Vol.2011-ARC-197 No.27
Vol.2011-HPC-132 No.27
2011/11/29

SAT アルゴリズムにおける BCP 処理の GPU を用いた並列化

藤井宏憲† 藤本典幸†

充足可能性判定問題（SAT）は，応用範囲の広い最も基本的な NP 完全問題の一つである．SAT を解くためには最悪指数時間かかってしまうが，重要な問題なのでできるだけ高速に解きたいという要求がある．そこで我々は，GPU を用いて並列計算を行うことで，その計算時間を節約しようと考えた．本論文では，SAT アルゴリズムの主要な高速化手法のひとつである BCP（Boolean Cons traint Propagation）処理を GPU を用いて並列化する手法を提案する．分割統治法に BCP 処理を組み合わせた SAT アルゴリズムに提案手法を適用し，CPU に 2.93GHz Intel Core i3，GPU に NVIDIA GeForce GTX480 を用いた環境で評価実験を行ったところ，SAT ソルバとしての速度向上率は 6.7 倍であった．

GPU Accelerated BCP Computation for SAT Algorithms

Hironori Fujii† and Noriyuki Fujimoto†

The Boolean s atisfiability p roblem is widely applicable and on e of the most bas ic NP-complete problems. This problem has been required to be solved as fast as possible because of its i mportance, but it t akes expon ential ti me in the worst c ase to so lve. Therefore, we ai m to sav e the computation time b y parallel co mputing on a GP U. We propose a parallelization of BCP (Boolean Constraint Propagation) computation, 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-embeded divide and conquer algorithm 6.7 times faster than the CPU.

1. はじめに

充足可能性判定問題（以降，SAT）[1]とは，入力として命題論理式が与えられ，式全体の値を真にする変数の真偽値割り当てが存在するかを問う問題で，存在すれば充足可能，存在しなければ充足不能が解となる．SAT は，変数集合 **V** に対して通常下記のような和積標準形（以降，CNF）で定式化される．

$$L_i = \{V_p, \neg V_p\}$$

$$C_j = \left(L_{j1} \vee L_{j2} \vee \cdots \vee L_{jm_j}\right)$$

$$C_1 \wedge C_2 \wedge \cdots \wedge C_n = 1$$

ここで， $V_p \in \mathbf{V}$ は変数，¬は否定，∨は論理和，∧は論理積である．L_i は変数 V_p の肯定または否定であり，リテラルという．C_j は１つ以上のリテラルの論理和をとったもので，節という．CNF は１つ以上の節の論理積で表される．

SAT は，人工知能や論理設計・検証など応用範囲の広い最も基本的な NP 完全問題の一つであり，解くためには最悪指数時間かかってしまうが，重要な問題なのでできるだけ高速に解きたいという要求がある．近年，SAT を解くための様々なアルゴリズム（以降，SAT アルゴリズム）の性能は飛躍的に改善され，数百万変数からなる論理式も数時間で判定できるようになってきている[1]．SAT アルゴリズムには complete 型と incomplete 型があり，complete 型は充足可能と充足不能の両方を，incomplete 型は充足可能のみを判定できる．complete 型でよく用いられる SAT アルゴリズムの１つに DPLL アルゴリズム[2][3]がある．DPLL アルゴリズムには，BCP(Boolean Constraint Propagation)という処理が含まれており，この BCP 処理は DPLL アルゴリズムの最も重い処理で全体の８～９割の計算時間を占める．これに注目して，この BCP 処理を，FPGA を用いて並列化することで高速化を実現する手法[4]が Davis らによって提案されている．Davis らは BCP 処理のみ並列化することにより，様々な SAT アルゴリズムに対応できるようにしている．Davis らは complete 型の zChaff[5][6][7]に BCP 処理を組み込み，CPU に 3.6 GHz Pentium 4，FPGA に Xilink Virtex 5 LX110T を用いた環境で CPU だけを用いる場合に比べて，速度向上率 5～16 倍を達成している．

本論文では，BCP 処理を CUDA 対応 GPU [8][9][10][11][12][13]を用いて並列化することで高速化する手法を提案する．本論文でも Davis らの手法と同様に，BCP 処理のみ並列化する方法を採用する．このため提案手法は様々な SAT アルゴリズムに組み込んで使用することができるが，本論文の評価実験では，complete 型の分割統治法によ

† 大阪府立大学 大学院理学系研究科
Graduate School of Science, Osaka Prefecture University

ⓒ 2011 Information Processing Society of Japan

る SAT アルゴリズム[14]における再帰呼び出しの前に BCP 処理を組み込んだものを SAT ソルバとして用いた．本論文では簡単のため節内のリテラルを 3 個に限定した SAT（3SAT）の問題のみを取り扱う．我々の手法では CPU に 2.93GHz Intel Core i 3，GPU に NVIDIA GeForce GTX480 を用いた環境で CPU だけを用いる場合に比べて，SAT ソルバとしての速度向上率 6.7 倍を達成した．

以降の論文の構成は次の通りである．第 2 章で提案手法を理解するために必要な，DPLL アルゴリズム，BCP 処理，および分割統治法による SAT アルゴリズムについて簡単に説明し，第 3 章で提案手法の詳細を示す．第 4 章で評価実験について述べる．第 5 章で関連研究を紹介する．第 6 章でまとめと今後の課題について述べる．ページ数の制限のため，本論文では，GPU のアーキテクチャおよびプログラミングの概要については述べない．これらに不慣れな読者は文献 8) 9) 10) 11) 12) 13) を参照されたい．

2. 準備

2.1 DPLL アルゴリズム

DPLL アルゴリズムは，基本的には全ての変数パターンを検証していき，いくつかの工夫によって省ける変数パターンを枝刈りしながら解を探索する．その工夫の 1 つに BCP 処理がある．図 1 に，BCP 処理を含む DPLL アルゴリズムの疑似コードを示す．DPLL アルゴリズムでは解を探索中，ヒューリスティックを用いて変数値を割り当てる Decision というフェーズのたびに BCP 処理を行う．なお，その他の工夫[2][3]については本研究では使用しないので省略する．

2.2 BCP 処理

BCP 処理は，DPLL アルゴリズムの過程で連鎖的なリテラル値の決定（以降，implication）と節内の変数が全て偽となっている状態（以降，conflict）の検知を行う処理である．図 2 に，逐次 BCP 処理のプログラムコードを示す．また，これ以降のコード上では，リテラルを配列 atom，節を配列 clause，リテラルの数を numatom，節数を numclause として扱い，またリテラル値の表現は，真（TRUE）を 1，偽（FALSE）を－1，未定（UNKNOWN）を 0 として扱う．

```
1:  前処理により自明に充足不可能な場合などを検出
2:  while( 1){
4:      Decision
5:      while (BCP 処理() == conflict) {
6:          if (これ以上バックトラックできない) return FALSE
7:          バックトラック
8:      }
9:  }
```

図 1　BCP 処理を含む DPLL アルゴリズムの疑似コード

```
1:  int   BCP(int numclause, int (*clause)[3], int *atom)
2:  {
3:          int conf, imp;
4:          conf = 0;
5:          do {
6:                  imp = 0;
7:                  BCPEngine(numclause, clause, atom, &conf, &imp);
8:          } while (!conf && imp);
9:          return conf;
10: }
```

図 2　逐次 BCP 処理のプログラムコード

BCP 処理は，新たな implication が起こらないもしくは conflict が発生するまで関数 BCPEngine（図 2 の 7 行目）の処理を繰り返す．図 3 に，BCPEngine のプログラムコードを示す．BCPEngine では，implication 後に全節内を 1 回見て（O(節数)の処理），新たな implication が起こるもしくは conflict が発生するかを探索する．節内の全てのリテラルが偽の場合，conflict が発生したとしてフラグ conf を更新し BCPEngine を終了させる（図 3 の 5～8 行目）．節内の全てのリテラルが偽の場合以外は，implication の探索を行う．implication の探索は，節内で未定リテラルが 1 つかつ他のリテラルは全て偽の場合，implication として検知する．まず，節内の未定リテラルの数をカウントする（図 3 の 11～13 行目）．節内の未定リテラルの数が 1 つの場合，次の処理に移る．節内のリテラルについて，真であるリテラルが存在すれば implication ではないのでこの節の処理を終了する（図 3 の 16～18 行目）．節内のリテラルが未定リテラル以外全て偽の場合，implication 検知となる．implication が検知されると，未定リテラルが真となる値を atom に割り当てる（図 3 の 20～22 行目）．atom に割り当て後は，割り当てた atom のインデックス値をバックトラックのための配列 implicated にスタックしておき implicated のカウンターsp を増やす（図 3 の 23 行目）．最後に implication を検知したとしてフラグ imp を更新する（図 3 の 24 行目）．

ⓒ 2011 Information Processing Society of Japan

```
 1:   void BCPEngine(int numclause, int (*clause)[3], int * atom, int *conf, int *imp)
 2:  {
 3:          *conf = 0; *imp = 0;
 4:          for (int i = 0; i < m; i++) {
 5:                  if (clause[i][0] * atom[abs(clause[i][0])] >= 0) goto F;
 6:                  if (clause[i][1] * atom[abs(clause[i][1])] >= 0) goto F;
 7:                  if (clause[i][2] * atom[abs(clause[i][2])] >= 0) goto F;
 8:                  *conf = 1; return;
 9:  F:
10:                  int numUnknown = 0; int idxUnknown;
11:                  if (atom[abs(clause[i][0])] == 0) { numUnknown++; idxUnknown = 0; }
12:                  if (atom[abs(clause[i][1])] == 0) { numUnknown++; idxUnknown = 1; }
13:                  if (atom[abs(clause[i][2])] == 0) { numUnknown++; idxUnknown = 2; }
14:                  if (numUnknown == 1) {
15:                      // clause[i]の未定リテラルはちょうど１つ
16:                      if (clause[i][0] * atom[abs(clause[i][0])] > 0) continue;
17:                      if (clause[i][1] * atom[abs(clause[i][1])] > 0) continue;
18:                      if (clause[i][2] * atom[abs(clause[i][2])] > 0) continue;
19:                      //  他のリテラルは偽
20:          int          litUnknown = clause[i][idxUnknown];
21:                      int val = (litUnknown > 0) ? TRUE : FALSE;
22:          ato          m[abs(litUnknown)] = val;
23:          im          plicated[sp++] = abs(litUnknown);
24:                      (*imp)++;
25:                  }
26:          }
27:  }
```

図 3　逐次 BCP 処理内の BCPEngine のプログラムコード

2.3 分割統治法による SAT アルゴリズム

分割統治法 3SAT の疑似コードを図 4 に示す. ここで, f は入力の命題論理式であり, f(x=真偽値)は f 中の変数 x の値を決定して f を簡略化した式を表す.

3SAT-DC()は，評価した節の中でリテラル数が最も少ない節について，各リテラル数に応じた再帰呼び出しによる処理をする．この分割統治法 3SAT の時間計算量は，変数の数を n，節数を m とすると $O(m \times 1.84^n)$[14]である．

```
 1:   3SAT-DC(f)
 2:  {
 3:   if (f == FALSE) return FALSE;
 4:   min_c=f でリテラル数が最も少ない節;
 5:  if(   min_c==空節 )  return TRUE;
 6:   if( min_c==(x) ) return 3SAT-DC( f(x=真) );
 7:  els  e if( min_c==(x∨y) )
 8:      return 3SAT-DC( f(x=真) ) ∨3SAT-DC( f(x=偽, y=真));
 9:  els  e /* min_c==( x∨y∨z  ) */
10:   r    eturn 3SAT-DC( f(x=真) ) ∨3SAT-DC( f(x=偽, y=真))
11:                  ∨3SAT-DC( f(x=偽, y=偽, z=真));
12:  }
```

図 4　分割統治法 3SAT の疑似コード

3. 提案手法

3.1 概要

BCP 処理は O(節数)時間の処理の繰り返しであり，各繰り返し間の逐次性は強い．このため，O(節数)の処理の部分のみを GPU で並列化し，その他の処理は全て CPU 上で行う．O(節数)の処理を並列化するために節集合を分割する．図 5 に，並列 BCP 処理を導入した DPLL アルゴリズムの CPU 側の疑似コードを示す.

並列 BCP 処理を導入した DPLL アルゴリズムでは, 前処理として与えられた節集合を分割してから解の探索を開始し，解の探索中，Decision のたびに現在の探索の状態を表すデータ（以降，状態データ）を CPU または GPU のメモリに転送しながら並列 BCP 処理を行う．2.1 節の処理との違いは，図 5 の 2 行目の節集合の分割が加わるのみである．その他の処理は 2.1 節の処理と等しい.

3.2 節集合の分割

並列に BCP 処理を行うために節集合を分割する．GPU 上の並列 BCP 処理では 1 つのスレッドブロックに 1 つの節グループを処理させる．そこで，負荷分散をするためグループ内の節数が均等になるよう分割する．同時にアクティブにできる最大スレッドブロック数はマルチプロセッサ数（numMP）の 8 倍なので，グループ内の節数（numsubclause）を

numsubclause = (numclause + numMP * 8 - 1) / (numMP * 8);

とする．これにより，グループ数が最大スレッドブロック数を超えないように節集合

ⓒ 2011 Information Processing Society of Japan

```
1:  前処理により自明に充足不可能な場合などを検出
2:  節集合を複数のグループに分割
3: while( 1){
4:     Decision
5:     while (BCP 処理() == conflict) {
6:        if (これ以上バックトラックできない) return FALSE
7:        バックトラック
8:     }
9: }
```

図 5　並列 BCP 処理を導入した DPLL アルゴリズムの CPU 疑似コード

を均等に分割できる．

3.3 BCP 処理の並列化

2.2 節の逐次 BCP 処理をどのように並列化したかを述べる．

3.3.1 CPU からの呼び出し

図 6 に，図 5 の 5 行目にあたる並列 BCP 処理のプログラムコードを示す．図 2 の逐次 BCP 処理と比べた大まかな違いとしては, 状態データをできるだけ同じ配列内に納めていることと，BCP 処理である do-while 文の前後とカーネル関数呼び出しの後にそれぞれ CPU - GPU 間の状態データ転送が含まれていることが挙げられる．

状態データをできるだけ同じ配列内に納めた理由は，cudaMemcpy の呼び出し回数をできるだけ減らすためである．cudaMemcpy は呼び出しのオーバーヘッドが大きいので呼び出し回数を減らすことで転送時間を短縮できる. conflict と implication のフラグである conf と imp を配列 flag，バックトラック用の sp と implicated を配列 sp_implicated にまとめている．状態データをできるだけ同じ配列内に納めることで全体の約 1 割計算時間を短縮できている．

BCP 処理前後の転送は，atom の内容と sp_implicated の内容を転送する（図 6 の 7〜10,18〜21 行目）．ただし，BCP 処理後の転送は conflict が発生していない場合に限る．これは，無駄な転送を避けるためである．カーネル関数呼び出しの後は，flag の内容を転送する（図 6 の 15 行目）．

3.3.2 カーネル関数

GPU のカーネル関数について説明する．図 7 に，図 6 の 13,14 行目のカーネル関数 BCPEngine のプログラムコードを示す. スレッドブロック数はグループ数である s を，スレッド数はあらかじめ定義している BLKSZ を用いる．引数の subset, o rderedClause はそれぞれ節集合分割後のグループのインデックスとデータが保存してある．orderedClause には先頭のグループから順に節データが保存され，subset には

```
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:   cudaM    emset(&d_flag[0], 0, sizeof(int));   // conf = 0;
7:   cudaM    emcpy(d_atom, atom,sizeof(int) * (numatom + 1),
8:                  cudaMemcpyHostToDevice);
9:   cudaM    emcpy(d_sp_implicated, sp_implicated, sizeof(int) * (numatom+1),
10:                 cudaMemcpyHostToDevice);
11:     do {
12:    cudaM      emset(&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:    c      udaMemcpy(flag, d_flag, sizeof(int)*2, cudaMemcpyDeviceToHost);
16:     } while (!flag[0] && flag[1]);   // conf == 0 && imp != 0
17:   if(!    flag[0]){   // conflict が起きてない
18:    cudaM      emcpy(atom, d_atom, sizeof(int) * (numatom + 1),
19:                     cudaMemcpyDeviceToHost);
20:    cudaM      emcpy(sp_implicated, d_sp_implicated, sizeof(int) * (numatom+1),
21:                     cudaMemcpyDeviceToHost);
22:     }
23:   re    turn flag[0];  //  return conf;
24: }
```

図 6　並列 BCP 処理のプログラムコード

orderedClause に保存されている各グループの先頭のインデックスが保存されている．さらに，orderedClause は int[4]の配列で渡す．3SAT なので本来は int[3]で扱えるが，コアレスアクセスを行うため，あらかじめパディングして int[4] の orderedClause に格納しておき，カーネル関数内では int[4]の配列として扱う．

カーネル関数は，１つのスレッドブロックに 1 つの節グループを処理させる．スレッドブロック内の各スレッドは，グループの先頭から担当の節を割り振られ，グループ内の節をすべて処理するまで動く．

各節を処理する前の準備として，subset からスレッドブロックの担当するグループの節数を size に計算しておく（図 7 の 4 行目）．また，各スレッドブロックが担当する

ⓒ 2011 Information Processing Society of Japan

```
 1:  __global__ void BCPEngine(int *subset, int (*orderedClause)[4],
 2:                                  int *atom, int *flag, int *sp_implicated)
 3: {
 4:   int     size = subset[blockIdx.x + 1] - subset[blockIdx.x];
 5:       orderedClause += subset[blockIdx.x];
 6:   int     myImplication = 0;
 7:       for (int i = threadIdx.x; i  <  size; i += blockDim.x) {
 8:            if (flag[0]) return;   //  他で conflict が発生
 9:            int c[4];
10:            *((int4 *) c) = *((int4 *) orderedClause[i]);
11:            if (c[0] * atom[abs(c[0])] >= 0) goto F;
12:            if (c[1] * atom[abs(c[1])] >= 0) goto F;
13:            if (c[2] * atom[abs(c[2])] >= 0) goto F;
14:            flag[0] = 1;    // conf = 1;
15:            return;
16: F:
17:            int numUnknown = 0; int idxUnknown;
18:            if (atom[abs(c[0])]==UNKNOWN) {numUnknown++; idxUnknown=0;}
19:            if (atom[abs(c[1])]==UNKNOWN) {numUnknown++; idxUnknown=1;}
20:            if (atom[abs(c[2])]==UNKNOWN) {numUnknown++; idxUnknown=2;}
21:            if (numUnknown == 1) {    // orderedClause[i]の未定リテラルはちょうど1つ
22:                 if (c[0] * atom[abs(c[0])] > 0) continue;
23:                 if (c[1] * atom[abs(c[1])] > 0) continue;
24:                 if (c[2] * atom[abs(c[2])] > 0) continue;
25:                 int litUnknown = c[idxUnknown];
26:                 int val = (litUnknown > 0) ? TRUE : FALSE;
27:                 int old = atomicCAS(&atom[abs(litUnknown)], UNKNOWN, val);
28:                 if (old == UNKNOWN) {myImplication++;
29:                                 sp_implicated[atomicAdd(&sp_implicated[0],1)]
30:                                 = abs(litUnknown);}
31:                 //else if (old != val) { flag[0] = 1; return; }
32:            }
33:       }
34:   if(     myImplication) flag[1] = 1;  //      imp = 1;
35:   }
```

図 7　並列 BCP 処理内の BCPEngine のプログラムコード

節の先頭にポインタ orderedClause を移動させる（図 7 の 5 行目）．5 行目以降では，orderedClause[0..size-1] が各スレッドブロックの処理対象となる．

各節の処理は図 3 の逐次 BCPEngine とほぼ等しいが，異なる部分が 4 つある．1 つ目は conflict のフラグ確認である．並列処理を行っているので他のスレッドで conflict が発生した場合，BCP 処理全体を止めるために処理スレッド自身で conflict のフラグを確認する．そこで，処理前に conflict のフラグ flag の確認を行い，conflict が発生していればスレッドの処理を終了する（図 7 の 8 行目）. 2 つ目は節データの扱いである．レジスタ上に int4 型の変数 c を用意し，そこにグローバルメモリ上の節データを int4 型で読み込む（図 7 の 10 行目）．このコアレスアクセスにより，転送速度を向上できる．以降は節の処理が終わるまで c を節データとして扱う．3 つ目は implication 検知時の処理である．並列処理をしているので implication の検知が同時に起こり，今回の BCP 処理に入る前は未割り当てだったリテラルの中で，今回の BCP 処理で 2 つ以上のスレッドが implication で同じリテラルに値を割り当てようとする場合が出てくる．そこで，CUDA に用意されているアトミック関数を用いる．アトミック関数は 1 つのスレッドが処理中に他のスレッドが同時に同様の処理をしないように用意されている関数である．そのアトミック関数の中で，atomicCAS は第 1 引数で指定したアドレスの値を第 2 引数の値と比較し，等しければ第 3 引数の値を第 1 引数のアドレスに格納し，異なれば第 1 引数のアドレスの値はそのままにして，最後に第 1 引数のアドレスに入っていた元の値を返す関数である．この atomicCAS を用いて，implication と判断されたリテラルの変数値と UNKNOWN を比較し, UNKNOWN であれば対応する atom の要素に値を割り当てる（図 7 の 27 行目）．これで同時に implication が検知されても変数値の更新処理は不可分に行われ，既に割り当てられているリテラルに新たに値が割り当てられることはなくなる．そして，implication が検知された場合 sp_implicated に implication 検知で割り当てたリテラルの変数のインデックス値をスタックしなければならないが，スタックは逐次に処理しなければならないので，ここではアトミック関数 atomicAdd を用いる．atomicAdd は第 1 引数で指定したアドレスの値に第 2 引数の値を加え，第 1 引数のアドレスに入っていた元の値を返す関数である．この atomicAdd を用いて, 図 3 の 23 行目と同様の処理を implication 検知ごとに不可分に処理できる（図 7 の 29,30 行目）．そして，この implication 検知時の処理で問題となるのが，1 回の BCP 処理中に 2 つ以上のスレッドが implication で同じリテラルに異なる値を割り当てようとした場合である．この場合は，いずれかのスレッドが割り当てようとした節で conflict が起きるので，図 7 の 31 行目のように conflict として検知すべきである．しかし，このコードを入れなくても，次回の全節内を検索するときに該当節で conflict が検知されるので問題ない．また，予備実験で図 7 の 31 行目のコードを省く方が数パーセント速いという結果が出たので，次回の全節内の検索で conflict を検知する方を採用した．4 つ目は implication のフラグの処理である．implication が検知

ⓒ 2011 Information Processing Society of Japan

されるたびに，毎回グローバルメモリにある flag を更新すると転送時間のための時間がかかってしまう．そこで，レジスタ上に myImplication を用意し，implication 検知時のフラグ更新を myImplication に対して行う．そして，各スレッドは全体の処理終了後に myImplication を見て implication の検知があれば, flag を更新する（図 7 の 34 行目）.

3.3.3 キャッシュの設定

図 7 のカーネル関数はシェアードメモリを一切使用していない．そのかわり，キャッシュに頼っている．現在最新のアーキテクチャであるフェルミでは，各マルチプロセッサが持つ 64KB の高速オンチップメモリをプログラマ側でシェアードメモリとキャッシュに分割できる. 具体的には, シェアードメモリを 16KB でキャッシュを 48KB, もしくはシェアードメモリを 48KB でキャッシュを 16KB と指定できる．今回はシェアードメモリを使用していないので，前者のキャッシュを 48KB 使用できる方を指定する．

4. 評価実験

GPU 版の並列 BCP 処理と CPU 版の逐次 BCP 処理の性能を比較する. CPU は 2.93GHz Intel Core i3 の 1 コアのみを使用し，GPU は NVIDIA GeForce GTX480，OS は Windows 7 Professional sp1 64bit, 開発環境は Visual Studio 2008 Professional, CUDA3.2 を用いた．GTX480 では 3.3.3 節の記述どおり，共有メモリは使用せずキャッシュを 48KB 使用している．また，どの問題も 1 スレッドブロックのスレッド数は 64 が比較的速いので，1 スレッドブロックのスレッド数は 64 で固定している．

比較方法としては，CPU も GPU も共に BCP 処理単体ではなく分割統治法 3SAT に組み込んで全体の計算時間を計る. 図 4 の 6,8,10,11 行目の再帰呼び出し直前が Decision にあたるので，そこに BCP 処理を組み込む．さらに，BCP 処理回数に上限（以降, BCPMAX）を設定する．BCP 処理回数が BCPMAX に到達するとその時点でプログラムを終了させ，その時間を終了時間として計測する．また，BCPMAX に到達する前にプログラムが終了すれば，その時間を終了時間として計測する．

4.1 問題サイズによる比較

変数の個数を基準に，問題サイズごとに GPU 版の並列 BCP 処理と CPU 版の逐次 BCP 処理の性能を比較する．使用する問題インスタンスは，Motoki の 3SAT i nstance generator G3(n,m) （以降，G3）[15][16]を用いて変数の個数の 4.2 倍が節の個数という設定で作成したインスタンスである. G3 は論理式を充足する変数割り当てが高い確率で 1 つしか存在しないような充足可能な問題インスタンスを生成する．充足する変数割り当ての組み合わせが少ないと解の探索は一般に難しくなる. G3 で作成したインスタンスで，GPU と CPU を比較した結果を図 8，9，10 に示す．図 8，9 は共に縦軸は計算時間[s]，横軸は変数の個数[個]を示す．図 8 は問題サイズが比較的小さい問題（変数の個数が 250～1000）で，BCPMAX は全て 5 万回に設定している．図 9 は問題サイズが比較的大きい問題（変数の個数が 2500～100000）で，BCPMAX は全て 1 万回に設定している．図 10 は図 8，9 の結果についての CPU に対する GPU の速度向上率を表している．図 10 の縦軸は速度向上率，横軸は変数の個数[個]を示す．

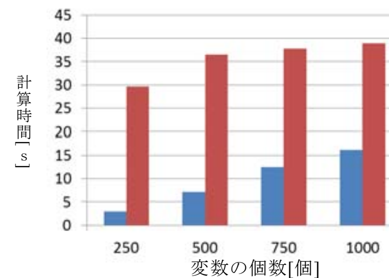

図 8 GPU と CPU の比較（小問題）

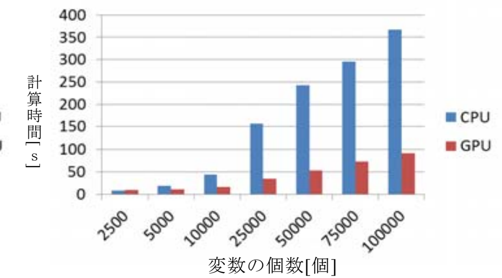

図 9 GPU と CPU の比較（大問題）

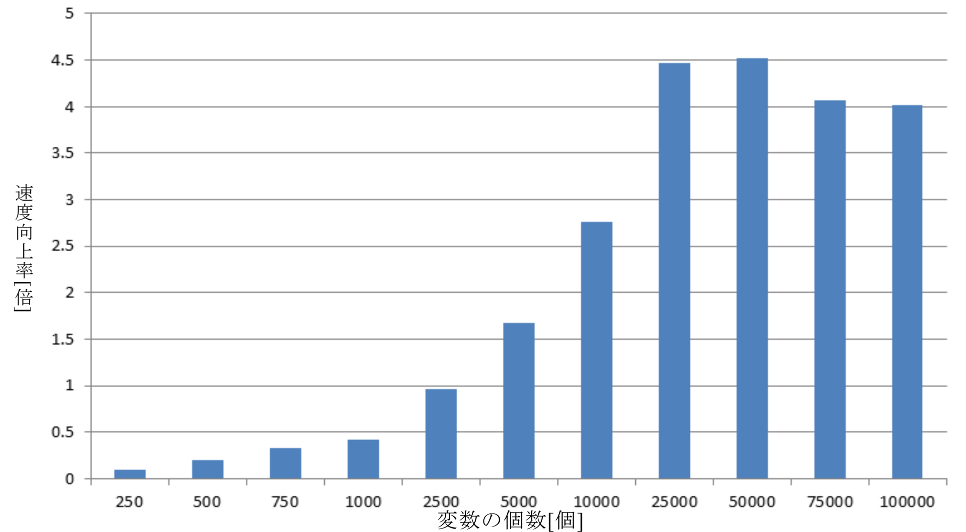

図 10 問題サイズを変化させたときの性能の変化

比較的小さな問題では，CPU が GPU の 2 倍以上速いという結果が出た．しかし，問題サイズを大きくするとスピード差は縮まり, 変数の個数が 2500 のときほぼ等しくなる. 問題サイズをさらに大きくすると, CPU と比べた GPU の速度は上がっていき，

ⓒ 2011 Information Processing Society of Japan

性能がもっとも良いのは変数の個数が 50000 のときで, GPU が CPU の 4.5 倍程度速いという結果が出た. 変数の個数を 50000 より大きくすると, 次第に GPU 性能は衰えていく傾向がある.

4.2 BCP 処理回数による比較

問題サイズを固定して，BCPMAX ごとに GPU 版の並列 BCP 処理と CPU 版の逐次 BCP 処理の性能を比較する．使用する問題インスタンスは，SAT11 Com petition[17]にてカテゴリ RANDOM で使用された 3SAT で変数の個数が 50000, 変数の個数が 210000 のインスタンスから無作為に抽出した 10 個である. このインスタンスで, GPU と CPU を比較した結果を図 11 に示す.

図 11 の縦軸は速度向上率，横軸はインスタンスファイル名の一部（“unif-k3-r4.2-v50000-c210000-”以降）を示す．比較する BCPMAX の回数は，1000,5000,7000,10000,100000,1000000 である. BCPMAX が 1000 のときは, GPU が CPU の平均 1.5 倍程度速いだけだが，BCPMAX を上げていくと CPU と比べた GPU の速度は上がっていき，BCPMAX が 7000 以上では問題ごとに少し性能の差が出てくる．性能がもっとも良いのは BCPMAX が 10000 のときで，GPU が CPU の 6.7 倍程度速いという結果が出た．BCPMAX が 10000 以上では，更なる性能向上は見られそうになく，BCPMAX が 10000 のときと同程度の性能になる.

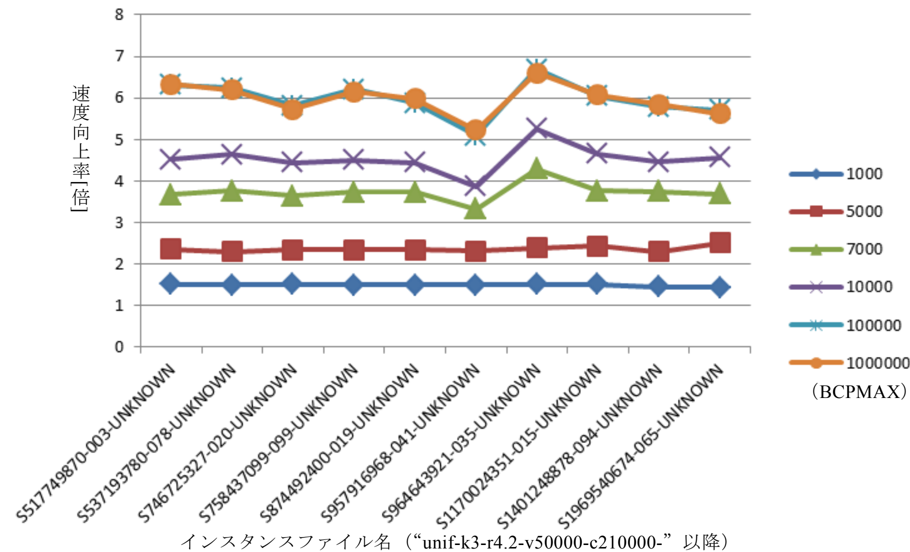

図 11　BCPMAX を変化させたときの性能の変化

5. 関連研究

CUDA を用いた SAT アルゴリズムの並列化の既存研究としては以下の５つがある.

文献 18) で，Meyer らが本論文で用いたのと同じ分割統治法 3SAT を並列化する complete 型の手法を提案している．Meyer らは BCP 処理は用いず，分割統治法自体を並列 に GPU 上で実行するための問題分割法と Decision フェーズで用いる独自のヒューリスティックを提案している.

文献 19) で, McDonald らが節学習を取り入れた WalkSAT アルゴリズムを並列化した incomplete 型の手法を提案している．並列化は，各スレッドで異なる乱数系列を用いて WalkSAT を実行する方法をとる.

文献 20) で，Gulati らが MESP (MiniSAT enhanced wi th Survey Propagation)という complete 型の手法を用いた並列化を提案している．CPU に 2.67 GHz Intel i7，GPU に NVIDIA GeFor ce 280 GT X を用いた環境での速度向上率が平均 2.35 倍という性能を出している.

文献 21) で，Wang がセルラ遺伝アルゴリズムとランダムウォークによる局所探索を組み合わせた incomplete 型 SAT アルゴリズムの並列化を提案している.

文献 22) で，Deleau らが SAT インスタンスを 0-1 行列で表現し，0-1 行列の行列積を用いて SAT の解を探索する incomplete 型の手法を提案している.

6. まとめ

本論文では，SA T アルゴリズムにおける BCP 処理の GPU を用いた並列化方法を提案し，その方法を基に実装したプログラムの評価実験を行った．実験の結果，CPU による逐次 BCP 処理の 6.7 倍の高速化を達成した.

今後の課題としては，未使用のメモリの有効活用や現在は単純に分割している節集合の分割方法を工夫することで，更なる高速化を目指すことが挙げられる.

参考文献

1) 藤田 昌宏： SAT アルゴリズムの最新動向，電子情報通信学会誌, V ol.90,No.12, pp.1067-1072 (2007).

2) Davis, M. and Pu tnum, H. : A Computing Pr ocedure for Quantification Th eory, Jou rnal of th e ACM, Vol.7, No.3, pp.201-215 (1960).

3) Davis, M., L ogemann, G ., and Loveland, D. : A Machine Pr ogram fo r T heorem Pr oving, Communications of the ACM, Vol.5, No.7, pp.394-397 (1962).

4) Davis, J. D., Tan, Z. , Y u, F ., an d Zhang, L . : Designing an Ef ficient Ha rdware Implication Accelerator for SAT Solving, LNCS Vol.4996, pp.48-62 (2008).

5) Princeton University: zChaff, http://www.princeton.edu/~chaff/zchaff.html

ⓒ 2011 Information Processing Society of Japan

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) NVIDIA Corp. : NVIDIA CUDA C Programming Guide, http://developer.nvidia.com/nvidia-gpu-computing-documentation (2011).
9) 青木尊之, 額田彰 : はじめての CUDA プログラミング, 工学社 (2009).
10) Kirk, D. B. and Hwu, W. W. : Programming Massively Parallel Processors: A Hands-on Approach, Morgan Kaufmann (2010).
11) 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).
12) Sanders, J. and Kandrot, E. : CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional (2010).
13) Garland, M. and Kirk, D. B. : Understanding Throughput-Oriented Architectures, Communications of the ACM, Vol.53, No.11, pp.58-66 (2010).
14) Monien, B. and Speckenmeyer, E. : Solving satisfiability in less than 2^n 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).

ⓒ 2011 Information Processing Society of Japan