host program executes the three outer loops. It executes the kernel program *one_spin_flip* for all the spins in all the Trotters for all MC steps serially.

B. TEMPORAL PARALLEL PROCESSING FOR MULTIPLE-SPIN-FLIPS USING CONCURRENT KERNEL EXECUTION

As shown in Algorithm 1, spins of two neighboring Trotters are required for the computation of *one spin flip* of a Trotter. For example, the computation of *spin k* of Trotter *m* requires *spin k* of Trotter *m−1* and *spin k* of Trotter *m+1*. The *spin k* of Trotter *m−1* is already computed, while the *spin k* of Trotter *m+1* is yet to be computed. As a result, we can compute *spin k+1* of Trotter *m−1* and *spin k* of Trotter *m* in parallel without violating the data dependency. Similarly, we can schedule parallel computations among Trotters as shown in Fig.3. This is called the "temporal parallelism".

Algorithm 2 Spatial Parallel Processing of the Local-Field Computation. The Spin and Trotter Numbers Are i and m Respectively

```
// Processing on CPU (host)
for t ← 1 to MC_steps do
    for m ← 1 to M do
        for i ← 1 to N do
            one_spin_flip(i, m)
        end
    end
end

// Processing on GPU
__kernel: one_spin_flip(i, m)
shared memory spin[...]
shared memory local_field[...]
// copy spin from global to local
__barrier__
// compute local-field in parallel
   using multiple threads
Parallel reduction for j ← 1 to N do
    local_field[i]+ = spin(m, j) × J(i, j)
end
__barrier__
if threadId == 0 then
    // compute transverse filed
    local_field[i] + =
    J_tran × (spin(m + 1, i) − spin(m − 1, i))
    energy_diff = local_field[i]× ···
    // one spin flip
    if exp(−energy_diff /..) > rand_num then
        spin[i] = ¬spin[i]
    end
end
// copy spin from local to global
```

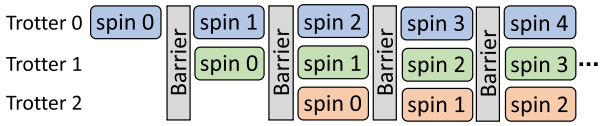


FIGURE 3. Parallel processing of spins belonging to three different Trotters.

We can implement this method on a GPU using "concurrent kernel execution" technique. We use independent "CUDA streams" for the computation of each Trotter as shown in Fig.4. A stream contains a kernel that is corresponding to the computation of a spin in a Trotter. As shown in Algorithm 3, the streams are executed by the host according to the schedule in Fig.3. Then we synchronize all streams and make sure all kernels are executed. After that, we launch another set of streams that contain kernels corresponding to the spins belonging to different Trotters.

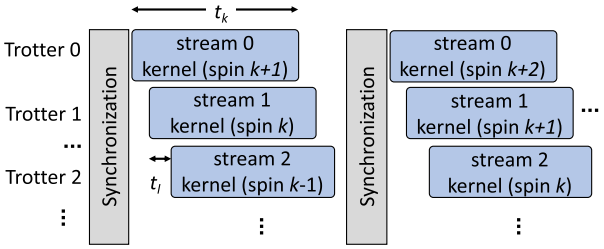


FIGURE 4. Concurrent execution of kernels using multiple CUDA streams.

Algorithm 3 Temporal Parallel Processing of Spin-Flips Belonging to Multiple Trotters

```
// Processing on CPU (host)
for t ← 1 to MC_steps do
    for i ← 1 to N do
        // Concurrent kernel execution
           for all Trotters.
        ...
        stream( one_spin_flip(i + 1, m − 1) )
        stream( one_spin_flip(i, m) )
        stream( one_spin_flip(i − 1, m + 1) )
        ...
        __barrier__
    end
end
```

Each stream is launched one after the other, so that the total launch time increases with the number of streams. Usually, the kernel launch time t_l is negligible compared to the kernel execution time t_k. However, when there are hundreds of streams, t_l can be comparable to t_k. Therefore, in order to execute M streams concurrently, the condition in Eq.(2) must be satisfied.

$$(M-1) \times t_l < t_k \tag{2}$$

For smaller number of spins, t_k can be too small, so that the condition in Eq.(2) is not satisfied. In order to satisfy this condition, we have to increase t_k by doing more computations in a kernel. Therefore, instead of computing one spin, we compute a block of spins serially in each kernel.

Fig.5 shows the concurrent execution of kernels where each kernel computes multiple spins serially. Blocks of spins called "spin-blocks" are shown in Fig.5a. A spin-block contains multiple spins that should be computed serially due to data dependency. However, different spin-blocks belonging to multiple Trotters can be computed in parallel as shown in Fig.5b. The size of a spin-block b can be chosen to satisfy the condition in Eq.(2). If b is too large, most of the spins are processed serially, and that reduces the degree of Trotter-level parallelism. For example, if b equals to the number of spins N, all Trotters are executed serially. Therefore, the condition in Eq.(3) also must be satisfied to compute all M Trotters in parallel.

$$b \leq \frac{N}{M} \tag{3}$$

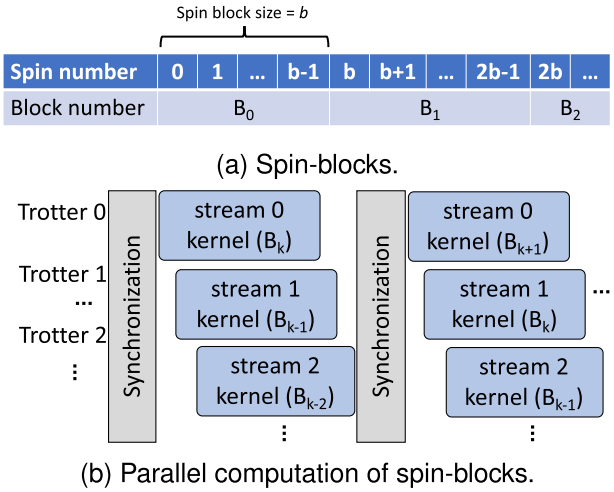


FIGURE 5. Concurrent execution of kernels that compute spin-blocks.

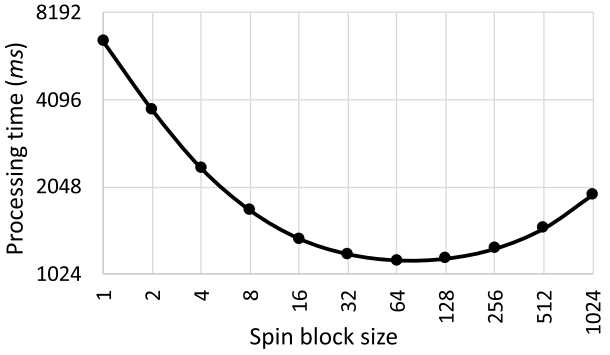


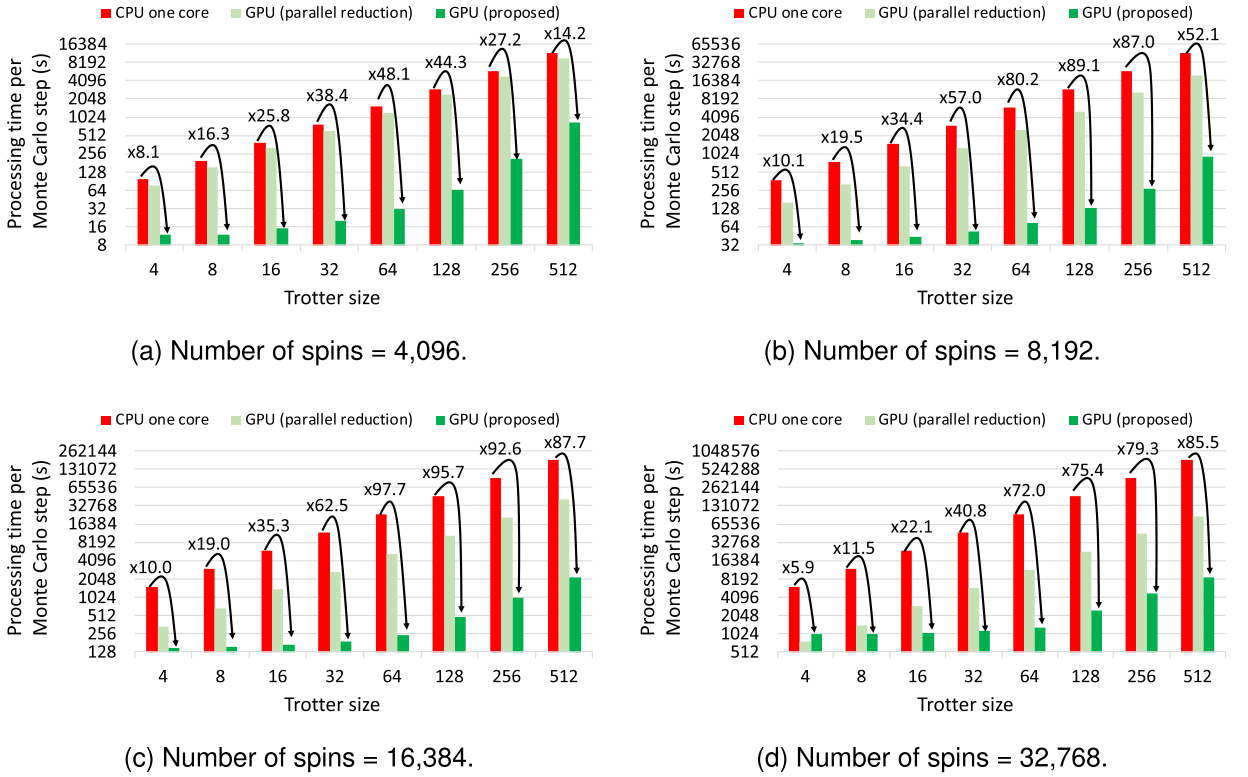
FIGURE 6. Spin-block size vs. processing time. Number of spins is 32,768 and the number of Trotters is 32.

IV. EVALUATION

The evaluation is done using an Nvidia Quadro GV100 GPU. CPU host code and GPU kernels are compiled using gcc 7.4.0 and CUDA 10.1 respectively on CentOS 7.5 operating system.

A. EVALUATION OF THE PROCESSING TIME

Fig.6 shows the relationship of the spin-block size and the processing time. In this example, the number of spins and Trotters are 32,768 and 32 respectively. According to Eq.(3) the maximum spin-block size for parallel computation is 1024. If we know the values of t_k and t_l, we can also calculate the minimum spin-block size for parallel computation using Eq.(2). However, it is difficult to measure t_k and t_l accurately. Therefore, we change the spin-block size from 1 to the maximum value and measure the processing time to find the optimal one. The processing time is minimized when the spin-block size is 64. When the spin-block



(a) Number of spins = 4,096.

(b) Number of spins = 8,192.

(c) Number of spins = 16,384.

(d) Number of spins = 32,768.

FIGURE 7. Trotter size vs. processing time.

size decreases, the number of kernels increases. Therefore, the kernel launch time and the total processing time increase. When the spin-block size increases, the degree of parallelism decreases and the processing time increases. As a result, we can see that the processing time is minimized at a certain spin-block size. In this paper, we evaluate various spin-block sizes and chose the one that provides the minimum processing time.

Fig.7 shows the comparison of processing time against "single-core CPU implementation" and "GPU implementation that use only spatial parallelism". Spatial parallelism is employed as described in section III-A. CPU codes are executed on Intel Xeon Silver 4116 CPU using gcc compiler version 9.2. The optimization options are *-O3* and *-march = native*. Figs.7a, 7b, 7c and 7d shows the processing time per MC step, when the numbers of spins are 4,096, 8,192, 16,384 and 32,768 respectively. We have achieved up to 97.7 times speed-up compared to single-core CPU implementation. We have also achieved up to 38.7 times speed-up compared to the GPU implementations with spatial parallelism. The processing times of CPU implementations and GPU implementations with spatial parallelism increase linearly with the number of Trotters. However, the processing time of the proposed GPU implementation is nearly a constant for relatively smaller Trotters sizes of less than 64. The reason for this is the temporal parallel computation of multiple Trotters. When the number of Trotters is large, there is not enough resources in the GPU to execute all kernels concurrently. Therefore, some of the kernels have to wait until the previous ones are completed. As a result, the processing time increases for larger Trotter sizes.

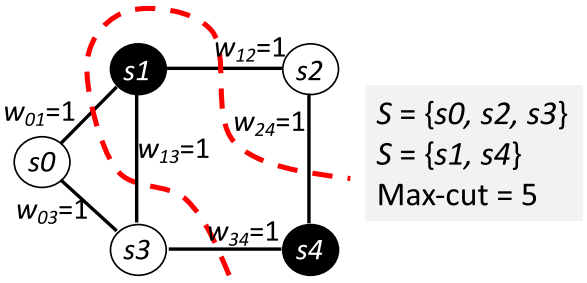


FIGURE 8. An example of a max-cut.

B. ACCURACY EVALUATION USING MAX-CUT PROBLEM

We use "max-cut problem" [32] benchmarks called 'Gset" [33] to evaluate the accuracy of the proposed accelerator. Those benchmarks are online available and many previous studies such as [34]–[37] use those benchmarks. In order to solve a max-cut problem using SQA, we have to map it to the Ising model. That means, we have to formulate the problem in the form of Eq.(1). The max-cut problem is defined as finding a subset S from a set of vertices such that the weighted edges between S and its complementary subset S' are as large as possible. Fig.8 shows an example of the max-cut which equals to 5. The max-cut problem of n vertices

TABLE 1. Comparison of accuracy.

Benchmark	Number of nodes	Number of edges	Best known cut [35]–[38]	Work in [23] 1000 MC steps average	CPU single-core 1000 MC steps best	CPU single-core 1000 MC steps average	Proposed (Qaudro GV100) 1000 MC steps best	Proposed (Qaudro GV100) 1000 MC steps average
G9	800	19,176	2,054	2,004 (97.6%)	2,037	2,006 (97.7%)	2,037	2,005 (97.6%)
G13	800	1,600	**582**	522 (89.7%)	**582**	580 (99.7%)	**582**	580 (99.7%)
G18	800	4,694	992	938 (94.6%)	988	974 (98.2%)	988	974 (98.2%)
G19	800	4,661	906	844 (93.2%)	903	889 (98.1%)	903	889 (98.1%)
G20	800	4,672	**941**	880 (93.5%)	**941**	926 (98.4%)	**941**	926 (98.4%)
G21	800	4,667	931	880 (94.5%)	929	913 (98.1%)	929	913 (98.1%)
G31	2,000	19,990	3,309	3,227 (97.5%)	3,278	3,241 (97.9%)	3,278	3,241 (97.9%)
G34	2,000	4,000	**1,384**	1,191 (86.1%)	**1,384**	1,378 (99.6%)	**1,384**	1,379 (99.6%)
G39	2,000	11,778	2,408	2,269 (94.2%)	2,379	2,345 (97.4%)	2,379	2,345 (97.4%)
G40	2,000	11,766	2,400	2,267 (94.5%)	2,379	2,338 (97.4%)	2,379	2,338 (97.4%)
G41	2,000	11,785	2,405	2,284 (95.0%)	2,394	2,337 (97.2%)	2,394	2,337 (97.2%)
G42	2,000	11,779	2,481	2,325 (93.7%)	2,452	2,410 (97.1%)	2,452	2,409 (97.1%)
G47	1,000	9,990	6,657	6,619 (99.4%)	6,656	6,628 (99.6%)	6,656	6,628 (99.6%)
G50	3,000	6,000	**5,880**	5,803 (98.7%)	**5,880**	5,876 (99.9%)	**5,880**	5,876 (99.9%)
G51	1,000	5,909	3,848	3,754 (97.6%)	3,845	3,828 (99.5%)	3,845	3,828 (99.5%)
G53	1,000	5,914	3,850	3,756 (97.6%)	3,842	3,830 (99.5%)	3,842	3,830 (99.5%)
G54	1,000	5,916	3,852	3,756 (97.5%)	3,848	3,827 (99.4%)	3,848	3,827 (99.4%)

is given by Eq.(4). The weight of the edge between vertices i and j is denoted by w_{ij}. Since $w_{ij}/2$ is a constant, we can write the Hamiltonian using Eq.(5), which is the Ising formulation of the max-cut problem. Using Eq.(5), we can determine the interaction coefficients among spins.

$$\begin{aligned} \textit{Maximize}: \quad \textit{Cut size} &= \sum_{1\le i<j\le n} w_{ij}\frac{(1-\sigma_i\sigma_j)}{2} \\ \textit{where}: \sigma_i, \sigma_j &= \{-1, 1\} \quad (4) \end{aligned}$$

$$H(\sigma) = -\sum_{ij}\frac{w_{ij}}{2}\sigma_i\sigma_j \quad (5)$$

Table 1 shows the accuracy comparison of the proposed accelerator using "Gset" [33] benchmarks. The "best known solutions" are obtained by investigating several previous studies such as [34]–[37]. In this evaluation, we use the same random numbers and the same initial spin values for both CPU and GPU implementations in each sample. Each benchmark is simulated for 100 samples, and different initial data are used for different samples. The number of Trotters is 16 for all implementations. The results of all benchmarks obtained after 1000 MC steps of the GPU implementation are similar to those of the single-core CPU implementation. This shows that there is no accuracy degradation due to parallel GPU implementation compared to serial CPU implementation. We have obtained the best known cut for "G13, G20, G34" and "G50" benchmarks, while over 99% accuracy for the rest. The average value of the max-cut is more than 97% of the best known solution for all benchmarks. We obtained better solutions for all benchmarks, compared to the most recent study of GPU acceleration in [23] that uses Tesla K40c. Note that the accuracy is compared with [23] under the same conditions of 1000 MC steps while taking the average max-cut value. We obtained such a high accuracy by preserving the data dependency of the computation, while implementing parallel operations.

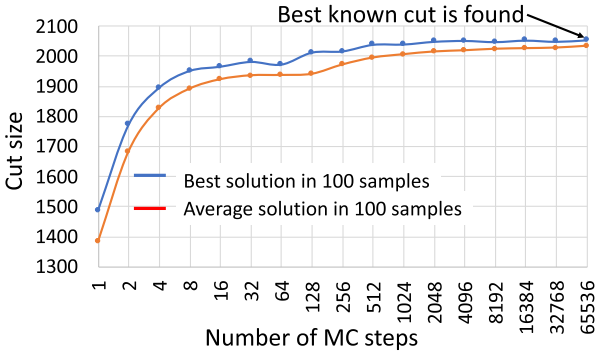


FIGURE 9. Number of MC steps vs. max-cut for G9 benchmark.

Fig.9 shows the relationship between the number of MC steps and the max-cut for "G9" benchmark. The max-cut gets larger when the number of MC steps are increased. The best known solution for "G9" is found when the simulation is executed for 65,536 MC steps. This shows that the solution is not converged to a local minimum and a better solution can be found by increasing the number of MC steps.

C. COMPARISON WITH MULTICORE CPU AND FPGA IMPLEMENTATIONS

In this section, we compare the proposed method against highly-optimized and recently published SQA accelerator in [17] and also with multicore CPU implementation that uses

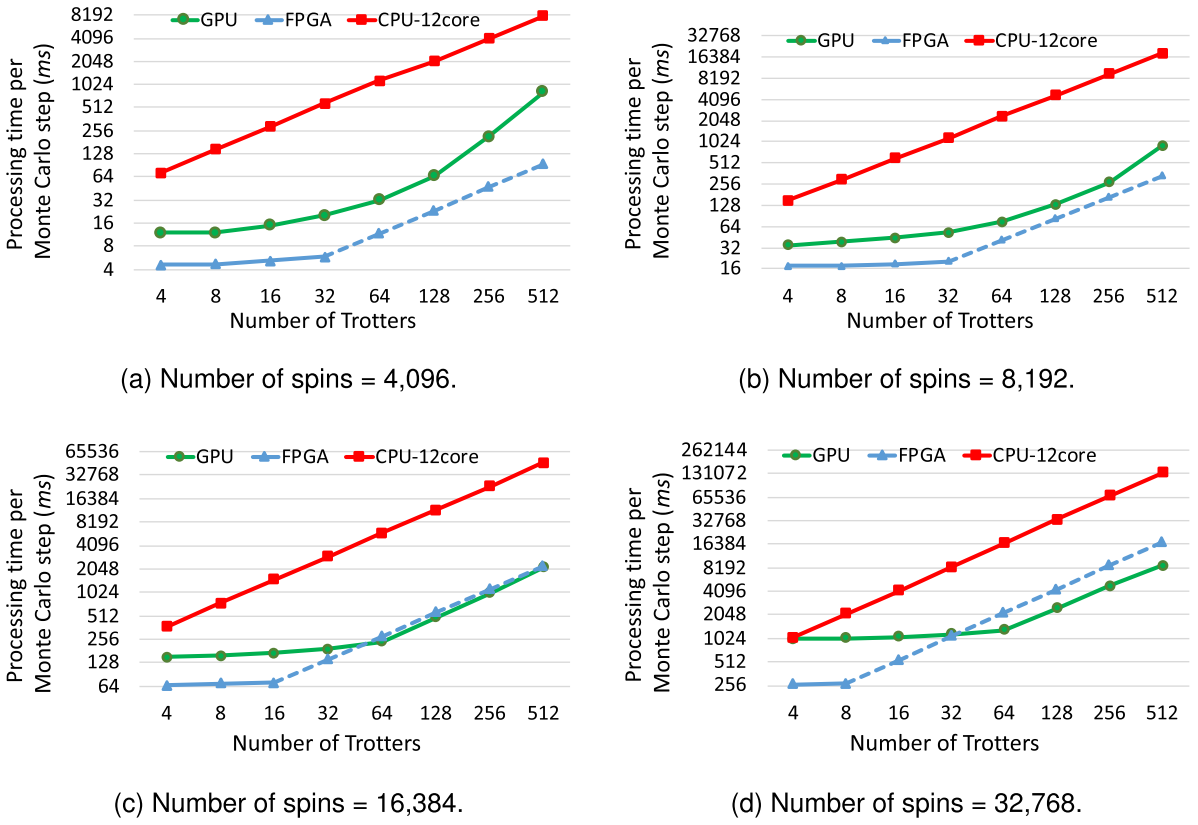


FIGURE 10. Comparison of the processing time of GPU, 12-CPU and FPGA implementations. Dashed-lines show the approximated processing time of FPGA implementation, that is calculated under the assumption of processing multiple Trotter-blocks serially and the Trotters in a block in parallel.

spatial parallelism. The accelerator [17] uses both spatial and temporal parallelism. It is implemented on a "Nallatech 385A accelerator board" [38] that contain Intel Arria 10 FPGA. In this accelerator, all Trotters are processed in parallel. The maximum number of Trotters that can be processed in parallel is less than or equals to 32 due to the FPGA resource constraint. Therefore, we can process a small block of Trotters in parallel, and multiple blocks serially. The multicore CPU implementation is also described in [17]. It computes the inner-most loop of algorithm 1 in parallel using *parallel reduction* pragma in OpenMP [39]. Multicore CPU implementations are done on Intel Xeon Silver 4116 CPU using all 12 cores. Compiler version is gcc 9.2, and the optimization options are $-O3$, *-march = native* and *-fopenmp*.

Fig.10 shows the processing time comparison against FPGA [17] and multicore CPU accelerators. Note that the dashed-lines show the approximated processing time of FPGA accelerators, that is calculated under the assumption of processing multiple Trotter-blocks serially and the Trotters in a block in parallel. Both GPU and FPGA accelerators are faster compared to the multicore CPU accelerators. On the one hand, the GPU accelerator is slower than the FPGA accelerator for smaller number of spins, as shown in Figs.10a and 10b. The reason for this is the synchronization overhead of multiple streams in GPU that occupies a relatively large portion of the total processing time. One the other hand, the GPU accelerator is faster than or equals to the FPGA accelerator for larger number of spins and Trotters, as shown in Figs.10c and 10d. When the number of spins and Trotters increases, FPGA accelerators require a large amount of internal memory resources to store the interaction coefficient data that are represented in single-precision floating-point. As a result, the degree of parallelism in FPGA decreases with the number of spins and the number of Trotters. The GPU accelerators store only the spin data in the internal memory those have a value either "−1" or "+1". Therefore, the required internal memory (shared memory) capacity is much smaller compared to that of FPGA accelerators. As a result, GPU accelerators have a relatively large degree of parallelism for larger number of spins and Trotters compared to that of the FPGA accelerators. Note that, when the number of spins increases over 32,768, the internal memory of GPU is not enough to store all the spin values. Similarly, FPGA also does not contain enough resources to process more than 32,768 spins.

Fig.11 shows the comparison of the power consumption of the GPU and FPGA accelerators. The power consumption of the FPGA is measured using MMD (memory-mapped device) library API (application programming interface) of the Nallatech BSP (board support package) 17.1 [38]. It reads

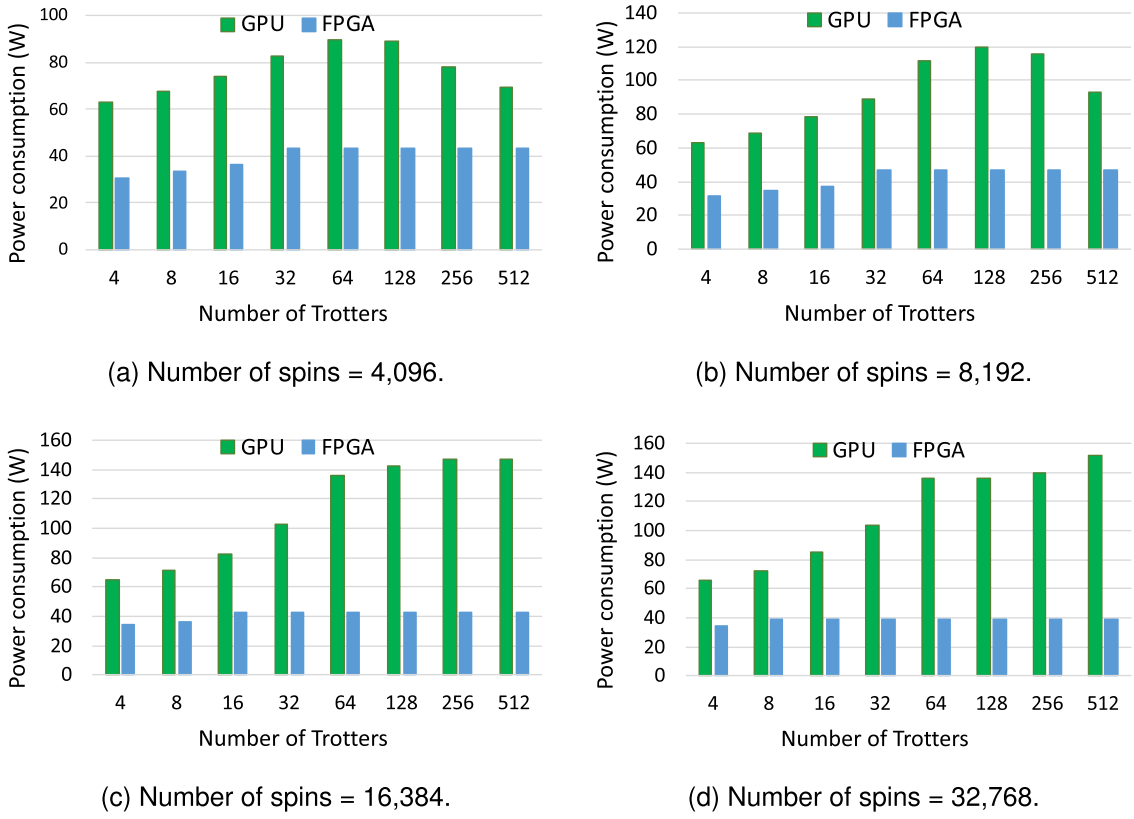


FIGURE 11. Comparison of the power consumption of GPU and FPGA implementations.

the current and voltage sensor data of the FPGA board in real time and provides the power consumption data. The power consumption of GPU is measured using NVML (NVIDIA management library) API [40] that also reads the sensor data of the GPU board and provides the power consumption data. Since the power consumption changes in each measurement, we plot all power consumption data and use the peak value. The power consumption varies from 60W to 150W, and from 30W to 50W, in GPU and FPGA respectively. For the same number of spins and Trotters, the power consumption of the GPU is 1.6 to 3.9 times larger compared to that of the FPGA.

We can see that the power consumption increases initially with the number of Trotters in Fig.11. This is due to the increase in parallel operations. However, we can also see that the power consumption is reduced after 128 Trotters in Figs.11a and 11b. We assume the following reason for this reduction. As shown in Fig.2, the total processing time composed of the kernel execution time in GPU and also the stream launch and synchronization times in CPU. When the number of spins are smaller, the kernel execution time in GPU is also smaller. In order to execute multiple steams concurrently, the condition in Eq.(2) must be satisfied. The possibility of satisfying this condition decreases with the number of streams. Therefore, when the host launches new CUDA streams, the execution of the previous streams is already completed. As a result, the degree of parallelism is reduced for larger number of Trotters, while decreasing the power consumption. In contrast, the power consumption is the same or increases with the number of Trotters for large number of spins, as shown in Figs.11c and 11d. When the number of spins is large, kernel execution time is sufficiently large for concurrent kernel execution. Therefore the degree of parallelism and the power consumption increase.

Fig.12 shows the comparison of the power-efficiency of the GPU and FPGA accelerators. Power-efficiency is calculated by Eq.6.

$$\text{power-efficiency} = \frac{\text{speed-up}}{\text{power consumption}} \tag{6}$$

The speed-up is calculated compared to the single-core CPU implementation. According to the results, the power-efficiency of the GPU is smaller than that of FPGA. The power-efficiency is peaked near 64 Trotters and then decreases or stays the same. When the number of Trotters increases, both power consumption and performance decrease for smaller number of spins, and power consumption increases for larger number of spins. In either case, the power-efficiency decreases.

In Table 2, we summarize the advantages and disadvantages of the proposed GPU accelerator compared to previous studies using FPGAs. Both the proposed GPU accelerator and the previous FPGA accelerator [17] handle quantum annealing simulations up to 32,768 spins, with similar accuracy-level. Both accelerators provide a large speed-up compared to

TABLE 2. Comparison of advantages and disadvantages of GPU and FPGA accelerators.

Category	FPGA accelerator in [17]	Proposed GPU accelerator
Speed-up	Very large compared to single-core CPU implementation	
Accuracy	Similar to single-core CPU implementation	
Problem size	Up to 32,768 spins	
Power consumption	Small	Relatively large
Power-efficiency	Large	Relatively small
Availability	Not common	Commonly available from PCs to supercomputers
Programmability	Require hardware design skills	Relatively easy using CUDA, OpenCL, OpenAcc, etc.
Compilation time	Several hours	Less than a minute
Design time	Large	Relatively small

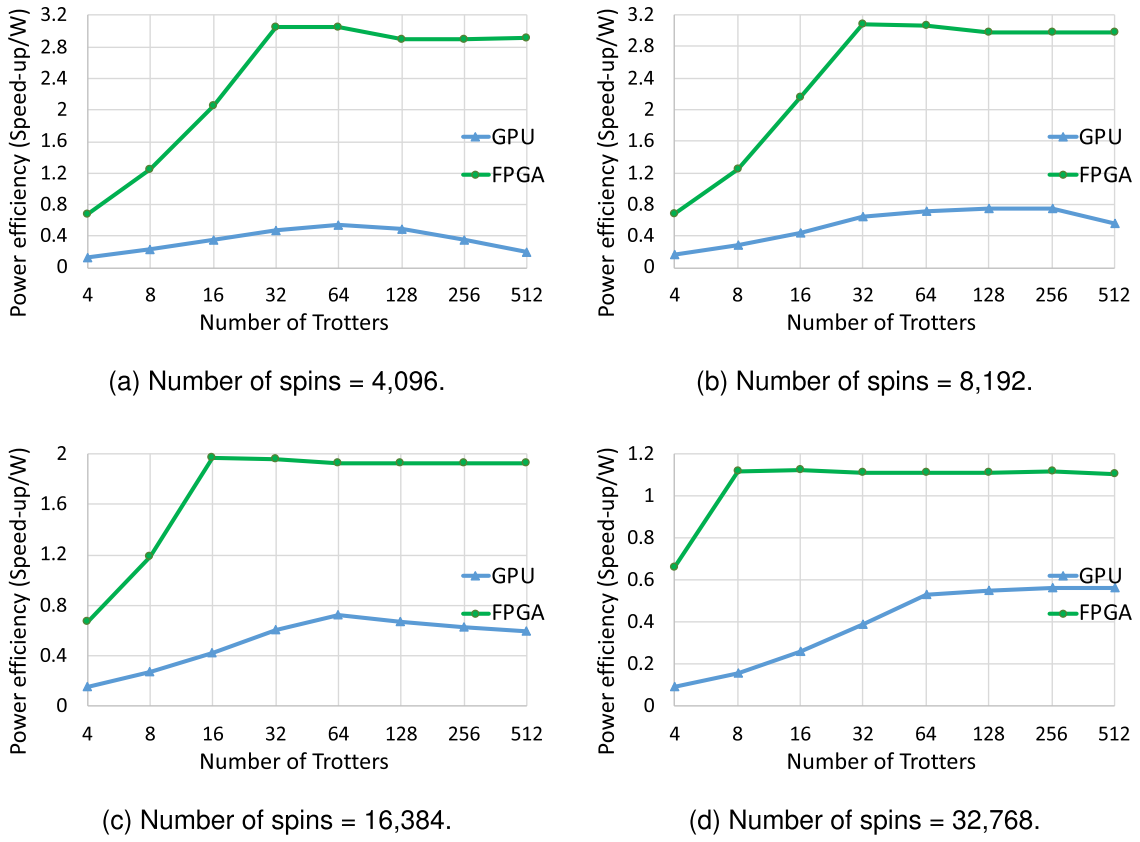


FIGURE 12. Comparison of the power-efficiency of GPU and FPGA implementations.

single-core CPU implementation. FPGA accelerator in [17] is more power-efficient and consumes less power compared to the proposed one. In addition, for smaller Ising models with less than ten thousand spins, FPGA implementation is faster. However, for bigger Ising models with over ten thousand spins and 64 Trotters, the performance of the proposed GPU accelerator is similar or even better compared to that of the FPGA accelerator. In addition, GPUs are commonly available for many different price-points and easily affordable compared to FPGAs. Many supercomputers are already equipped with GPUs, including "Summit" [18] and "Siera" [19], the world's number 1 and 2 from top 500 supercomputers list of 2019 [41]. There are other high-performance GPU clusters such as [20], [21]. Most users are already familiar with different GPU programming methods such as CUDA [42], OpenCL [43], OpenACC [44], OpenHMPP [45] etc. The compilation time of a GPU program is less than a minute while FPGA takes several hours. Therefore, the design time of GPU accelerators is smaller compared to that of FPGAs. For those reasons, it is extremely important to consider GPU acceleration of SQA. Depending on the application, design goal, project schedule, availability of hardware and software resources, cost and design knowledge, we can chose between FPGA and GPU accelerations.

V. CONCLUSION

We have proposed a highly-parallel GPU accelerator for SQA, exploiting temporal and spatial parallelism of the quantum Monte Carlo simulation. Parallel operations are implemented on a GPU using concurrent kernel execution and parallel reduction. The processing speed of the proposed implementation is more than 97.7 times larger compared to the single-core CPU implementation. The experimental results using max-cut benchmarks show that the accuracy of the proposed accelerator is similar to that of a single-core CPU implementation. We have found the best known solution for four benchmarks, while the accuracy of the rest is over 99%. We expect to find even better solutions by simulating for a long period of time.

The proposed accelerator provides the same or better performance compared to FPGA acceleration, when using bigger Ising models with over ten thousand spins and 64 Trotters. However, proposed accelerator is less power-efficient while consuming from 1.6 to 3.9 times of power compared to FPGA. Therefore, various factors such as design goal, project schedule, design knowledge, budget should be considered to choose between FPGA and GPU accelerations.

REFERENCES

[1] T. Kadowaki and H. Nishimori, "Quantum annealing in the transverse ising model," *Phys. Rev. E, Stat. Phys. Plasmas Fluids Relat. Interdiscip. Top.*, vol. 58, no. 5, pp. 5355–5363, Nov. 1998.

[2] T. Kadowaki, "Study of optimization problems by quantum annealing," Ph.D. dissertation, Dept. Phys., Tokyo Inst. Technol., Tokyo, Japan, 1998.

[3] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003, vol. 24.

[4] F. Neukart, G. Compostella, C. Seidel, D. von Dollen, S. Yarkoni, and B. Parney, "Traffic flow optimization using a quantum annealer," *Frontiers ICT*, vol. 4, p. 29, Dec. 2017.

[5] R. Orús, S. Mugel, and E. Lizaso, "Quantum computing for finance: Overview and prospects," *Rev. Phys.*, vol. 4, Nov. 2019, Art. no. 100028.

[6] 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 Nat. Lab., Oak Ridge, TN, USA, Tech. Rep., 2017.

[7] O. Titiloye and A. Crispin, "Quantum annealing of the graph coloring problem," *Discrete Optim.*, vol. 8, no. 2, pp. 376–384, 2011.

[8] H. Ushijima-Mwesigwa, C. F. A. Negre, and S. M. Mniszewski, "Graph partitioning using quantum annealing on the D-Wave system," in *Proc. 2nd Int. Workshop Post Moores Era Supercomput. (PMES)*. New York, NY, USA: ACM, 2017, pp. 22–29.

[9] (2019). *D-Wave*. [Online]. Available: https://www.dwavesys.com

[10] M. Suzuki, S. Miyashita, and A. Kuroda, "Monte Carlo simulation of quantum spin systems. I," *Prog. Theor. Phys.*, vol. 58, no. 5, pp. 1377–1387, 1977.

[11] A.-H. Abdel-Aty, A. N. Khedr, Y. B. Saddeek, and A. A. Youssef, "Thermal entanglement in quantum annealing processor," *Int. J. Quantum Inf.*, vol. 16, no. 01, Feb. 2018, Art. no. 1850006.

[12] 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 J. Solid-State Circuits*, vol. 51, no. 1, pp. 303–309, Jan. 2016.

[13] T. Okuyama, M. Hayashi, and M. Yamaoka, "An ising computer based on simulated quantum annealing by path integral Monte Carlo method," in *Proc. IEEE Int. Conf. Rebooting Comput. (ICRC)*, Nov. 2017, pp. 1–6.

[14] 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 *Proc. Int. Symp. Intell. Signal Process. Commun. Syst. (ISPACS)*, Nov. 2018, pp. 336–340.

[15] H. M. Waidyasooriya, M. Hariyama, M. J. Miyama, and M. Ohzeki, "OpenCL-based design of an FPGA accelerator for quantum annealing simulation," *J. Supercomput.*, vol. 75, no. 8, pp. 5019–5039, Aug. 2019.

[16] C.-Y. Liu, H. M. Waidyasooriya, and M. Hariyama, "Data-transfer-bottleneck-less architecture for FPGA-based quantum annealing simulation," in *Proc. 7th Int. Symp. Comput. Netw. (CANDAR)*, Nov. 2019, pp. 164–170.

[17] H. Waidyasooriya and M. Hariyama, "Highly-parallel FPGA accelerator for simulated quantum annealing," *IEEE Trans. Emerg. Topics Comput.*, early access, Dec. 2, 2019, doi: 10.1109/tetc.2019.2957177.

[18] (2020). *SUMMIT, Oak Ridge National Laboratory's 200 Petaflop Supercomputer*. [Online]. Available: https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/

[19] (2020). *Sierra*. [Online]. Available: https://hpc.llnl.gov/hardware/platforms/sierra

[20] (2020). *NVIDIA DGX-1*. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-1/

[21] (2020). *NVIDIA DGX-2*. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-2/

[22] M. Weigel, "Performance potential for simulating spin models on GPU," *J. Comput. Phys.*, vol. 231, no. 8, pp. 3064–3082, Apr. 2012.

[23] 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," 2018, *arXiv:1807.10750*. [Online]. Available: http://arxiv.org/abs/1807.10750

[24] R. B. Stinchcombe, "Ising model in a transverse field. I. Basic theory," *J. Phys. C, Solid State Phys.*, vol. 6, no. 15, pp. 2459–2483, Aug. 1973.

[25] P. Pfeuty and R. J. Elliott, "The Ising model with a transverse field. II. Ground state properties," *J. Phys. C, Solid State Phys.*, vol. 4, no. 15, pp. 2370–2385, Oct. 1971.

[26] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[27] 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," *Prog. Theor. Phys.*, vol. 56, no. 5, pp. 1454–1469, Nov. 1976.

[28] A. Zaribafiyan, D. J. J. Marchand, and S. S. Changiz Rezaei, "Systematic and deterministic graph minor embedding for Cartesian products of graphs," *Quantum Inf. Process.*, vol. 16, no. 5, p. 136, May 2017.

[29] M. Booth, S. P. Reinhardt, and A. Roy, "Partitioning optimization problems for hybrid classical/quantum execution," D-Wave, Burnaby, BC, Canada, Tech. Rep. 14-1006A-A, 2017, pp. 01–09.

[30] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming*. Hoboken, NJ, USA: Wiley, 2014.

[31] *Optimizing Parallel Reduction in CUDA*. Accessed: 2020. [Online]. Available: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

[32] E. Boros and P. L. Hammer, "The max-cut problem and quadratic 0–1 optimization; polyhedral aspects, relaxations and bounds," *Ann. Oper. Res.*, vol. 33, no. 3, pp. 151–180, Mar. 1991.

[33] *2019. Gset*. [Online]. Available: https://web.stanford.edu/~yyye/yyye/Gset/

[34] Q. Wu and J.-K. Hao, "A memetic approach for the max-cut problem," in *Proc. Int. Conf. Parallel Problem Solving Nature*. Berlin, Germany: Springer, 2012, pp. 297–306.

[35] Y. Wang, Z. Lü, F. Glover, and J.-K. Hao, "Probabilistic GRASP-tabu search algorithms for the UBQP problem," *Comput. Oper. Res.*, vol. 40, no. 12, pp. 3100–3107, Dec. 2013.

[36] G. A. Kochenberger, J.-K. Hao, Z. Lü, H. Wang, and F. Glover, "Solving large scale max cut problems via tabu search," *J. Heuristics*, vol. 19, no. 4, pp. 565–571, Aug. 2013.

[37] U. Benlic and J.-K. Hao, "Breakout local search for the max-cutproblem," *Eng. Appl. Artif. Intell.*, vol. 26, no. 3, pp. 1162–1173, Mar. 2013.

[38] (2018). *Nallatech 385A Accelerator Card*. [Online]. Available: http://www.nallatech.com/store/fpga-accelerated-computing/pcie-accelerator-cards/nallatech-385a-arria10-1150-fpga/

[39] (2020). *OpenMP*. [Online]. Available: https://www.openmp.org

[40] (2020). *NVML API Reference Guide*. [Online]. Available: https://docs.nvidia.com/deploy/nvml-api/index.html

[41] (2020). *Top500*. [Online]. Available: https://www.top500.org/lists/2019/11/

[42] (2020). *CUDA Toolkit*. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[43] (2020). *OpenCL Overview*. [Online]. Available: https://www.khronos.org/opencl/
[44] (2020). OpenACC. [Online]. Available: https://www.openacc.org
[45] J. M. Andión, M. Arenaz, F. Bodin, G. Rodríguez, and J. Touriño, "Locality-aware automatic parallelization for GPGPU with OpenHMPP directives," *Int. J. Parallel Program.*, vol. 44, no. 3, pp. 620–643, Jun. 2016.

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

MASANORI HARIYAMA (Member, IEEE) 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.