Technical Reports on Information and
Computer Science from Kochi
Vol. 5 (2013), No. 2

# GPGPU におけるファンアウトコーンに基づく 並列論理シミュレーション法の研究

A Study of Parallel Logic Simulation Method based on Fan-out Cone Sub-Circuit on GP-GPU

那須 升亮 1） 大菊 祥子 2) 豊永 昌彦 3）

Shosuke Nasu 1） Sachiko Ogiku 2) Masahiko Toyonaga 3）

1）高知大学理学部 2）高知大学大学院理学専攻 3)高知大学 情報講座

Information Science Division, Faculty of Science, Kochi University

**あらまし**

**本論文は，GPGPU を用いたゲートレベルの論理シミュレーションを行う並列論理シミュレーション法の提案と，大規模回路での本手法の有効性の検証を行なう．具体的には，レベルソート法（レベリング法）を基本に，ファンアウトコーンによって GPGPU の複数の Block（MP）を使用した大規模ゲート数の回路の並列シミュレーションアルゴリズムを考案し，GPGPU で実装して，その性能の評価を行う．評価回路として 4bit Adder×16 から 4bit Adder×640 を用いて評価実験を行ったところ，CPU の論理シミュレータと比較してほぼ同等の処理速度であることを確認できた．また 1 つの GPGPU の Block で扱える並列処理数を超える論理ゲート数の回路で並列論理シミュレーションが可能であることを確認できた．**

**キーワード：論理シミュレーション，並列処理，レベルソート法，GPGPU**

## 1. はじめに

情報化社会の発展により，従来に比べ高性能で安価な電子機器が求められている．このような高性能で安価な電子機器を実現するため VLSI(Very Large Scale Integration)回路が大規模化している．例えばスマートフォンではディスプレイの高精細化，高精細カメラ，ワンセグ受信機能，ブルートゥース内蔵，オートフォーカス・ズーム機能，電子決済機能，グローバル通信機能などを搭載するために VLSI 回路が大規模化している[1].

一方電子機器の市場は，製品開発サイクルが短期化している [2]．スマートフォンの販売台数は増加し，その他の携帯電話の出荷個数は減少すると推計されている．それに伴い，スマートフォンの市場が拡大し，利用者からは新機種をより早く手に入れたいという要求がある．これに応えるため，VLSI 回路の設計期間の短縮が望まれるが，回路規模増大に伴って設計期間が長期化している．VLSI の正常動作を検証する機能・論理シミュレーションは全設計期間の 7 割以上を占めているが，その検証時間は更に長くなっている．

このように設計検証が長期化すると，製品提供（リリース）が遅れるため，市場の要望にこたえることができない．したがって製品開発サイクルを短くするには，設計検証の高速化が不可欠である．

最近になり，汎用並列計算機としてグラフィック処理用コア GPU(Graphics Processing Unit)を利用する技術 GPGPU(General Purpose GPU)が普及し，これを利用する論理シミュレーション法が提案されてきている．例えばGPGPU を利用したイベントドリブン法による論理シミュレーション[3]や SystemC を並列シミュレーションする方法[4]，GPGPU のメモリ再利用で高速化を目指した並列シミュレーション法[5]や，独立性の高い回路分割で各回路動作を GPGPU で並列シミュレーションする方法[6]が提案されている．論理素子を並列計算したレベルソート法の論理シミュレーション法 [7] も提案されている．

これらの中で，レベルソート法の並列論理シミュレーション法では，並列計算の同期が保証できる GPGPU の 1 つのマルチプロセッサ(Block)内の処理フロー(Thread)を用いて，イベントドリブン法の商用の ModelSim と同等の高速処理を達成していることが報告された[8]．しかし，GPGPU では，1 つの Block 内の Thread 数に制限がある．そのため回路規模(論理ゲート数)が制限され，より大規模な回路の高速化の障害となっていた．より大規模な回路でこれを実行するためには，複数の Block を活用するイベントドリブン法の並列論理シミュレーション法が望まれる．

本研究では，Block 内の Thread 数の制限問題を解決するため回路を分割して複数の Block に割り当て並列処理を行う論理シミュレーションを提案する．具体的には，レベルソート法を基本とし， GPGPU の多数の Block の Thread へ分割した回路の論理ゲートを割り当てて，並列計算することで論理シミュレーションを行う方法を提案するものである．

提案手法で分割は，論理回路のネットリスト（使用する論理ゲートや入出力端子と接続，ゲート種類情報）を入力とし，回路の各出力に関する部分回路（ファンアウトコーン）を抽出し，これらファンアウトコーンごとに Block を割り当てて並列計算する．この有効性を確かめるため，提案手法を NVIDIA 社の GeForce や Tesla が使える GPGPU 計算環境を用いて並

列論理シミュレータを実装し，CPU に実装したレベルソート法と比べて処理速度の評価を行う．

回路規模による効果を評価するため，対象とする回路は 4bit Adder×16 から 4bit Adder×640 を用いた．実験結果によれば，評価回路の範囲では CPU によるレベルソート法と比べて同等の処理速度となることを確認した．

以下，本論文の構成は，次の第 2 章において，以後の議論の準備として従来の論理シミュレーションの概要，GPGPU 環境での実装法の概要および先行研究のレベルソート法を説明する．第 3 章で提案する並列計算アルゴリズムによる論理シミュレータについて説明する．第 4 章で評価実験について説明し，評価結果を示す．第 5 章でまとめをおこなう．

## 2. 準備

### (1) 論理シミュレーションとは

AND や OR などの論理素子で構成した論理回路(ゲートレベル回路)の検証では，論理シミュレーションを用いる．論理シミュレーションの方法は，回路入力パターン(テストベクタ)を定義して，各素子の機能に応じて入力から出力を計算する．実装に用いるアルゴリズムは，市販シミュレーションを含め，ここ数 10 年にわたりイベントドリブン法を主流として高速手法が研究されてきた．

イベントドリブン法は，入力信号変化を事象変化(イベント)とみなして，入力信号がイベントとして判明した論理ゲートのみを演算することで出力を求める．具体的には，入力信号が 1 から 0，あるいは 0 から 1 へ遷移したものをイベント信号としている．

図 2.1 を用いてイベントドリブン法による論理シミュレーションの動作を説明する．論理ゲートを四角で示し，特にイベント信号を入力する論理ゲートを灰色で表す．各ゲート間の有向枝は信号伝搬 (結線)を示し，破線の有向枝はイベント信号を示す．

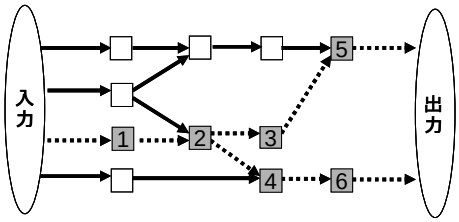



図 2.1 イベントドリブン法

“1”とラベル付された四角とその入力がイベントとする．すると，ラベル1の論理ゲートの演算を行う．続いてラベル 1 の論理ゲートの出力がイベントとなり，その先のラベル 2 の論理ゲートの演算を行う．このように入力信号がイベントと判明した論理ゲートの演算を順番に行うことで回路の出力を得ることができる．

以上のようにイベントドリブン法では，論理回路の出力は，イベントのあった論理ゲートのみである．

図 2.1 ではラベル付された論理ゲートのみを入力順に時間に応じて演算する．

CPU 回路などではクロック 1 サイクル内で，全論理ゲートの入力信号のイベント発生率は数%程度といわれている．そのため，論理シミュレーションをイベントドリブン法で実行するとわずかな演算で，出力を効率よく計算できる．一方，イベント発生を管理し，また時間経過を管理するために複雑なデータ構造や機構が必要となる．

図 2.2 を用いて，レベルソート法による論理シミュレーションの動作を説明する．回路を入力から順に接続段ごとに整理し，同一段の論理ゲートを楕円で囲んでいる．ラベル 1, 2, 3, 4 は入力から 1 段目の論理ゲートであり，ラベル 5, 6 は 2 段目，ラベル 7, 8, 9 は 3 段目，ラベル 10, 11 は第 4 段の論理ゲートである．

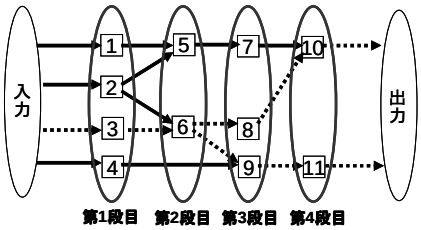



図 2.2 レベルソート法

ラベル 3 の論理ゲートの入力にイベントがあったとする．レベルソート法では，他の信号のイベントにかかわらず第 1 段目の論理ゲート全ての演算行う．続いて第 2 段目の全ての論理ゲートの演算をおこなう．このとき，第 1 段目の出力のイベントの有無は考慮しない．同様に最終段まで論理ゲートを順に演算すると，回路の出力が計算できる．

レベルソート法では，イベントのない入力をもつ論理ゲートも演算するため，無駄な計算が生じる．一方，管理が容易で計算が単純にできる．

### (2) GPGPU による並列計算環境(CUDA)

#### ① GPGPU

GPU(Graphic Processing Unit)は多数の画像プロセッサを有する画像処理装置の名称である GPGPU(General Purpose GPU)は，GPU が有する多数のプロセッサを並列計算機として利用することをいう．CPU(Central Processing Unit)に比べて，プロセッサの性能は劣るが多数のプロセッサで並列計算ができるため処理を高速化できる可能性がある．

GPGPU を利用するプログラムを実装するには，並列計算環境 CUDA や OpenCL などを利用する．CUDA は，NVIDIA 社が提供したプログラミング環境であり，OpenCL は，他の GPU でも利用可能な開発環境である．本研究では，実績が高く解説資料が豊富な CUDA を用いることにした．

#### ② GPGPU(ハード)の構成

GPGPU を利用するプログラムは，2 種類のプログラムを組み合わせて構成する．1 つは，パソコンの CPU(Host)

で動作させるプログラム，もう1つは GPU(Device)で動作させる Kernel と呼ぶプログラムである[8]. GPGPU のハードの構成を図 2.3 に示す. Device 側のメモリは，Host と独立しており， GPU の全 Thread からアクセス可能な Device Memory，Thread ごとに用意された Local Memory，Block 内の Thread が共有してアクセス可能な Share Memory などがある.

GPU のプログラム Kernel の入出力は，大容量の Device Memory のみである. そのため，Kernel を動作する前に，入力を Host が Device へコピーし，Kernel による並列計算結果は，Device から Host へコピーし，最終的に Host が出力する. Share Memory は，16KB 程度で小容量であるが高速にアクセスできる.

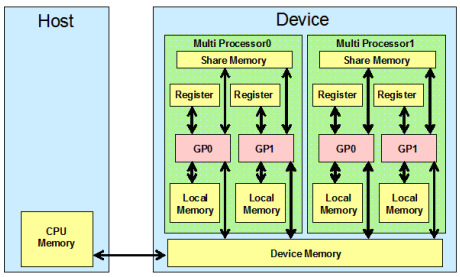



図 2.3 GPGPU のハードの構成

### ③ GPGPU(ソフト)の構成(CUDA)

NVIDIA 社の提供するプログラミング環境 CUDA では，単一の処理 Thread，いくつかの Thread を管理する Block で形成している. 1 つの Block は，最大 512(CUDA3.2)または 1024(CUDA3.3 以降)の Thread を同期して利用することができる. Block は最大 65535 まで利用することができる. Device 側の各 Thread で動作させるプログラムを Kernel と呼ぶ.

図 2.4 は，Host 処理におけるデータの流れと，Kernel 処理の呼び出し，さらに Kernel が Device の Thread で実行される様子を示している.

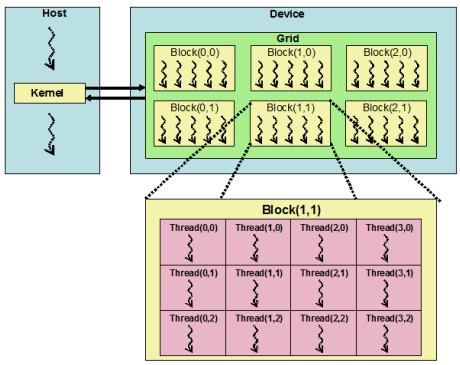



図 2.4 Kernel で各 Thread が実行される様子

### (3) 先行研究

先行研究では GPGPU の 1 つの Block を用い，Block 内の各 Thread に論理シミュレーションを行う回路の論理ゲートと入出力を割り当て，レベルソート法を基本とし並列論理シミュレーションを行った. ここでは先行研究の並列論理シミュレーションについて図 2.5 と図 2.6 を用いて説明する. 入出力を含めた論理ゲートを四角で示し，1 から 11 とラベル付けし，入出力を IN1~IN4，OUT1 と OUT2 と名前付けしている. 特にイベント信号を入力する論理ゲートを灰色で表す. 各ゲート間の有向枝は信号伝搬 (結線)を示し，破線の有向枝はイベント信号を示す.

ここでは，論理ゲートだけでなく，入出力も回路段数に数えることとする. そのため第 2 章(1)で紹介したレベルソート法による回路段数に，入出力を回路段数として加える. シミュレーションを行う回路を，入出力も回路段数として加え，接続段ごとに整理して楕円で囲んだ模式図を図 2.5 に示す. また GPGPU の Block 内の Thread に，図 2.5 のラベル 1 から 11，IN1 から IN4，OUT1 と OUT2 の論理ゲート(入出力を含む)が割り当てられる様子を図 2.13 に示す. このとき，Thread に割り当てられる論理ゲート(入出力を含む)はランダムである. また図 2.6 では割り当てられる論理ゲート(入出力を含む)の数が 12 を超えているため，1 つの Thread に複数の論理ゲート(入出力を含む)が割り当てられる.

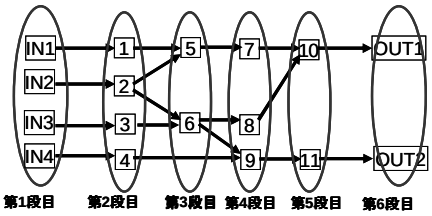



図 2.5 シミュレーションを行う回路の模式図

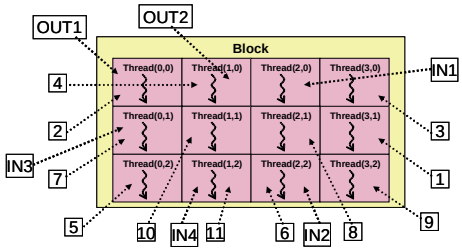



図 2.6 Thread への割り当て

先行研究では GPGPU の 1 つの Block を使用し並列論理シミュレーションを行っていた. しかし第 2 章(2)の GPGPU(ソフト)の構成(CUDA)で説明したように，1 つの Block に定義可能な Thread の最大数は 512 または 1024(先行研究では 512)までとなっている.

そのため先行研究では GPGPU の Block を複数利用することにより，論理シミュレーションを行う回路の論理ゲート数が 512 を超える回路で並列論理シミュレーションを行うことが課題であった.

## 3. GPGPU におけるファンアウトコーンに基づく並列論理シミュレーション法

本手法は，GPGPU の Block の複数利用により，先

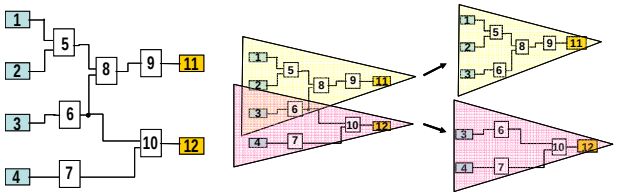
行研究による並列論理シミュレーションでは扱えなかった論理ゲート数 512 を超える論理回路での論理シミュレーションを目指す. 具体的にはレベルソート法（レベリング法）を基本に並列処理向きアルゴリズムを考案し, ファンアウトコーンによって回路を分割して, 各分割した回路を GPGPU の複数の Block（MP）で計算する方法で実現する.

### (1) ファンアウトコーン

GPGPU で 1 つの Block でシミュレーションを行う理由は, 各 Block 内のみで同期して並列に演算を行えるからである. 提案手法では回路を分割する方法としてファンアウトコーンを作成する方法を使う.

ファンアウトコーンとは論理回路において, 各出力に影響する論理ゲートを集めた部分回路のことである. またファンアウトコーンにより回路を分割することで 1 つの Block に割り当てる論理ゲートの数を少なくすることができる. そのため回路の論理ゲートの数が 1 つの Block で定義できる Thread 数(512Thread)を超える回路であっても 1 つのファンアウトコーン内の論理ゲートの数が 1 つの Block で定義できる Thread 数を超えなければシミュレーション可能になる.

図 3.1 を用いてファンアウトコーンの作成方法について説明する. ファンアウトコーンを作成する回路を, 入力は青の四角, 論理ゲートは白の四角, 出力は黄色の四角として図 3.1 に示す.




(a) 回路図　　(b) ファンアウトコーン

図 3.1 ファンアウトコーンの作成手順

図 3.1 の(a)において黄色い四角で示したラベル 11 の出力に影響する論理ゲートと入力を出力側から取り出すと, ラベル 9, 8, 6, 5 の論理ゲートとラベル 3, 2, 1 の入力である. 同様にラベル 12 の出力に対しても影響する論理ゲートと入力を取り出すと, ラベル 10, 7, 6 の論理ゲートとラベル 4, 3 の入力である.

ラベル 11, 12 の出力に対するファンアウトコーンを作成し, 独立した回路に分割した様子を図 3.1 の(b)に示す. クリーム色の三角形はラベル 11 の出力から作成したファンアウトコーンを表している. ピンク色の三角形はラベル 12 の出力から作成したファンアウトコーンを表している.

ラベル 6 の論理ゲートとラベル 3 の入力は 2 つのファンアウトコーンに含まれているが, それぞれのファンアウトコーンで別々に定義される.

### (2) アルゴリズム

本研究で提案する GPGPU におけるファンアウトコーンに基づく並列論理シミュレーションのアルゴリズムを図 3.2 に示す. 図 3.2 の Host は CPU により行われる処理のことであり, Device は GPGPU により行われる並列処理のことである.

```
<Host>
Step0.  ファンアウトコーン(FC)作成
Step1.  論理回路の段数計算
Step2.  テストベクタをDeviceに転送
Step3.  以下Step4, Step5, Step6をテストベクタ長回繰り返す
Step4.  カーネル関数を呼び出す
<Device>
Step5.  テストベクタの1パターンを入力に設定
Step6.  FCごとにBlock割り付け,以下FCごとに並列処理
        Step6.1.  Threadが1つの論理ゲート演算を以下最大段数まで繰り返す
                Step6.1.1.  論理ゲートの入力値を読み込み
                Step6.1.2.  論理ゲート演算
                Step6.1.3.  論理ゲートの出力値の書き込み
<Host>
Step7.  全出力をDeviceから受け取る
Step8.  出力ベクタを出力
```

図 3.2 アルゴリズム

Step0.から Step4.までは Host 側で行われる処理を示している. まず Step0.ではファンアウトコーンを作成する, 次に Step1.で論理回路の段数計算を行う, 論理回路の段数計算とは, 論理シミュレーションを行う回路をレベルソート法によって入出力を含め, 接続段ごとに整理し, 出力まで最大で何段あるかを計算することである. Step2.はテストベクタと呼ばれる論理シミュレーションを行う回路に入力として与える入力信号のパターンを Host のメモリから Device のメモリへ転送する. Step3.は Step4.から Step6.までの処理をテストベクタ長回繰り返す. テストベクタ長回とは, シミュレーション回数のことであり, 10,000 パターンのテストベクタでシミュレーションを行うとき, テストベクタ長は 10000 であり, シミュレーション回数は 10,000 回である. Step4.では Device 側で行う処理を表すカーネル関数の呼び出しを行う.

Step5.と Step6.は Device での処理を表している. Step5.はテストベクタの 1 パターンを, 論理シミュレーションを行う回路の入力としてデータを格納する状態テーブルに設定する. Step6.は Device(GPGPU)の Block に Step0.で作成したファンアウトコーンを割り付ける. Step6.1 ではファンアウトコーン内の論理ゲートを Block 内の Thread に割り付け Step1 で計算した論理回路の最大段数回 Step6.1.1.から Step6.1.3.までの処理を繰り返す. Step6.1.1.で論理ゲートの入力値を読み込み, Step6.1.2.ではそれぞれの論理ゲートの種類に応じて演算を行う. Step6.1.3.は演算の結果を出力する.

Step7.と Step8.は再び Host 側で行われる処理となる. Step7.では Device 側から転送されるシミュレーションを行った回路の全出力を受け取り, Step8 で出力ベクタを出力し論理シミュレーションを終了する. 出力ベクタとは論理シミュレーションを行った回路の全出力のことである.

### (3) 具体例

提案する GPGPU におけるファンアウトコーンに基づく並列論理シミュレーションについて図 3.4 から 3.13 を用いて説明する.

具体例として本研究の提案手法で並列論理シミュレーションを行う回路図を図 3.4 に示す. IN1 から IN4 は入力を示している. 各論理ゲートの下に論理ゲートの種類を示している. OUT1 と OUT2 は出力を示している.

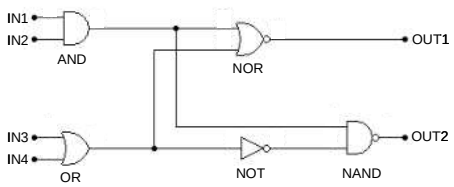


図 3.3 具体例に用いる回路図

Step0.として図 3.3 の出力 OUT1 と OUT2 についてファンアウトコーンを作成し，Step1.として回路の段数も計算する.

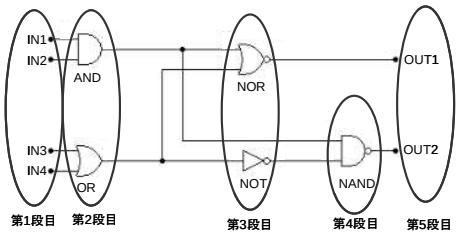


図 3.4 回路の段数計算

ここでは図 3.4 に示すように段数は 5 段であることがわかる.

出力 OUT1 に影響する論理ゲートと入力は NOR, OR, AND, IN1, IN2, IN3, IN4 である. OUT2 に影響する論理ゲートと入力は NAND, NOT, OR, AND, IN1, IN2, IN3, IN4 である. これらの出力 OUT1 のファンアウトコーンを**コーン 1**, 出力 OUT2 のファンアウトコーンを**コーン 2** と呼ぶ. 作成したコーン 1 とコーン 2 を Step5.以降で Device の Block へ割り当てる様子を図 3.5 に示す.

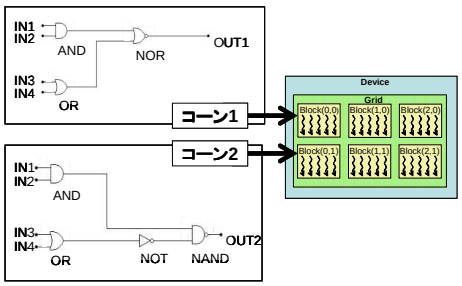


図 3.5 ファンアウトコーンを Block に割り当て

コーン 1 内の論理ゲートを Block(0,0)内の Thread へ割り当てる様子を図 3.6 に示す. 論理ゲート内のラベル 1 から 3 は論理ゲートの番号を示している. 灰色の点線の有向枝はコーン 1 内の論理ゲートを Block(0,0)内の各 Thread へ割り当てる様子を模式的に示している. コーン 1 にはラベル 1 の AND, 2 の OR, 3 の NOR の 3 つの論理ゲートと IN1, IN2, IN3, IN4 の入力, OUT1 の出力が存在している. これらの全ての論理ゲートを Block(0,0)内の Thread に割り当てる. 図 3.8 ではラベル 1 の AND は Thread(0,1), 2 の OR は Thread(1,0), 3 の NOR は Thread(3,0), IN1 は Thread(0,0), IN2 は Thread(1,1), IN3 は Thread(0,2), IN4 は Thread(1,2), OUT1 は Thread(3,1)へと割り当てられる. 論理ゲートの割り当てられた Thread はピンク色, 論理ゲートを割り当てられなかった Thread は黄色で表している.

同様にコーン 2 についても, コーン 2 内の論理ゲートを Block(0,1)の各 Thread へ割り当てる.

全てのコーンが Block に割り当てられたとき, すなわち各 Block 内の Thread がどのコーンの, どの論理ゲートについて処理を行うかが決まると, Step6.1.として, 入力ベクタが IN1, IN2, IN3, IN4 に与えられ各 Thread は自分に割り当てられている論理ゲートに対応した処理を行う.

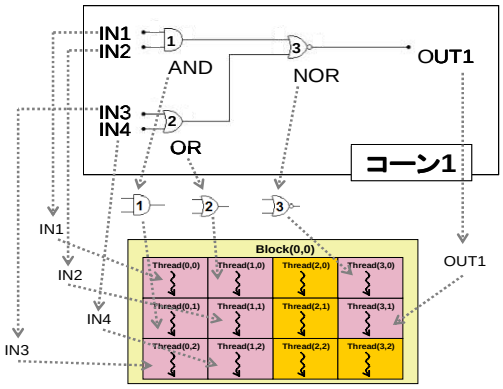


図 3.8 コーン 1 の論理ゲートを Block 内の Thread の割り当て

本手法はレベルソート法を基本としたアルゴリズムである. そのため図 3.4 より論理回路の段数である 5 段, すなわち 5 回各 Thread は並列処理を行う.

## 4. 実験と考察

### (1) 実験内容

CPU における論理シミュレーションを H, CPU におけるファンアウトコーンを用いた論理シミュレーションを HFC, 本手法である GPGPU におけるファンアウトコーンに基づく並列論理シミュレーションを FCP とする. この H, HFC, FCP の論理シミュレーションを 4bit Adder を複数個並べた回路で行うことで, 大規模な回路でのシミュレーションが可能であるかどうかと, 論理シミュレーション時間の比較により, 本手法による大規模回路での並列化の効果について評価する.

### (2) 実験環境と評価回路

提案手法をプログラム開発環境 Microsoft Visual C++ 2008 Express Edition と NVIDIA 社の CUDA

で C 言語により実装し，人工的に作成した回路で評価する実験を行う．

使用したハードウェアのスペックを表 4.1 に示す．

表 4.1 CPU と GPU の環境

| CPU | |
|---|---|
| CPU | Intel(R)Core(TM)2 Duo CPU E8400 |
| 周波数 | 2.99GHz |
| メモリサイズ | 2.00GB |
| FSB(メモリクロック) | 1.3GHz |

| GPU | GeForce GTX 480 |
|---|---|
| CUDAコア数 | 480 |
| ブロック数 | 15 |
| 1ブロックあたりのコア数 | 32 |
| プロセッサクロック | 1.4GHz |
| メモリサイズ | 1.5GB |
| メモリクロック | 1.8GHz |
| メモリバンド幅(GB/sec) | 177.4 |

評価実験として論理シミュレーションを行う回路は 4bit Adder を複数個並べた論理回路を用いる．X[0:3],Y[0:3]は 4bit の入力，Cin はキャリーインを表している．S[0:3]は 4bit の出力を表し，Cout はキャリーアウトを表している．図 4.1 に 4bit Adder を N 個並べた模式図を示す．4bit Adder を用いる理由としては，先行研究の評価実験では 4bit Adder を 16 個並べた回路までしか論理シミュレーションを行うことができなかったため，本手法にて 4bit Adder を 16 個以上並べた回路の論理シミュレーションを行うことにより，大規模回路での論理シミュレーションが可能であるかどうかを確認するためである．また複数個並べることによって論理回路の論理ゲート数の増加が容易に行えるからである．

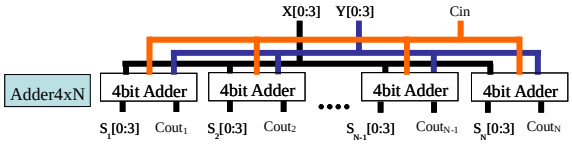



図 4.1　4bit Adder×N の模式図

4bit Adder を複数個並べた回路の論理ゲート(GATE)数，入力(in)数，出力(out)数，インスタンス(inst)数，端子(pin)数，配線(net)数，論理回路の最大段数を表 4.2 に示す．inst 数とは論理ゲート数に入出力数を加えたものである．

表 4.2　4bit Adder ×　N の内容

| N | GATE | in | out | inst | pin | net | 最大段数 |
|---|---|---|---|---|---|---|---|
| 1 | 20 | 9 | 5 | 34 | 82 | 29 | 10 |
| 16 | 320 | 9 | 80 | 409 | 1,177 | 329 | 10 |
| 40 | 800 | 9 | 200 | 1,009 | 2,929 | 809 | 10 |
| 80 | 1,600 | 9 | 400 | 2,009 | 5,849 | 1,609 | 10 |
| 160 | 3,200 | 9 | 800 | 4,009 | 11,689 | 3,209 | 10 |
| 320 | 6,400 | 9 | 1,600 | 8,009 | 23,369 | 6,409 | 10 |
| 640 | 12,800 | 9 | 3,200 | 16,009 | 46,729 | 12,809 | 10 |

### (3) 実験結果

H と HFC と FCP で，テストベクタ長 1,000 から 10,000 において 4bit Adder×16，×160，×640 の評価用回路を論理シミュレーションした処理時間と，FCP の H と HFC との速度比を表 4.3 から表 4.5 に示す．

表 4.3 の 4bit Adder×16 における実験結果では，FCP は H と HFC に比べてテストベクタ長 1,000 のとき H は 0.031 秒，HFC は 0.093 秒，FCP は 0.678 秒となり，FCP は H に比べ約 22 倍，HFC に比べ約 7.3 倍遅いことが確認された．またテストベクタ長 10,000 のとき H は 0.281 秒，HFC は 0.875 秒，FCP は 1.030 秒となり FCP は H に比べ約 3.7 倍，HFC に比べ約 1.2 倍遅いことが確認された．

表 4.4 の 4bit Adder×160 における実験結果では，FCP は H と HFC に比べてテストベクタ長 1,000 のとき H は 0.250 秒，HFC は 3.750 秒，FCP は 0.866 秒となり，FCP は H に比べ約 3.5 倍遅いが，HFC に比べると約 4.3 倍早いことが確認された．またテストベクタ長 10,000 のとき H は 2.422 秒，HFC は 37.406 秒，FCP は 3.010 秒となり FCP は H に比べ約 1.2 倍遅いが，HFC に比べ約 12.5 倍早いことが確認された．

表 4.5 の 4bit Adder×640 における実験結果では，FCP は H と HFC に比べてテストベクタ長 1,000 のとき H は 0.953 秒，HFC は 56.312 秒，FCP は 1.525 秒となり，FCP は H に比べ 1.6 倍遅いが，HFC に比べると約 37 倍早いことが確認された．またテストベクタ長 10,000 のとき H は 9.813 秒，HFC は 565.938 秒，FCP は 9.553 秒となり FCP は H に比べほぼ同等の処理時間となり，HFC と比べると約 59 倍早いことが確認された．

表 4.3　4bit Adder×16 における実験結果

| テストベクタ長 | H(sec) | HFC(sec) | FCP(sec) | FCP/H | FCP/HFC |
|---|---|---|---|---|---|
| 1,000 | 0.031 | 0.093 | 0.678 | 21.871 | 7.290 |
| 5,000 | 0.140 | 0.468 | 0.839 | 5.993 | 1.793 |
| 10,000 | 0.281 | 0.875 | 1.030 | 3.665 | 1.177 |

表 4.4　4bit Adder×160 における実験結果

| テストベクタ長 | H(sec) | HFC(sec) | FCP(sec) | FCP/H | FCP/HFC |
|---|---|---|---|---|---|
| 1,000 | 0.250 | 3.750 | 0.866 | 3.464 | 0.231 |
| 5,000 | 1.203 | 18.734 | 1.852 | 1.539 | 0.099 |
| 10,000 | 2.422 | 37.406 | 3.010 | 1.243 | 0.080 |

表 4.5　4bit Adder×640 における実験結果

| テストベクタ長 | H(sec) | HFC(sec) | FCP(sec) | FCP/H | FCP/HFC |
|---|---|---|---|---|---|
| 1,000 | 0.953 | 56.312 | 1.525 | 1.600 | 0.027 |
| 5,000 | 4.812 | 281.563 | 5.097 | 1.059 | 0.018 |
| 10,000 | 9.813 | 565.938 | 9.553 | 0.974 | 0.017 |

### (4) 考察

まず FCP と HFC について考察する．4bit Adder×16 のテストベクタ長 10,000 のとき，FCP の処理時間は，HFC の処理時間に比べ，1.2 倍遅いことがわかる．これは 4bit Adder×16 において FCP で行う並列処理が少なかったためだと考えられる．4bit Adder×160 のテストベクタ長 10,000 のとき FCP の処理時間は，HFC の処理時間に比べ，12.5 倍早いことがわかる．また 4bit Adder×640 のテストベクタ長 10,000 のとき FCP の処理時間は，HFC の処理時間に比べ，約 59 倍早いことがわかる．これは 4bit Adder×160，×640 において，FCP で行う並列処理が増加したためだと考えられる．

次に FCP と H について考察する．4bit Adder×16 のテストベクタ長 1,000 のとき，FCP の処理時間は，H に比べ約 22 倍遅いことがわかる．これは，メモリの準備などのオーバーヘッドの比率が大きいためだと考えられる．そのためテストベクタ長 10,000 のとき，並列処理の時間が増え，オーバーヘッドの比率が緩和

されたと考えられる．4bit Adder×160 のテストベクタ長 10,000 のとき，FCP の処理時間は，H の処理時間に比べ 1.2 倍遅いが，4bit Adder×640 のテストベクタ長 10,000 のとき，FCP の処理時間は，H の処理時間とほぼ同等であった．これにより，大規模回路を並列論理シミュレーションすることで，H に比べ，速度優位性が得られる．

先行研究に比べて大規模な回路を論理シミュレーションすることができた．これは論理シミュレーションを行う回路を，ファンアウトコーンによって分割することで，1 つの Block で並列論理シミュレーションを行う論理ゲート数を Thread 数制限内に抑えることができたためだと考えられる．

## 5. まとめ

本論文では，GPGPU におけるファンアウトコーンに基づく並列論理シミュレーション法を提案した．具体的には，論理シミュレーションを行う回路をファンアウトコーンにより分割し，分割した回路を GPGPU の Block に割り当て並列論理シミュレーションを行うものである．提案手法を 4bit Adder を複数個並べた回路を用いて評価実験を行ったところ，論理ゲート数 16,009(入出力含む)の 4bit Adder×640 個の回路において，CPU の論理シミュレータと比較してほぼ同等の処理速度であることを確認した．また 1 つの Block で扱える論理ゲートの数を超える回路での並列論理シミュレーションが可能であることを確認した．
今後の課題としては，商用シミュレータとの比較と，更なる高速化にむけたアルゴリズムの改良があげられる．

## 謝辞

本研究は，科学研究費補助金（JSPS 科研費 22500049）の助成をうけたもので，独立行政法人日本学術振興会に感謝いたします．

## 参考文献

[1] EE Times Japan 「高機能化進むスマホ、高まるデジタル家電市場活性化への期待」
http://eetimes.jp/ee/articles/1211/30/news051.html
[2] 浜銀総研 News Release「スマートフォン市場拡大の恩恵を受ける電子部品業界」
http://www.yokohama-ri.co.jp/html/report/pdf/pr121010.pdf
[3] Debapriya Chatterjee, Andrew DeOrio, Valeria Bertacco, “Event-Driven Gate-Level Simulation with GP-GPUs”, DAC’09, pp557-562, July 26-31, 2009.
[4] Sara Vinco, Debapriya Chatterjee, Valeria Bertacco, Franco Fummi, “SAGA: SystemC Acceleration on GPU Architectures”, DAC 2012, pp115-120, June 3-7, 2012.
[5] Bo Wang, Yuhao Zhu, Yangdong Deng, ”Distributed Time, Conservative Parallel Logic Simulation on GPUs”, DAC’10, pp761-766, June 12-18, 2010.
[6] Debapriya Chatterjee, Andrew DeOrio, Valeria Bertacco, “GCS: HighPerformance Gate Level Simulation with GPGPUs”, 978-3-9810801-5-5/DATE09 © 2009 EDAA
[7] 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
[8] Nvidia CUDA3.2 マニュアル
[9] 那須升亮，大菊祥子，村岡道明，豊永昌彦，“GP-GPU におけるファンアウトコーンに基づく並列論理シミュレーション法の検討”，平成 24 年度 電気関係学会四国支部連合大会 四国支部連合大会講演論文集 1-20(2012)