Adaptive Bulk Search: Solving Quadratic Unconstrained Binary Optimization Problems on Multiple GPUs

Ryota Yasudo
yasudo@cs.hiroshima-u.ac.jp
Hiroshima University
Hiroshima, Japan

Koji Nakano
nakano@cs.hiroshima-u.ac.jp
Hiroshima University
Hiroshima, Japan

Yasuaki Ito
yasuaki@cs.hiroshima-u.ac.jp
Hiroshima University
Hiroshima, Japan

Masaru Tatekawa
Masaru.Tatekawa@nttdata.com
NTT DATA Corporation
Tokyo, Japan

Ryota Katsuki
Ryota.Katsuki@nttdata.com
NTT DATA Corporation
Tokyo, Japan

Takashi Yazane
Takashi.Yazane@nttdata.com
NTT DATA Corporation
Tokyo, Japan

Yoko Inaba
Yoko.Inaba@nttdata.com
NTT DATA Corporation
Tokyo, Japan

ABSTRACT

The quadratic unconstrained binary optimization (QUBO) is recently gathering attention in conjunction with quantum annealing (QA), since it is equivalent to finding the ground state of an Ising model. Due to the limitation of current QA systems, classical computers may outperform them. Researchers have thus been proposed to solve QUBO on FPGAs, GPUs, and special purpose processors. In this paper, we propose an adaptive bulk search (ABS), a framework for solving QUBO that can perform many searches in parallel on multiple GPUs. It supports fully-connected Ising models with up to 32k spins and 16-bit weights. In our ABS, a CPU host performs genetic algorithm (GA) while GPUs asynchronously perform local searches. A bottleneck for solving QUBO exists in the evaluation of the energy function, which requires $O(n^2)$ computational cost for each solution. We show this can be reduced to $O(1)$ in our ABS. The experimental results show that, with four NVIDIA GeForce RTX 2080 Ti GPUs, our framework can search up to 1.24×10^{12} solutions per second. We also show that our system quickly solves maximum cut and traveling salesman problems.

CCS CONCEPTS

• **Theory of computation** → **Optimization with randomized search heuristics**; • **Computing methodologies** → **Graphics processors**; • **Computer systems organization** → *Quantum computing*.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

KEYWORDS

Ising model, quantum annealing, QUBO, combinatorial optimization, GPGPU

ACM Reference Format:
Ryota Yasudo, Koji Nakano, Yasuaki Ito, Masaru Tatekawa, Ryota Katsuki, Takashi Yazane, and Yoko Inaba. 2020. Adaptive Bulk Search: Solving Quadratic Unconstrained Binary Optimization Problems on Multiple GPUs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Quantum computation has recently attracted attention in the field of both academic [14] and industrial [2] communities. It can be divided into two main types: universal quantum computing [4] and quantum annealing (QA) [11, 17]. QA, on which we focus in this paper, is specially designed for solving combinatorial optimization problem based on statistical mechanical models such as Ising models. The Ising model simply represents ferromagnetism consisting of spins $S = (s_0, s_1, \ldots, s_{n-1})$ where $s_i = \pm 1$, interaction J_{ij} between s_i and s_j, and the external magnetic field $h_0, h_1, \ldots, h_{n-1}$. The goal of QA is to find the ground state of the Ising model, i.e., the state of the spins that minimizes the Hamiltonian $\mathcal{H} = -\sum J_{ij} s_i s_j - \sum h_i s_i$.

The Ising model has a potential for solving a wide range of combinatorial optimization problems, including Karp's 21 NP-complete problems [20, 21], real-world applications [23, 28], and machine learning [8, 9]. For solving these problems, QA has already been developed by D-Wave systems [16]. However, a current commercial system called D-Wave 2000Q provides a limited number of spins (2048 spins) and sparse interaction (given by a Chimera graph). In the graph, a vertex and an edge denote a spin and an interaction, respectively. There exist no interactions if two spins are not connected in the graph. Hence, minor-embedding, which is NP-hard [5], is needed for fitting the Chimera graph. If the interactions are fully-connected, D-Wave 2000Q solves at most 64-spin problems. In addition, the ground state is not always attained due to environmental effects [1]. A few studies suggest that classical simulated

annealing is better than QA [3]. It is thus difficult to obtain good solutions for practical problems at least now.

The optimization problem for the Ising model can equivalently be converted into the *quadratic unconstrained binary optimization* (*QUBO*) problem, and vice versa. QUBO has been studied from 1960s [12] in the field of combinatorial optimization. QUBO is known as NP-hard [26]. While exact methods are studied, the size of problems is restricted; for example, up to 200 bits [19]. We hence need heuristic algorithms such as simulated annealing (SA) and genetic algorithm (GA). Many researchers and developers recently study on solving QUBO or Ising models on special purpose processors and hardware accelerators such as field programmable gate arrays (FPGAs) and graphical processing units (GPUs).

Inagaki *et al.* proposed a coherent Ising machine, an Ising model solver based on photonic technologies [15]. Experimental results are currently reported for small Ising models with 2000 spins. Matsubara *et al.* proposed to use an FPGA that efficiently performs a Markov-chain Monte-Carlo search [22]. It supports only 1,024 spins and attains 20.4G search rate, which our proposed system in this paper outperforms by 60×. Yamaoka *et al.* proposed Ising computer using CMOS circuits [30]. It supports 20k (=20,480) spins, but the possible interactions can have only three values (+1, 0, −1). Goto *et al.* proposed a simulated bifurcation (SB), which is an alternative algorithm to conventional QA. They implement them on a GPU cluster [13] and an FPGA [29]. Okuyama *et al.* proposed momentum annealing (MA) [25], which updates all spins simultaneously by converting an Ising model to bipartite graphs with two times larger spins. This means that the required memory size for storing interactions increases by two times. MA also requires mapping to a bipartite graph and carefully setting parameters called momentum coupling.

An instance of a QUBO problem is given by a *weight matrix* $W = (W_{ij})$ $(0 \le i, j < n)$, an $n \times n$ symmetric matrix of numbers called *weights* such that $W_{ij} = W_{ji}$ holds. Each bit corresponds to a spin of the Ising model, but its value is 0 or 1, instead of +1 or −1. The objective of the QUBO is to find an n-bit vector $X = x_0x_1 \cdots x_{n-1}$ such that the *energy function* $E(X)$ is minimized for given weight matrix. The energy function is defined as

$$E(X) \coloneqq X^{\mathrm{T}}WX = \sum_{0 \le i,j < n} W_{ij}x_ix_j. \tag{1}$$

Figure 1 shows an example of QUBO. Eq. (1) clearly suggests that the computation of the energy requires $O(n^2)$ computational cost, i.e., the total number of executed instructions. Thus, the computational cost prevents us from solving QUBO.

In this paper, we consider an algorithm that enables us to reduce the computational cost to $O(1)$ per one solution. We call the computational cost for evaluating the energy per one solution *search efficiency*. The idea here is to compute the energy of a neighbor solution based on the change of the energy. For later references, we define two functions. For an n-bit vector $X = x_0x_1 \cdots x_{n-1}$, let $\mathrm{flip}_k(X)$ be a function that returns an n-bit vector obtained by flipping the k-th bit of X, i.e.,

$$\mathrm{flip}_k(X) \coloneqq x_0x_1 \cdots x_{k-1}\overline{x_k}x_{k+1} \cdots x_{n-1}. \tag{2}$$

weight matrix W

	0	1	2	3
0	1	−1	2	−2
1	−1	5	3	4
2	2	3	−3	−5
3	−2	4	−5	4

$$E(X) = X^{\mathrm{T}}WX$$

e.g.,

$$\begin{aligned} E(1011) &= 1 + (-3) + 4 + 2(2 + (-2) + (-5)) \\ &= -8 \end{aligned}$$

Figure 1: An example of a QUBO problem with $n = 4$.

For a bit x, let $\varphi(x)$ be a function defined as

$$\varphi(x) \coloneqq \begin{cases} +1 & \text{if } x = 0, \\ -1 & \text{if } x = 1. \end{cases} \tag{3}$$

Clearly, $\varphi(x) = 1-2x$ holds. Let us now consider $\Delta_k(X) = E(\mathrm{flip}_k(X)) - E(X)$, i.e., the amount of change of the energy if the k-th bit in X is flipped. When x_k is flipped, the energy changes by $W_{ki} + W_{ik} = 2W_{ki}$ for $i \neq k$ and $x_i = 1$, and also by W_{kk}. Thus we can write

$$\Delta_k(X) = \varphi(x_k)\left(2\sum_{i \neq k} W_{ki}x_i + W_{kk}\right). \tag{4}$$

The idea here is to retain the values of $\Delta_k(X)$ for all k $(0 \le k < n)$. Consequently we can compute all of the energy $E(\mathrm{flip}_0(X))$, $E(\mathrm{flip}_1(X))$, ..., $E(\mathrm{flip}_{n-1}(X))$ for all of the n neighbor solutions by

$$E(\mathrm{flip}_k(X)) = E(X) + \Delta_k(X). \tag{5}$$

The value of $\Delta_i(\mathrm{flip}_k(X))$ can be computed by

$$\Delta_i(\mathrm{flip}_k(X)) = \begin{cases} -\Delta_i(X) & \text{if } k = i, \\ \Delta_i(X) + 2W_{ik}\varphi(x_i)\varphi(x_k) & \text{if } k \neq i. \end{cases} \tag{6}$$

The derivation of this formula will be shown in Section 2. In this paper, we propose an algorithm where this $O(1)$ search efficiency is always possible while it can be combined with GA.

In our algorithm, the evaluation of the energy is performed only by GPUs in parallel. A GPU is a specialized circuit originally designed to accelerate computation for building and manipulating images. Latest GPUs are designed for general-purpose computing, and hence they are now used for accelerating various applications. Since GPU architecture is built around a scalable array of multithreaded processors, each thread can compute the value of $\Delta_i(X)$ by Eq. (6) in parallel, and consequently the time complexity becomes small. Moreover, all of the values are stored in a register file, which keeps the access latency low. Thus GPU can efficiently perform a local search.

Based on the algorithm above, we propose an *adaptive bulk search* (*ABS*), a framework for solving QUBO consisting of a CPU host and multiple GPUs. The main concept of ABS is to perform a genetic

algorithm and local searches in bulk by exploiting GPU resources. Starting from random bit vectors, a CPU performs genetic algorithm for a global search and asks GPUs to perform local searches with selected solutions (called *target* solutions). A GPU performs a large number of local searches for the target solutions in parallel, and returns the best-found solutions. Here, we can flip arbitrary bits on the basis of the values of $\Delta_i(X)$ for all i $(0 \le i < n)$ with any policy, including a greedy algorithm and SA. Thus, we can adaptively change the local search algorithm. Combining GA and a local search makes it impossible to keep $O(1)$ search efficiency, because iterative local search steps must start with different solutions. To avoid this drawback, we propose a *straight search*, a search algorithm from generation to generation. In addition, a host never computes the energy function. In this way our ABS maintains $O(1)$ search efficiency. Our implementation on NVIDIA GeForce RTX 2080 Ti GPUs supports any QUBO problems with up to 32k variables (bits) and 16-bit weights. We exploit GPU resources so that the occupancy becomes 100% and show that our ABS can search up to 1.24T (1.24×10^{12}) solutions per second.

The structure of the paper is as follows. Section 2 discusses algorithms for solving QUBO by introducing search efficiency. After we show a naive local search algorithm, we improve the search efficiency, which will finally be reduced to $O(1)$. We also propose a method for combining genetic algorithm and local search algorithms. In Section 3, we show the implementation of the proposed ABS written in CUDA C, which exploits resources of GPU architecture. The proposed system is experimentally evaluated in terms of time-to-solution and throughput in Section 4. Finally Section 5 concludes the paper.

2 ALGORITHMS

In this section, we propose a local search algorithm, a genetic algorithm, and their combination. Asymptotic analysis using search efficiency shows the computational cost of our algorithm.

2.1 Local search algorithm

Let us start with a naive local search algorithm. Again, we use a function $\mathrm{flip}_k(X)$ that returns an n-bit vector obtained by flipping the k-th bit of $X = x_0x_1\cdots x_{n-1}$. Since the Hamming distance between X and $\mathrm{flip}_k(X)$ is one, we regard $\mathrm{flip}_k(X)$ as a neighbor solution of X. Using this neighbor function, we can perform a local search shown in Algorithm 1.

The ACCEPT function depends on metaheuristics. Simulated annealing (SA) is one of the well-studied metaheuristics. In SA, a neighbor solution is probabilistically accepted according to the amount of the change of the energy, denoted by ΔE [18]. In Algorithm 1, ΔE is equal to $E(\mathrm{flip}_k(X)) - E(X)$. The acceptance probability $p(\Delta E)$ is

$$p(\Delta E) = \begin{cases} 1 & \text{if } \Delta E \le 0, \\ \exp(-\Delta E/k_B t) & \text{if } \Delta E > 0, \end{cases} \tag{7}$$

where t is temperature and k_B is some constant (in nature, Boltzmann's constant). The temperature t is initially set to be high and reduced gradually. To obtain good solutions, we must carefully set such parameters and functions for cooling temperature.

Algorithm 1 A naive local search with $O(n^2)$ search efficiency

```
E(X) ← Σ_{0≤i,j<n} W_ij x_i x_j
E(B) ← E(X)                                   ▷ best energy
loop
    randomly select a bit x_k (0 ≤ k < n) in X = x_0 x_1 ··· x_{n-1}.
    generate a neighbor solution flip_k(X) = X'.
    E(flip_k(X)) ← Σ_{0≤i,j<n} W_ij x'_i x'_j.
    if ACCEPT(E(X), E(flip_k(X))) == true then
        X ← flip_k(X)
        if E(X) < E(B) then
            B ← X                                 ▷ update best solution
        end if
    end if
end loop

function ACCEPT(E(X), E(flip_k(X)))
    return true or false                          ▷ depending on metaheuristics
end function
```

Let us evaluate the computational cost for the SA. Let m be the number of iterations of the search step. For a given n-bit vector X, the value of $E(X)$ can be computed with $O(n^2)$ computational cost by evaluating Eq. (1). Hence, the SA with m search steps requires $O(n^2m)$ computational cost. Next, let us introduce a performance index for evaluating the performance of search algorithms called a *search efficiency*.

Definition 1. Search efficiency *is the computational cost divided by the number of evaluated solutions. In other words, it denotes the expected number of operations necessary to evaluate the energy for one solution.*

In a naive local search, $O(n^2)$ computational cost is required for each solution. Thus, we have

Lemma 1. *The search efficiency of a local search in Algorithm 1 is* $O(n^2)$.

Suppose that we know the value of energy function $E(X)$ for an instance of QUBO with $W = (W_{ij})$ $(0 \le i, j < n)$ and an n-bit vector $X = x_0x_1\cdots x_{n-1}$. Let us consider the value of $E(\mathrm{flip}_k(X))$, i.e., the energy when x_k is flipped. This value depends on the value of x_k. If $x_k = 0$, then the energy increases by W_{kk}, W_{kj}, and W_{jk} for all j such that $j \ne k$ and $x_j = 1$. Thus, if $x_k = 0$, then $E(\mathrm{flip}_k(X))$ can be written as

$$E(\mathrm{flip}_k(X)) = E(X) + 2\sum_{j\ne k} W_{kj}x_j + W_{kk}. \tag{8}$$

Similarly, if $x_k = 1$, then the energy increases by $-W_{kk}$, $-W_{kj}$, and $-W_{jk}$ for all k such that $j \ne k$ and $x_j = 1$. Thus $E(\mathrm{flip}_k(X))$ can be written as

$$E(\mathrm{flip}_k(X)) = E(X) - 2\sum_{j\ne k} W_{kj}x_j - W_{kk}. \tag{9}$$

Combining Eqs. (8) and (9), we have

$$E(\mathrm{flip}_k(X)) = E(X) + \varphi(x_k)\left(2\sum_{j\ne k} W_{kj}x_j + W_{kk}\right). \tag{10}$$

Recall that $\varphi(x)$ is a function $\varphi(x) : \{0,1\} \rightarrow \{+1,-1\}$. Using Eq. (10), we can compute $E(\text{flip}_k(X))$ with $O(n)$ computational cost if we know $E(X)$ in advance. Therefore the SA with m search steps can evaluate $E(X)$ for at most $m+1$ solutions X with $O(n^2+mn)$ computational cost.

This difference computation reduces computational cost from Algorithm 1 to Algorithm 2. Since the algorithm can search $m+1$ solutions with $O(n^2+mn)$ computational cost, we have

Lemma 2. *The search efficiency of a local search in Algorithm 2 with m search steps is*

$$\frac{O(n^2+mn)}{m+1} = O\left(n + \frac{n^2}{m}\right).$$

Algorithm 2 A local search with $O(n + \frac{n^2}{m})$ search efficiency

```
1: E(X) ← Σ_{0≤i,j<n} W_ij x_i x_j
2: E(B) ← E(X)                                   ▷ best energy
3: loop
4:     randomly select a bit x_k (0 ≤ k < n) in X = x_0 x_1 ··· x_{n-1}.
5:     generate a neighbor solution flip_k(X).
6:     E(flip_k(X)) ← E(X) + φ(x_k)(2 Σ_{j≠k} W_kj x_j + W_kk).
7:     if ACCEPT(E(X), E(flip_k(X))) == true then
8:         X ← flip_k(X)
9:         if E(X) < E(B) then
10:            B ← X                              ▷ update best solution
11:        end if
12:    end if
13: end loop
14:
15: function ACCEPT(E(X), E(flip_k(X)))
16:    return true or false                       ▷ depending on metaheuristics
17: end function
```

Next, we reduce the search efficiency by computing $E(\text{flip}_i(X))$ for all i $(0 \le i < n)$. From Eq. (10), it takes $O(n) \times n = O(n^2)$ computational cost to compute all of them. To reduce the cost, we retain

$$\Delta_k(X) \coloneqq E(\text{flip}_k(X)) - E(X) \tag{11}$$

$$= \begin{cases} 2\sum_{j\neq k} W_{kj}x_j + W_{kk} & \text{if } k = 0, \\ -2\sum_{j\neq k} W_{kj}x_j - W_{kk} & \text{if } k = 1 \end{cases} \tag{12}$$

$$= \varphi(x_k)\left(2\sum_{j\neq k} W_{kj}x_j + W_{kk}\right). \tag{13}$$

for all k $(0 \le k < n)$. Obviously $\Delta_k(X)$ denotes the change of energy after we flip x_k in X. Since $E(\text{flip}_k(X)) = E(X)+\Delta_k(X)$ holds, we can compute the energy function $E(\text{flip}_k(X))$ for all k $(0 \le k < n)$ with $O(1)\times n = O(n)$ computational cost if we know $E(X)$ and $\Delta_k(X)$ in advance. In the following, we show the algorithm to achieve this.

Suppose that we know the value of $\Delta_i(X)$ for all i $(0 \le i < n)$. Now let us consider $\Delta_i(\text{flip}_k(X))$, i.e., the change of Δ_i after we flip x_k in X. This value depends on the situations which can be divided into the following five cases.

Case 1: $i = k$. In this case, $\Delta_i(\text{flip}_k(X))$ is clearly equal to $-\Delta_i(X)$, regardless of the value of $x_i (= x_k)$.

In the remaining cases, we assume that $i \neq k$ holds.

Case 2: $x_i = 0, x_k = 0 \rightarrow 1$. Before x_k is flipped, W_{ik} and W_{ki} never affect Δ_i; however, after x_k is flipped to be 1, they start to affect Δ_i. Thus, we have

$$\Delta_i(\text{flip}_k(X)) = \Delta_i(X) + 2W_{ik}. \tag{14}$$

Case 3: $x_i = 0, x_k = 1 \rightarrow 0$. Contrary to Case 2, W_{ik} and W_{ki} cease to affect Δ_i after x_k is flipped. Thus, we have

$$\Delta_i(\text{flip}_k(X)) = \Delta_i(X) - 2W_{ik}. \tag{15}$$

Case 4: $x_i = 1, x_k = 0 \rightarrow 1$. If $x_i = 1$, then we just have to invert signs in Eq. (14) or (15); in Case 4, Eq. (14) because x_k is the same as Case 2. As a result, Case 4 satisfies Eq. (15).

Case 5: $x_i = 1, x_k = 1 \rightarrow 0$. Contrary to Case 4, Case 5 satisfies Eq. (14).

In summary, Eq. (14) holds if $x_i = x_k$, and Eq. (15) holds if $x_i \neq x_k$. Thus, we can write

$$\Delta_i(\text{flip}_k(X)) = \Delta_i(X) + 2W_{ik}\varphi(x_i)\varphi(x_k). \tag{16}$$

Note that $\varphi(x)^2 = 1$ and $\varphi(x)\varphi(\overline{x}) = -1$ hold. To exploit Eq. (16), let us consider to start initialization with a zero vector $\mathbf{0} = 00\cdots0$. Clearly, $E(\mathbf{0}) = 0$ and $\Delta_i(\mathbf{0}) = W_{ii}$ hold, and thus they can be computed in $O(n)$. Subsequently, we can search at most n solutions until a solution becomes the given solution X' as shown in Algorithm 3. Since the computation for n solutions until an initial solution and subsequent m solutions respectively require $O(n^2)$ and $O(mn)$ computational costs, we have

Lemma 3. *The search efficiency of a local search in Algorithm 3 with m search steps is*

$$\frac{O(n^2+mn)}{n+m} = O(n).$$

As we stated above, one of the drawbacks of a conventional local search is that a neighbor solution would hardly be accepted, especially if a solution is near local minimum. This drawback decreases the number of flips per unit time. To avoid this, we force to flip a bit in every iteration. Fortunately, it is easy to do this by using Δ_i again. Suppose that we know Δ_i for all i $(0 \le i < n)$. This means that we know n neighbor solutions, and thus we can arbitrary choose one of them on the basis of their energy. From $E(\text{flip}_j(X)) = E(X) + \Delta_j(X)$, we can evaluate the energy of n neighbor solutions such that 1 bit is flipped with $O(n)$ computational cost. In the initialization step and m iterations of the search step, the energy of at most $(n+m)n$ solutions is evaluated with $O(n^2+mn)$ computational cost. Ultimately, we can design Algorithm 4, and we have

Theorem 1. *The search efficiency of the local search in Algorithm 4 with m search steps is*

$$\frac{O(n^2+mn)}{n^2+mn} = O(1).$$

In Algorithm 4, we can use any selection policy. In this paper, we adopt the following selection policy shown in Figure 2. The main concept is extracting some bits from a bit vector and then flipping a bit x_i with the minimum Δ_i. It forces a bit flip obviously. If the

Algorithm 3 A local search with $O(n)$ search efficiency

Require: $X = 00\cdots0$ and $d_i = 0$ for all i $(0 \le i < n)$ ▷ d_i retains $\Delta_i(X)$
$E(X) \leftarrow 0$
$E(B) \leftarrow E(X)$ ▷ best energy
repeat
  select a k-th bit such that $x'_k = 1$.
  for all i $(0 \le i < n$ and $i \ne k)$ **do**
    $d_i \leftarrow d_i + 2W_{ik}\varphi(x_i)\varphi(x_k)$
  end for
  $E(X) \leftarrow E(X) + d_k$
  $d_k \leftarrow -d_k$
  $x_k \leftarrow \overline{x}_k = 1 - x_k$ ▷ flip x_k
  if $E(X) < E(B)$ **then**
    $B \leftarrow X$ ▷ update best solution
  end if
until $X = X'$
loop
  randomly select a bit x_k $(0 \le k < n)$ in $X = x_0x_1\cdots x_{n-1}$.
  generate a neighbor solution $\mathrm{flip}_k(X)$.
  for $i \leftarrow 0, n-1$ **do**
    $d_i \leftarrow d_i + 2W_{ik}\varphi(x_i)\varphi(x_k)$
  end for
  evaluate $E(\mathrm{flip}_k(X))$.
  if ACCEPT$(E(X), E(\mathrm{flip}_k(X)))$ == **true then**
    $X \leftarrow \mathrm{flip}_k(X)$
    if $E(X) < E(B)$ **then**
      $B \leftarrow X$ ▷ update best solution
    end if
  end if
end loop

function ACCEPT$(E(X), E(\mathrm{flip}_k(X)))$
  return **true** or **false** ▷ depending on metaheuristics
end function

Algorithm 4 Proposed local search with $O(1)$ search efficiency

Require: $X = 00\cdots0$ and $d_i = 0$ for all i $(0 \le i < n)$ ▷ d_i retains $\Delta_i(X)$
$E(X) \leftarrow 0$
$E(B) \leftarrow E(X)$ ▷ best energy
repeat
  select a k-th bit such that $x'_k = 1$.
  for all i $(0 \le i < n$ and $i \ne k)$ **do**
    $d_i \leftarrow d_i + 2W_{ik}\varphi(x_i)\varphi(x_k)$
    if $E(X) + d_i < E(B)$ **then**
      $B \leftarrow \mathrm{flip}_i(X)$ ▷ update best solution
    end if
  end for
  $E(X) \leftarrow E(X) + d_k$
  $d_k \leftarrow -d_k$
  $x_k \leftarrow \overline{x}_k = 1 - x_k$ ▷ flip x_k
until $X = X'$
loop
  arbitrarily select a bit x_k $(0 \le k < n)$ in X ▷ based on selection policy
  for all i $(0 \le i < n$ and $i \ne k)$ **do**
    $d_i \leftarrow d_i + 2W_{ik}\varphi(x_i)\varphi(x_k)$
    if $E(X) + d_i < E(B)$ **then**
      $B \leftarrow \mathrm{flip}_i(X)$ ▷ update best solution
    end if
  end for
  $E(X) \leftarrow E(X) + d_k$
  $d_k \leftarrow -d_k$
  $x_k \leftarrow \overline{x}_k = 1 - x_k$ ▷ flip x_k
end loop

number of extracted bits is n (i.e., all of the bits in a bit vector), the algorithm becomes the same as a greedy algorithm because the best neighbor solution is always selected. In contrast, if the number of extracted bits is 1, a bit is selected randomly. Thus, the number l of extracted bits is similar to the temperature of SA. Note, however, that the larger l corresponds to the lower temperature. As with parallel tempering [10], we can set a different temperature for each search. Extracted bits can be randomly selected, but we deterministically select them by introducing an *offset*. When an offset is a, the bits $x_a, x_{a+1}, \ldots, x_{a+l-1}$ are selected, and then the offset changes to $(a + l) \mod n$. This algorithm requires no random number generation unlike conventional SA, and consequently the computational complexity are small, while it provides good solutions because it is combined with GA introduced below and GPU-based highly parallel processing. Note that, while this policy compares l solutions, n solutions are evaluated at once and a solution other than compared l solutions can be selected if the best solution is updated by selecting it.

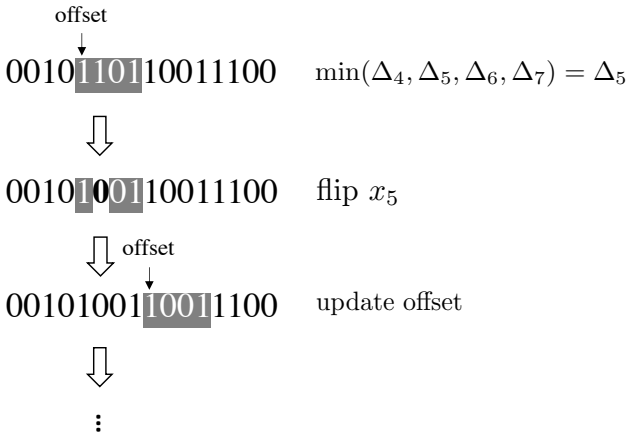

Figure 2: A selection policy without random numbers used in our ABS.

2.2 Genetic algorithm

2.2.1 Genetic algorithm in a host. A *genetic algorithm* (*GA*) is one of the well-known metaheuristics inspired by biological evolutions

with mutations, crossover, and selection. Several solutions are represented as a *genetic representation*, which is typically a bit vector, and each solution has *fitness*. The average fitness increases as with natural selection. A GA generally consists of the following steps.

Step 1: Initialize solutions.
Step 2: Compute fitness of the solutions.
Step 3: Perform mutation, crossover, or copy for the next generation.
Step 4: Go back to Step 2.

A solution of QUBO is originally a bit vector, and hence the solution itself can be used for a genetic representation. A fitness corresponds to the energy function. A mutation generates a new solution from one selected solution with flipping some random bits. A crossover generates a new solution from two solutions called parents so that each bit is randomly selected from either of the parents.

We combine GA with a local search; a host CPU and a device GPU perform GA and local search, respectively. After a new solution X is generated, typical GA adopts it for the next generation as it is. However, we can obtain better solutions similar to X by using a local search. Thus, we perform a local search between two generations and consequently the objective function improves. Furthermore, a CPU does not need to compute the fitness, because a CPU just stores the solution to a target buffer.

One of the drawbacks of GA is that it may lead to premature convergence when an extremely good solution is found, especially it is combined with a local search. Thus, we keep the solutions distinct to avoid premature convergence. After a local search ends, the best solution during the search is inserted only if the same solution does not exist in the solution pool. Hence, a solution pool is always sorted by the value of energy, and binary search checks whether the same solution exists and computes the index to insert with $O(\log n)$ computational cost. At this point, the worst solution in the solution pool is replaced with the new solution.

2.2.2 Straight search in a GPU. Unfortunately, combining GA and a local search prevents the difference computation of the energy described in Section 2.1. This is because a local search must start with a new solution generated by GA. Thus, thirdly, we propose an algorithm called a *straight search* to enable the difference computation. Suppose that there is a known solution X and a new solution X' Starting from a known solution X, it flips a bit so that the Hamming distance to X' decreases until X becomes the same as X'. Thus, the number of flips corresponds to the Hamming distance between X and X'. Since it starts from a known solution, the difference computation is possible, and besides a local search is performed at the same time. A bit k is greedily selected so that Δ_k becomes minimum. A straight search also enables to escape from a local minimum because returning to the visited solutions is prohibited. In other words, a straight search generalizes the first half of Algorithm 4 so that X is arbitrary. Figure 3 illustrates a concept of a straight search, and a pseudo-code is shown in Algorithm 5.

In our ABS, each CUDA block performs straight search and a local search alternately as shown in Figure 4.

Algorithm 5 Straight search

Require: $X = x_0x_1 \cdots x_{n-1}$ and $X' = x'_0x'_1 \cdots x'_{n-1}$
1: $E(B) \leftarrow E(X)$
2: **repeat**
3: compute minimum Δ_k such that $x_k \neq x'_k$. ▷ greedily select a bit
4: **for all** i $(0 \leq i < n$ and $i \neq k)$ **do**
5: $d_i \leftarrow d_i + 2W_{ik}\varphi(x_i)\varphi(x_k)$
6: **end for**
7: $E(X) \leftarrow E(X) + d_k$
8: $d_k \leftarrow -d_k$
9: $x_k \leftarrow \overline{x}_k = 1 - x_k$ ▷ flip x_k
10: **if** $E(X) < E(B)$ **then**
11: $B \leftarrow X$ ▷ update best solution
12: **end if**
13: **until** $X = X'$

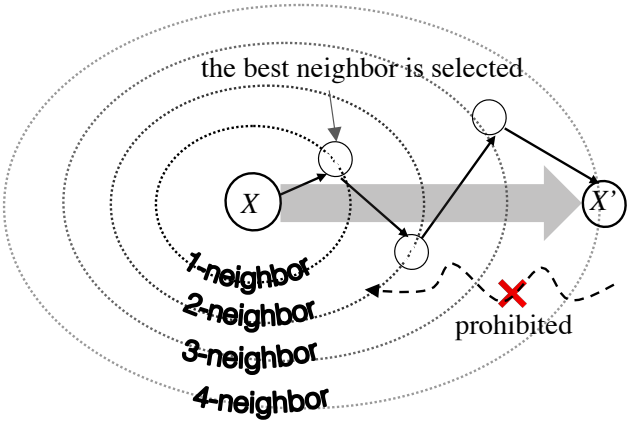

Figure 3: An overview of a straight search from X to X' such that the Hamming distance between them is four.

3 IMPLEMENTATION

Figure 5 illustrates an overview of our proposed system. It consists of a CPU host and multiple GPUs. Target and solution buffers are located in a global memory that can be accessed by both a host and a device, and thus they can exchange data via this memory. In general, the host manages the solutions and stores a *target* solution T generated by GA, and a CUDA block in a GPU executes a local search from one of the target solutions and stores the best-found solution $\mathcal{B}$ and its energy $\mathcal{E}_{\mathcal{B}}$. A host and a device do not communicate with each other directly, but rather exchange data indirectly via a global memory. As a result, each CUDA block runs asynchronously, and hence the overhead for synchronization is avoided. In the following we describe the details of the system.

3.1 CPU Host

A CPU host manages a *solution pool* consisting of m solutions X_0, X_1, ..., X_{m-1} and corresponding values of the energy function

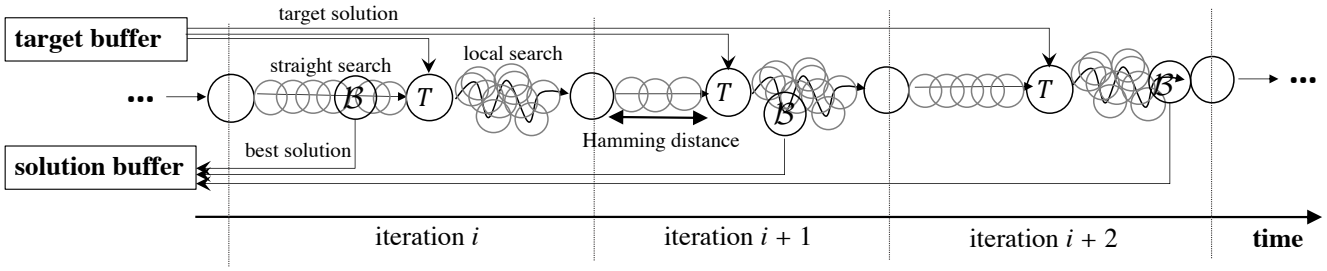

Figure 4: Iterative search with a straight search and a local search. Note that the iteration i starts with the last solution in the iteration $i-1$, which enables efficient computation of the energy.

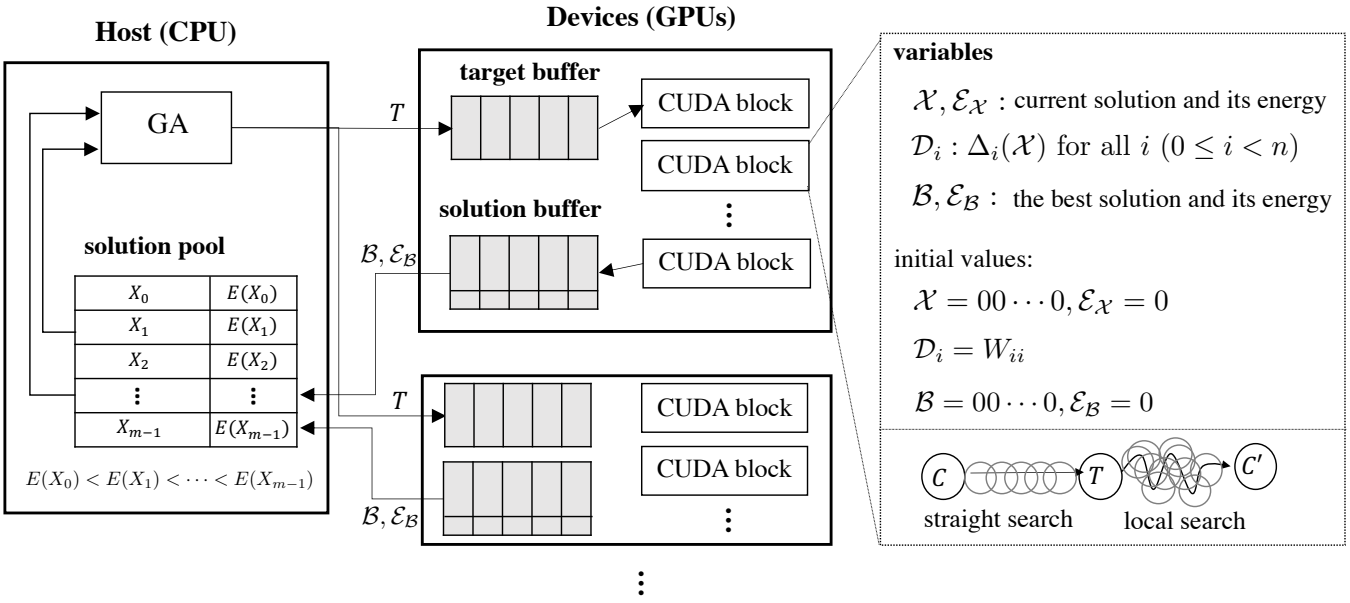

Figure 5: An overview of our proposed system consisting of a CPU and multiple GPUs.

$E(X_0)$, $E(X_1)$, ..., $E(X_{m-1})$. They are kept distinct from each other and sorted to be $E(X_0) < E(X_1) < \cdots < E(X_{m-1})$.

The operations of the host consist of the following steps.

Step 1: Initialize the solution pool and the target buffer.

Step 2: Wait for new solutions stored by GPU.

Step 3: If new solutions have been stored, insert them to the solution pool.

Step 4: Generate and store new target solutions, and go back to Step 2.

In Step 1, the solutions are initially set to be random bit vectors, and the energy values are $+\infty$ in the sense that they are not computed. In the ABS, a host never computes the energy function. In Step 2, a host repeatedly reads the value of the global counter by using `cudaMemcpyAsnyc` function. Once the value increases, which means a GPU have stored new solutions to a solution buffer, a host goes to Step 3. In Step 3, a host inserts new solutions to the solution pool. As we describe above, the solutions in the solution pool must be sorted by the value and differ from each other. To satisfy these conditions, we use binary search, which enables us to check whether the new solution is in the solution pool and detect the location to insert in $O(\log m)$ time. In Step 4, a host generates and stores new target solutions by GA. The number of generated solutions is set to be the same as the number of newly arrived solutions generated by a device.

3.2 GPU Device

Our method is implemented on a graphics processing unit (GPU) consisting of multiple streaming multiprocessors. NVIDIA provides a parallel computing architecture called *compute unified device architecture* (*CUDA*), which includes a general-purpose computing platform and programming model [6]. CUDA enables us to access a virtual instruction set and memories using high-level languages such as C/C++. CUDA programming model consists of three layers: threads, blocks, and grids. A block consists of several threads, and a grid consists of several blocks. A multithreaded program is

partitioned into blocks of threads so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors. We call a block of threads in terms of CUDA programming model a *CUDA block*. A GPU can retain data in three memory locations: a global memory, a shared memory, and a register file. We store a best solution $\mathcal{B}$ and energies $\mathcal{E}_{\mathcal{B}}$ and $\mathcal{E}_{X}$ in a shared memory and store a current solution X and Δ_i ($0 \leq i < n$) in a register file. We implement the ABS on NVIDIA GeForce RTX 2080 Ti GPU (Turing Architecture, Compute Capability 7.5) [7] in CUDA C. It has 64-KB shared memory, 1024 threads (32 warps), 64K 32-bit registers per multiprocessor, an 11GB global memory (GDDR6 SDRAM), and 68 multiprocessors.

Each CUDA block independently executes local searches. Each thread handles one or more bits of a solution. Let p denote *bits per thread*, that is, the number of bits handled by a thread. In a GPU we use, a CUDA block has at most 1024 threads. Thus p is equal to one for n-bit QUBO problems, and it exceeds one if n is more than 1024. A thread has the value of p bits and corresponding values of Δ (i.e., they are stored in a register file). Using a register file, we can update the energy with low latency. Since each thread has 64 registers, our system can support up to 32k-bit QUBO problems. As a result, a thread i performs the computation of $\Delta_{ip}, \Delta_{ip+1}, \ldots, \Delta_{ip+p-1}$. The total size of data limits the number of warps (a group of 32 threads) executed in parallel. Our implementation attains 100% occupancy, i.e., the ratio of the number of resident warps to the maximum number of warps supported by the hardware.

The operations of a device consist of the following steps.

Step 1: Initialize the variables.
Step 2: Read a target solution T.
Step 3: Reset the best solution $\mathcal{B}$ and its energy $\mathcal{E}_{\mathcal{B}}$.
Step 4a: Perform a straight search from a current solution C to a target solution T.
Step 4b: Perform a local search from T to C', which corresponds to C in the next iteration.
Step 5: Store the best solution $\mathcal{B}$ found in Step 4 and its energy $\mathcal{E}_{\mathcal{B}}$ to the solution buffer, and go back to Step 2.

In Step 1, the variables are initialized as shown in Figure 5. Recall that starting from a zero vector enables $O(1)$ search efficiency. At this point, Δ_i is set to be W_{ii} by accessing the global memory. In Step 2, a CUDA block reads a target solution T from the target buffer. In Step 3, the best solution $\mathcal{B}$ is reset for avoiding premature convergence. In Step 4a, a CUDA block performs a straight search from C to T. Here the number of flips is equal to the Hamming distance between C and T, and thus the execution time varies. This variation may produce an overhead for synchronization between CUDA blocks, but it is avoided because each CUDA block operates asynchronously. In Step 4b, a CUDA block performs an arbitrary local search from T with the fixed number of flips. The resulting solution C' corresponds to C in the next iteration, and hence the search efficiency remains $O(1)$. In Step 5, a CUDA block stores the best solution $\mathcal{B}$ found in Step 3 and its energy $\mathcal{E}_{\mathcal{B}}$ to the solution buffer. Since the best solution is reset in Step 2, the already stored solution is ignored, and hence various solutions can be stored.

4 EXPERIMENTAL RESULTS

This section provides our experimental results and discussion. We evaluate our ABS in terms of two metrics: the time-to-solution and the search rate. For each parameter, we show the average in ten times of measurement. Time-to-solution denotes the time to obtain a target solution, which is typically the exact solution or the best-known solution. The *search rate* is the number of total solutions searched by a system per second, which is used in [22]. Our ABS is implemented on a multi-GPU system consisting of four NVIDIA GeForce RTX 2080 Ti GPUs and an 18-core Intel Core i9-9980XE CPU (3.00 GHz).

4.1 Benchmarks

Our proposed system is evaluated for QUBO problems with up to 32k spins and 16-bit weights. We use the following three benchmarks for QUBO, including easy and hard instances.

4.1.1 Max-Cut problem. The Max-Cut problem is one of Karp's 21 NP-complete problems, a problem of finding a maximum cut in a graph. It is known that the Max-Cut problem is equivalent to solving Ising models: a spin and an interaction correspond to a vertex and a weight of an edge in a graph, respectively.

The Max-Cut can be formulated by QUBO as follows. Let a vertex in a graph correspond to a bit x_i in QUBO and divide the vertices into two groups by their values (0 or 1). We set weights by

$$W_{ij} = \begin{cases} G_{ij} & \text{if } i \neq j, \\ -\sum_{0 \leq k < n} G_{ik} & \text{if } i = j, \end{cases} \tag{17}$$

where $G_{ij} (= G_{ji})$ denotes a weight between vertex i and vertex j in an original graph. Obviously, W_{ii} is the negation of the degree of a vertex i and W_{ij} ($i \neq j$) is the edge weight between two vertices i and j. Let V_0 and V_1 be the set of a vertex i such that $x_i = 0$ and $x_i = 1$, respectively. By the values of W_{ii}, the negation of the energy includes the sum of weights from the vertices in V_1. They include the weights between a vertex in V_1 and that in V_0, and the weights between two vertices in V_1. The latter weights are subtracted by $W_{ij}+W_{ji} = 2G_{ij}$, and consequently the cut weights remain. Figure 6 shows an example of QUBO formulation of the Max-Cut. In the figure, the edge weights of an original graph are one. The Max-Cut is widely used as a benchmark for QUBO solvers [13, 15, 25]. In particular, G-set benchmark [31] is commonly used, and we also adopt it.

4.1.2 TSP. The traveling salesman problem (TSP) is a well-known NP-hard problem in combinatorial optimization. It is reported that c-city TSP can be converted into a $(c-1)^2$-bit QUBO problem [21], and we convert some symmetric TSPs obtained from traveling salesman problem library (TSPLIB) [24, 27] into QUBO problems. Figure 7 illustrates an example of TSP represented by QUBO formulation. When we arrange a solution of TSP QUBO in a matrix, a column and a row denote the order and the city, respectively. A city i is visited in the j-th order if a bit in the column j and the row i is one. A solution is hence valid only if each row and each column has exactly one bit whose value is 1. To guarantee this condition, we add penalties whose value is twice as much as the maximum distance. Hence, the Hamming distance of two different solutions is at least four, and four bit-flips are needed to obtain a valid different solution

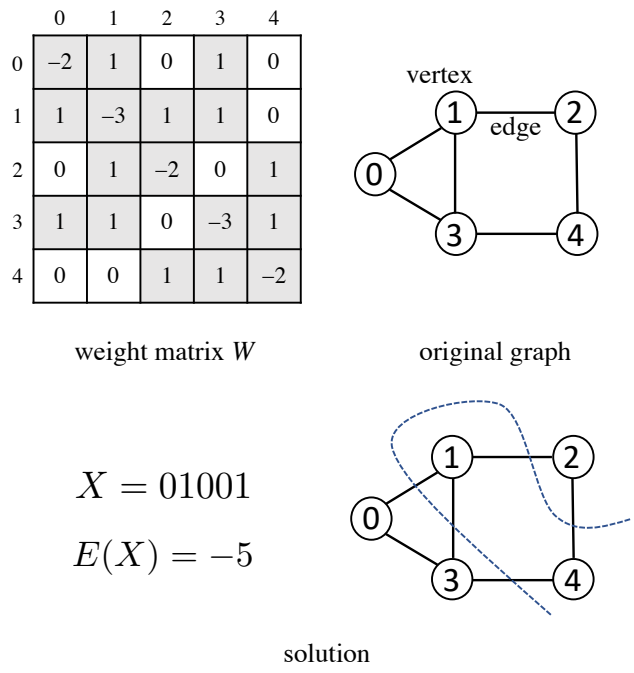

Figure 6: An example of Max-Cut formulated by QUBO.

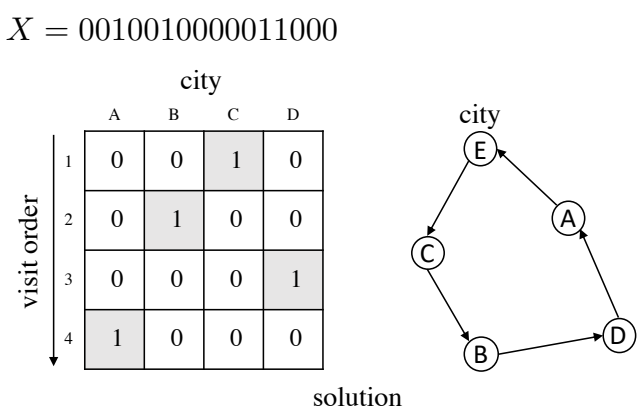

Figure 7: An example of a solution of 5-city TSP formulated by QUBO. Note that the visit order of city E is omitted for reducing the number of bits.

when a local search is used. In general, TSP QUBO problems are thus hard QUBO instances.

4.1.3 Synthetic random problems. A synthetic random problem is a QUBO problem such that all of the weights in W is randomly set in 16 bits, i.e., $W_{ij} \in [-32768, 32767]$. While the optimal solution is unknown, heuristic methods obtain a certain solution. In contrast with TSP, generated weight matrices are essentially dense, and in general they are easy problems.

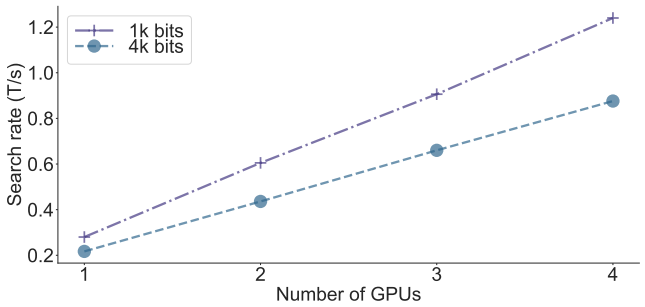

Figure 8: Scaling of the search rate when increasing the number of GPUs.

4.2 Time-to-solution

Table 1 summarizes the results of time-to-solution. Target values are regarded as the best-known solution or those of close to it. Table 1 (a) shows the results of the Max-Cut, including various graphs with 800–10000 vertices. The results show that our ABS provides the exact solution for an 800-bit problem and 95%-accurate solutions for up to 10000-bit problems within one second. G-set includes two types of graphs in terms of weights. Intuitively, weighted graphs are harder to solve than unweighted graphs, and it takes much time.

Table 1 (b) shows the results of TSPLIB. Our ABS provides the best-known solutions for problems of less than 1k bits. When the number of bits exceeds 1k, problems rapidly becomes hard, and thus we target solutions with 10% more values than best-known ones. In particular, it takes about six seconds to solve berlin52. This result suggests the hardness of TSP.

Table 1 (c) shows the results of synthetic random problems. Since these problems are randomly generated, we compute good solutions by repeating searches until convergence and regard them as best-known solutions. The results show that our ABS quickly finds good solutions for problems with 1k–32k bits.

4.3 Throughput

Table 2 shows the results of search rate for synthetic random problems with various bits per thread. Bits per thread determines the number of threads in a CUDA block and the number of active CUDA blocks in a GPU, and consequently the execution time change. For example, if the number of bits per thread increases, sequential processing in each thread increases. On the other hand, computing the minimum value between threads takes less time. The number of active blocks is automatically selected so that the occupancy becomes 100%. The search rate denotes the number of evaluated solutions per unit time introduced in [22]. The results show that our ABS attains at most 1.24T solutions search per second. It is about 60 times faster than FPGA-based QUBO solver reported in [22] with the same bit size. As shown in Figure 8, the search rate linearly increases with the number of GPUS, and thus our ABS is scalable.

Lastly, Table 3 compares our ABS and existing systems. As shown in the table, our ABS supports a large number of bits and provides high search rate and various benchmark results. A GPU implementation of the simulating adiabatic bifurcations [13] supports the maximum number of bits because spins are divided into eight GPUs,

Table 1: Results of time-to-solution.

(a) Max-Cut from G-set

Graph	# Bits (vertices)	Type	Edge weight	Target value		Time (s)
G1	800	random	+1	11624	(best-known)	0.0723
G6	800	random	±1	2178	(best-known)	0.106
G22	2000	random	+1	13225	(99% of best-known)	0.110
G27	2000	random	±1	3308	(99% of best-known)	0.721
G35	2000	planar	+1	7611	(99% of best-known)	0.208
G39	2000	planar	±1	2384	(99% of best-known)	1.89
G55	5000	random	+1	9785	(95% of best-known)	0.150
G70	10000	random	+1	9112	(95% of best-known)	0.360

(b) TSP from TSPLIB

Problem	# Bits	Target value		Time (s)
ulysses16	225	6859	(best-known)	0.11
bayg29	784	1610	(best-known)	0.69
dantzig42	1681	734	(best-known +5%)	1.25
berlin52	2601	7919	(best-known +5%)	1.79
st70	4621	742	(best-known + 10%)	4.19

(c) Synthetic random problems

# Bits	Target energy		Time (s)
1024	-182208337	(best-known)	0.0172
2048	-518114192	(best-known)	0.0413
4096	-1466369859	(best-known)	1.04
16384	-11631426556	(99% of best-known)	0.417
32768	-33115098990	(99% of best-known)	1.79

Table 2: Throughput results for synthetic random problems with 100% of occupancy.

# Bits	Bits per thread	# Threads/block	# Active blocks/GPU	Search rate (T/s)
1k	1	1024	68	0.221
	2	512	136	0.480
	4	256	272	0.924
	8	128	544	1.12
	16	64	1088	**1.24**
2k	2	1024	68	0.304
	4	512	136	0.564
	8	128	272	0.821
	16	64	544	**1.01**
	32	32	1088	0.807
4k	4	1024	68	0.407
	8	512	136	0.590
	16	256	272	**0.732**
	32	128	544	0.495
8k	8	1024	68	0.421
	16	512	136	**0.537**
	32	256	272	0.427
16k	16	1024	68	**0.578**
	32	512	136	0.513
32k	32	1024	68	**0.439**

each of which performs searches for 13,100 spins. In contrast, each GPU in our ABS independently performs local searches for all of the spins, which increases the search rate.

5 CONCLUSIONS

In this paper, we have proposed the *adaptive bulk search* (*ABS*), a framework for solving quadratic unconstrained binary optimization (QUBO) problems, which is equivalent to solving fully-connected Ising models. ABS consists of a CPU host and GPU devices and they asynchronously perform a genetic algorithm and local searches in parallel. For GPUs, we have proposed an efficient algorithm to compute the energy function using difference computation combined with genetic algorithm and a straight search. Combining their algorithms enables us to search a large number of various solutions with $O(1)$ search efficiency, i.e., $O(1)$ computational costs for evaluating one solution.

Table 3: Comparison between our system and main existing systems.

	D-Wave	Ref. [22]	Ref. [29]	Ref. [13]	Our ABS
Number of bits	2,048	1,024	4,096	100,000	32,768
Connection	Chimera graph	fully-connected	fully-connected	fully-connected	fully-connected
Search rate	N/A	20.4G	N/A	N/A	1.24T
Benchmark	N/A	TSP	Random Max-Cut	Random Max-Cut	G-set Max-Cut, TSPLIB, 16-bit synthetic random
Technology	D-Wave 2000Q	Intel Arria 10 GX FPGA	Intel Arria 10 GX1150 FPGA	NVIDIA Tesla V100-SXM2 GPU ×8	NVIDIA GeForce RTX 2080 Ti GPU ×4

We have implemented our ABS on a multi-GPU system with four GPUs. Our ABS with NVIDIA GeForce RTX 2080 Ti GPU supports QUBO problems with up to 32k variables and 16-bit weights. Our experimental results show that our ABS solves Max-Cut, TSP, and synthetic random problems. The results also show that our ABS implementation attains up to 1.24T search rate, which is 60× faster than the existing system, due to our efficient algorithms and high parallelism of multi-GPUs (at most $1088 \times 4 = 4352$ CUDA blocks perform searches in parallel). Since our ABS is scalable, more powerful systems would attain higher performance.

Future work will focus on applying our ABS to other applications and hardware. By studying a specific application, tailored algorithms for each application can be designed. As suggested in our results, the hardness of QUBO instances depends on the applications. Alternatively, an application-agnostic universal QUBO solver can be considered. To this end, each CUDA block would perform different algorithms and possibly they are changed automatically.

ACKNOWLEDGMENTS

The authors are grateful to Makoto Motoka for collaboration on the early stages of this work.

REFERENCES

[1] Mohammad H Amin. 2015. Searching for quantum speedup in quasistatic quantum annealers. *Physical Review A* 92, 5 (2015), 1–5.
[2] Frank Arute et al. 2019. Quantum supremacy using a programmable superconducting processor. *Nature* 574, 7779 (2019), 505–510.
[3] Demian A Battaglia, Giuseppe E Santoro, and Erio Tosatti. 2005. Optimization by quantum annealing: Lessons from hard satisfiability problems. *Physical Review E* 71, 6 (2005), 066707.
[4] Paul Benioff. 1980. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of statistical physics* 22, 5 (1980), 563–591.
[5] Jun Cai, William G Macready, and Aidan Roy. 2014. A practical heuristic for finding graph minors. *arXiv preprint arXiv:1406.2741* (2014).
[6] NVIDIA Corporation. 2019. *CUDA C++ programming guide (version 10.2)*.
[7] NVIDIA Corporation. 2020. NVIDIA Turing GPU architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf.
[8] Daniel Crawford, Anna Levit, Navid Ghadermarzy, Jaspreet S Oberoi, and Pooya Ronagh. 2016. Reinforcement learning using quantum Boltzmann machines. *arXiv preprint arXiv:1612.05695* (2016).
[9] Vincent Dumoulin, Ian J Goodfellow, Aaron Courville, and Yoshua Bengio. 2014. On the challenges of physical implementations of RBMs. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*.
[10] David J Earl and Michael W Deem. 2005. Parallel tempering: Theory, applications, and new perspectives. *Physical Chemistry Chemical Physics* 7, 23 (2005), 3910–3916.
[11] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. 2000. Quantum computation by adiabatic evolution. *arXiv preprint quant-ph/0001106* (2000).
[12] Gary Kochenberger, and Jin-Kao Hao and Fred Glover and Mark Lewis and Zhipeng Lü and Haibo Wang and Yang Wang. 2014. The unconstrained binary quadratic programming problem: a survey. *Journal of Combinatorial Optimization* 28, 1 (2014), 58–81.
[13] Hayato Goto, Kosuke Tatsumura, and Alexander R. Dixon. 2019. Combinatorial optimization by simulating adiabatic bifurcations in nonlinear Hamiltonian systems. *Science advances* 5, 4 (2019).
[14] Laszlo Gyongyosi and Sandor Imre. 2019. A survey on quantum computing technology. *Computer Science Review* 31 (2019), 51–71.
[15] Takahiro Inagaki, Yoshitaka Haribara, Koji Igarashi, Tomohiro Sonobe, Shuhei Tamate, Toshimori Honjo, Alireza Marandi, Peter L. McMahon, Takeshi Umeki, Koji Enbutsu, Osamu Tadanaga, Hirokazu Takenouchi, Kazuyuki Aihara, Ken ichi Kawarabayashi, Kyo Inoue, Shoko Utsunomiya, and Hiroki Takesue. 2016. A coherent Ising machine for 2000-node optimization problems. *Science* 354, 6312 (2016), 603–606.
[16] M. W. Johnson, M. H. S. Amin, S. Gildert, T. Lanting, F. Hamze, N. Dickson, R. Harris, A. J. Berkley, J. Johansson, P. Bunyk, E. M. Chapple, C. Enderud, J. P. Hilton, K. Karimi, E. Ladizinsky, N. Ladizinsky, T. Oh, I. Perminov, C. Rich, M. C. Thom, E. Tolkacheva, C. J. S. Truncik, S. Uchaikin, J. Wang, B. Wilson, and G. Rose. 2011. Quantum annealing with manufactured spins. *Nature* 473, 7346 (2011), 194–198.
[17] Tadashi Kadowaki and Hidetoshi Nishimori. 1998. Quantum annealing in the transverse Ising model. *Physical Review E* 58, 5 (1998), 5355.
[18] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. 1983. Optimization by simulated annealing. *Science* 220, 4598 (1983), 671–680.
[19] Duan Li, X. L. Sun, and C. L. Liu. 2012. An exact solution method for unconstrained quadratic 0–1 programming: a geometric approach. *Journal of Global Optimization* 52, 4 (2012), 797–829.
[20] Bas Lodewijks. 2019. Mapping NP-hard and NP-complete optimisation problems to Quadratic Unconstrained Binary Optimisation problems. *arXiv preprint arXiv:1911.08043* (2019).
[21] Andrew Lucas. 2014. Ising formulations of many NP problems. *Frontiers in Physics* 2 (2014), 5.
[22] Satoshi Matsubara, Hirotaka Tamura, Motomu Takatsu, Danny Yoo, Behraz Vatankhahghadim, Hironobu Yamasaki, Toshiyuki Miyazawa, Sanroku Tsukamoto, Yasuhiro Watanabe, Kazuya Takemoto, and Ali Sheikholeslami. 2017. Ising-model optimizer with parallel-trial bit-sieve engine. In *Proc. of the International Conference on Complex, Intelligent, and Software Intensive Systems*. 432–438.
[23] Florian Neukart, Gabriele Compostella, Christian Seidel, David Von Dollen, Sheir Yarkoni, and Bob Parney. 2017. Traffic flow optimization using a quantum annealer. *Frontiers in ICT* 4 (2017), 29.
[24] Ruprecht Karl University of Heidelberg. 2020. TSPLIB. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.
[25] Takuya Okuyama, Tomohiro Sonobe, Ken ichi Kawarabayashi, and Masanao Yamaoka. 2019. Binary optimization by momentum annealing. *Physical Review E* 100, 1 (2019).
[26] Panos M Pardalos and Somesh Jha. 1992. Complexity of Uniqueness and Local Search in Quadratic 0-1 Programming. *Oper. Res. Lett.* 11, 2 (1992), 119–123.
[27] Gerhard Reinelt. 1991. TSPLIB: A traveling salesman problem library. *ORSA journal on computing* 3, 4 (1991), 376–384.
[28] Gili Rosenberg, Poya Haghnegahdar, Phil Goddard, Peter Carr, Kesheng Wu, and Marcos López De Prado. 2016. Solving the optimal trading trajectory problem using a quantum annealer. *IEEE Journal of Selected Topics in Signal Processing* 10, 6 (2016), 1053–1060.
[29] Kosuke Tatsumura, Alexander R. Dixon, and Hayato Goto. 2019. FPGA-Based Simulated Bifurcation Machine. In *International Conference on Field Programmable Logic and Applications (FPL)*. 59–66.
[30] Masanao Yamaoka, Chihiro Yoshimura, Masato Hayashi, Takuya Okuyama, Hidetaka Aoki, and Hiroyuki Mizuno. 2015. 20k-spin Ising chip for combinational optimization problem with CMOS annealing. In *IEEE International Solid-State Circuits Conference (ISSCC)*. 1–3.
[31] Yinyu Ye. 2020. G-set. https://web.stanford.edu/~yyye/yyye/Gset/.