Vol.2014-SLDM-165 No.19
Vol.2014-EMB-32 No.19
2014/3/15

GP-GPUを用いた高速並列論理シミュレーション手法

橋口 拓哉　　豊永 昌彦　　村岡 道明
高知大学大学院 理学専攻 情報科学分野
〒780-8520 高知県高知市曙町 2-5-1
{hashiguc, toyonaga, muraoka}@is.kochi-u.ac.jp

概要　本研究では，GP-GPU (General Purpose Graphics Processing Unit) を用いた並列論理回路シミュレーションアルゴリズムおよびその高速化手法について提案し、それに基づいたプロトタイプを開発した．さらに，大規模回路のシミュレーションを高速で行うために，本アルゴリズムを基に論理回路を並列処理向きのロジックコーンに変換し，GPU の複数 SM (Streaming Multiprocessor) および複数 GPU を用いて演算の並列度を向上させることにより高速化を図った．その性能を商用シミュレータと比較したところ，組合せ回路で最大43倍，順序回路で最大6.3倍の高速性が得られた．
キーワード：GP-GPU，論理シミュレーション，並列処理，並列アルゴリズム，ストリーミングマルチプロセッサ（SM）

A High-Speed Parallel Logic Simulation Method Using GP-GPU

Takuya Hashiguchi, Masahiko Toyonaga, Michiaki Muraoka
Graduate School of Science, Kochi University
2-5-1 Akebono-cho, Kochi, 780-8520, Japan

Abstract: In this paper, a parallel logic simulation algorithm using GP-GPU and the acceleration method of the algorithm have been proposed. The prototype of logic simulator was developed based on the algorithm. And, the algorithm was improved for faster simulation using fan-out cones converted from the original logic circuit and don't generate data communications with each other. The multi-SMs of a GPU and Multi-GPUs were applied to accelerate the parallel simulation using the fan-out cones. The performance evaluation results show that the Parallel Simulator compared with a commercial logic simulator obtained up to 43x speedup on the combinational circuits and 6.3x speedup on the sequential circuits.
Keywords: GP-GPU, Logic Simulation, Parallel Processing, Parallel Algorithm, Streaming Multiprocessor (SM)

1. はじめに

近年，システムの大規模化や半導体の微細化技術の技術の進歩により，設計の規模や複雑性が急激に増加しており，その検証がますます重要になってきている．しかしながら，大規模な設計の検証には膨大な時間がかかるためその高速化が望まれている．本研究では論理回路検証のための論理シミュレーションに GPU を用いて並列処理することで高速化を図ることを目的とする．この数年間，GP-GPU を用いた論理シミュレーションの並列化[1][2][3]や SystemC の並列シミュレーション[4]などの研究が進められている．しかしながら，これらの先行研究では GPU を使用した高速化手法の詳細があまり示されておらず，高速な市販論理シミュレータなどとの速度比較が明瞭には示されていない．本論文では，GP-GPU を用いた論理シミュレーションの並列化アルゴリズムを提案し，その高速化手法について述べる．また，複数 GPU を使用した場合の性能評価を行い，並列化前のシミュレーションアルゴリズムおよび市販論理シミュレータとの速度比較の結果を報告する．

2. 論理シミュレーションアルゴリズム

論理シミュレーションアルゴリズムは大別してイベントドリブン法とレベルソート法がある．本章では 2 つのアルゴリズムの説明を行う．

2.1 イベントドリブン法

論理シミュレーションを行う際に毎テストパターンに論理回路内の全ての論理ゲートの出力値が変化するわけではないため，イベントドリブン法では入力信号の変化のある論理素子のみの演算を行なう．

Vol.2014-SLDM-165 No.19
Vol.2014-EMB-32 No.19
2014/3/15

2.2 レベルソート法（レベライズド法）

レベルソート法では論理素子の入力信号の変化の有無に関わらずに全論理素子の演算を行なう．この際，論理素子を入力側から接続順に並べ（レベルソート），論理段数（レベル）順に論理素子の演算を実行する．

レベルソート法による論理シミュレーションアルゴリズムを図 1 に示す．図中，四角は論理ゲート，実線は入力信号の変化がないもの，点線は入力信号の変化したものを示す．また，論理段上の論理演算順序を太線で示す．レベルソート法では，外部入力から論理段順にすべての論理ゲートについて論理演算を行うことによりシミュレーションを行う．(i) まず，論理ゲートを外部入力から順に接続段ごとに整列（レベルソート）させる．(ii) 第 1 段目の論理ゲートの演算を行なう．このとき入力信号の変化の有無にかかわらず段上の全ての論理ゲートの演算を行う．(iii) 第 2 段目の論理ゲートの演算を同様に行う．以上の(ii)，(iii) の処理を最終段まで行うことで論理回路の出力を得る．

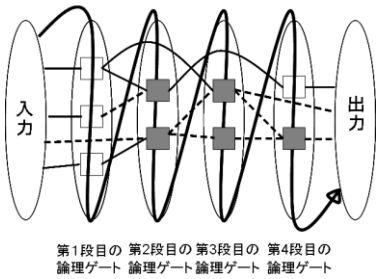

図 1 レベルソート法の手順

2.3 CUDA 概要

CUDA は NVIDIA 社が提供する GP-GPU のプログラム開発環境であり，GP-GPU の並列計算を仮想的に行う[8]．図 2 は本並列論理シミュレーションにおける CUDA の Host 側 (CPU) と Device 側 (GPU) 側のメモリ構成を説明したものである．Device 側は，複数 SM (Streaming Multiprocessor) から構成され，各 SM からアクセスできる 1GB 程度 (Geforce GTX480 の場合) の大容量な Global メモリ (共有メモリ) があり，各 SM 内には高速アクセス可能な Shared メモリ (ローカルメモリ) が実装されている．Host 側からのデータ転送は Device 側の Global メモリに行われる．Shared メモリに Host 側のデータを格納するためには，一旦 Global メモリに転送してから各 SM の Shared メモリへコピーする．

CUDA プログラムの並列演算処理の最小単位をスレッドと呼び，スレッドが並列に処理される．スレッドのまとまりはブロックと呼ばれ，ブロックが自動的に GPU（ハードウェア）の SM に割り当てられる．

Shared メモリは，1 つのブロック内でのみアクセスできるメモリであり，1SM あたり 48KB の容量制限 (Geforce GTX480 の場合) があるため，プログラミングにおいてはデータ構造の工夫が必要となる．本並列論理シミュレーションでは Host 側でシミュレーションの前処理としてネットリストテーブルやテストベクタなどを準備し，それらを Device 側に転送し，並列処理を行う．Device 側では頻繁に使用するデータを Shared メモリに格納し並列論理シミュレーションを行なう．

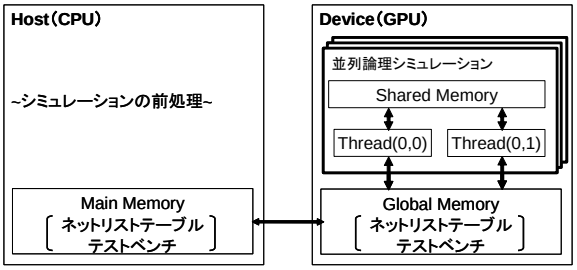

図 2 Host (CPU) と Device (GPU)のメモリ構成

3. 先行研究

先行研究[5][6]ではレベルソート法に基づき並列論理シミュレーションアルゴリズムを提案した．図3に本並列論理シミュレーションの処理手順を示す．

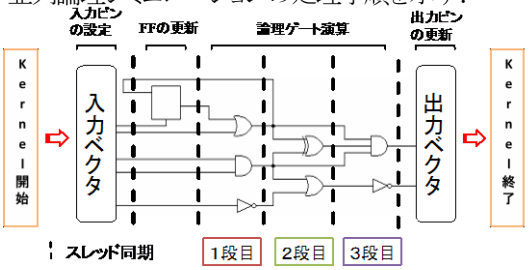

図 3 並列論理シミュレーションの処理手順

本アルゴリズムでは 1 つのゲート (1インスタンス) が CUDA の 1 スレッドに対応する．

(i) 入力ベクタを外部入力端子に設定する．

(ii) FF の出力値の更新を行なう．

(iii) ゲートの論理演算を行う．本並列論理シミュレータの論理演算部分では一度に一段分の論理ゲートを並列に演算するため，最大段数回演算を行なうことで全論理ゲートの演算が完了する．

(iv) 最後に外部出力端子から出力ベクタの出力値を更新する．FF の場合には内部状態を更新する．

入力端子の設定，FF の値の更新，論理ゲート演算，出力端子の更新の4つのそれぞれの処理の終了時にスレッド間の同期をとることにより，全スレッドの処理が完了していない時点で次の処理に移ることを防ぐことができる．

4. 高速化手法

4.1 高速化手法

並列論理シミュレーションアルゴリズムの高速化手法

Vol.2014-SLDM-165 No.19
Vol.2014-EMB-32 No.19
2014/3/15

として「**条件分岐数の削減**」,「**メモリアクセスの高速化**」,「**論理演算時の実行サイクル数の削減**」の 3 つの方法を提案する. 高速化する前のバージョンを (ver.1) とし, 高速化方法に対応したバージョンを (ver.n) と示す.

1) 条件分岐数の削減

1 つ目の高速化方法として, “条件分岐の削減” を提案する. これは, kernel 関数 (GPU 側のプログラム) 内の CUDA スレッドは SIMD 方式で実行されるため, 条件分岐が存在すると下の図 4 に示すようにすべての分岐に対応する処理が実行されるため, 処理性能が低下する. これを避けるため, kernel 関数内の条件分岐数を極力削減する.

図 4 は本論文で実施した並列論理シミュレータの条件分岐数の改良前と改良後を示す. 改良前では kernel 内の論理ゲート演算部分でゲートの種類を switch 文で条件分岐させているため, 分岐数は論理ゲートの種類数となり, 分岐数が 11 であった. 改良後ではあらかじめ入力ピンの本数(1～3)毎に AND や OR などの基本的な論理ゲートの LUT (Look up Table) を作成しておき, 論理演算部分ではその配列にアクセスすることで出力値を求めるように変更した. この改良により条件分岐数を 11 から 3 に削減することができた. (ver.2)

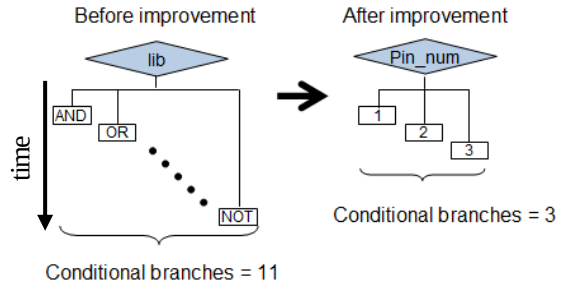

図 4 CUDA での条件分岐処理

2) メモリアクセスの高速化

2 つ目の高速化方法として, メモリアクセスの高速化を提案する. GPU は L1 キャッシュと呼ばれるキャッシュを持っており, L1 キャッシュはグローバルメモリよりもアクセス速度が高速であるため, 頻繁にアクセスされるデータは L1 キャッシュにおくことでデータアクセスを高速に行なうことが可能となる.

本並列論理シミュレーションアルゴリズムでは論理ゲートの演算部分が処理全体の大半を占めているため, キャッシュのヒット率を向上させるために論理ゲート演算の際にアクセスされるネットリストテーブル内のデータ構造を改良するために以下の 2 つの方法を考えた.

ⅰ) 構造体を配列に置換

図 5 の左部分が構造体, 左部分が配列のメモリ配置を表わす. 構造体ではメンバ毎に離散的にメモリ領域が割り当てられるため, データアクセスに時間がかかるが, 配列を使用すると連続した領域にアクセスすることになりアクセス速度の向上が期待できる. (ver.3)

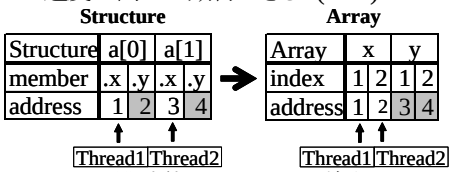

図 5 構造体を配列に置き換える

ⅱ) ネットリストテーブルのソート

ネットリストテーブルをインスタンスのレベル(論理段数)順にソートすることで, レベル上のインスタンスを連続させ, アクセス速度を向上させる. (ver.4)

3) 論理演算時の実行サイクル数の削減

ⅰ) 条件分岐の削除

条件分岐を削除するため, 入力ピン数に依存しない LUT を使用することで条件分岐をなくす. これは, ゲートの入力側のゲートに出力値に影響しないダミーゲートを挿入し, 回路内の全ゲートが持つ入力側のゲート数を揃えることで実現できる (ver.5)

ⅱ) ネットリストデータの連結

論理演算を行なう際にアクセスされる M 個のデータを連結し N 個にする方法を考える. (M > N)
従来のシミュレーションアルゴリズムにおいて, n 番目に演算対象となる論理ゲートの状態値 v[n]の論理演算の計算式を以下に示す.

$$v[n] = LUT\,[\,\mathbf{gfunc[n]} + v[\,\mathbf{in1[n]}\,] + v[\,\mathbf{in2[n]}\,] + v[\,\mathbf{in3[n]}\,]\,];$$

式中の *v[n]*は対象ゲートの状態値, *LUT[ ]*は基本論理ゲートの LUT, *gfunc[n]*はゲートの機能, *in1[n]* / *in2[n]* / *in3[n]*はそれぞれ 1 入力目 / 2 入力目 /3 入力目のファンアウト元のゲート番号を表している. このとき ”n” はすでにレジスタにあるとして式中のメモリアクセス回数を左から順に数えると, ストアは左辺の v[ ]で 1 回, ロードは式中の右辺で, *LUT[gfunc[ ], in1[ ], v[in1[ ], in2[ ], v[in2[ ]], in3[ ], v[in3[ ]]*の計 8 回である
ここで, *gfunc[ ](8bit), in1(16bit), in2(16bit), in3(16bit)* の 4 つのデータをそれぞれ 2 つの 32bit データ (C1, C2) に連結することでメモリアクセス数を削減する.
図 6 にその処理過程を示す. 図 6 の前処理で C1 の上位 16bit に *gfunc[ ]*, 下位 16bit に *in1[ ]*, C2 の上位 16bit に *in2[ ]*, 下位 16bit に *in3[ ]*のデータ内容がそれぞれ保持される. シミュレーション実行時には C1, C2 の上位 16bit のデータは 16bit 右にシフトし, 2^{16}-1 と論理積をとることで元のデータ *gfunc[ ]*, *in2[ ]*が得られ, 下位 16bit のデータは 2^{16}-1 と論理積をとることで元のデータ

Vol.2014-SLDM-165 No.19
Vol.2014-EMB-32 No.19
2014/3/15

*in1[ ], in3[ ]*が得られる. 以下に図6の処理後の論理演算の計算式を示す.

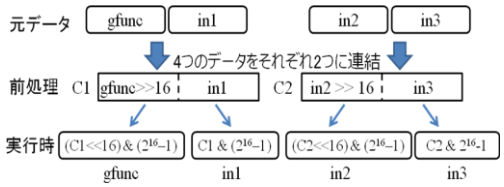

図6 データの連結

$$v[n] = LUT\,[\,(\,C1[n] << 16\,\&\,2^{16}-1\,) + v[\,C1[n]\,\&\,2^{16}-1\,] + v[\,(\,C2[n] << 16\,\&\,2^{16}-1\,)\,] + v[\,C2[n]\,\&\,2^{16}-1\,]\,];$$

上記の計算式のメモリアクセス回数を左から順に数えると, ストアは変わらず左辺の 1 回, ロードは *LUT[ ], C1[ ], v[C1] , v[C2[ ]], C2[ ], v[C2[ ]]*の **6** 回となり, ロード回数が 2 回削減できた. (ver.6)
しかしながら, 16bit シフト演算が 2 回と 16 ビットのマスクビットとの論理積が 4 回増加した.

4.2 高速化手法の評価

並列論理シミュレータの高速化の前後の処理時間の比較を行い, 高速化手法の有効性を評価した. 評価に使用したシミュレータのバージョンを次に示す.

ver.1: 高速化手法適応前のシミュレータ
ver.2: ver.1 + 1) 条件分岐数を 11→3 に削減
ver.3: ver.2 + 2)- ⅰ) 構造体を配列に置換
ver.4: ver.3 + 2)- ⅱ) ネットリストテーブルのソート
ver.5: ver.4 + 1) 条件分岐の削除
ver.6: ver.5 + 2) ネットリストデータの連結

本評価では高速化を図るために, シミュレーション実行時に頻繁にアクセスされる論理ゲートの状態値を Shared メモリに格納した. その結果, 既発表[6]の Shared メモリ未使用のものと比べ, どのバージョンにおいても 1.3～1.5 倍高速になった. 表 1 に ver.1～ver.6 の処理時間を示す. 表 1 が示すように, シミュレータのバージョンおよび回路の並列度が上がるほど高速化率が高くなり, 最も高速である ver.6 では ver.1 と比べ最大で 10 倍以上(cpux16 の場合 7.8 msec→0.7msec)の高速化率が得られた.

表 1 評価結果

10k cycle	処理時間[sec]						ratio
評価回路	ver.1	ver.2	ver.3	ver.4	ver.5	ver.6	ver.1 / ver.6
cpu x 1	1.50	1.45	1.19	0.58	0.31	0.30	5.00
cpu x 4	3.07	2.61	2.11	0.75	0.42	0.41	7.56
cpu x 16	7.97	7.66	5.35	1.47	0.78	0.69	11.58

また, ver.6 でシフト演算, 論理積演算が増加した影響を調べるために, ver5 と ver6 の評価結果を比較すると, cpu x 1 と cpu x 4 では処理速度がほぼ同等で, cpu x 16 では ver.5 の 0.78[msec]から ver.6 では 0.69[msec]となり, ver.6 の方が ver.5 よりも高速であった. この結果より, GPU では論理演算命令数が多少増加してもメモリアクセス命令を削減することにより高速化が期待できる.

5. 複数 SM の使用

前述の並列論理シミュレーションアルゴリズムの高速化については, GPU の単体 SM 上での性能評価が目的であったが, 大規模回路のシミュレーションを高速に行うためには単体 SM では性能に限界があるため, 複数 SM を用いた大規模回路向きのアルゴリズムの確立が必要である. 本章ではそれを可能とする並列シミュレーションアルゴリズムについて述べる.

5.1 ファンアウトコーン

論理回路の並列処理向きの表現方法としてファンアウトコーンや AND/OR プレーン[7]などがあるが, ここではファンアウトコーンを用いた並列化方法を示す.

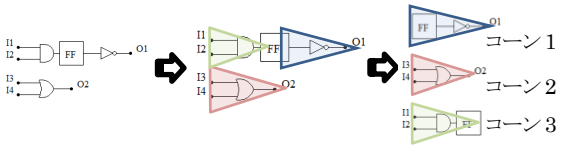

図 7 ファンアウトコーンの生成

図 7 はファンアウトコーンの生成例を示す. ファンアウトコーンは互いにデータ通信が発生しないため並列処理にむくデータ構造である. 図 7 に示した回路では, 外部出力端子 O1 を始点に FF まで辿るコーン 1, O2 を始点に外部入力端子まで辿るコーン 2, FF を始点に外部入力端子まで辿るコーン 3 の合計 3 つのコーンが作成され, それぞれのコーン内には辿る際に遭遇したゲートが存在する. なお, この 3 つのコーン間では互いにデータ通信が発生しないため, 1 つのコーンに対し CUDA の1ブロックを割り付けることでブロック間の通信がない高速並列論理シミュレーションが可能となる. しかしながら, 1つのコーンを CUDA の1ブロック (GPU の1SM) に割り当てる場合, コーン数分のブロックが必要となる. CUDA ではブロック数が SM 数より大きい場合には, 複数のブロックが 1SM に割り当てられて処理されるため処理オーバヘッドが発生する. これを避けるため, ブロック数が SM 数を超えないように複数のコーンをグループ化する方法を考案した.

5.2 コーングループ作成アルゴリズム

複数のコーンを 1 つのグループにまとめたものをコーングループと呼ぶ. GPU の処理時間は最も時間のかかるグループ (CUDA の 1 ブロックに対応) に律速されるため, 各コーングループの処理時間を均等化する. 処理時間はインスタンス数に比例するため, 各グループ内のインスタンス数の均等化が必要である.

Vol.2014-SLDM-165 No.19
Vol.2014-EMB-32 No.19
2014/3/15

図8にコーングループの作成手順を示す. STEP1では論理回路からコーンを抽出する. STEP2 ではコーンをインスタンス数の降順に並べる. STEP3 ではソート後のコーンを順にグループに割り付ける. その際に, 対象のコーンをすでに割り付けられたインスタンス数が最も少ないグループに割り付ける. 以上の処理をすべてのコーンに行ない, 各グループのインスタンス数を均等化させ, 生成されたコーングループをGPUのSMに1:1で割り当て, 複数SMの並列演算を行なう.

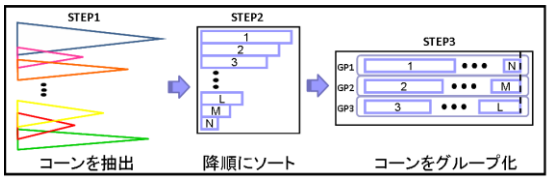

図8 コーングループの作成手順

5.3 複数GPU使用

GPU を複数枚使用することで演算の並列性を更に高めるためには, 各 SM 用のデータ準備と複数GPU(Device)の呼び出しの2つが必要である.

本並列論理シミュレーションアルゴリズムでは 1SM を CUDA の 1 ブロックに対応させる前提で, グループ数 = SM 数と考える. 本評価環境で使用した Geforce GTX480 は 15 個の SM を持つため, コーングループ生成アルゴリズムではグループの数を 15 としている. ここで, GTX480を2枚使用する場合を考えると, SM数が15*2=30個となるためグループ数を30にすることでデータが準備できる. 1つのPCに複数のGPUが接続されている場合, それぞれのGPUはDevice(index)という個別の番号で識別されている. そして CUDA 関数である ”cudaSetDevice(index)” で使用する GPU を指定することができる. 以下に複数 GPU を使用する際のHOST側の擬似コードを示す.

```
for(idx = 0; idx < GPU_NUM; idx++){    //GPU_NUM : GPU数
    cudaSetDevice(idx);     //使用するGPUの指定
    kernel<<< blocks, threads >>>(); //kernel関数の呼出し
}
```

上記の擬似コード中の for 文は接続されている GPU 数回ループさせる. ループ内の処理は使用する GPU を指定し, kernel 関数を呼出す. この記述により kernel 関数は GPU 数分並列に呼出され実行される. 以下に複数GPU使用時のHOSTとDeviceの構成図を示す. 図中, GPU (Device) は各々メモリを持っており, HOSTから必要なデータ(コーングループ)をGPUに転送し, 各GPU内で並列処理を行い, 出力値をHOST側へ転送する. HOST 側では転送されてきたデータを結合する. 以上が複数GPUを使用する場合の手順である.

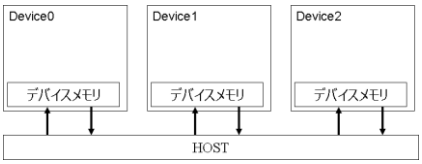

図9 複数GPU使用時のHOSTとDeviceの構成図

6. 評価実験

6.1 実験環境

本アルゴリズムをプロトタイプソフトウェアとして実装し, GPU 実行時間と CPU 実行時間を比較した. 実験環境は以下のとおりである.

・商用シミュレータ ModelSim (VDEC 提供)
- ModelSim SE 6.2e
- PC 環境: Windows XP SP3,
Intel® Core™ i7-950 3.07GHz

・シーケンシャルシミュレータ SEQSim (内部開発のシーケンシャルアルゴリズムレベルソートシミュレータ)
- PC 環境: 商用シミュレータと同様

・提案シミュレータ GPUSim-S (単体 SM 使用[ver.6])
- GPU 環境:CUDA5, Geforce GTX480(480 コア), 1.15GHz(プロセッサクロック), 1.5GHz(メモリクロック)

・提案シミュレータ GPUSim-M (複数 SM 使用)
- GPU 環境:CUDA5, Geforce GTX480 **(x 1 枚)**

・提案シミュレータ GPUSim-x2 (GTX480 x 2 枚)

・提案シミュレータ GPUSim-x3 (GTX480 x 3 枚)

6.2 実験内容および評価データ

単体 SM を使用する GPUsim-S と複数 SM を使用する GPUsim-M, および, GTX480 をそれぞれ 2, 3 枚使用する GPUSim-x2, GPUSim-x3 のシミュレーション実行時間を高速な市販シミュレータと比較した. 実行時間の測定方法は, GPU 環境では NVIDIA CUDA Compute Visual Profiler が示すエラプス時間を用い, PC 環境ではシミュレーションのエラプス時間を用いた. 評価用の論理回路は, 組合せ回路として, 4bit-Adder を 640 個並列に並べた adder4 x 640 と低密度パリティビットエンコーダである LDPC エンコーダ[9] を, 順序回路として, 8bit-CPU を 1, 20, 40 個並列に並べた 3 種類の回路 (cpu x1, cpu x20, cpu x40) を用いた.

表2 評価回路の一覧

評価回路	インスタンス数	入力ピン数	出力ピン数	FF数	ゲート数	論理段数
adder4x640	16009	9	3200	0	12800	10
LDPC エンコーダ	78806	1723	2048	0	75035	12
cpu x 1	2148	19	18	173	2111	56
cpu x 20	42599	19	360	3460	42220	56
cpu x 40	85179	19	720	6920	84440	56

テストベクタ長は 100,000 テストサイクルとし, テストベクタとして, 2 つの組合せ回路はランダム生成したもの, 順序回路である 8bit-CPU はロード, ストア, 加算などの命令を使用した. 表2に評価回路の一覧を示す.

6.3 評価結果

表 3 はレベルソート法ベースの(シーケンシャル / 並列)シミュレータの評価結果を示しており, 表 4 では SEQSim, GPUSim-x3 と市販論理シミュレータである ModelSim の速度の比較結果を示しており, 回路の種類対応に処理時間[sec], 速度比率を表す. なお, 表 3 中の GPUSim-S における LDPC エンコーダと cpux40 の 2 つの回路は結果の斜線部分は評価不可能であることを示す. 理由は, GPU の Shred メモリの容量の上限が 48KB であり, 本アルゴリズムではインスタンスの状態値(1Byte)を Shred メモリに置いているため, インスタンス数が 48K 個を超える回路は Shared メモリの容量不足によるためである. 一方, GPUSim-S 以外の GPUSim では論理回路をコーングループに分割しているため, 48K 個以内に収まり評価が可能である.

表 3 でシーケンシャルなシミュレータ SEQSim(a)と GPU を用いた並列シミュレータ中で最速である GPUSim-x3(e)の処理時間との比較結果を示す. 図中, 組合せ回路では最大で約 25 倍(LDPC エンコーダ), 順序回路では最大で 7.5 倍(cpu x 40)となり, レベルソート法を並列化することで組合せ回路, 順序回路共にシミュレーション時間を高速化することができた.

表 3 レベルソート法シミュレータの比較結果

100k cycle	処理時間[sec]					ratio			
評価回路	SEQSim (a)	GPUSim-S (b)	GPUSim-M	GPUSim-x2 (d)	GPUSim-x3 (e)	a/b	a/c	a/d	a/e
adder4x640	9.20	3.03	1.33	1.30	1.24	3.04	6.92	7.10	7.45
LDPC エンコーダ	44.80		2.78	1.95	1.77		16.11	22.97	25.38
cpu x 1	1.22	3.00	3.22	3.28	3.29	0.41	0.38	0.37	0.37
cpu x 20	24.97	8.78	7.17	6.20	5.45	2.84	3.48	4.03	4.58
cpu x 40	50.62		9.86	7.42	6.75		5.13	6.82	7.50

表 4 からシーケンシャルなレベルソートシミュレータである SEQSim は市販シミュレータと比べ約 1～3.8 倍程度の高速化率となっているが, Geforce GTX480 を 3 枚使用する GPUSim-x3 では約 0.4～43 倍の高速化率となっている. なかでも LDPC エンコーダが約 43 倍と最も高い高速化率となった. これは, 論理段数が 12 段で, 全ゲート数が 75,000 もあるため, 段上のゲート数が多く回路の並列性が高いためである.

表 4 市販シミュレータ (イベント法) との比較結果

100k cycle	処理時間[sec]			ratio		
評価回路	ModelSim (a)	SEQSim (b)	GPUSim-x3 (c)	a/b	b/c	a/c
adder4x640	35.0	9.20	1.24	3.80	7.45	28.34
LDPC エンコーダ	76.5	44.80	1.77	1.71	25.38	43.34
cpu x 1	1.4	1.22	3.29	1.12	0.37	0.42
cpu x 20	20.4	24.97	5.45	0.82	4.58	3.74
cpu x 40	42.5	50.62	6.75	0.84	7.50	6.30

一方, 8bit-CPU を 1 つ並べた cpu x 1 では市販シミュレータの方が高速であったが, これは全ゲート数が 2,000 しかなく, 回路の並列度が低いため GPU の性能を活かすのが難しいためと推測される. 本評価結果より, 本並列論理シミュレータは小規模化回路以外では高速な市販論理シミュレータよりも高速であることを示せた.

7. まとめ

本稿では GP-GPU を用いた並列論理シミュレーションアルゴリズムを基に, GPU アーキテクチャを効率的に利用する目的で, キャッシュのヒット率向上やメモリアクセス回数の削減および複数 GPU を利用するなどの並列論理シミュレータの高速化手法を開発した.

いくつかの評価回路を用いて開発した高速並列論理シミュレータの性能を評価した結果、市販の高速論理シミュレータと比較して組合せ回路で最大 43 倍 (LDPC エンコーダ), 順序回路で最大 6.3 倍 (cpu x40) の高速化率を得ることができた。また, 本評価結果より大規模な回路ほど, 市販論理シミュレータと比較して高速化率が高くなる傾向が見られた.

今後は実用的かつ大規模な回路での評価を行いたいと考えている. また, 複数の論理ゲートを複合化することで論理回路の最大論理段数および論理ゲート数を削減できること[7]より, 本アルゴリズムにゲートの複合化を行うことでさらなる高速化が期待される.

謝辞

本研究は東京大学大規模集積システム設計教育研究センターを通し, メンター・グラフィックス・ジャパン株式会社の協力で行われたものである.

参考文献

[1] Debapriya Chatterjee, Andrew DeOrio, Valeria Bertacco, “Event-Driven Gate-Level Simulation with GP-GPUs”, DAC’09, July 26-31, 2009

[2] Debapriya Chatterjee, Andrew DeOrio, Valeria Bertacco “GCS: HighPerformance Gate Level Simulation with GPGPUs”, 978-3-9810801-5-5/DATE09 © 2009 EDAA

[3] Bo Wang, Yuhao Zhu, Yangdong Deng, “Distributed Time, Conservative Parallel Logic Simulation on GPUs”, DAC’10, June 12-18, 2010

[4] Sara Vinco, Debapriya Chatterjee, “SAGA: SystemC Acceleration on GPU Architectures”, DAC 2012, June 3-7, 2012

[5] 大菊祥子, 橋口拓哉, 豊永昌彦, 村岡道明 ”GP-GPU を用いた並列論理シミュレーションアルゴリズムの評価”, DA シンポジウム 2012 論文集, pp.109-114, 2012 年 8 月 29 日

[6]橋口拓哉, 豊永昌彦, 村岡道明 ”GP-GPUを用いた並列論理シミュレーション手法” DA シンポジウム 2013 論文集, pp. 97-102, 2013 年 8 月 22 日

[7] 竹内勇矢, トウブンチク, 村岡道明 ”並列化アルゴリズムによる論理シミュレーションの高速化手法の提案”, DA シンポジウム 2013 論文集, pp.91-96, 2013 年 8 月 22 日

[8] NVIDIA CUDA Compute Unified Device Architecture

[9] Open Cores. htpp://www.opencores.org