Solving QUBO with GPU Parallel MOPSO

Noriyuki Fujimoto
fujimoto@cs.osakafu-u.ac.jp
Osaka Prefecture University
Sakai, Osaka, Japan

Kouki Nanai
sab01101@edu.osakafu-u.ac.jp
Osaka Prefecture University
Sakai, Osaka, Japan

ABSTRACT

The Quadratic Unconstrained Binary Optimization problem (QUBO) is an NP-hard optimization problem. QUBO can be reduced from many other combinatorial optimization problems. Hence, if we can solve QUBO, we can also solve many other problems. The paper proposes a novel method to solve QUBO by reducing it into the Bi-objective Bound-constrained Continuous Optimization problem (BBCO) and then solving the BBCO with Multi-Objective Particle Swarm Optimization (MOPSO). Using 45 benchmark problem instances, the paper also shows that a GPU parallel implementation of the proposed method on the CUDA architecture finds solutions with low relative errors for the benchmark instances at most 100 variables (76% of the benchmark instances) and runs up to 202 times faster than the corresponding multi-threaded CPU program on four physical cores.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**.

KEYWORDS

quadratic unconstrained binary optimization, multi-objective optimization, particle swarm optimization, GPU computing, CUDA

ACM Reference Format:
Noriyuki Fujimoto and Kouki Nanai. 2021. Solving QUBO with GPU Parallel MOPSO. In *2021 Genetic and Evolutionary Computation Conference Companion (GECCO '21 Companion), July 10–14, 2021, Lille, France.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3449726.3463208

1 INTRODUCTION

The Quadratic Unconstrained Binary Optimization problem (QUBO) is an NP-hard optimization problem defined as follows:

$$\min_{\boldsymbol{x}} \boldsymbol{x}^t Q \boldsymbol{x} \tag{1}$$

where Q is a given n-by-n matrix, $\boldsymbol{x}$ is an n dimensional 0-1 vector (i.e., $\boldsymbol{x} \in \{0,1\}^n$), and $\boldsymbol{x}^t$ is the transposed vector of $\boldsymbol{x}$. QUBO can be reduced from the Ising problem and vice versa by a naive change of variables. The Ising problem is the only problem that Quantum Annealing machines [14, 15] (QA machines) can solve natively. QA machines have recently attracted much attention. Therefore, QUBO is also gathering attention as a target problem for QA machines. Thus, QUBO has been actively studied these days and it turned out that many important combinatorial optimization problems including the optimization counter parts of the 21 NP-complete problems shown by Karp can be reduced to QUBO [9, 20]. That is, although the formal definition of QUBO is very brief, it can represent many problems. Hence, if we can solve QUBO, then many other important combinatorial optimization problems also can be solved.

QUBO can be solved by QA machines if we reduce QUBO to the Ising problem. However, due to the connectivity limitations among quantum qubits, current QA machines can solve small problem instances only. For example, QA machines by D-Wave Systems with 2,048 qubits can solve the traveling salesman problems with at most eight cities. It is unclear that the limitations can be improved in the near future. Since QUBO is an important optimization problem, we should develop new methods to fast solve QUBO on classical (i.e., non-quantum) computers or devices without waiting for the development of QA machines.

In November 2006, GPUs that were originally developed as dedicated chips to draw images on display monitors became able to programmable not only for graphics but also for general purpose computing. The latest GPUs on the CUDA architecture [6] are many-core processors equipped with 10,496 cores and can run multi-threaded parallel programs with more than several tens of millions of threads. They provide 35.7 Tera FP32 FLOPS with relatively low cost. Therefore, GPUs have been gathering attention as high cost-performance computing platforms and have been applied to various application problems in various fields in this decade.

Recently, for the single-objective optimization problem classes of maximizing non-monotone submodular functions without constraints, maximizing submodular and approximately monotone functions with a size constraint, and maximizing monotone and approximately submodular functions with a size constraint, Qian et al. proved that multi-objective evolutionary algorithms given multi-objective optimization problem instances transformed from original single-objective problem instances can generally achieve good approximation guarantees for the original problem instances in polynomial expected time [22]. The above problem classes include maximum cut [8], maximum facility location [2], variants of the maximum satisfiability problem [8], sensor placement [18], sparse regression [7], dictionary selection [17], and Bayesian experimental design [3].

Inspired by Qian et al.'s work [22], the paper proposes a novel method to solve QUBO by reducing it into the Bi-objective Bound-constrained Continuous Optimization problem (BBCO). Note that QUBO is a discrete problem and BBCO is a continuous problem. The proposed method lets one of the two objective function be the single objective function of QUBO and uses another objective

GECCO '21 Companion, July 10–14, 2021, Lille, France

ACM ISBN 978-1-4503-8351-6/21/07…$15.00
https://doi.org/10.1145/3449726.3463208

function to force the optimal solution of the bi-objective problem to be integral.

To solve BBCO, the paper uses Multi-Objective Particle Swarm Optimization (MOPSO). In 2018, the authors developed a GPU program [12, 13] to solve BBCO with Multi-Objective Particle Swarm Optimization (MOPSO). The GPU program solves BBCO up to 182 times faster than the corresponding single-threaded CPU program. The paper shows an implementation of the proposed method with an improved version of the GPU program, which is specialized to QUBO solving.

The remainder of the paper is organized as follows. First, Section 2 briefly reviews our GPU parallel MOPSO method parallelized on the CUDA architecture. Next, Section 3 describes the proposed method and its implementation in detail. Then, Section 4 shows several experimental results to evaluate the proposed method. Finally, Section 5 gives some concluding remarks and future works.

The paper does not explain GPU programming. The readers not familiar with GPU programming are recommended to read [6].

2 A BRIEF REVIEW OF OUR GPU PARALLEL MOPSO METHOD

MOPSO is an extension of Particle Swarm Optimization (PSO) [16] to deal with multi-objective optimization problems. PSO is a population based stochastic search method for single-objective optimization problems. The population consists of particles such that each particle moves in the search space to find a better solution and maintains the best solution it found so far as its *personal best solution*. The best solution among the personal best solutions is called the *global best solution*. The velocity $\boldsymbol{v}(t)$ and the position $\boldsymbol{x}(t)$ of a particle at iteration t are updated according to its personal best solution $\boldsymbol{x}_{pbest}(t)$ and the global best solution $\boldsymbol{x}_{gbest}(t)$ respectively by the following equations where w, c_1, and c_2 are constants and r_1 and r_2 are random real numbers in $[0, 1]$:

$$\boldsymbol{v}(t+1) = w\boldsymbol{v} + c_1 r_1(\boldsymbol{x}_{pbest}(t) - \boldsymbol{x}(t)) + c_2 r_2(\boldsymbol{x}_{gbest}(t) - \boldsymbol{x}(t)) \tag{2}$$

$$\boldsymbol{x}(t+1) = \boldsymbol{x}(t) + \boldsymbol{v}(t+1) \tag{3}$$

In our experiments in Section 4, w, c_1, and c_2 are respectively fixed to 0.729, 1.4595, and 1.4595 according to [13]. The final global best solution is returned as the output of PSO.

In multi-objective optimization problems, for given two solutions, we cannot always say which one is better than another. Therefore, in general we cannot determine 'the best' solution in the multi-objective settings. Thus, MOPSO requires to extend the notion of 'the best' solution in some way. There are several ways to extend [19, 23].

Our MOPSO method [12, 13] replaces the personal best solution with a dominant solution each particle newly found and replaces the global best solution with a solution each particle randomly selected from the set (called an *archive*) of solutions non-dominated by 'the personal best solution' of the first particle. After the t loop is stopped, assuming bi-objective optimization problems, our MOPSO method sorts the solutions in the final archive in terms of one of the two objectives and then removes solutions dominated by other solutions from the final archive to generate a Pareto front. In common MOPSO methods, an archive is defined as the set of solutions non-dominated each other. However, such an archive is difficult to fast update in parallel. In contrast, although an archive computed by our MOPSO method is no more than a rough approximation of a non-dominated set, its computation can be easily parallelized and its parallel efficiency is high. That is, our MOPSO method intends to keep balance between computational time and accuracy of the computed Pareto front. Regardless of its simplicity and high speed, our MOPSO method was shown to generate a Pareto front close to the true Pareto front for some test functions [12, 13].

A pseudo code of our MOPSO method is shown in Algorithm 1. Our MOPSO solves the following BBCO:

$$\min_{\boldsymbol{x}}(f_1(\boldsymbol{x}), f_2(\boldsymbol{x})) \text{ s.t. } \boldsymbol{\ell} \le \boldsymbol{x} \le \boldsymbol{u} \tag{4}$$

where $\boldsymbol{x}$ is an n dimensional real vector such that each element is given a constant lower bound and a constant upper bound respectively by n dimensional constant real vectors $\boldsymbol{\ell}$ and $\boldsymbol{u}$. The construct **for**...**in parallel do** represents that iterations in the for loop can be executed in parallel. There are four for loops in the pseudo code. The three for loops except the for loop to count t in line 4 are executed in parallel. Note that using the scan operation [11] Step 10 of Algorithm 1 is efficiently parallelized [12, 13]. In summary, most part of our MOPSO is parallelized.

Algorithm 1 A pseudo code of our GPU parallel MOPSO method

Input: objective functions f_1 and f_2 of $\boldsymbol{x} \in \mathbb{R}^d$, constant vectors $\boldsymbol{\ell}, \boldsymbol{u} \in \mathbb{R}^d$, a number m of used particles, the maximum number t_{max} of iterations
Output: a Pareto front for multi-objective optimization problem $\min_{\boldsymbol{x}}(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}))$ s.t. $\boldsymbol{\ell} \le \boldsymbol{x} \le \boldsymbol{u}$

1: **for** $i = 1 \ldots m$ **in parallel do**
2: Initialize the position vector of the particle i randomly.
3: **end for**
4: **for** $t = 1 \ldots t_{max}$ **do**
5: **for** $i = 1 \ldots m$ **in parallel do**
6: Let $\boldsymbol{x}$ be the position vector of the particle i.
7: Compute $f_1(\boldsymbol{x})$ and $f_2(\boldsymbol{x})$.
8: Replace the personal best solution of the particle i with $\boldsymbol{x}$ if the personal best solution is dominated by $\boldsymbol{x}$.
9: **end for**
10: Let the set of all personal best solutions non-dominated by the personal best solution of the first particle (i.e., particle 1) be the current archive.
11: **for** $i = 1 \ldots m$ **in parallel do**
12: Randomly select a particle in the current archive as a substitute for the global best of the particle i.
13: Update velocities and positions of the particle i according to the velocity update equation (2) and the position update equation (3), which depends on the current personal best solutions and the current global best solution.
14: **end for**
15: **end for**
16: Sort the current archive in the ascending order of f_2.
17: Construct a Pareto front from the sorted archive.

3 THE PROPOSED METHOD

The proposed method solves a given QUBO instance $\min_{\boldsymbol{x}} \boldsymbol{x}^t Q \boldsymbol{x}$ by reducing it into the following BBCO:

$$\begin{cases} \min_{\boldsymbol{x}}(f_1(\boldsymbol{x}), f_2(\boldsymbol{x})) \text{ s.t. } \mathbf{0} \le x \le \mathbf{1} \\ f_1(\boldsymbol{x}) = \boldsymbol{x}^t Q \boldsymbol{x} \\ f_2(\boldsymbol{x}) = \sum_{i=1}^{n} \boldsymbol{x}_i(1 - \boldsymbol{x}_i) \end{cases} \tag{5}$$

where $\boldsymbol{x}$ is an n dimensional real vector, $\boldsymbol{x}_i$ is the ith element of $\boldsymbol{x}$, $\mathbf{0}$ is an n dimensional zero vector, and $\mathbf{1}$ is an n dimensional vector such that every element is one. Minimizing the objective function $f_2(\boldsymbol{x})$ guarantees that $\boldsymbol{x} \in \{0,1\}^n$. This follows because $f_2(\boldsymbol{x}) \ge 0$ for $\mathbf{0} \le \forall \boldsymbol{x} \le \mathbf{1}$ and "$f_2(\boldsymbol{x}) = 0$ if and only if $\boldsymbol{x} \in \{0,1\}^n$." The basic idea behind the proposed method is that the proposed method returns as a result a vector with the minimum f_1 value of vectors with f_2 value zero in the obtained Pareto front. If we can always obtain the true Pareto front of a given BBCO, the basic idea goes well. However, it is not realistic. Therefore, the actual implementation shown in the following takes measures to address this problem.

BBCO is solved with an improved version of our GPU parallel MOPSO described in Section 2. We specialized our MOPSO to solve QUBO. When we naively apply our MOPSO to QUBO, the most time-consuming part is evaluation of $f_1(\boldsymbol{x}) = \boldsymbol{x}^t Q \boldsymbol{x}$ for each particle. To accept any function as f_1 (i.e., f_1 is given as a parameter), evaluation of f_1 have to be naively implemented as shown in Algorithm 2. Although Algorithm 2 is embarrassingly parallelized among particles, if the time complexity of evaluation of f_1 is high, Algorithm 2 is still time-consuming. However, if $f_1(\boldsymbol{x})$ is fixed to $\boldsymbol{x}^t Q \boldsymbol{x}$, evaluation of f_1 can be implemented as shown in Algorithm 3. Notice that Algorithm 2 and Algorithm 3 have the same total work because the total work of Algorithm 2 in the case that $f_1(\boldsymbol{x}) = \boldsymbol{x}^t Q \boldsymbol{x}$ is $O(mn^2)$ and Algorithm 3 consists of matrix multiplication with the total work $O(mn^2)$ and a simple parallel loop with the total work $O(mn)$. The parallel loop can be efficiently implemented by a naive GPU kernel function. For matrix multiplication on CPUs, there exist several highly optimized implementations compatible with the famous BLAS library [1]. For matrix multiplication on GPUs, GPU vendor NVIDIA provides a highly optimized implementation compatible with the BLAS library, called cuBLAS [4]. If we use such a highly optimized implementation for matrix multiplication, Algorithm 3 runs much faster than Algorithm 2 on both a CPU and a GPU.

A pseudo code of the whole proposed method is shown in Algorithm 4. Note that the proposed method parallelizes not only matrix multiplication but also most other part of the algorithm. That is, most part of the proposed method is parallelized. The solutions in the true Pareto front of the bi-objective problem is forced to be a 0-1 vector by the second objective function f_2. However, our MOPSO does not always generate the true Pareto front. In general, our MOPSO generates an approximation of the true Pareto front. Therefore, a Pareto front generated by our MOPSO includes vectors such that each element is a real number at least zero and at most one. Hence, in Step 2 of Algorithm 4, we round off each element of every vector in the generated Pareto front to obtain the corresponding 0-1 vector. Finally, Algorithm 4 returns a 0-1 vector with the minimum f_1 value of such rounded off vectors. Ties are broken arbitrarily.

Algorithm 2 A naive evaluation of f_1

Input: a function f_1 of $\boldsymbol{x} \in \mathbb{R}^d$, a set of m particles
Output: the f_1 value for every particle in the given set
1: **for** $i = 1 \ldots m$ **in parallel do**
2: Let $\boldsymbol{x}$ be the position vector of the particle i.
3: Compute $f_1(\boldsymbol{x})$ directly.
4: **end for**

Algorithm 3 A fast evaluation of f_1 with matrix multiplication

Input: a function f_1 of $\boldsymbol{x} \in \mathbb{R}^d$, a set of m particles
Output: the f_1 value for every particle in the given set
1: Let X be an n-by-m matrix such that the column i is the position vector of the particle i.
2: Compute $Y = QX$ with a highly optimized matrix multiplication subroutine.
3: **for** $i = 1 \ldots m$ **in parallel do**
4: Let $\boldsymbol{x}$ be the position vector of particle i.
5: Compute the inner product of the ith column vector of Y and the ith column vector of X as $f_1(\boldsymbol{x})$.
6: **end for**

Algorithm 4 The proposed method

Input: a positive integer n, an n-by-n matrix Q, the maximum number t_{max} of iterations
Output: an n dimensional 0-1 vector that approximates $\min_{\boldsymbol{x}} \boldsymbol{x}^t Q \boldsymbol{x}$ where $\boldsymbol{x}$ is an n dimensional 0-1 vector
1: Using our GPU parallel MOPSO method enhanced with fast matrix multiplication , generate a Pareto front $\{\boldsymbol{s}_1, \boldsymbol{s}_2, \cdots, \boldsymbol{s}_m\}$ for BBCO $\min_{\boldsymbol{x}}(f_1(\boldsymbol{x}) = \boldsymbol{x}^t Q \boldsymbol{x}, f_2(\boldsymbol{x}) = \sum_{i=1}^{n} \boldsymbol{x}_i(1 - \boldsymbol{x}_i))$ with $0 \le \boldsymbol{x}_i \le 1$ $(i \in \{1, 2, \cdots, n\})$.
2: **return** $\boldsymbol{r}_k$ where $k = \arg\min_i f_1(\boldsymbol{r}_i)$ and $\boldsymbol{r}_i$ is an n dimensional 0-1 vector such that each element of $\boldsymbol{s}_i$ is rounded off to zero or one. Ties are broken arbitrarily.

4 EXPERIMENTS

We implemented the proposed method over a GPU based on the CUDA architecture. To evaluate its parallel performance, we implemented the proposed method also over a CPU. In this section, using 45 benchmark problem instances, we evaluate the proposed method in terms of accuracy of found solutions and the parallel performance. Accuracy is measured by the relative error of a found solution to the optimal solution. The relative error err_{rel} is defined as follows:

$$err_{rel} = (f(s) - f(x^*))/f(x^*) \tag{6}$$

where f is the objective function of QUBO, s is the solution found by our program, and x^* is the optimal solution. The parallel performance is measured by the speedup ratio, which is the ratio of execution time of our CPU program to that of our GPU program.

Table 1: The parameters to generate problem instances

problem instance	# of variables	Density	Starting Seed	Linear Coef.		Quadr. Coef.	
	(n)	(Den)	(Seed)	(c-)	(c+)	(q-)	(q+)
A1	50	0.1	10				
A2	60	0.1	10				
A3	70	0.1	10				
A4	80	0.1	10				
A5	50	0.2	10	-100	100	-100	100
A6	30	0.4	10				
A7	30	0.5	10				
A8	100	0.0625	10				
B1	20	1	10				
B2	30	1	10				
B3	40	1	10				
B4	50	1	10				
B5	60	1	10				
B6	70	1	10	0	100	-63	0
B7	80	1	10				
B8	90	1	10				
B9	100	1	10				
B10	125	1	10				
C1	40	0.8	10				
C2	50	0.6	70				
C3	60	0.4	31				
C4	70	0.3	34	-50	50	-100	100
C5	80	0.2	8				
C6	90	0.1	80				
C7	100	0.1	142				
D1	100	0.1	31				
D2	100	0.2	37				
D3	100	0.3	143				
D4	100	0.4	47				
D5	100	0.5	31				
D6	100	0.6	47	-50	50	-75	75
D7	100	0.7	97				
D8	100	0.8	133				
D9	100	0.9	307				
D10	100	1	1311				
E1	200	0.1	51				
E2	200	0.2	43				
E3	200	0.3	34	-50	50	-100	100
E4	200	0.4	73				
E5	200	0.5	89				
F1	500	0.1	137				
F2	500	0.25	137				
F3	500	0.5	137	-50	50	-75	75
F4	500	0.75	137				
F5	500	1	137				

4.1 The Used Benchmark Problem Instances

For computational experiments about QUBO, an approach followed by many researchers is to generate random instances by using a test problem generator (P&R generator) [21] introduced by Pardalos and Rodgers [24]. The algorithm of P&R generator is strictly defined and opened, including the random number generator used. Therefore, we can reproduce any random instance generated by P&R generator if we know the used input parameter for P&R generator. According to the approach, in order to evaluate the proposed method, we used the test problems in the families A, B, C, D, E, and F proposed by Glover et al. [10]. The number of variables of generated QUBO instances ranges from 20 to 500. The used parameters to generate the problem instances with P&R generator are shown in Table 1.

4.2 Experimental Setup

Experiments were conducted using our GPU and CPU server. Our GPU server has an NVIDIA TITAN V (5120 cores, 12GB VRAM), a 2.8GHz Intel Core i7-6700T (8MB L3 cache, 4 physical cores), 16GB main memory, and Windows 10 Pro. Our CPU server has a 3.0GHz Intel Xeon E3-1220V5 (8MB L3 cache, 4 physical cores), 16GB main memory, and Windows Server 2019 Standard. Our GPU program was written in CUDA C/C++ and our CPU program was written in the standard C language. For compilation, we used Microsoft Visual Studio 2019 Community and CUDA Toolkit 11.2 [5]. Matrix multiplication was implemented by cuBLAS included in the CUDA toolkit for our GPU program and OpenBLAS 0.3.13 [25] for our CPU program. OpenBLAS is a highly optimized open source BLAS implementation. We implemented two versions of our CPU program. One is the single-threaded version which uses the single-threaded version of OpenBLAS. The other is the multi-threaded version which uses the multi-threaded version of OpenBLAS. Note that our multi-threaded CPU program uses multiple threads only in matrix multiplication implemented by OpenBLAS. That is, our multi-threaded CPU program parallelizes only the most time-consuming part of the program and the other parts remain single-threaded. In contrast, our GPU program uses many threads for the whole program, i.e., not only for matrix multiplication implemented by cuBLAS but also computing archive, updating velocities and positions of particles, and so on.

4.3 Experimental Results

To evaluate the parallel performance of our GPU program by comparing the corresponding CPU program, we show in Table 2 execution time of our single-threaded CPU program and our multi-threaded CPU program for 1,024 to 32,768 particles and 2,500 iterations. For smaller problem instances, the single-threaded program is faster than the multi-threaded program. However, for larger problem instances, which are more time-consuming, the multi-threaded program runs faster than the single-threaded program. Therefore, in the following, we compare our GPU program only with our multi-threaded CPU program.

Accuracy of the solution of QUBO generated by the proposed GPU parallel MOPSO is shown in Table 3. Table 3 shows relative errors of solutions found by our GPU program and CPU program when we use 1,024 to 32,768 particles and 2,500 iterations in our MOPSO. The relative errors of our GPU program are not completely identical to those of our CPU program. This is because parallelization of matrix multiplication of floating point number changes the order of executed arithmetic operations. The difference of the order yields the difference of rounding errors and truncation errors in floating point arithmetic and thus changes trajectories of particles between our GPU MOPSO and our CPU MOPSO. Roughly speaking, the relative errors tend to decrease with an increase of the number of used particles. The proposed method found optimal solutions of QUBO for 16 of the 45 problem instances when we use 32,768 particles. 32 (respectively 27) of the 45 problem instances have relative errors at most 2% under the same condition in the case of GPU (respectively CPU). If the number of variables is at most 100, accuracy of the proposed method seems to be relatively good. However, accuracy for the problem instances with 500 variables is

Table 2: Execution time of our single-threaded CPU program and our multi-threaded CPU program.

		# of particles											
		1,024		2,048		4,096		8,192		16,384		32,768	
problem instance	# of variables	single (s)	multi (s)	single (s)	multi (s)	single (s)	multi (s)	single (s)	multi (s)	single (s)	multi (s)	single (s)	multi (s)
A1	50	2.68	2.70	5.11	5.13	9.16	9.40	17.79	31.24	37.80	36.38	76.34	84.75
A2	60	3.09	3.00	5.16	5.04	11.45	11.14	22.79	23.37	50.15	46.61	94.88	102.65
A3	70	3.74	3.41	7.81	7.75	11.73	13.47	27.01	26.50	53.10	53.41	107.05	119.73
A4	80	4.21	3.84	12.30	9.23	15.67	15.00	40.82	35.32	63.27	66.93	133.03	129.01
A5	50	2.62	2.56	4.59	8.30	9.51	9.18	18.03	17.25	36.55	35.43	78.50	81.85
A6	30	2.52	2.52	7.81	7.77	15.36	15.23	31.18	30.83	63.15	62.52	125.42	124.68
A7	30	3.27	3.26	3.98	3.73	5.65	5.51	11.09	10.68	24.17	23.60	44.59	46.36
A8	100	5.40	5.00	10.73	10.07	19.85	20.32	55.54	42.87	91.44	84.90	162.89	164.39
B1	20	2.24	2.26	4.99	5.00	2.75	2.73	20.93	20.86	40.42	40.19	79.50	79.13
B2	30	4.10	4.10	5.42	5.39	12.23	12.08	28.46	28.13	39.18	38.54	94.25	93.29
B3	40	4.89	4.90	4.07	4.04	19.07	18.99	39.64	39.47	59.46	59.37	146.52	148.18
B4	50	6.64	6.58	10.69	7.45	25.27	24.98	50.48	50.31	98.31	98.03	199.52	200.13
B5	60	2.59	2.45	15.28	15.13	27.29	27.73	61.33	60.72	129.38	128.74	239.74	240.87
B6	70	9.20	9.06	4.72	4.50	31.99	31.47	53.73	55.91	117.85	122.33	122.47	103.50
B7	80	10.31	10.17	20.01	19.66	41.28	40.82	65.81	60.79	161.64	160.48	320.07	314.43
B8	90	11.70	10.19	22.59	22.15	45.71	46.53	143.76	95.63	57.58	75.75	348.34	296.02
B9	100	6.09	5.79	27.75	22.38	62.32	49.28	68.02	55.31	160.69	142.58	374.28	407.27
B10	125	23.46	16.38	54.50	35.86	115.59	74.85	218.58	148.81	366.53	276.37	862.20	549.81
C1	40	2.15	2.14	8.30	8.26	8.46	8.32	32.27	35.98	62.54	66.63	125.73	64.81
C2	50	6.34	6.17	12.70	12.56	24.40	24.71	47.75	26.78	50.90	56.66	155.05	97.61
C3	60	2.88	2.82	6.09	6.22	11.18	11.03	27.48	18.06	47.81	58.13	141.97	115.70
C4	70	3.69	3.56	12.54	10.67	13.80	13.43	29.94	28.70	60.59	60.55	119.12	122.57
C5	80	4.14	3.99	7.69	7.35	21.29	19.50	31.16	30.60	59.46	73.91	128.56	128.79
C6	90	4.62	4.38	9.30	8.64	18.88	17.70	37.68	35.96	75.87	70.82	153.21	155.26
C7	100	5.37	5.08	9.81	9.26	20.54	19.58	42.26	41.86	82.44	80.64	156.80	152.77
D1	100	4.52	5.03	10.51	9.21	19.97	18.73	42.61	41.15	80.94	84.07	175.21	165.07
D2	100	11.93	10.59	21.31	18.89	22.49	22.90	44.08	40.70	77.71	95.20	170.78	165.21
D3	100	11.11	11.44	21.80	23.67	50.49	48.83	90.72	84.70	202.88	190.09	301.64	330.88
D4	100	10.46	7.85	18.73	18.60	33.05	29.93	57.56	80.42	106.42	96.57	174.12	174.69
D5	100	9.94	11.85	15.32	11.54	19.09	17.29	41.29	41.08	84.28	83.99	238.54	176.32
D6	100	10.28	11.27	16.97	16.21	21.94	21.19	41.21	35.00	146.06	145.57	290.48	247.55
D7	100	5.34	6.68	10.59	9.43	20.58	18.72	40.71	39.62	82.52	81.41	170.28	287.25
D8	100	9.56	10.03	21.80	21.93	35.97	30.75	78.70	73.31	154.41	107.92	154.14	145.93
D9	100	11.96	10.07	10.37	9.77	41.79	41.43	78.97	89.98	174.04	124.06	320.12	286.96
D10	100	4.51	4.99	9.56	8.85	22.32	17.08	46.09	50.50	91.69	84.16	180.32	180.09
E1	200	12.17	15.21	26.51	25.32	47.22	41.22	94.90	85.03	188.90	177.40	376.83	330.55
E2	200	11.04	9.66	22.75	20.13	46.16	41.09	93.15	84.24	189.08	171.16	381.15	345.55
E3	200	18.82	14.79	30.19	24.00	46.74	40.76	99.81	86.83	241.24	212.08	438.73	336.11
E4	200	11.28	9.62	23.65	23.73	61.12	49.86	95.48	86.07	181.41	167.00	370.85	327.08
E5	200	11.07	9.66	22.48	19.73	47.27	42.29	93.34	84.02	187.03	169.49	375.18	339.74
F1	500	36.19	26.75	72.82	54.16	144.91	108.16	289.26	215.95	582.28	436.88	1160.77	868.64
F2	500	36.75	27.32	73.75	55.07	147.44	110.64	295.13	222.04	590.48	445.79	1181.01	890.57
F3	500	36.75	27.34	73.84	55.16	147.58	110.75	294.90	221.86	590.22	445.25	1182.58	891.71
F4	500	36.74	27.31	73.74	55.06	147.56	110.79	295.33	222.13	590.75	445.05	1181.80	891.87
F5	500	36.75	27.31	73.70	55.02	147.62	110.77	295.49	222.35	591.09	445.65	1181.44	891.38

much worse than other problem instances. Thus, we need further improvement of the proposed method.

Table 4 shows execution time and speedup ratios of our GPU program to the multi-threaded CPU program for 1,024 to 32,768 particles and 2,500 iterations. Our GPU program runs much faster than our multi-threaded CPU program on four physical cores. The problem instances with up to 100 variables runs within only a few seconds even if we use 32,768 particles. The maximum speedup ratio to the multi-threaded CPU program is about 202.

5 CONCLUSION AND FUTURE WORK

We have presented a novel method to solve QUBO, which seems to be one of the most important combinatorial optimization problems. The proposed method is suitable to be efficiently parallelized. We have also implemented the proposed method on the GPU computing platform based on the CUDA architecture. Furthermore, we have empirically confirmed that QUBO can be approximately solved fast for many of 45 benchmark problem instances.

Future work includes to improve accuracy of the proposed method for larger problem instances with at least 500 variables and to compare the proposed method with other methods.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Numbers 17K00171, 20K11842.

Table 3: Relative errors of solutions found by our GPU program and our multi-threaded CPU program.

		# of particles											
		1024		2048		4096		8192		16384		32768	
problem instance	# of variables	GPU (%)	multi (%)	GPU (%)	multi (%)	GPU (%)	multi (%)	GPU (%)	multi (%)	GPU (%)	multi (%)	GPU (%)	multi (%)
A1	50	0.76	2.90	0.00	0.00	0.76	0.76	0.00	0.76	0.00	0.76	0.00	0.76
A2	60	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
A3	70	4.52	5.17	1.06	3.99	3.28	3.33	3.33	3.33	1.49	3.33	1.14	3.33
A4	80	1.77	5.59	1.77	1.99	1.77	3.05	1.76	1.99	1.77	1.99	1.76	1.77
A5	50	3.35	4.93	2.79	4.97	3.35	3.38	2.48	3.35	3.35	3.35	2.58	3.35
A6	30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
A7	30	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
A8	100	3.47	3.31	1.28	3.55	0.07	0.98	0.81	0.07	0.00	0.07	0.00	0.07
B1	20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
B2	30	0.00	2.48	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
B3	40	13.56	13.56	13.56	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
B4	50	0.00	31.01	8.53	0.00	0.00	20.16	0.00	0.00	0.00	0.00	0.00	0.00
B5	60	24.67	22.67	29.33	0.00	0.00	8.67	0.00	0.00	0.00	0.00	0.00	0.00
B6	70	50.68	40.41	28.77	19.86	22.60	34.25	18.49	29.45	18.49	22.60	0.00	0.00
B7	80	60.63	63.13	35.63	28.13	36.88	60.63	0.00	21.88	0.00	0.00	0.00	0.00
B8	90	57.24	60.00	45.52	26.90	27.59	56.55	51.03	27.59	15.17	16.55	16.55	8.28
B9	100	100.00	100.00	48.18	54.01	54.01	100.00	25.55	15.33	62.04	37.96	12.41	7.30
B10	125	85.71	100.00	100.00	100.00	100.00	100.00	100.00	100.00	60.39	62.99	61.04	100.00
C1	40	0.00	3.34	0.00	0.00	0.00	3.26	0.00	0.00	0.00	0.00	0.00	0.00
C2	50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
C3	60	0.00	1.35	1.35	1.35	0.00	1.35	0.00	0.00	0.00	0.00	0.00	0.00
C4	70	1.23	3.53	0.07	0.07	0.07	2.05	0.00	2.89	0.00	0.00	0.00	0.00
C5	80	2.76	5.00	2.36	3.60	1.75	0.43	1.63	1.63	0.43	0.43	0.43	0.43
C6	90	7.90	7.54	1.68	4.19	1.60	5.91	1.56	5.08	1.06	1.60	1.06	2.27
C7	100	3.35	6.87	2.69	3.76	3.40	6.67	1.91	6.48	1.69	1.98	1.98	3.11
D1	100	3.27	3.82	3.16	3.17	2.56	3.16	1.64	1.91	1.50	2.97	0.65	2.97
D2	100	1.69	1.25	3.66	1.87	3.07	1.66	1.64	1.82	1.66	1.66	1.64	1.66
D3	100	2.53	0.77	0.60	0.77	0.52	0.60	0.73	0.60	0.60	0.60	0.60	0.60
D4	100	1.25	2.15	0.90	2.15	1.60	2.15	2.15	1.34	0.95	2.15	1.34	2.11
D5	100	6.54	5.69	3.97	6.29	4.15	6.97	0.73	4.47	0.73	4.96	3.22	3.58
D6	100	4.03	4.50	1.79	4.86	3.96	3.53	1.65	2.79	1.65	1.79	1.65	1.65
D7	100	8.68	8.21	6.49	8.86	6.47	7.50	5.85	7.87	5.75	7.35	5.24	6.50
D8	100	3.52	0.52	2.62	2.97	0.34	2.34	0.83	0.52	0.83	2.07	0.83	0.83
D9	100	2.73	3.09	1.02	2.60	0.63	0.59	0.63	0.00	0.63	0.89	0.63	1.76
D10	100	3.59	7.25	1.02	3.83	0.79	4.09	0.99	2.33	0.81	2.00	0.81	0.90
E1	200	3.75	1.66	0.93	0.63	0.77	0.76	0.49	0.59	0.76	0.42	0.76	0.63
E2	200	6.08	9.14	20.08	3.55	14.58	2.53	22.00	2.77	20.38	13.23	10.34	7.10
E3	200	3.70	3.42	4.58	2.41	2.23	2.56	3.14	2.27	5.50	2.41	1.90	1.38
E4	200	4.88	6.03	1.34	4.51	1.87	4.26	1.45	2.27	2.17	1.89	1.17	1.88
E5	200	5.29	5.22	21.64	4.15	15.64	4.27	13.13	4.32	17.74	4.42	9.19	3.99
F1	500	49.67	45.32	46.88	43.68	42.50	43.32	41.88	39.21	43.94	43.72	43.54	42.06
F2	500	50.10	49.27	51.20	52.36	53.53	45.14	48.88	44.54	53.70	49.12	44.13	42.83
F3	500	52.39	51.29	53.27	30.52	53.09	51.59	51.72	47.44	50.66	48.16	49.44	52.98
F4	500	56.39	55.38	49.23	54.95	53.13	49.74	51.13	53.74	50.74	48.70	50.36	47.84
F5	500	56.13	53.78	60.64	55.07	61.32	52.37	51.54	56.18	52.87	55.04	51.64	48.83

REFERENCES

[1] 2021. *BLAS (Basic Linear Algebra Subprograms).* Retrieved April 12, 2021 from http://www.netlib.org/blas/

[2] Alexander A. Ageev and Maxim I. Sviridenko. 1999. An 0.828-approximation algorithm for the uncapacitated facility location problem. *Discrete Applied Mathematics* 93, 2 (1999), 149–156.

[3] Kathryn M. Chaloner and Isabella Verdinelli. 1995. Bayesian experimental design: A review. *Statist. Sci.* 10, 3 (1995), 273–304.

[4] NVIDIA Corp. 2021. *cuBLAS.* Retrieved April 12, 2021 from https://docs.nvidia.com/cuda/cublas/index.html

[5] NVIDIA Corp. 2021. *CUDA Toolkit.* Retrieved April 12, 2021 from https://developer.nvidia.com/cuda-toolkit

[6] NVIDIA Corp. 2021. *CUDA toolkit documentation.* Retrieved April 12, 2021 from https://docs.nvidia.com/cuda/

[7] Abhimanyu Das and David Kempe. 2011. Submodular meets spectral: Greedy algorithms for subset selection, sparse approximation and dictionary selection. In *Proc. of the 28th International Conference on Machine Learning (ICML).* 1057–1064.

[8] Michael R. Garey and David S. Johnson. 1990. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., USA.

[9] Fred Glover, Gary Kochenberger, and Yu Du. 2018. A tutorial on formulating and using QUBO models. , Article arXiv:1811.11538 (2018).

[10] Fred Glover, Garey A. Kochenberger, and Bahram Alidaee. 1998. Adaptative memory tabu search for binary quadratic programs. *Management Science* 44, 3 (1998), 336–345.

[11] Mark Harris, Shubhabrata Sengupta, and John D. Owens. 2007. *Parallel Prefix Sum (Scan) with CUDA.* GPU Gems, Vol. 3. Addison-Wesley Professional, Chapter 39.

[12] Md Maruf Hussain and Noriyuki Fujimoto. 2018. Parallel Multi-Objective Particle Swarm Optimization for Large Swarm and High Dimensional Problems. In *Proc. of the 2018 IEEE Congress on Evolutionary Computation (CEC).* 1–10.

[13] Md Maruf Hussain and Noriyuki Fujimoto. 2020. GPU-based Parallel Multi-objective Particle Swarm Optimization for Large Swarms and High Dimensional Problems. *Parallel Comput.* 92, Article 102589 (2020), 19 pages.

[14] M. W. Johnson, M. H. S. Amin, and S. Gildert et al. 2011. Quantum annealing with manufactured spins. *Nature* 473 (2011), 194–198.

Table 4: Execution time and speedup ratios of our GPU program to the multi-threaded CPU program.

		# of particles											
		1,024		2,048		4,096		8,192		16,384		32,768	
problem instance	# of variables	time (s)	speedup to multi	time (s)	speedup to multi	time (s)	speedup to multi	time (s)	speedup to multi	time (s)	speedup to multi	time (s)	speedup to multi
A1	50	0.71	3.82	0.68	7.55	0.72	13.04	0.78	40.14	0.95	38.40	1.32	64.31
A2	60	0.67	4.47	0.68	7.38	0.73	15.31	0.81	28.83	0.99	46.88	1.44	71.15
A3	70	0.69	4.97	0.70	11.10	0.76	17.64	0.88	30.22	1.09	48.93	1.66	72.31
A4	80	0.73	5.25	0.76	12.22	0.81	18.46	0.92	38.43	1.14	58.92	1.67	77.11
A5	50	0.66	3.90	0.68	12.27	0.71	12.93	0.77	22.35	0.93	38.21	1.31	62.41
A6	30	0.64	3.91	0.65	11.99	0.66	22.92	0.71	43.26	0.79	78.89	1.04	120.38
A7	30	0.64	5.10	0.65	5.76	0.66	8.33	0.71	14.99	0.79	29.81	1.03	45.12
A8	100	0.72	6.91	0.76	13.21	0.85	23.86	1.01	42.50	1.32	64.28	2.24	73.46
B1	20	0.63	3.58	0.64	7.85	0.65	4.19	0.67	30.99	0.75	53.88	0.90	87.82
B2	30	0.64	6.42	0.65	8.26	0.66	18.19	0.71	39.64	0.80	48.46	1.03	90.31
B3	40	0.65	7.50	0.67	6.06	0.69	27.56	0.75	52.95	0.86	68.76	1.16	127.55
B4	50	0.66	10.05	0.67	11.08	0.71	35.17	0.77	65.24	0.93	105.64	1.32	151.84
B5	60	0.67	3.68	0.68	22.16	0.72	38.25	0.80	75.94	0.98	130.83	1.42	169.86
B6	70	0.68	13.34	0.70	6.47	0.77	40.83	0.86	64.71	1.10	111.66	1.66	62.25
B7	80	0.73	14.01	0.75	26.18	0.82	50.08	0.92	66.27	1.12	142.98	1.68	186.97
B8	90	0.71	14.37	0.73	30.36	0.81	57.48	0.95	100.32	1.21	62.68	1.96	150.99
B9	100	0.73	7.91	0.75	29.67	0.85	57.95	1.01	54.95	1.32	107.72	2.25	181.41
B10	125	0.75	21.95	0.82	43.74	0.91	82.14	1.11	134.33	1.53	181.22	2.71	202.78
C1	40	0.66	3.26	0.67	12.35	0.68	12.23	0.75	48.21	0.86	77.81	1.17	55.61
C2	50	0.66	9.34	0.67	18.71	0.71	34.92	0.77	34.69	0.92	61.41	1.30	74.92
C3	60	0.67	4.20	0.68	9.11	0.73	15.20	0.80	22.46	0.98	59.07	1.42	81.55
C4	70	0.68	5.21	0.70	15.31	0.77	17.46	0.86	33.17	1.10	55.23	1.64	74.59
C5	80	0.73	5.46	0.75	9.81	0.82	23.81	0.92	33.17	1.14	64.89	1.68	76.62
C6	90	0.71	6.14	0.74	11.71	0.81	21.78	0.93	38.65	1.22	58.15	1.96	79.24
C7	100	0.73	6.97	0.76	12.18	0.86	22.84	1.01	41.54	1.33	60.83	2.22	68.78
D1	100	0.72	6.94	0.76	12.13	0.85	22.05	1.01	40.87	1.33	63.25	2.24	73.76
D2	100	0.73	14.58	0.77	24.65	0.86	26.69	1.01	40.37	1.33	71.53	2.22	74.27
D3	100	0.72	15.85	0.76	31.34	0.85	57.24	1.00	84.39	1.32	143.58	2.23	148.56
D4	100	0.73	10.80	0.76	24.47	0.85	35.07	1.00	80.19	1.33	72.47	2.25	77.73
D5	100	0.72	16.43	0.76	15.10	0.85	20.32	1.01	40.60	1.32	63.41	2.24	78.84
D6	100	0.73	15.52	0.77	21.13	0.85	24.90	1.00	34.89	1.33	109.11	2.23	111.01
D7	100	0.72	9.23	0.76	12.41	0.85	21.90	1.01	39.24	1.33	61.24	2.22	129.21
D8	100	0.72	13.93	0.76	28.86	0.86	35.95	1.01	72.70	1.32	81.47	2.24	65.17
D9	100	0.72	13.98	0.76	12.80	0.85	48.62	1.01	89.49	1.33	93.04	2.22	129.26
D10	100	0.73	6.84	0.76	11.64	0.85	19.99	1.01	50.21	1.33	63.41	2.23	80.91
E1	200	0.89	17.05	1.01	25.10	1.13	36.62	1.43	59.41	2.12	83.74	4.21	78.44
E2	200	0.89	10.83	1.01	20.03	1.12	36.80	1.42	59.48	2.11	81.30	4.18	82.67
E3	200	0.89	16.65	1.01	23.87	1.12	36.36	1.43	60.84	2.11	100.33	4.24	79.18
E4	200	0.90	10.70	1.01	23.49	1.12	44.33	1.42	60.69	2.11	79.14	4.17	78.38
E5	200	0.90	10.77	1.00	19.67	1.12	37.92	1.42	59.31	2.10	80.53	4.17	81.55
F1	500	1.30	20.59	1.59	34.00	2.07	52.30	3.05	70.81	5.33	81.91	11.61	74.85
F2	500	1.30	20.97	1.60	34.50	2.06	53.65	3.04	73.00	5.32	83.76	11.60	76.75
F3	500	1.30	21.05	1.59	34.64	2.07	53.59	3.04	72.88	5.33	83.52	11.61	76.79
F4	500	1.30	21.03	1.60	34.34	2.07	53.60	3.05	72.92	5.33	83.42	11.61	76.79
F5	500	1.30	21.07	1.60	34.49	2.06	53.67	3.05	73.02	5.33	83.62	11.61	76.76

[15] Tadashi Kadowaki and Hidetoshi Nishimori. 1998. Quantum annealing in the transverse Ising model. *Physical Review E* 58, 5 (1998), 5355–5363.

[16] James Kennedy and Russell C. Eberhart. 1995. Particle Swarm Optimization. In *Proc. of International Conference on Neural Networks (ICNN)*. 1942–1948.

[17] Andreas Krause and Volkan Cevher. 2010. Submodular dictionary selection for sparse representation. In *Proc. of 27th International Conference on Machine Learning (ICML)*. 567–574.

[18] Andreas Krause, Ajit Paul Singh, and Carlos Guestrin. 2008. Near-optimal sensor placements in Gaussian processes: Theory, efficient algorithms and empirical studies. *Journal of Machine Learning Research* 9 (2008), 235–284.

[19] Soniya Lalwani, Sorabh Singhal, Rajesh Kumar, and Nilama Gupta. 2013. A comprehensive survey: applications of multi-opbjective particle swarm optimization (MOPSO) algorithm. *Transactions on Combinatorics* 2, 1 (2013), 39–101.

[20] Andrew Lucas. 2014. Ising formulations of many NP problems. *Frontiers in Physics* 2 (2014).

[21] Panos M. Pardalos and Gregory P. Rodgers. 1990. Computational aspects of a branch and bound algorithm for quadratic 0-1 programming. *Computing* 45 (1990), 131–144.

[22] Chao Qian, Yang Yu, Ke Tang, Xin Yao, and Zhi-Hua Zhou. 2019. Maximizing submodular or monotone approximately submodular functions by multi-objective evolutionary algorithms. *Artificial Intelligence* 275 (2019), 279–294.

[23] Margarita Reyes-Sierra and Carlos A. Coello Coello. 2006. Multi-objective particle swarm optimizers: a survey of the state-of-the-art. *International Journal of Computational Intelligence Research* 2, 3 (2006), 287–308.

[24] Gabriel Tavares. 2008. *New algorithms for Quadratic Unconstrained Binary Optimization (QUBO) with applications in engineering and social sciences.* Ph.D. Dissertation. New Brunswick Rutgers, the State University of New Jersey.

[25] Zhang Xianyi. 2021. *OpenBLAS.* Retrieved April 12, 2021 from https://www.openblas.net/