Improved Survey Propagation on Graphics Processing Units

Yang Zhao(✉), Jingfei Jiang, and Pengbo Wu

National University of Defense Technology, Changsha, Hunan, China
{zhaoyang10nudt,pengbo026}@163.com, jingfeijiang@126.com

Abstract. The development of graphic processing units (GPUs) ensures a significant improvement in parallel computing performance. However, it also leads to an unprecedented level of complexity in algorithm design because of its physical architecture. In this paper, we propose an improved survey propagation (SP) algorithm to solve the Boolean satisfiability problem on GPUs. SP is a CPU-based incomplete algorithm that can solve hard instances of k-CNF problems with large numbers of variables. In accordance with the analysis on NVIDIA Kepler GPU architecture, a more efficient algorithm is designed with methods of changing data flow, parallel computing, and hiding communication. For NVIDIA K20c and Intel Xeon CPU E5-2650, our proposed algorithm can obtain speed 4.76 times faster than its CPU counterpart.

1 Introduction

Boolean satisfiability problem (SAT) plays an important role across a broad spectrum of computer science areas, including computational complexity theory [5], coding theory [8], and artificial intelligence [6,13]. Obtaining substantial assignments is an important problem in this field. The well-known k-SAT problem is a classical NP complete problem [5] for all $k \geq 3$. The problem is challenging because of the difficulty in deciding if a random formula can obtain a satisfactory assignment for a random formula [9,15]. In statistical physics, Mézard, Parisi and Zecchina proposed a new algorithm *Survey Propagation* (SP) to solve k-SAT problems [12]. SP effectively solves large-scale random k-SAT problems. Based on this advantage, many studies have been proposed in statistical physics and computer science communities [1–4].

With the development of graphic processing units (GPUs), parallel computing on GPUs have promoted all kinds of algorithms in recent years [14,16]. For SAT problems, meaningful works have been proposed. In pSATO, Zhang et al. [18] proposed a parallel and distributed solver based on their previous serial solver SATO [17]. This solver uses a master-slave model where the master aims to balance the work of the slaves to achieve acceleration. Fujii and Fujimoto [7] used GPU to speedup clause analysis. Luo and Liu [10] implemented cellular genetic algorithm and local search for 3-SAT problem based on GPU. The performance of an appropriately-designed algorithm based on GPU can be ensured. However, not all works on SP can perform effectively even in serial computing.

X. Huang et al. (Eds.): GPC 2016, LNCS 9663, pp. 30–41, 2016.
DOI: 10.1007/978-3-319-39077-2_3

Most well-known studies involve parallelized SP from the LonestarGPU benchmark suite, which is automatically parallelized by Galois. Obviously, this version has not adequately considered GPU architecture. Manolios and Zhang [11] implemented SP on GPU; however, their work was performed on an NVIDIA GTX 7900 GPU and parallel computation was considered while the time-consuming part of data transfer was ignored.

Considering the lack of research in this area, we propose an improved SP based on GPU. We focus not only on parallel computation but also on hidden communication. We first design an appropriate data structure to store propagation information based on GPU, then divide tasks for the multicore GPU to balance the load. Furthermore, stream technology is used in hiding communication, which evidently improves performance.

The rest of this paper is organized as follows. We review the k-SAT problem and SP in Sect. 2. Then, we describe our proposed algorithm in Sect. 3. Section 4 shows the results of our experiments, and the last section is conclusion.

2 Background and Problem Set-Up

In this section, we introduce notation and terminologies required in the k-SAT problem and explain SP in detail.

2.1 The k-SAT Problem and Factor Graphs

We define C as index sets for the clauses, V as index sets for variables; they satisfy $|C| = m$ and $|V| = n$. We denote variables as letters i, j, k, and so on, and clauses as a, b, c, and so on. x_s denotes the subset of variables $\{x_i : i \in S\}$. In the k-SAT problem, the clause indexed by $a \in C$ is specified by the pair $(V(a), J_a)$, where $V(a) \subset V$ consists of k elements, and $J_a := (J_{a,j} : i \in V(a))$ is a k-tuple of $\{0, 1\}$-valued weights. The clause indexed by a is *satisfied* by the assignment x if and only if $x_{V(a)} \neq J_a$. Equivalently, $\delta(y, z)$ denotes an indicator function for the event $\{y = z\}$, if we define the function

$$\Psi_{J_a}(x) := 1 - \prod_{i \in V(a)} \delta(J_{a,i}, x_i), \tag{1}$$

then the clause a is satisfied by x if and only if $\Psi_{J_a}(x) = 1$. The overall formula consists of the AND of all the individual clauses, and is satisfied by x if and only if $\prod_{a \in C} \Psi_{J_a}(x) = 1$.

We call the graphical representation of any k-SAT problem provided by the formalism of constraints as factor graphs. As illustrated in Fig. 1, we use circular nodes to describe variables, square nodes to describe clauses. If variable i is in clause a, there is an edge (a, i). If variable i in clause a is a positive presentation, the edge (a, i) is a solid line, while for negative presentation, the edge is a dotted line.

For each $i \in V$, we define the set $C(i) := \{a \in C : i \in V(a)\}$, which corresponds to clauses that impose constraints on variable x_i. This set of clauses

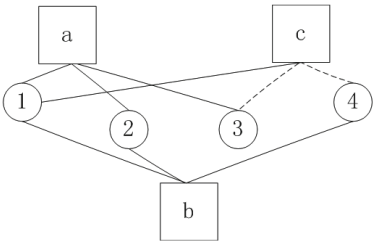

Fig. 1. An example of a factor graph with 4 variable nodes $i = 1, \ldots, 4$, 3 function nodes a, b, c. The formula which is encoded is : $F = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \bar{x}_3 \vee \bar{x}_4)$.

can be decomposed into two disjoint subsets, according to whether the clause is satisfied by $x_i = 0$ or $x_i = 1$ respectively.

$$C^-(i) := \{a \in C(i) : J_{a,i} = 1\} \tag{2}$$

$$C^+(i) := \{a \in C(i) : J_{a,i} = 0\} \tag{3}$$

Moreover, for each pair $(a, i) \in E$, the set $C(i) \backslash a$ can be divided into two (disjoint) subsets, depending on whether their preferred assignment of x_i *agrees* (in which case $b \in C_a^s(i)$) or *disagrees* (in which case $b \in C_a^u(i)$) with the preferred assignment of x_i corresponding to clause a. More formally, we define

$$C_a^s(i) := \{b \in C(i) \backslash \{a\} : J_{a,i} = J_{b,i}\} \tag{4}$$

$$C_a^u(i) := \{b \in C(i) \backslash \{a\} : J_{a,i} \neq J_{b,i}\}. \tag{5}$$

2.2 Survey Propagation

In contrast to the naive BP approach, a marginalization-decimation approach based on SP appears to be effective in solving random k-SAT problems even close to the threshold. In this case, we provide an explicit description of what we refer to as the SP(ρ) family of algorithms, where setting the parameter $\rho = 1$ yields the pure form of survey propagation. For any given $\rho \in [0, 1]$, the algorithm involves updating messages from clauses to variables, as well as from variables to clauses. Each clause $a \in C$ passes a real number $\eta_{a \to i} \in [0, 1]$ to each of its variable neighbors $i \in V(a)$. In the other direction, each variable $i \in V$ passes three real numbers $\prod_{i \to a} = (\prod_{i \to a}^u, \prod_{i \to a}^s, \prod_{i \to a}^*)$ to each of its clause neighbors $a \in C(i)$ (that is the set of clauses that impose constraints on variable x_i). The precise form of the updates is given as follow:

Message from clause a to variable i:

$$\eta_{a \to i} = \prod_{j \in V(a) \backslash i} \left[\frac{\prod_{j \to a}^u}{\prod_{j \to a}^u + \prod_{j \to a}^s + \prod_{j \to a}^*}\right] \tag{6}$$

Message from variable i to clause a:

$$\prod_{i \to a}^u = [1 - \rho \prod_{b \in C_a^u(i)} (1 - \eta_{b \to i})] \prod_{b \in C_a^s(i)} (1 - \eta_{b \to i}) \tag{7}$$

$$\prod\nolimits_{i\to a}^{s} = [1-\rho \prod_{b\in C_a^s(i)} (1-\eta_{b\to i})] \prod_{b\in C_a^u(i)} (1-\eta_{b\to i}) \tag{8}$$

$$\prod\nolimits_{i\to a}^{*} = [\prod_{b\in C_a^s(i)} (1-\eta_{b\to i})] \prod_{b\in C_a^u(i)} (1-\eta_{b\to i}) \tag{9}$$

The following are comments on these SP(ρ) updates:

1. Although the time-step index was omitted for simplicity, Eqs. (6–9) should be interpreted as defining a recursion on $(\eta, \prod)$. The initial values for ρ are chosen randomly in the interval $(0, 1)$.
2. The idea of the ρ parameter is to provide a smooth transition from the original naive belief propagation algorithm to the SP algorithm. In Eq. (6), setting $\rho = 0$ yields the belief propagation updates applied to the probability distribution [3], whereas setting $\rho = 1$ yields the pure version of SP.

Intuitive "Warning" Interpretation. To gain insight into these updates, the pure SP setting of $\rho = 1$ must be considered. As described by Braunstein et al. [3], the messages in this case have a natural interpretation in terms of probabilities of warnings. In particular, at time $t = 0$, we suppose that clause a sends a warning message to variable i with probability $\eta^0_{a\to i}$, and a message without a warning with probability $1 - \eta^0_{a\to i}$. After receiving all messages from clauses in $C(i)\backslash a$, variable i sends a particular symbol to clause a, which indicates that it cannot satisfy ("u"), that it can satisfy ("s"), or that it is indifferent ("*") depending on what messages it received from the other clauses. The following are the four cases:

1. If variable i receives a warning from $C_a^u(i)$ and no warning from $C_a^s(i)$, then it cannot satisfy a and sends a "u" message.
2. If variable i receives a warning from $C_a^s(i)$ but no warning from $C_a^u(i)$, then it sends an "s" message to indicate that it is inclined to satisfy the clause a.
3. If variable i receives no warnings from either $C_a^u(i)$ or $C_a^s(i)$, then it is indifferent and sends a "*" message.
4. If variable i receives warning from both $C_a^u(i)$ or $C_a^s(i)$, a contradiction has occurred.

Updates from clauses to variables are simple; in particular, any given clause sends a warning if and only if it receives "u" symbols from all the other variables.

In this context, real-valued messages involved in pure SP(1) all have natural probabilistic interpretations. In particular, the message $\eta_{a\to i}$ corresponds to the probability that clause a sends a warning to variable i. The quantity $\prod^u_{j\to a}$ and $\prod^*_{j\to a}$. Normalization by the sum $\prod^u_{j\to a} + \prod^s_{j\to a} + \prod^*_{j\to a}$ reflects the fact that the fourth case is a failure, and therefore is excluded a priori from the probability distribution. We suppose that all possible warning events were independent. In this case, the SP message update Eqs. (6–9) would be the correct estimates for the probabilities. This independence assumption is valid on a graph without cycles, and in which case the SP updates have a rigorous probabilistic interpretation. Whether or not the equations have a simple interpretation in the case $\rho \neq 1$ is not clear.

Decimation Based on SP. We suppose that these SP updates are applied and converged, and the overall conviction of a value at a given variable is computed from the incoming set of equilibrium messages as

$$\mu_i(1) \propto [1-\rho \prod_{b\in C^+(j)}(1-\eta_{b\to j})] \prod_{b\in C^-(j)}(1-\eta_{b\to j}).$$

$$\mu_i(0) \propto [1-\rho \prod_{b\in C^-(j)}(1-\eta_{b\to j})] \prod_{b\in C^+(j)}(1-\eta_{b\to j}).$$

$$\mu_i(*) \propto \prod_{b\in C^+(j)}(1-\eta_{b\to j}) \prod_{b\in C^-(j)}(1-\eta_{b\to j}).$$

To be consistent with their interpretation as (approximate) marginals, the three variables $\mu_i(0), \mu_i(*), \mu_i(1)$ at each node $i \in V$ are normalized to obtain a sum of one. We define the *bias* of a variable node as $B(i) := |\mu_i(0) - \mu_i(1)|$.

The marginalization-decimation algorithm based on SP [3] consists of the following steps:

1. Run SP(1) on the SAT problem. Extract fraction β of variables with the largest biases, and set them to their preferred values.
2. Simplify the SAT formula, and return to Step 1.

Once the maximum bias over all the variables falls below a pre-specified tolerance, the Walk-SAT algorithm is applied to the formula to find the remainder of assignment(if possible). Intuitively, the goal of the initial phases of decimation is to locate a cluster; once inside the cluster, the induced problem has a simple solution, which means that any "local" algorithm should perform effectively within a given cluster.

Algorithm Analysis. The algorithm is described as follow Algorithm 1:

The calculation in line [10] is irrelevant among different variables. The same condition applies that the calculation in line [7] is irrelevant among different clauses. However, the second part has a large number of iterations (from line [5] to line [19] except [17]), which consumes most of the execution time. All of the iterations are included in the total iteration procedure seeking for convergence. The execution time of different scales of datasets is shown in Table 1:

Optimization Target. As shown by Amdahl's law, given $n \in N$, the number of threads of execution, $B \in [0, 1]$, the fraction of the algorithm that we optimize. $T(n)$ is the time that an algorithm takes to finish execution of n threads, which corresponds to:

$$T(n) = T(1)(1 - B + \frac{1}{n}B) \tag{10}$$

Therefore, the theoretical speedup $S(n)$ that can be obtained by executing a given algorithm on a system capable of executing n threads of execution is:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)(1 - B + \frac{1}{n}B)} = \frac{1}{1 - B + \frac{1}{n}B} \tag{11}$$

Algorithm 1. Survey Propagation

procedure SURVEY PROPAGATION(n variables v, m clauses c, maximum iteration number $maxIteration$, stripe per cycle $stripe$, bias threshold $biasThreshold$)
  Initialize all messages variables received randomly
  $iterationNum \leftarrow 0$
  while not converge **and** $iterationNum < maxIteration$ **do**
    for all v **do**
      calculate the message $m_{c \to v}$
    end for
    for all c **do**
      update messages $m_{v \to c}$
    end for
    $iterationNum \Leftarrow iterationNum + 1$
    fix the $stripe$ most-biased variables
    $sofb \Leftarrow$ sum of the bias of all unfixed variables
    $sofv \Leftarrow$ sum of the number of all unfixed variables
    if $sofb/sofv < biasThreshold$ **then**
      use local algorithm and exit
    end if
  end while
end procedure

Table 1. Percentage of iteration time in execution time

Input variables	360	560	900	2000	4000	6000
Total time (s)	0.37	0.81	7.75	5.30	32.13	60.93
Iteration time (s)	0.36	0.78	7.02	4.94	29.86	56.20
(%) of iteration	97.3	95.3	90.5	93.2	92.9	92.2
Input variables	8000	10000	12000	14000	16000	18000
Total time (s)	86.20	142.81	196.13	283.58	346.67	422.59
Iteration time (s)	79.49	131.34	180.26	259.05	315.14	386.70
(%) of iteration	92.2	91.9	91.9	91.3	90.9	91.5

Speedup is directly proportional to the percentage rate of total execution time. According to the Table 1, taking the iteration procedure as the optimization target would produce the best speedup time. Thus, our group converted the two procedures to calculate messages into GPU.

3 Improved SP on GPU

In SP, iteration execution time is more than 90 % of total execution time, while iteration operations have the potential to run in parallel. Based on the analysis in Sects. 2.2 and 2.2, we present a framework of the improved SP on GPU.

3.1 Data Structure

In SP, each variable v_i may exist in several clauses, while each clause c_a may have many variables. For every v_i, its bias (i.e., true or false) needs to be calculated in all the clauses it exists in. Therefore, data correlation exists between different clauses. To avoid this problem, we use a method to make a copy of every variable in every clause and gather the messages after message collection. In the meantime, to quickly find the clauses that a certain variable exists in, we add the index of clauses to every variable.

In factor graph, we use two edges to present the indirected edge $e_{i \leftrightarrow a}$:

1. $e_{i \to a}$: edge from v_i to c_a also provides the index of variables in clause c_i.
2. $e_{a \to i}$: edge from c_a to v_i also provides the index of clauses that variable v_a exists in.

Every pair of edges in opposite directions creates a *one-to-one mapping* in massage-passing edges. By storing messages in edges, we can achieve the message-passing with no correlation using no extra memory and calculation as illustrated in Fig. 2.

3.2 Memory Hierarchy Optimization

Memory access latency is also a problem in high-performance calculation. Memory hierarchy can be classified as register, shared memory, local memory, texture and surface memory, constant memory, and global memory.

Global memory resides in device memory, which is the slowest memory on GPU.

Local memory space resides in device memory; thus, local memory access has the same high latency and low bandwidth as the global memory access and is subject to the same requirements for memory coalescence.

Texture and surface memory spaces reside in the device memory and are cached in the texture cache. Thus, a texture fetch or surface read costs one

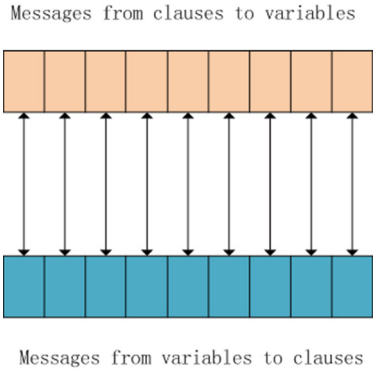

Fig. 2. In factor graph, for the indirected edge, we use two edges to present it with two directions: one is from variables to clauses, the other is from clauses to variables.

memory read from the device memory only on a cache miss. Otherwise, it only costs one read from the texture cache and is an alternative optimization option.

Shared memory is on-chip; therefore, it has significantly higher bandwidth and significantly lower latency than the local or global memory. Shared memory appears to be the best option to optimize our program. However, its disadvantage is its 48 KB capacity, which is insufficient for this program.

Consequently, we focus on maximizing the use of the register. A main difference between CPU and GPU is the method of mapping registers. CPU uses register renaming and stack to execute multiple threads. The context switch procedure must save data in registers and load new data. By contrast, GPU aims to allocate all registers to every thread. It only needs to change the pointer of the register group, which involves no cost.

3.3 Data Communication and Calculation

Communication influences parallel computing. Although communication is necessary to ensure the correctness of the program, it always consumes part of the speedup gained using multi-threads. Our group reduced communication costs by hiding communication, which clearly improves performance.

Cutting Down Communication. Transferring all data between CPU and GPU is not necessary when using GPU. We need all information on variables, clauses, and their copies in the optimization procedure, while we only need part of the information in other parts of the algorithm. Therefore, we can initialize the data of copies of clauses and transfer them to GPU *only once* and then leave it in GPU to reduce communication time.

Communication Hidden. After the messages from clauses to variable optimization procedure, the calculation of messages for every variable is performed. No change in this procedure in GPU occurs, while calculation is performed thereafter, which is needed in CPU. Copies of variables need to be transferred from GPU to CPU. Thus, we can perform parallel communication and GPU calculation as illustrated in Fig. 3.

4 Experiment

We applied our proposed methods to the data set of SAT2009. We evaluated the optimizing method using different data set sizes and achieved performance improvement. Specifically, hidden communication is different from preceding work and shows improvement, which leads to a new way of optimizing the algorithm.

4.1 Experimental Platform

We implement our improved survey propagation on an Intel Xeon CPU E5-26500 @ 2.00GHz 2.00GHz(double core) with a NVIDIA Tesla K20c GPU. The operating system consists of 64-bit Windows 7 with a total memory of 64 GB RAM. Both CPU and GPU run at high speed.

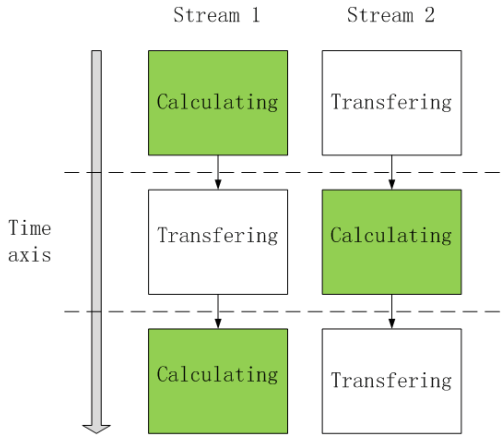

Fig. 3. Two streams can execute in parallel using one when one is calculating while the other is transfering data.

4.2 Data Structure Optimization Results

We use the data structure we previously defined to make the algorithm execute concurrently. Using parameters $BLOKS = 2, THREADS = 1024$, we obtain satisfactory results, as shown in Figs. 4 and 5.

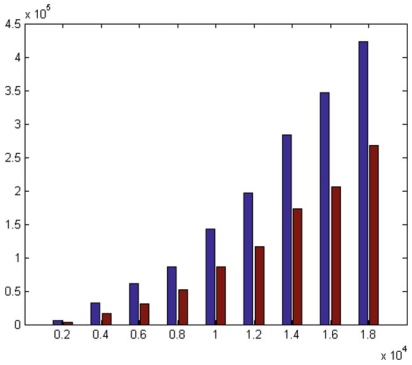

Fig. 4. The x-axis represents the number of variables, the y-axis represents the execution time(ms). The blue ones stand for CPU, the others stand for GPU only with data structure optimization. (Color figure online)

From the figures, the algorithm using our data structure running in GPU is faster than that in CPU using the entire data set. This task is only the beginning of our optimization process; speedup is highly significant because of the availability of high-speed CPU today.

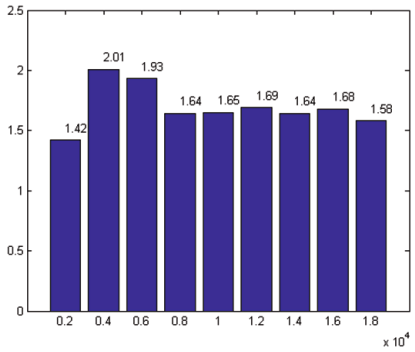

Fig. 5. The x-axis represents the number of variables, the y-axis represents the speedup of GPU to CPU.

Table 2. This table gives the influence of allocation of register by changing the combination of BLOCKS and THREADS.

Number of variables	2000	4000	6000	8000
Initial $THREADS \times BLOCKS$	2×1024	2×1024	2×1024	2×1024
Execution time of GPU (ms)	3736	15983	31539	52627
Best $THREADS \times BLOCKS$	64×128	64×256	256×64	128×128
Best execution time of GPU (ms)	1999	7686	14778	24971
Speedup to initial GPU	1.87	2.08	2.13	2.11
Number of variables	10000	12000	14000	18000
Initial $THREADS \times BLOCKS$	2×1024	2×1024	2×1024	2×1024
Execution time of GPU (ms)	86597	116021	172914	267539
Best $THREADS \times BLOCKS$	64×128	64×64	16×512	32×256
Best execution time of GPU (ms)	42097	60635	89458	127950
Speedup to initial GPU	2.06	1.91	1.93	2.09

4.3 Memory Hierarchy Optimization Results

We try different combinations of $BLOCKS$ and $THREADS$, in which GPU would change the allocation of registers to different threads.

The best performance is more surprising than the initial ones. Detailed data are reported in Table 2.

By choosing the proper allocation of registers, the entire data set performed faster, which means that memory hierarchy is necessary in optimization. The speedup to the initial combination ($BLOCKS * THREADS = 2 * 1024$) can reach up to 2.13, which cuts processing time in half, enabling the speed of our program to run at a new level.

Table 3. This is the speedup *using stream method hiding communication* to *without using stream method* when number of variables is 8000. It includes different BLOCKS and THREADS combinations. Almost all conditions are improving the performance.

BLOCKS \ THREADS	8	16	32	64	128	256	512	1024
4								1.11
8							1.05	0.93
16						1.25	1.169	0.94
32					1.11	0.92	1.04	1.12
64				1.30	1.27	1.08	1.09	1.15
128			1.27	1.20	1.06	1.08	1.21	
256		1.22	1.27	1.13	1.02			
512	1.09	1.13	1.05	1.14	1.03			

4.4 Data Communication Optimization Results

When the stream method on hidden communication is used as described in Sect. 3.3, the algorithm executes faster. Data communication optimization contributes to the data set as shown in Table 3.

Speed gain is not obvious as in other optimization methods. First, we have yet to investigate certain parallelism on data transfer and calculation. Second, hidden data transfer does not take a large amount of time. However, reducing data transfer time enables our program to run faster.

4.5 Final Optimization Results

By using all the methods we previously mentioned, we obtain satisfactory results. All the methods can be applied in one program, which improves SP to enable GPU to run faster than CPU. The final optimization results are shown in Table 4. The highest speed can reach 4.76 times faster on Intel Xeon CPU 5-2650.

Table 4. Speedup to CPU in final optimization

Number of variables	2000	4000	6000	8000	10000	12000	14000
Speedup to CPU	2.85	**4.76**	4.45	3.77	3.70	3.83	3.40

5 Conclusion

In this paper, we propose an improved SP-based GPU. According to the analysis of GPU architecture, a new data structure is defined to adapt the calculation. Then, we equally divide the tasks for every processor and use stream technology to save time on calculation and data transfer. As the experiments demonstrate, our proposed algorithm can achieve 4.76× speedup in NVIDIA K20c to Intel Xeon E5-2650.

Acknowledgements. This work is funded by National Science Foundation of China (number 61303070). Dr. Jingfei Jiang was an academic visitor at University of Manchester. We acknowledge TianHe-1A supercomputing system service.

References

1. Achlioptas, D., Moore, C.: Random k-SAT: two moments suffice to cross a sharp threshold. SIAM J. Comput. **36**(3), 740–762 (2006)
2. Braunstein, A., Mézard, M., Weigt, M., Zecchina, R.: Constraint satisfaction by survey propagation. In: Computational Complexity and Statistical Physics, p. 107 (2005)
3. Braunstein, A., Mézard, M., Zecchina, R.: Survey propagation: an algorithm for satisfiability. Random Struct. Algorithms **27**(2), 201–226 (2005)
4. Braunstein, A., Zecchina, R.: Survey propagation as local equilibrium equations. J. Stat. Mech. Theory Exp. **2004**(06), P06007 (2004)
5. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing, pp. 151–158. ACM (1971)
6. Dechter, R.: Constraint Processing. Morgan Kaufmann, Burlington (2003)
7. Fujii, H., Fujimoto, N.: GPU acceleration of BCP procedure for SAT algorithms. IPSJ SIG Notes **8**, 1–6 (2012)
8. Gallager, R.G.: Low-density parity-check codes. IRE Trans. Inf. Theory **8**(1), 21–28 (1962)
9. Levin, L.A.: Average case complete problems. SIAM J. Comput. **15**(1), 285–286 (1986)
10. Luo, Z., Liu, H.: Cellular genetic algorithms and local search for 3-SAT problem on graphic hardware. In: IEEE Congress on Evolutionary Computation, CEC 2006, pp. 2988–2992. IEEE (2006)
11. Manolios, P., Zhang, Y.: Implementing survey propagation on graphics processing units. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 311–324. Springer, Heidelberg (2006)
12. Mézard, M., Parisi, G., Zecchina, R.: Analytic and algorithmic solution of random satisfiability problems. Science **297**(5582), 812–815 (2002)
13. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, Burlington (2014)
14. Tao, T., Xuejun, Y., Yisong, L.: Locality analysis and optimization for stream programs based on iteration sequence. J. Comput. Res. Dev. **6**, 027 (2012)
15. Wang, J.: Average-case computational complexity theory. In: Complexity Theory Retrospective II, pp. 295–328 (1997)
16. Wen, M., Su, H., Wei, W., Wu, N., Cai, X., Zhang, C.: High efficient sedimentary basin simulations on hybrid CPU-GPU clusters. Cluster Comput. **17**(2), 359–369 (2014)
17. Zhang, H.: SATO: an efficient prepositional prover. In: McCune, W. (ed.) CADE 1997. LNCS, vol. 1249, pp. 272–275. Springer, Heidelberg (1997)
18. Zhang, H., Bonacina, M.P., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. J. Symbolic Comput. **21**(4), 543–560 (1996)