Vol.2012-MPS-89 No.8
2012/7/16

GPU Acceleration of BCP Procedure for SAT Algorithms

Hironori Fujii[1] Noriyuki Fujimoto[1,a)]

Abstract: The satisfiability problem (SAT) is widely applicable and one of the most basic NP-complete problems. This problem has been required to be solved as fast as possible because of its significance, but it takes exponential time in the worst case to solve. Therefore, we aim to save the computation time by parallel computing on a GPU. We propose parallelization of BCP (Boolean Constraint Propagation) procedure, one of the most effective techniques for SAT, on a GPU. For a 2.93GHz Intel Core i3 CPU and an NVIDIA GeForce GTX480, our experiment shows that the GPU accelerates our SAT solver based on our BCP-embedded divide and conquer algorithm 6.7 times faster than the CPU counterpart.

1. Introduction

The satisfiability problem (SAT for short) [1] is a problem of determining if a given boolean expression (usually in the conjunctive normal form) can be true by assigning some boolean values into the boolean variables in the expression. Formally, SAT is defined as follows:

- Let $V = \{V_1, V_2, \cdots, V_n\}$ be a set of n boolean variables.
- Let $C = \{C_1, C_2, \cdots, C_m\}$ be a set of m *clauses* where
  - $C_j = (L_{j_1} \vee L_{j_2} \vee \cdots \vee L_{j_{k_j}})$
  - $L_i \in \{V_p, \neg V_p\}$
- Question: Is there assignment to the variables in V such that $C_1 \wedge C_2 \wedge \cdots \wedge C_m =$ true ?

where $\neg$ is a logical not operator, $\vee$ is a logical or operator, $\wedge$ is a logical and operator. L_i is called a *literal* which is either affirmation or negation of a boolean variable. SAT problem such that clauses are of exactly three literals is called 3SAT. Any SAT instance can be transformed to a 3SAT instance with the same solution. Therefore, without loss of generality, for simplicity, our SAT solver accepts 3SAT instance only.

SAT is widely applicable and one of the most basic NP-complete problems. This problem has been required to be solved as fast as possible because of its significance, but it takes exponential time in the worst case to solve. Recent SAT algorithms can solve problem instances with several million variables in a few hours. SAT algorithms can be categorized into complete-type and incomplete-type. Complete-type algorithms can identify whether any given problem instance is satisfiable or unsatisfiable. In contrast, incomplete-type algorithms can not identify unsatisfiability but satisfiability. Many of common complete-type algorithms can be categorized into so-called DPLL algorithm [2], [3]. DPLL algorithm performs BCP(Boolean Constraint Propagation) procedure, which is a dominant part of the algorithm. BCP procedure occupies 80% to 90% of the whole execution. On the other hand, Davis et al. proposed a method [4] to accelerate BCP procedure by parallel processing with an FPGA. Davis et al. parallelized only BCP procedure for their method to be applicable to various DPLL algorithms. With Xilinx Virtex 5 LX110T and 3.6GHz Intel Pentium 4, Davis et al. experimentally showed that zChaff [5], [6], [7] with FPGA accelerated BCP procedure runs 5 to 16 times faster than the zChaff with the CPU only.

This paper proposes parallelization of BCP procedure on a GPU, rather than an FPGA. Similar to Davis et al.'s method, the proposed method can be used with various DPLL algorithms. However, we adopt divide-and-conquer algorithm [14] (3SAT-DC for short) with BCP procedure as our SAT solver used in the experiments in this paper. 3SAT-DC is complete-type. For a 2.93GHz Intel Core i3 CPU and an NVIDIA GeForce GTX480, our experiment showed that the GPU accelerates our SAT solver based on our BCP-embedded divide and conquer algorithm 6.7 times faster than the corresponding CPU implementation.

The remainder of this paper is organized as follows. Section 2 briefly reviews DPLL algorithm, BCP procedure, and SAT-DC. Section 3 presents the proposed algorithm. Experiments to show the performance of the proposed algorithm are reported in Section 4. Section 5 briefly surveys the related works. Section 6 gives some concluding remarks and future works. Due to the limited space, this paper include no description on CUDA. Readers unfamiliar with CUDA GPU architecture are recommended the literature [8], [9], [10], [11], [12], [13].

2. Preliminaries

2.1 DPLL Algorithm

Basically, DPLL algorithm finds an optimal solution by checking all the patterns of boolean value assignment to the variables in a given boolean expression while discarding hopeless sets of patterns without checking them. One of the methods to identify a hopeless set of patterns is BCP procedure. DPLL algorithm performs BCP procedure every *decision phase* which heuristically determines the value of a variable. Other features of DPLL al-

[1] Osaka Prefecture University, Sakai, Osaka 599–8531, Japan
[a)] fujimoto@mi.s.osakafu-u.ac.jp

© 2012 Information Processing Society of Japan

gorithm are not used in the proposed method, and therefore not described in this paper. See the literatures [2], [3] for more detail of DPLL algorithm.

2.2 BCP Procedure

BCP procedure consists of ”implication” process and ”conflict” process. The *”implication” process* finds a literal whose value is consequently determined by other literals in the same clause, and then repeat this process until no such a literal is found. The *”conflict” process* finds a clause with all literals false. In the following, literals and clauses are respectively stored in arrays named ”atom” and ”clause”. BCP procedure repeatedly calls function BCPEngine until no more implication occurs or conflict is detected. After ”implication” process, BCPEngine scans all clauses to detect implication or conflict in $O(m)$. If every literal in a clause is false, then BCPEngine finishes after updating flag variable ”conf” to indicate that conflict occurs. Otherwise, BCPEngine performs ”implication” process. If exactly one literal is unknown and any other literal is false, then BCPEngine detects implication and assigns the value which makes the unknown literal true into the corresponding element of array ”atom” .Then, BCPEngine pushes the index value of the modified element of array ”atom” to array ”implicated” and increments the stack pointer ”sp”.Finally, BCPEngine updates flag variable ”imp” to indicate the detection of implication.

2.3 3SAT-DC

Listing 1 shows a pseudo code of 3SAT-DC where the parameter ”f” of function ”3SAT-DC” is a given logical expression and notation ”f(x=propositional constant)” represents the logical expression such that ”f” is simplified by substituting the constant to propositional variable ”x” . 3SAT-DC is a recursive procedure for a clause with the minimum unknown literals at that time. The time complexity of 3SAT-DC is $O(m1.84^n)$ [14].

3. The Proposed Algorithm

3.1 An Overview

BCP procedure is inherently sequential because it consists of iterations of $O(m)$ time. Therefore, we parallelize only each iteration on a GPU and performs the other part on a CPU. To do so, we partition m clauses. Listing 2 shows a pseudo code of the proposed CPU code for DPLL algorithm with parallelized BCP procedure. After partitioning given clauses, DPLL algorithm with parallelized BCP procedure searches a solution while transferring data on the current status of search to memory on a CPU or a GPU every decision. The difference between serial DPLL algorithm in Section 2.1 and the parallel version is addition of clause partitioning in line 2 of Listing 2.

3.2 Clause Partitioning

We partition clauses into groups in order to parallelize BCP procedure. Each group is processed by a thread block. Hence, we partition clauses into groups the same number of clauses for load balance. The maximum number of active thread blocks is eight times the number numMP of multiprocessors. Hence, we set the number numsubclause of clauses in a group at (numclause + numMP * 8 - 1) / (numMP * 8). Due to this, the number of groups is at most the maximum number of active thread blocks.

3.3 Parallelized BCP Procedure

This section describes how to parallelize the serial BCP procedure in Section 2.2.

3.3.1 Calling from a CPU

Listing 3 shows a CUDA C code of the proposed parallelized BCP procedure called from line 5 in Listing 2. The differences between the serial BCP and the parallel version are:

- Status data are packed into the same array as many as possible.
- Data transfers between a CPU and a GPU are inserted before and after do-while statement and kernel function call.

The reason why status data are packed into the same array is to reduce the number of cudaMemcpy execution. The overhead of cudaMemcpy invocation is heavy. Therefore, memory transfer time can be reduced by reduce the nuber of the invocations. In the proposed CUDA C code, flag variables ”conf” and ”imp” are packed into array ”flag”. Also, stack pointer ”sp” and array ”implicated” are packed into array ”sp_implicated”. Due to these packings, the whole execution time is reduced to about 90%. The proposed CUDA C code transfers array ”atom” and ”sp_implicated” before and after executing parallelized BCP procedure (line 7 to 10 and 18 to 21 in Listing 3). However, if no conflict occurs, the latter transfer is not performed to avoid excess transfer. After executing kernel function BCPEngine, variable ”flag” is transfered (line 15 in Listing 3).

3.4 Kernel Function BCPEngine

Listing 4 shows the CUDA C code of the proposed kernel function BCPEngine. The number of thread blocks is the number s of groups. The number of threads in a thread block is predefined constant BLKSZ. The arguments subset and orderedClause represent partitioned groups together. Each element of orderedClause holds a clause. Clauses in the same group are consecutively stored in array orderedClause. The head index of elements of each group is stored array ”subset”. The type of orderedClause is int[4] rather than int[3]. Since every clause of 3SAT instance has exactly three literals, int[3] is sufficient to hold each clause. However, the proposed kernel stores each clause into int[4] with padding in order to enable coalesce access.

The kernel function BCPEngine works as follows. As preparation, the number ”size” of clauses in the group assigned to each thread block is calculated (line 4 in Listing 4). Also, pointer ”orderedClause” is moved so as to point at the head clause of the assigned group (line 5 in Listing 4). Hence, in the lines below line 5, each thread block can access the assigned group via orderedClause[0..size-1]. The processing for each clause is almost same as the serial BCPEngine, but there are four different points. First, flag ”conflict” is checked before each clause is processed and if conflict is detected then each thread is finished .Second, each clause is loaded into variable ”c” of type int4 to realize coalesced access .Third, detection of simultaneous implication by multiple threads is processed by CUDA atomic functions (line 27 to 30 in Listing 4). If multiple threads detect implications for the

ⓒ 2012 Information Processing Society of Japan

Listing 1 A pseudo code of 3SAT-DC

```
1  3SAT–DC(f)
2  {
3    if (f == FALSE) return FALSE;
4    min_c = a clause with minimum number of literals;
5    if( min_c == an empty clause ) return TRUE;
6    if( min_c == (x) ) return 3SAT–DC( f(x = true) );
7    else if( min_c == (x or y) )
8      return 3SAT–DC( f(x = true) ) || 3SAT–DC( f(x = false, y = true));
9    else /* min_c==( x or y or z ) */
10     return 3SAT–DC( f(x = true) ) || 3SAT–DC( f(x = false, y = true))
11             || 3SAT–DC( f(x = false, y = false, z = true));
12 }
```

Listing 2 A pseudo code of the proposed CPU code for DPLL algorithm with parallelized BCP procedure

```
1  preprocess to detect trivial unsatisfiability
2  partition clauses into groups
3  while(1){
4    Decision
5    while (BCP() == conflict) {
6      if (no more backtrack) return FALSE
7      backtrack
8    }
9  }
```

Listing 3 A CUDA C code of the proposed parallelized BCP procedure

```
1  #define BLKSZ 64
2  int BCP_GPU(int numatom, int *atom, int s, int *sp_implicated, int *d_subset,
3              int *d_orderedClause, int *d_atom, int *d_flag, int *d_sp_implicated)
4  {
5    int flag[2];   // flag[0]:conf, flag[1]:imp
6    cudaMemset(&d_flag[0], 0, sizeof(int));   // conf = 0;
7    cudaMemcpy(d_atom, atom,sizeof(int) * (numatom + 1),
8               cudaMemcpyHostToDevice);
9    cudaMemcpy(d_sp_implicated, sp_implicated, sizeof(int) * (numatom+1),
10             cudaMemcpyHostToDevice);
11   do {
12     cudaMemset(&d_flag[1], 0, sizeof(int));   // imp = 0;
13     BCPEngine<<< s, BLKSZ >>>( d_subset, d_orderedClause,
14                               d_atom, d_flag,d_sp_implicated);
15     cudaMemcpy(flag, d_flag, sizeof(int)*2, cudaMemcpyDeviceToHost);
16   } while (!flag[0] && flag[1]);   // conf == 0 && imp != 0
17   if(!flag[0]){   // no conflict
18      cudaMemcpy(atom, d_atom, sizeof(int) * (numatom + 1),
19                 cudaMemcpyDeviceToHost);
20      cudaMemcpy(sp_implicated, d_sp_implicated, sizeof(int) * (numatom+1),
21                 cudaMemcpyDeviceToHost);
22   }
23   return flag[0];   //return conf;
24 }
```

ⓒ 2012 Information Processing Society of Japan

same variable and attempt to assign the variable different values, then conflict should be detected as shown in line 31 in Listing 4. However, even if line 31 is deleted, no problem occurs because it will be detected as conflict at the next invocation of BCPEngine. Preliminary experiments show that the code without line 31 is faster by a few percent. Therefore, we deleted line 31. Fourth, implication flag variable on global memory is not updated immediately. Instead, each thread updates myImplication flag variable on a register at each iteration and finally updates implication flag variable once (line 34 in Listing 4).

3.4.1 Cache Configuration

The proposed kernel function in Listing 4 never use shared memory. Instead, it relies on cache memory of Fermi architecture. Fermi GPUs can configure size of shared memory and L1 cache memory either 16KB/48KB or 48KB/16KB. Our current implementation never use shared memory. Therefore, we set the size of L1 cache memory at 48KB.

4. Experiments

This section compares the performance of the proposed CUDA program with a CPU program that performs the same computation. We measured the execution time (average of 10 trials for each test) of not only BCP procedure but also whole execution of 3SAT-DC with BCP procedure. We embedded invocations of BCP procedure just before recursive calls in line 6, 8, 10, and 11 in Listing 1, which correspond to Decisions. Furthermore, we set up the maximum number BCPMAX of invocations of BCP procedure. If our SAT solver executed BCP procedure BCPMAX times, we stopped our SAT solver and measured the execution time at that time. If our SAT solver found a solution or exhausted the search space before BCP procedure was executed BCPMAX times, the execution time at that time is measured. We fixed the number of threads in a thread block at 64 because preliminary experiments showed the number is better.

For each test, a single core of 2.93 GHz Intel Core i3 and NVIDIA GeForce GTX480 was used. The OS used is Windows7 Professional SP1 with NVIDIA graphics driver Version 285.62. For compilation, Microsoft Visual Studio 2008 Professional Edition with optimization option /O2 and CUDA 3.2 SDK were used.

4.1 Performance and Problem Size

In this section, we compare the performance of GPU with that of CPU for various problem sizes. The used problem instances were generated by Motoki's 3SAT instance generator G3 (n,m) [15], [16] (G3 for short). G3 generates a boolean expression that has exactly one solution with high probability. In general, to solve 3SAT instance with many (a few) solutions is easy (hard).

Figure 1, 2, and 3 show a performance comparison between CPU and GPU with problem instances generated by G3. As for Figure 1 and 2, x-axis is the number of variables and y-axis is the execution time in second. Figure 1 shows performance for relatively small instances with BCPMAX 50000. Figure 2 shows performance for relatively large instances with BCPMAX 10000. Figure 3 shows the speedup ratios in case of Figure 1 and 2. As for Figure 3, x-axis is the number of variables and y-axis is speedup ratio. Our GPU solver runs faster with an increase of the problem size (i.e., the number of variables). Although GPU is slower than CPU for small problems, it is reversed for 2500 variables. The best GPU performance is about 4.5 times than CPU for 50000 variables. For more than 50000 variables, the performance of GPU tends to decrease with an increase of the problem size.

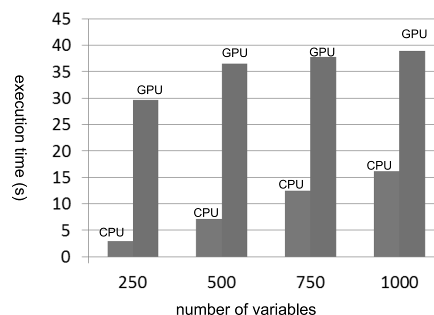

Fig. 1 A performance comparison between CPU and GPU (small instance)

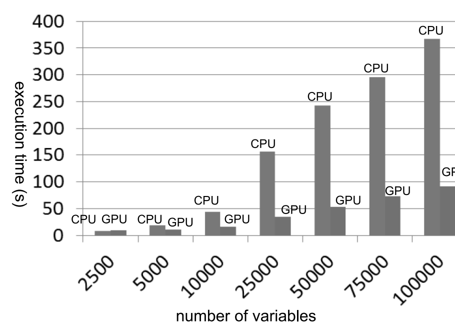

Fig. 2 A performance comparison between CPU and GPU (large instance)

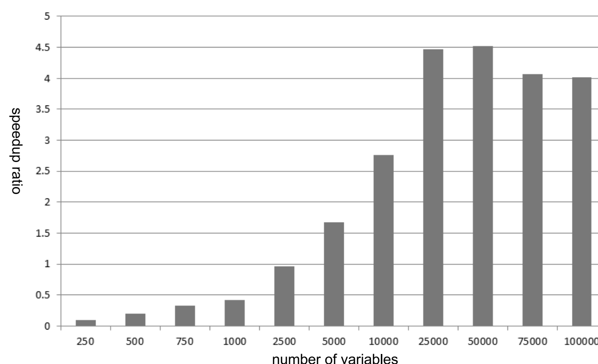

Fig. 3 Speedup ratios of GPU to CPU

4.2 Performance and the Number of BCP Procedure Done

In this section, we fix problem size and compare the performance of GPU with that of CPU for various values of BCPMAX. We randomly selected 10 instances from SAT11 Competition [17] with category RANDOM, 50000 variables, and 210000 clauses.

ⓒ 2012 Information Processing Society of Japan

Listing 4 A CUDA C code of the proposed parallelized BCP procedure

```
__global__ void BCPEngine(int *subset, int (*orderedClause)[4],
                          int *atom, int *flag, int *sp_implicated)
{
   int size = subset[blockIdx.x + 1] - subset[blockIdx.x];
   orderedClause += subset[blockIdx.x];
   int myImplication = 0;
   for (int i = threadIdx.x; i < size; i += blockDim.x) {
     if (flag[0]) return;  // Another thread detected conflict
     int c[4];
     *((int4 *) c) = *((int4 *) orderedClause[i]);
     if (c[0] * atom[abs(c[0])] >= 0) goto F;
     if (c[1] * atom[abs(c[1])] >= 0) goto F;
     if (c[2] * atom[abs(c[2])] >= 0) goto F;
     flag[0] = 1;  // conf = 1;
     return;
F:
     int numUnknown = 0; int idxUnknown;
     if (atom[abs(c[0])]==UNKNOWN) {numUnknown++; idxUnknown=0;}
     if (atom[abs(c[1])]==UNKNOWN) {numUnknown++; idxUnknown=1;}
     if (atom[abs(c[2])]==UNKNOWN) {numUnknown++; idxUnknown=2;}
     if (numUnknown == 1) {  // exactly one literal of unknown value in orderedClause[i]
       if (c[0] * atom[abs(c[0])] > 0) continue;
       if (c[1] * atom[abs(c[1])] > 0) continue;
       if (c[2] * atom[abs(c[2])] > 0) continue;
       int litUnknown = c[idxUnknown];
       int val = (litUnknown > 0) ? TRUE : FALSE;
       int old = atomicCAS(&atom[abs(litUnknown)], UNKNOWN, val);
       if (old == UNKNOWN) {myImplication++;
                           sp_implicated[atomicAdd(&sp_implicated[0],1)]
                           = abs(litUnknown);}
       //else if (old != val) { flag[0] = 1; return; }
       }
     }
     if(myImplication) flag[1] = 1;  // imp = 1;
   }
}
```

© 2012 Information Processing Society of Japan

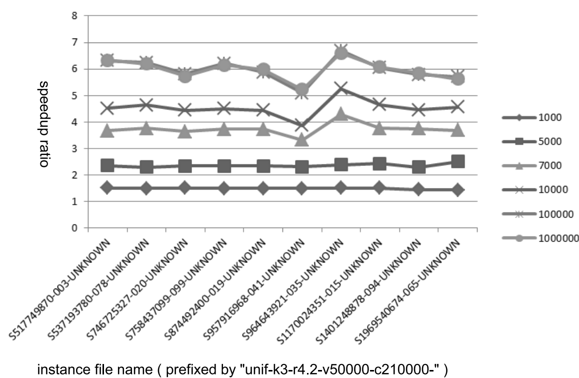

Fig. 4 Relation between speedup ratio and BCPMAX

Figure 4 shows the result. The x-axis shows the name of instance file name and the y-axis is speedup ratio. The speedup ratio is improved with an increase of BCPMAX. Even if BCPMAX is 1000, GPU is 1.5 times faster than CPU in average. The best performance (6.7 times speedup) of GPU is obtained when BCPMAX is 10000. For BCPMAX larger than 10000, the performance improvement is negligible.

5. Related Works

As far as we know, there exist five existing research on SAT algorithm parallelized on a CUDA GPU, as shown below.

In [18], Meyer et al. proposed a complete-type method to parallelize 3SAT-DC like us. However, they did not use BCP procedure. Instead, they proposed a problem partitioning method to parallelize divide-and-conquer itself and their own heuristic for Decision phase.

In [19], McDonald et al. proposed an incomplete-type method to parallelize WalkSAT algorithm with clause learning. Their parallelization method is to run many threads such that each thread executes serial WalkSAT algorithm with pseudo random sequence different any other thread.

In [20], Gulati et al. proposed a complete-type method named MESP (MiniSAT enhanced with SurveyPropagation). They experimentally showed speedup ratio of 2.35 in average with 2.67GHz Intel i7 CPU and NVIDIA GeForce 280GTX GPU.

In [21], Wang et al. proposed an incomplete-type method of a celler genetic algorithm with random walk local search.

In [22], Deleau et al. proposed an incomplete-type method such that SAT instance is represented by a 0-1 matrix and 0-1 matrix multiplication is used to search a solution for a given SAT instance.

6. Conclusion and Future Work

This paper has proposed a method to parallelize BCP procedure for SAT algorithm and has implemented a CUDA GPU program based on the proposed method. Experimental results show that the proposed GPU SAT solver runs maximum 6.7 times faster than the corresponding CPU solver. One of future works is to realize more speedup by improving clause partitioning method and so on.

Acknowledgments This manuscript is based on the previous style file for A4 landscape papers. The Editorial Board of the JIP sincerely thanks its members for their effort in preparing the previous version.

References

[1] Garey, M. R. and Johnson, D. S. : Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman (1979)
[2] Davis, M. and Putnum, H. : A Computing Procedure for Quantification Theory, Journal of the ACM, Vol.7, No.3, pp.201-215 (1960)
[3] Davis, M., Logemann, G., and Loveland, D. : A Machine Program for Theorem Proving, Communications of the ACM, Vol.5, No.7, pp.394-397 (1962)
[4] Davis, J. D., Tan, Z., Yu, F., and Zhang, L. : Designing an Efficient Hardware Implication Accelerator for SAT Solving, LNCS Vol.4996, pp.48-62 (2008)
[5] SAT Research Group, Princeton University : zChaff, http://www.princeton.edu/ chaff/zchaff.html
[6] Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., and Malik, S. : Chaff: Engineering an Efficient SAT Solver, 39th Design Automation Conference (DAC), (2001)
[7] Mahajan, Y. S., Fu, Z., and Malik S. : Zchaff2004: An efficient SAT solver, International Conference on Theory and Applications of Satisfiability Testing (SAT), pp.360-375 (2004)
[8] Kirk, D. B. and Hwu, W. W.; Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann (2010)
[9] Sanders, J. and Kandrot, E.; CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional (2010)
[10] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro, Vol.28, No.2, pp.39–55 (2008)
[11] Garland, M. and Kirk, D. B.: Understanding Throughput-Oriented Architectures, Communications of the ACM, Vol.53, No.11, pp.58-66 (2010)
[12] NVIDIA: CUDA Programming Guide Version 4.0. http://www.nvidia.com/object/ cuda_develop.html (2011)
[13] NVIDIA: CUDA Best Practice Guide 4.0. http://www.nvidia.com/object/ cuda_develop.html (2011)
[14] Monien, B. and Speckenmeyer, E. : Solving satisfiability in less than 2n steps, Discrete Applied Mathematics, Vol.10, No.3, pp.287-295 (1985)
[15] Motoki, M. : SAT Instance Generation Page, http://www.is.titech.ac.jp/ watanabe/gensat/
[16] Motoki, M. and Uehara, R. : Unique Solution Instance Generation for the 3-Satisfiability (3SAT) Problem, International Conference on Theory and Applications of Satisfiability Testing (SAT), pp.293-307 (2000)
[17] Jarvisalo, M., Berre, D. L. and Roussel, O. : SAT Competition 2011, http://www.satcompetition.org/2011/ (2011)
[18] Meyer, Q., Schönfeld, F., Stamminger, M., and Wanka, R. : 3-SAT on CUDA: Towards a Massively Parallel SAT Solver, High Performance Computing and Simulation Conference (HPSC), pp.306-313 (2010)
[19] McDonald, A. and Gordon, G. : ParallelWalkSAT with Clause Learning, Data Analysis Project Presentation, School of Computer Science, Carnegie Mellon University, http://www.ml.cmu.edu/research/dap-papers/dap_mcdonald.pdf (2009)
[20] Gulati, K. and Khatri, S. P. : Boolean Satisfiability on a Graphics Processor, the 20th Great Lakes Symposium on VLSI (GLSVLSI), pp.123-126 (2010)
[21] Wang, Y. : NVIDIA CUDA Architecture-based Parallel Incomplete SAT Solver, Master Project Final Report, Faculty of Rochester Institute of Technology (2010)
[22] Deleau, H., Jaillet, C., and Krajecki, M. : GPU4SAT: Solving the SAT Problem on GPU, http://para08.idi.ntnu.no/docs/submission_49.pdf (2008)

ⓒ 2012 Information Processing Society of Japan