平成 28 年度修士論文
FPGA タブレットを用いた
人工知能アプリケーションの高速化

電気通信大学大学院　情報理工学研究科

情報・通信工学専攻

1531048　佐藤 知哉

指導教員　成見 哲　教授

副指導教員　佐藤 証　教授

平成 29 年 1 月 30 日

概要

本研究では、FPGA を用いた Android タブレット端末を用いることで、2 種類の人工知能アプリケーションの高速化を行った。

近年、スマートフォンなどの携帯型端末の普及や、利用方法が多様化したことにより、アプリケーション自体の機能や性能への需要も高まってきている。一方で、計算機そのものの性能が向上したことにより人工知能分野の研究が盛んになってきている。人工知能に関する研究分野には、囲碁や将棋、ポーカーなどの AI(Artificial Intelligence)、Deep Learning という多層で構成されたニューラルネットワークによる機械学習方法がある。これらの分野において、それ自体のアルゴリズムの改良はもちろん以前までは現実的でなかった計算量を処理することができるようになったことにより、その性能は飛躍的に向上してきている。

本研究では、これらの処理を行う際に必要となるアクセラレータとして FPGA を選択し、その FPGA をベースに作成された Android タブレットを用いることで、これらの人工知能技術を含んだ携帯端末用アプリケーションの高速化を行った。本研究で作成したオセロアプリの AI では、CPU と比較して平均約 2 倍程度の高速化、Deep Learning を用いた画像分類アプリでは、本研究で用いるタブレットシステムの資源等を考慮したモデル式の結果で約 32〜72 倍の高速化の可能性を示した。また、VivadoHLS を用いた高位合成によって生成された部分専用回路の場合、CPU と比較して平均 2.9 倍程度の高速化が望めることを示した。

目次

1　はじめに

1.1　研究背景

近年、スマートフォンなどの携帯型端末の普及や、利用方法が多様化したことにより、アプリケーション自体の機能や性能への需要も高まってきている。この要求に応えるにあたり、ソフトウェア側からの解決策とハードウェア側からの解決策が考えられる。ソフトウェア側からの解決策では、処理の並列化やアルゴリズムの改善などが考えられる。ハードウェア側からの解決策では、専用の周辺回路やモジュール、チップを搭載するなどの端末自体のスペックや機能を向上する方法が考えられる。

一方で、計算機そのものの性能が向上したことにより人工知能分野の研究が盛んになってきている。人工知能を利用する一つの研究分野に、囲碁や将棋、ポーカーなどの AI(Artificial Intelligence) がある。AI 自体のアルゴリズムの改良はもちろん、以前までは現実的でなかった計算量の処理が可能になったことにより、その強さは飛躍的に向上している。

また、Deep Learning という多層で構成されたニューラルネットワークによる機械学習方法に注目が集まっている。Deep Learning では、画像や音声などの大規模なデータを扱うことから、CPU だけでの処理能力では不足していることもあり、GPU(Graphic Processing Unit) を用いた高速化を行うことが多い。しかし、GPU は携帯端末に搭載されておらず、広く普及しているスマートフォンやタブレット端末が持つ電力スペックでは、実現性は低い。これらを踏まえて、本研究は FPGA 評価ボードをベースに作成された Android タブレット端末を用いることで、FPGA の回路資源を利用することで、ゲームの AI や Deep Learning 手法を実装した人工知能アプリケーションの高速化を目指す。

1.2　FPGA

FPGA（Field Programmable Gate Array）はプログラマブルロジックデバイスの一種で、通常の専用ハードウェアとは異なり製造後に回路の構成を変更することができる集積回路である。回路の論理記述をユーザ変更するには Verilog-

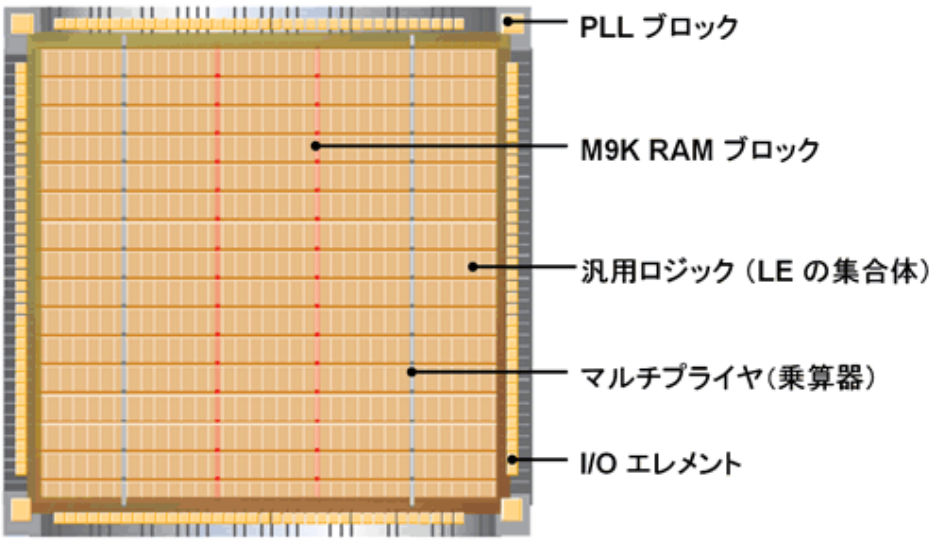

図 1　FPGA のチップの構成例 (参考文献 [1])

HDL や VHDL などのハードウェア記述言語 (Hardware Description Language, HDL) 等が用いられる。FPGA は何度でも論理の再構成可能であるため、製造後に論理を変更できない ASIC(Application Specific Integrated Circuit) の動作検証や頻繁に回路を変更する必要のある製品などに使用されている。

図 1 にあるように、FPGA の内部は、数千個から数十万個ものロジックエレメントや PLL ブロック、IO エレメントなどが集積しており、また最近の FPGA のコアには，乗算器（マルチプライヤ）の回路ブロックや，ユーザー回路内のメモリとして使用する RAM ブロックも搭載されている。

1.3　目的

近年の計算機や専用ハードウェアの処理性能の向上により、既に GPU を搭載した計算機でゲームの AI や DeepLearning を実装したアプリケーションの高速化の事例は存在しているが、本研究では、これらのアプリケーションを FPGA をベースに作成された Android タブレット (以下、FPGA タブレット) 上で実装する。本研究では、オセロアプリと Deep Learning による画像識別プログラムを実装し、それらの処理の高速化を行う。

1.4 本論文の構成

本論文の次章以下の構成を次に示す。

1. FPGA タブレット
   本研究で使用する FPGA タブレットに関して、その構造や部分専用回路の作成手法について述べる。
2. オセロアプリケーションの高速化
   AI とのオセロ対戦ができる Android アプリケーションを実装し、AI の思考処理の一部を専用回路化し高速化する手法を提案する。
3. DeepLearning アプリケーションの高速化
   畳み込みニューラルネットワークを構築し、単純な画像分類アプリケーションに実装し、畳み込み層の順伝播部分の処理の高速化する手法を提案する。

4. まとめ
   本研究で行ったこと、達成したこと、また今後の課題について述べる。

2　FPGA タブレット

本研究では、通常の Android タブレットではなく、FPGA をベースに構成されたタブレットを使用する。この章では、この FPGA タブレットについての説明を行う。

2.1　先行研究

FPGA タブレット (図 2) は塩谷によって開発された、Xilinx 社の Programmable SoC を搭載した FPGA 評価ボード ZedBoard[3] をベースとする Android タブレット型端末のことである [2]。ZedBoard のチップ上の ARM コアで Android OS を動作させることにより、タブレット端末として利用できる。また、FPGA の論理記述を部分的に再構成するダイナミックパーシャルリコンフィギュレーション機能を用いることで、FPGA の回路の一部を OS 動作中に書き換えることが可能である。これにより、アプリケーションごとに専用回路を用意し、それらを必要に応じて書き換えていくことで、アプリケーション全体の高速化を目指している。

2.2　FPGA タブレットの構成

FPGA タブレットは、ハードウェア、ソフトウェア、ソフトハードから成り立っている (図 3)。本研究では、ソフトハードは FPGA の再構成可能回路のことを意味する。

2.2.1　ハードウェア

ハードウェアは、制御部、電源部、表示部、入力部、無線部、外装部で構成される。制御部は FPGA 及び ARM コアを搭載した Xilinx 社の ZedBoard を使用している。このボード上の ARM コアで Android OS を動作させる。電源部にはモバイルバッテリを使用しており、USB による給電が可能である。表示部にはタッチディスプレイを採用しており、通常のタブレット端末同様タッチ操作が可能となっている。入力部には、ZedBoard 上の汎用 I/O 端子の PMOD を二つ使用で

図 2　FPGA タブレット

きる。これらの入出力は構成されている専用回路との接続も可能となっている。

2.2.2　ソフトウェア

ソフトウェアは、OS 部、ドライバ部、API 部、アプリ部に分けられる。ドライバは 3 つのドライバを使用する。一つ目は FPGA へ回路を書きこむドライバである。2 つ目はアプリケーションと回路間でデータをやり取りするための DMA ドライバである。DMA は CPU が持つものではなく、FPGA 上に構成されている AXI DMA Engine を使用している。3 つ目はアプリケーションからメモリを操作するためのドライバである。API 部では、3 つのドライバをアプリケーションから利用するためのライブラリを用意する。アプリケーション部では、実際に Android アプリが記述され、通常の機能に加えて API を用いて FPGA とのアクセスを行うことができる。アプリケーションが FPGA を利用する場合、アプリケーションの持つ回路データを API とドライバを通して FPGA 上に動的に部分再構成し、DMA を通じて計算に必要なデータのやり取りを行う。

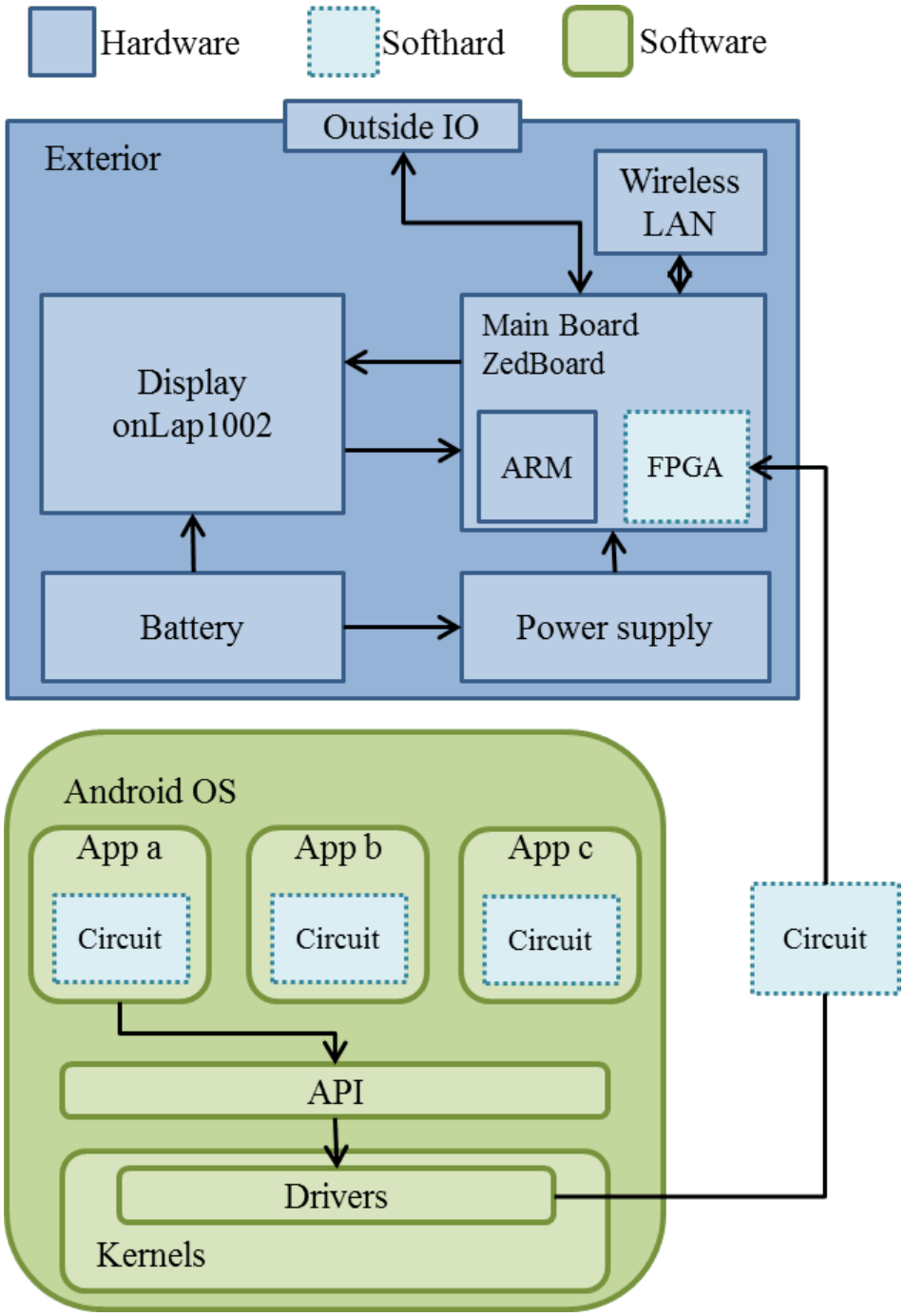

図 3　FPGA タブレットの構成

2.2.3　ソフトハード

ソフトハードは、FPGA 上の OS 使用部とアプリ使用部に分けられる。OS 使用部では OS からの画面データを HDMI 信号へと変換し出力する回路や、音声

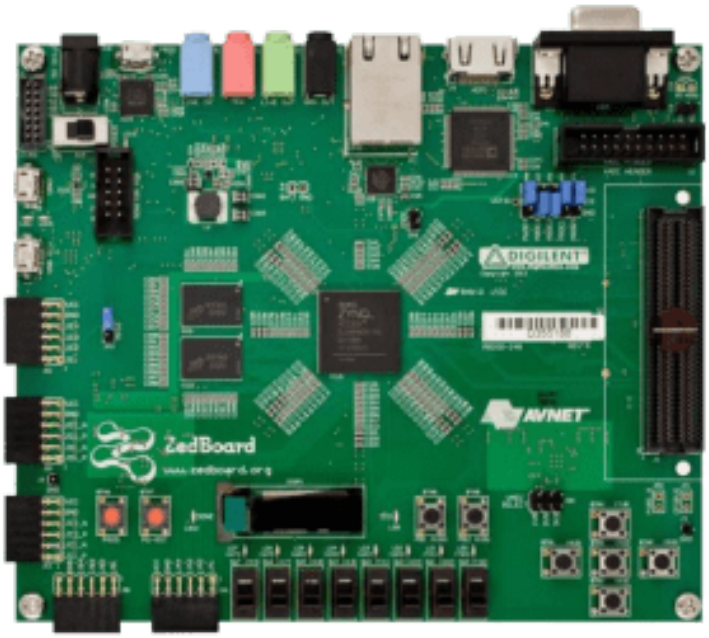

図 4　ZedBoard(参考文献 [3])

を出力する回路が存在する。アプリ使用部では、データをやり取りするための DMA とグローバルクロックを使用するための迂回回路や、外部 IO を使用するための回路がある。

2.3　部分再構成

2.3.1　部分再構成とは

部分再構成 (Patial Reconfiguration) とは、FPGA で構築されている回路の一部を部分的に書き換える技術である。従来の FPGA の回路の再構成では、全ての回路を書き換えるため動作を止める必要があった。これに対して、部分動的再構成は動作中の回路はそのまま、他の回路を部分的に書き換えることができる。

2.3.2　部分再構成領域における資源

FPGA タブレット上の部分再構成可能領域で、使用できる資源は表 1 の通りである。部分専用回路を作成する際はこの領域の資源を利用することになる。表 1 の資源は FPGA で確保できる資源が最大になるようにとってある。

表 1　部分再構成領域で利用することができる資源と個数

LUT	14,400
FD_LD	28,800
SLICEL	2,100
SLICEM	1,500
DSP48E1	120
FIFO18E1	60
RAMB18E1	60
RAMBFIFO36E1	60

2.4　書き換え回路の実装方法

この節では、FPGA タブレット上で動作する専用回路の基本的な作成方法を述べる。

2.4.1　回路の設計原理

部分再構成で書き換える回路は、通常の回路とは異なる方法で生成する必要がある。部分再構成可能な専用回路を作成するには、ベースとなる回路と書き換えたい回路を含んだ回路との差分をとり、その差分回路をアプリケーションに持たせることになる（図 5)。この時、部分再構成を行う部分はブラックボックスのように扱い、それ以外の部分の構成は完全に一致しており、なおかつその領域への信号線も一致してなくてはいけない。この専用回路の作成には Xilinx 社の FPGA 開発環境 PlanAhead を用いた。基本的な回路の作成手順を以下に示した。

1. 書き換え領域にベースとなる回路を含んだ基本的な回路と、書き換え後の

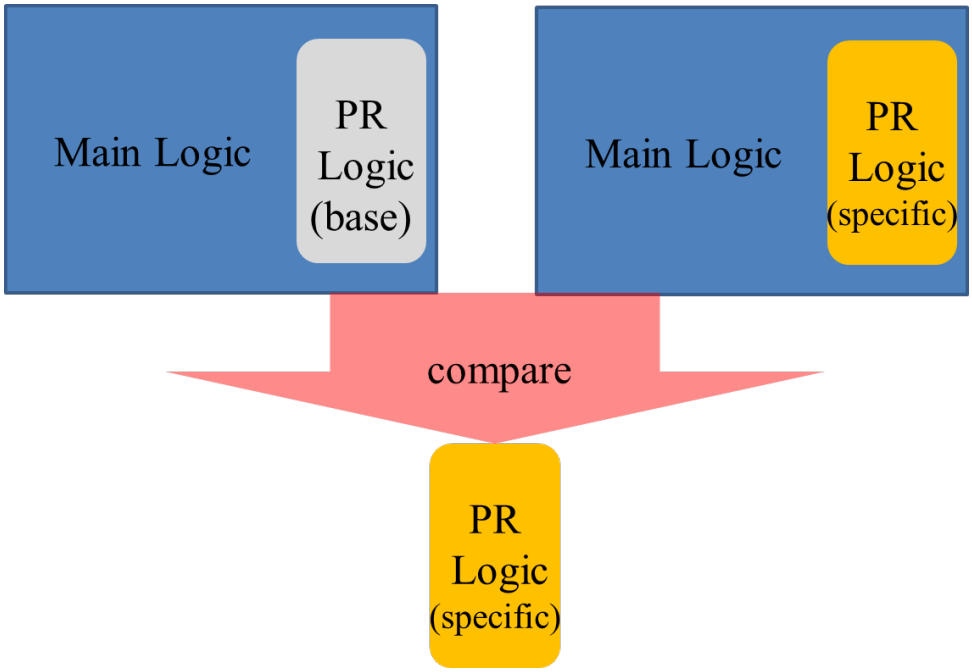

図 5　部分専用回路の作成方法

回路を含んだ回路のネットリストをそれぞれ作成し、部分専用回路のネットリストを取り出しておく。

2. PlanAhead に、部分専用回路以外のネットリストを読み込み、書き換え領域を Reconfigurable として設定しておく。
3. 手順 1 で作成したネットリストを Reconfigurable Module として登録する。
4. 登録したモジュールに対してインプリメンテーションを行い、bit ファイルを生成する。
5. アプリケーションから書き込むために、bit ファイルを promgen を用いて bin ファイルに変換する。

2.4.2　専用回路のインタフェース

FPGA タブレットの専用回路と、それ以外の回路との通信のために用意されているインタフェースは表 2 の通りである。

専用回路は、内部でいくつかの状態を持って処理を行っている。
アプリケーションからデータを受け取る状態、処理を行う状態、データをアプリ

表 2　専用回路のインタフェース

ピン	IN/OUT
ACLK	input
ARESETN	input
S_AXIS_TREADY	output
S_AXIS_TDATA	input 64bit
M_AXIS_TVALID	output
S_AXIS_TLAST	input
S_AXIS_TVALID	input
M_AXIS_TDATA	output 64bit
M_AXIS_TLAST	output
M_AXIS_TREADY	input
GSA	output
GSB	output
GCA	input
GCB	input
PMODOUT	output 8bit
PMODIN	input 8bit

ケーションに送信する状態、アイドル状態の 4 つが基本的な状態である。アプリケーションからデータを受け取る状態では、S_AXIS_TDATA に 64 ビットずつデータが転送されてくる。このとき、転送元はアプリケーションが動作しているプロセッサではなく、DMA 領域からの転送である。また、逆の通信として、部分専用回路からの処理結果の送信には M_AXIS_TDATA を用いて 64 ビットずつの送信を行う必要がある。この場合にも、直接プロセッサへと送信するのではなく、DMA 領域へと書き込みを行い、アプリケーション側で DMA 領域から計算結果を受け取ることになる。

2.5 アプリケーションと回路の通信

2.5.1 アプリで使用する API

アプリケーションから専用回路を利用するために C と Java 用の API がいくつか用意されている。

■PartialReconfiguration

このクラスは部分専用回路を FPGA に書き込むためのメソッドを扱うクラスである。アプリが持っている部分専用回路の bit データを FPGA に書き込む。

- changeLogic(byte[] data) as void
  引数のバイトデータ（部分専用回路データ）を FPGA に書き込む。
- changeLogicFromResource(String name, Resource res) as void
  引数の名前を持つデータをリソース res から取得して FPGA に書き込む。

■SimpleDMA

このクラスは DMA コントローラを操作するためのメソッドを扱うクラスである。基本的に、DMA 領域のアドレスの設定および取得や DMA 領域から専用回路へのデータ通信命令を実行することができる。アプリの持つデータの書き込みや読み込みはこのクラスで扱わず後述する MemoryAccess クラスが行う。

- initialize() as boolean
  DMA の初期化を行う。
- cleanUp() as void

DMA の後始末を行う。

- setBaseAddr() as long
  DMA のアドレスをセットする。
- getBaseAddr() as long
  DMA のアドレスを取得する。
- setRxAddr() as long
  受信データのアドレスを設定する。
- getRxAddr() as long
  受信データのアドレスを取得する。
- setTxAddr() as long
  送信データのアドレスを設定する。
- getTxAddr() as long
  送信データのアドレスを取得する。
- isRxBusy() as Boolean
  受信中かどうかを取得する。
- isTxBusy() as Boolean
  送信中かどうかを取得する。
- makeTransferToDevice(long offset, long len, boolean wait) as boolean
  引数で指定したオフセットと長さをメモリから回路へ送信する。wait 引数の真偽値によって送信処理が終了するまで待つかどうかを指定することができる。
- makeTransferToDMA(long offset,long len, boolean wait) as boolean
  引数で指定したオフセットと長さのデータを回路へ要求する。wait 引数の真偽値によって送信処理が終了するまで待つかどうかを指定することができる。

■MemoryAccess

このクラスはメモリの確保や読み書きを行うメソッドを扱うクラスである。扱うデータ型には整数型、整数配列、単精度浮動小数点数、単精度浮動小数点数配列を扱うことが可能である。このクラスの API を用いて DMA 領域への書き込みや読み込みを行い、前述の SimpleDMA クラスの API を用いて部分専用回路

へのアクセスを行う。

- Constructor(long addr, long size)
  MemoryAccess クラスのコンストラクタである。引数のアドレスとサイズでメモリ領域を確保する。
- map(long addr, long size) as void
  引数のアドレスとサイズでメモリ領域を確保する。
- unmap() as void
  確保したメモリ領域を解放する。
- read(long offset) as void
  引数で指定したオフセットから 32bit 整数としてデータを読み込む。
- readArray(long offset, int[] dest, long length) as long
  引数で指定したオフセットと長さ length にあるデータを 32bit 整数型配列として dest へとデータを読み込む。
- readf(long offset) as void
  引数で指定したオフセットから 32bit 浮動小数点数としてデータを読み込む。
- readArrayf(long offset, float[] dest, long length) as void
  引数で指定したオフセットと長さ length にあるデータを 32bi 浮動小数点数型配列として dest へとデータを読み込む。
- write(long offset, int value) as void
  引数で指定したオフセットに 32bit 整数としてデータを書き込む。
- writeArray(long offset, int[] value) as void
  引数で指定したオフセットにあるデータを 32bit 整数型配列として書き込む。
- writef(long offset) as void
  引数で指定したオフセットに 32bit 浮動小数点数としてデータを書き込む。
- writeArrayf(long offset, float[] value) as void
  引数で指定したオフセットにあるデータを 32bi 浮動小数点数型配列として書き込む。

表 3　アプリから部分専用回路を書き込む処理時間

Max Time	77.83ms
Min Time	70.29ms
Average Time	73.19ms

2.5.2　回路の書き換え時間の検証

アプリケーションの書き換えは、メソッド changeLogic や changeLogicFrom-Resource を用いて行うが、この API の実行時間を測定した。API を 100 回実行して平均を取る、その試行を 5 回行った。測定した結果を表 3 に示す。　結果から、952KB の回路を書き込むためにかかる処理時間は約 73ms であることが分かった。部分専用回路自体は、ブラックボックスとしてある一定の領域を指定しているためか、生成された bin ファイルのサイズは基本的に 952KB となっていることも確認できた。アプリが持つことができる部分専用回路の個数には制限がないため、多種類の回路を書き換えながら動作させることも可能である。ただし、逐次実行で別々の処理をいくつかの回路に分けて処理する場合、回路の書き換え時間の約 73ms とデータ転送の時間がオーバーヘッドとなるため、回路化するに相応しい処理でないと性能が期待できない。

3　オセロアプリケーションの高速化

3.1　先行研究

3.1.1　An FPGA-based Othello endgame solver

Wong らは FPGA 上にオセロゲームを実行することに成功している [4]。オセロを実装する上で、石が置けるかどうかの確認や、挟んだ石を裏返す処理等が必要となる。この研究では、特に対戦相手として FPGA 上で $\alpha - \beta$ 法を用いた AI を実装している。ゲーム全体の進行はステートマシンによるループで行われている。このステートマシンでは $\alpha - \beta$ 法をもとにして葉や部分木、パス、木の評価、の 4 状態を持ち、この状態をループしながら処理を行っている。本研究は、AI のアルゴリズムに Min-Max 法を用いている点、ゲーム木全体の回路化ではなくゲーム木中の盤面の評価のみを専用回路化することによってハードウェアを簡単化しつつ、ある程度の高速化を目指すという点が従来研究と異なる。

3.2　実装するオセロの定義づけ

3.2.1　題材とするゲーム

本研究で扱うオセロの定義づけを行う。オセロは二人零和完全情報ゲームである。8 × 8 マスの盤面に黒と白の石を交互に置いていき、盤面が全て石で埋まった時に石が多い方が勝ちとなるボードゲームである。

ゲームの盤面の初期配置は図 6 の通りである。手番は黒の石を持つプレイヤーから始める。手番プレイヤーは盤面の石が置かれていない場所に石を置くことができる。ただし、縦・横・斜めで相手の石を挟むように置かなければならない。もし、相手の石を挟むことができる位置が無い場合はパスとなる。石を置いた際に挟まれた相手の石は裏返されプレイヤーの石となる。通常、オセロは対人戦であるが、本研究では、相手プレイヤーを仮想プレイヤーとし、A に次の手を計算させる。よって、人間プレイヤー対 AI の勝負とする。

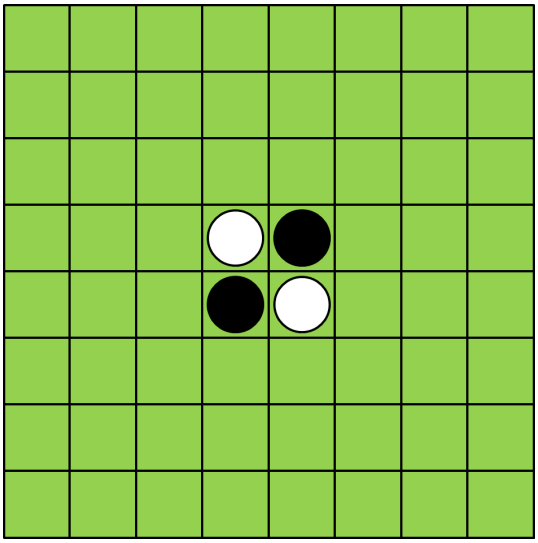

図 6　オセロの初期配置

3.2.2　実装するアプリケーション

実装したオセロアプリケーションを図 7 に示す。手番は黒 (ブラウン) からスタートし、プレイヤーが黒を担当する。配置可能なマスにタッチすると、石が置かれる。プレイヤー側の石が配置されると、自動的に AI の手番となり、以降交互に手番となる。左下にある'Reset' ボタンをタッチすると盤面が初期配置にリセットされる。起動時は AI の処理は CPU のみの計算となっているが、右下のスイッチをタッチすると、FPGA を利用するモードに切り換わる。この状態で AI の手番に移ると、後述する専用回路で指し手が計算される。CPU のみの処理に戻すにはもう一度タッチする。

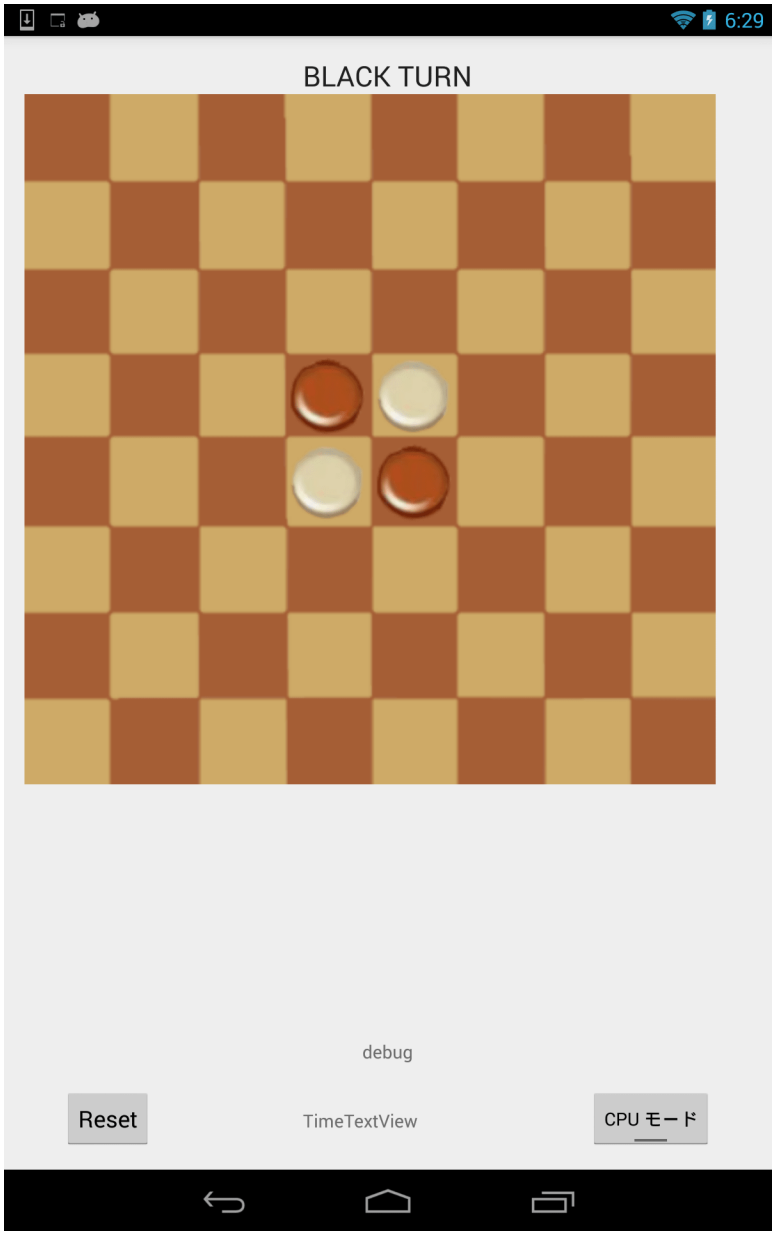

図 7　実装したオセロアプリケーション

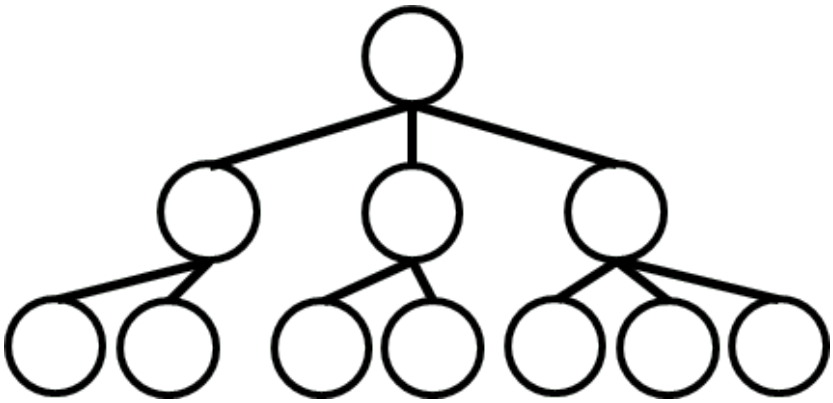

図 8　ゲーム木

3.3　AI のアルゴリズム

本研究では実装が簡単な Min-Max 法を AI による計算法として採用した。Min-Max 法は現在の盤面から想定される最大の損害を最小にするような手を選択するアルゴリズムである。図 8 に示す通り、ゲーム木の根の部分を現在の盤面とし、各節点からはその盤面から取りうる次の手の数だけ枝分かれし、ゲーム木をくだる。Min-Max 法では各節点から一つくだることで手番が入れ替わる。例えば、現在の節点の手番がプレイヤーであれば、一つ下にくだった節点は相手の手番になる。このゲーム木の各節点ではその盤面の評価値を持つ。評価値はプレイヤーに対する下記のような要素に対し重みづけされた値の合計値を採用する。

1. 石の数
   現在の盤面上の石の数。ゲームの勝利条件でもある。
2. 石の位置
   置いてある石の位置には優劣がある。例えば、四隅にある石は必ず取られない為有利な位置といえる。
3. 石を置くことができる位置の数
   手番時に石を置くことができる位置の数が多いほど戦略を立てることができる。

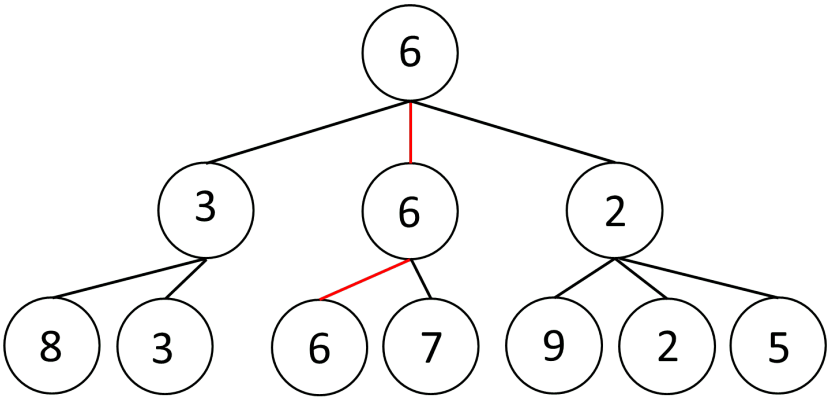

図 9　Min-Max 法

ただし、この評価値はその盤面のみを評価したものであって、これから先の盤面を考慮したものではない。そこで、Min-Max 法によって各節点の評価値を子の評価値を用いて計算する。Min-Max 法では、プレイヤーの手番では次の手番で最も評価値が高いものを選択し、逆に相手の手番では相手にとって最も評価値が高いもの、つまりプレイヤーにとって評価値が最も低いもの（プレイヤーに不利なもの）を選択すると仮定する。例として図 9 を示す。図 9 では根をプレイヤーの手番としている。この場合、第一節点は AI の手番、葉はプレイヤーの手番である。葉の評価値はそれより下の盤面が無いため、その盤面の評価値を採用する。第一節点は AI の手番の為、プレイヤーにとって最も不利な手である評価値が最低のものを選択する。図 9 ではそれぞれ 3, 6, 2 を選択することになる。最後に根の評価値を求める。根はプレイヤーの手番のため、最も有利な手である評価値が最大なものを選択する。つまり、6 を選択する。根は現在の手番であるから、この場合では評価値 6 である手を選択するとよいことになる。

階層が深くなればなるほど先の展開を加味した手の選択が可能であるが、計算機のメモリの関係上深さを制限してある程度で打ち切る。

3.4 書き換え回路の設計

3.4.1 回路の対象となる処理の決定

前節のアルゴリズムを実装するにはゲーム木を構築していくが、AI の手を決めるものなので、ゲーム木の根は AI の現在の手番となる。そこから深さ d までゲーム木を成長させる。AI の一連の計算中に最も多く出てくる処理は、与えられた盤面からあるプレイヤーが置くことができる石の位置を算出する処理である。この処理は、ゲーム木の節点の子を生成するときや評価値の算出の際に呼び出される。ソフトウェアでは、プログラム 1 のようなアルゴリズムになる。

プログラム 1　石の位置を算出する処理

```
1   for i:=0 to 7 do
2   begin
3       for j:=0 to 7 do
4       begin
5           for dx=−1 to 1 do
6           begin
7               for dy=−1 to 1 do
8               begin
9                   if 相手の石が存在しその一つ先に自分の石があるかどうか
10                  begin
11                      return true;
12                  end;
13              end;
14          end;
15      end;
16  end;
17  return false;
```

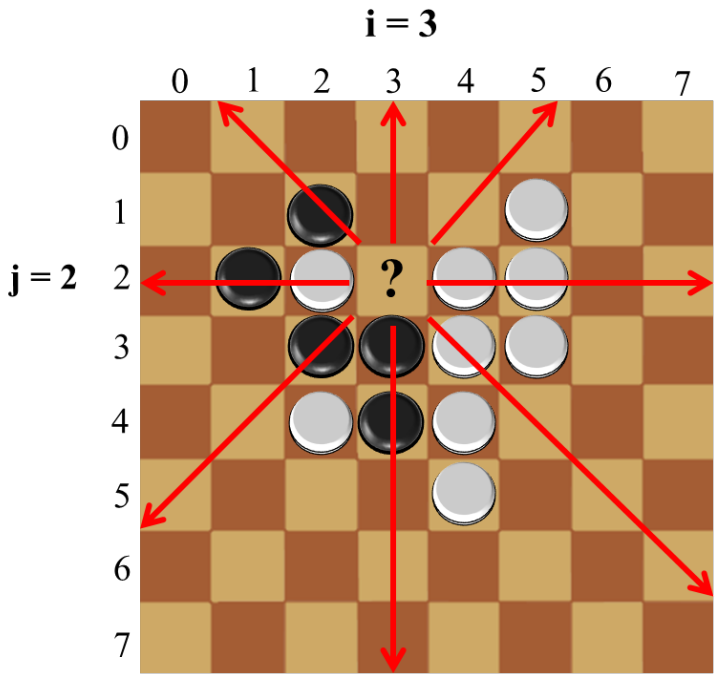

図 10　石を置くことができるかどうかの判定例

プログラム 1 の通り、盤面全体に対して走査を行い、(i, j) の位置で上下左右斜めを走査していき、相手の石を挟む位置に自身の石が存在すれば置くことができる。例えば、図 10 のような盤面で $(i, j) = (3, 2)$ の位置への、黒石の配置を調べる判定には、赤の矢印方向に対して白石を挟めるかどうかを調べる。この場合、左向きの矢印の方向に対して、白石を挟むことができ、この $(3, 2)$ の位置に黒石を置くことができる。同様の判定を $(i, j)(0 \leq i, j \leq 7)$ に対して行い、盤面全体で黒石が置ける位置を求める。しかし、このアルゴリズムは多重ループであり、四則演算などと比較すると重い処理といえる。そこで、この処理を専用回路化して計算の高速化を行った。

3.4.2　回路の設計方針

プログラム 1 の専用回路を実装するため、例として図 11 のような 1 ライン上の盤面を想定する。このとき、黒の手番プレイヤーは a の位置に石を置くことがで

図 11　オセロの具体例 (1 ライン)

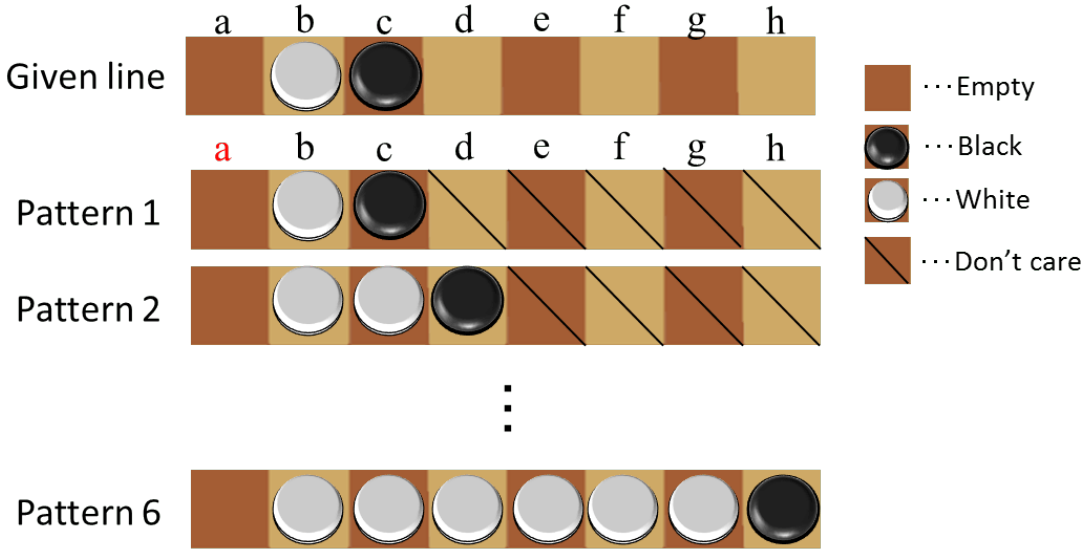

図 12　オセロのパターン例

きる。逆に、a の位置に黒石を置くことができるパターンを調べると図 12 のようにな 6 つのパターンとなる。したがって、このいずれかに一致するパターンであれば、黒石を a に置くことができる。ただし、パターン 1 からパターン 5 は d から h までの石の状態を考えない Don't care を含んでいる。これらのパターンを事前に用意しておき現在の盤面とのマッチングを行う。よって、実際には次のような論理式となる。

$$P_1+P_2+P_3+P_4+P_5+P_6 = True+False+False+False+False+False = True$$

$P_n(n = 1, 2, ...)$ はそれぞれのパターンとのマッチングである。P_1から P_6 は a に関する論理式である。図 12 において P_1 とのマッチング結果は $True$ のため最終的な結果も True になる。もし $P_1 = False$ であった場合、上述の論理式の結果は $False$ になり、a には黒石を置けないということになる。しかし、これは a を含

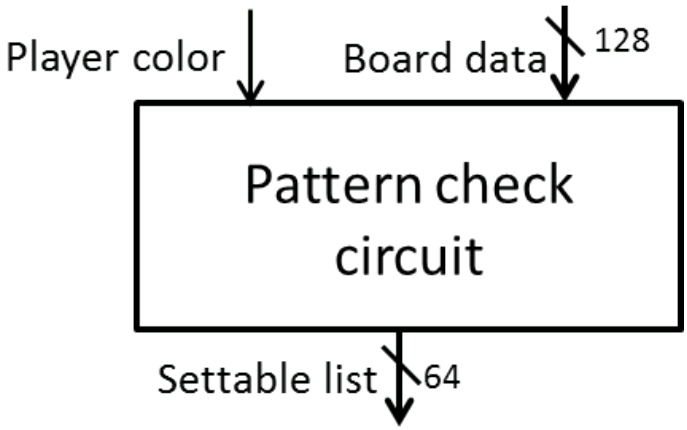

図 13　パターンチェック回路

んだ横列に関するマッチング結果であり、他にも a を含んだ縦列や斜めの列も存在する。よって、縦・横・斜め 2 種の全 4 方向のうちどれかの列に関する結果が $True$ であれば、a に黒石を置ける位置であるということになる。

3.4.3　回路の作成

前節の回路の設計方針から、実際に図 13 のような専用回路を構築していく。石を置くことができるかどうかの判定回路に必要なデータは、盤面の状況と判定するプレイヤーの石の色である。プレイヤーの石の色は白か黒の 2 種類の 1 ビットの情報 (Player Color) である。盤面の状況は、各マスに対して「黒石」「「なし」の 3 つの状態があるので、2 ビットの情報が必要になる。また、盤面は全体で 8 × 8 のマスで構築されているため、全体で 128 ビットの情報となる。

出力も盤面の情報を返却するが、1 つのマスに対する情報は石を置けるか否かの 1 ビットの情報となり、全体で 64 ビットの出力となる。

マッチング判定回路自体はあらかじめ用意したパターンとのマッチングの可否は組み合わせ回路で行い、レジスタは不要である。したがって、評価を行う部分だけであれば 1 クロックで処理が可能である。

3.5 実験・評価

作成した専用回路に対する評価実験を行った。その結果について説明を行う。

3.5.1 評価手法

Min-Max 法による AI の計算時にゲーム木を構築する。その際に、どこまで枝を伸ばすかの制限を設け、その制限の深さを d とした。評価実験ではこの深さ $d = 4, d = 5$ の場合で、プレイヤーが石を置いて手番が交代した直後から、手を決定して石を置くまでの AI の手番の計算時間を測定した。

3.5.2 結果

結果を図 14、15 に示す。それぞれの計算時間は 3 ゲーム行った時の平均を取った。縦軸は計算時間 (ms)、横軸は AI の手番の推移である。つまり、左側はゲーム開始序盤を表しており、右側に行けばいくほどゲームが中盤、終盤となっていく。

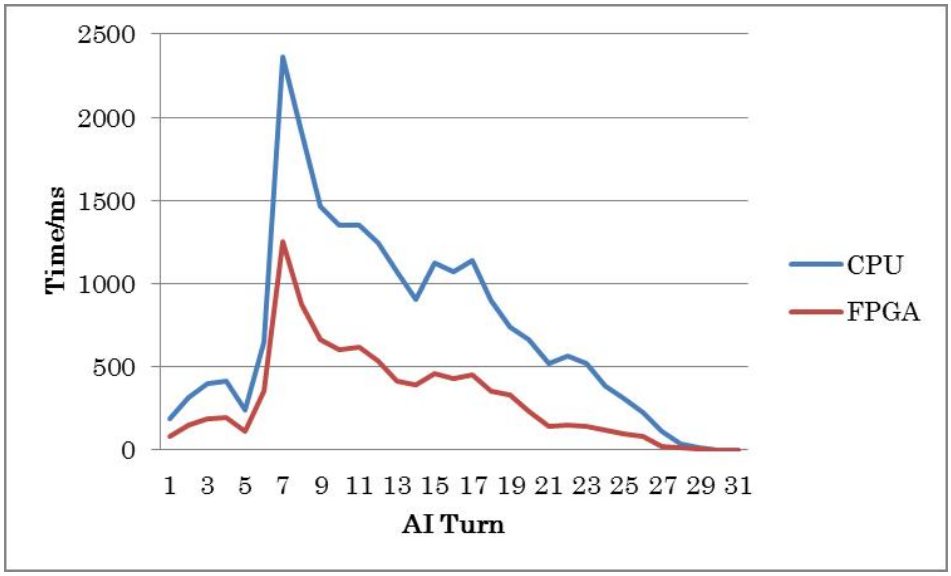

図 14　$d = 4$ の測定結果

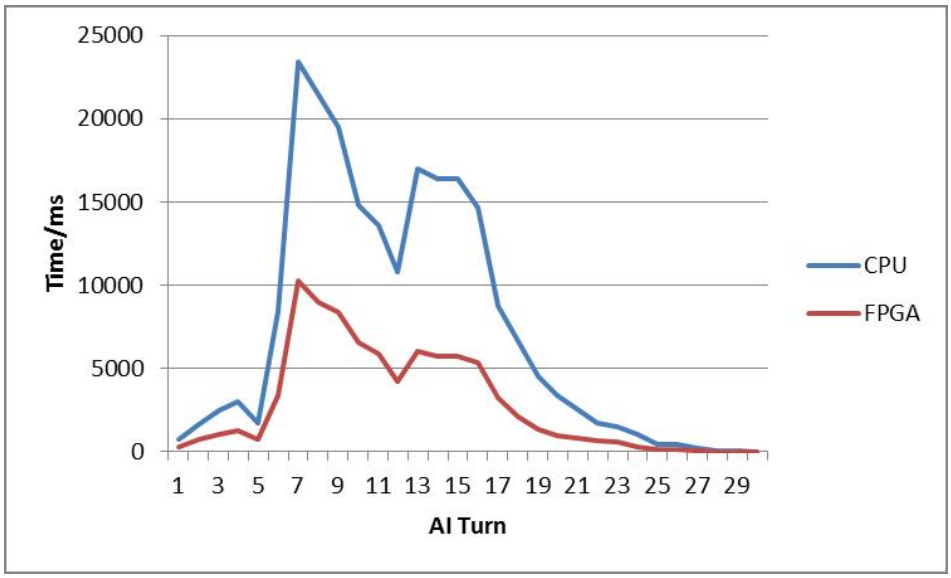

図 15　$d = 5$ の測定結果

表 4　$d = 4, 5$ におけるそれぞれのピーク時の計算時間

Depth	4	5
CPU	2,364.67ms	23,419.33ms
FPGA	1,256.00ms	10,230.67ms
relative time(CPU/FPGA)	1.88	2.29

また、各 d に対して、ピーク時の計算時間を比較したものを表 4 に示した。

3.6 考察

前節の評価実験の結果を受けて考察を行う。

3.6.1 高速化の比較

表 4 から、$d = 4$ において、FPGA は CPU に対してピークで約 1.9 倍の性能を示している。ピーク時以外の AI の手番では平均して 2.5 倍の高速化となった。また、$d = 5$ のときでは、FPGA が約 2.3 倍の性能を示している。ピーク時以外の AI の手番では平均して 2.8 倍の高速化を行うことができた。

通常、FPGA は CPU に対して数十倍以上の高速化なことが多いが、今回は 2 倍程度にしかならなかった。これは、AI の処理に関して、全体に対する石が置ける位置の検出処理の割合やハードウェアとアプリケーションとのデータのやり取りにによるオーバヘッドが大きいためだと考えられる。実際に、判定部分のみの処理時間は 100 倍以上高速化されている。

$d = 4$ と $d = 5$ を比較すると、d が 1 階層深くなることで計算時間が約 10 倍に増大している。より先の展開を考慮した手の探索は、計算時間が指数関数的に増加し、$d = 6$ ではピーク時に約 230 秒もの計算時間を要することが予想される。AI の処理時間の短縮には、ソフトウェア・ハードウェアによる以下の改良が有効であると考えられる。

ソフトウェアの観点から、ゲーム木の構築方法の改良を考える。現在の木の構築方法は Min-Max 法を採用しているが、これを $\alpha - \beta$ 法に変更すると高速化が望める。ここで、$\alpha - \beta$ 法は、Min-Max 法のようにゲーム木を構築する際にその部分木で計算しなくても、親ノードの値が自明になるパターンを考慮したアルゴリズムである。このアルゴリズムを用いて $d = 5$ の場合の計算時間のグラフを図 16 に示す。この場合、最大で Mini-Max 法の CPU 実装に対して $\alpha - \beta$ 法の FPGA 実装は 10.92 倍、平均して 6.70 倍高速化された。

ハードウェアの改良は、部分専用回路の拡大である。ゲーム木の構築部分まで含めると大幅な高速化が期待できるが、それにはかなり多くの資源が必要で、さらには開発にかかる時間も大きくなると予想される。この改良はオセロアプリケーションの専用回路に対する改良となる。他には、FPGA タブレット自体の改良も

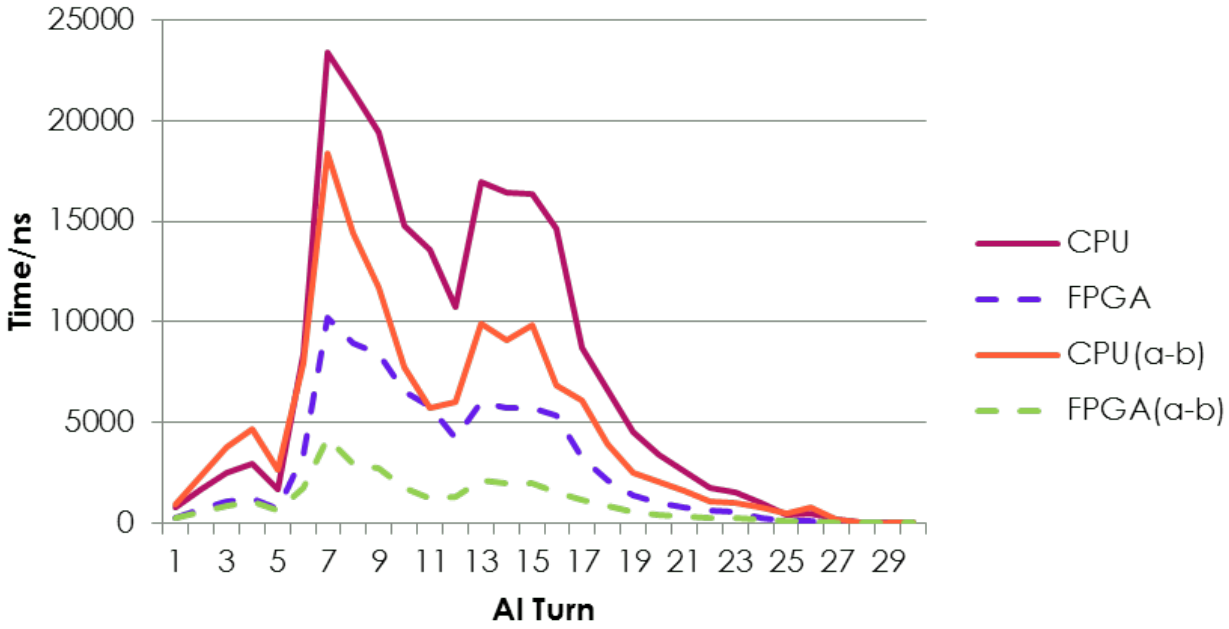

図 16　$d = 5$ における $\alpha - \beta$ 法を用いた場合の計算時間

考えられる。現在の FPGA タブレットの仕組みでは、FPGA とアプリケーションとのデータのやり取りの際に、一度 DMA を通す必要がある。そこで、アプリケーション-FPGA 間で PIO(Programmable IO) を用いて直接データ通信を行うことによって、オーバーヘッドを削減できると考えられる。タブレットに用いている FPGA 評価ボードには ACP（Accelerator Coherency Port）があり、直接プロセッサが専用回路と通信でき、DMA を仲介することによるオーバヘッド問題を解決することが可能であると考えられる。

3.6.2　AI における部分専用回路化

本研究ではオセロの AI の高速化を行ったが、他のボードゲームの AI への応用も期待できる。オセロは二人零和完全情報ゲームであり、お互いの戦況と優劣が公開されている。ここで、ゲームの戦況が一部しか公開されない非完全情報ゲームの場合の AI への応用を考えてみたい。

そのようなゲームの一つとしてポーカー (図 17) がある。ポーカーには様々な種類があるが、一般に広く知られているのは、一組のトランプを使用して各プレ

図 17　ポーカー (テキサスホールデム) 参考文献 [5]

イヤーが 5 枚の手札を交換しながら役をつくり、その強さを競いあうものである。ポーカーでは、最後の役公開 (Show Down) まで各プレイヤーは手札を見せない。つまり、現在の盤面の状況に関する情報はプレイヤーの手札、対戦相手が交換した枚数となる。したがって、盤面の評価関数を定めることが困難で、オセロのように Min-Max 法や $\alpha - \beta$ 法を用いたゲーム木の構築はあまり効果が見込めない。ポーカーでは、いくつかの役が存在し、それぞれに強さが設定されている。例えば、同種のスートの 10,J,Q,K,A を集めたロイヤルストレートフラッシュや、同じ数字のペアを二種類集めたツーペアなどがあり、ツーペアよりもロイヤルストレートフラッシュの方が役の強さは上となる。しかし、ポーカーにおける勝ち負けは役の強さだけで決定するわけではなく、チップの掛け方が重要となる。カードの役を揃える操作以外にも、掛けるチップの枚数のつり上げやその過程でのラウンドの棄権などが存在する。つまり、あるプレイヤーよりも強い役を作った相手がラウンドを棄権した場合は、その役よりも弱いプレイヤーの勝利となる。したがって、評価関数を定める際に、単純に手札の役の強さだけで盤面の優劣を判断することはできない。

ポーカーの AI に関する研究では、Zinkevich らはリミットテキサスホールデムにおいて CFR(CounterFactual Regret minimization) を用いた手法の提案をしている [6]。これは、ゲーム結果から得られる後悔（こういう選択をすればよかったというフィードバック）をもとに判断基準を修正していくものである。ただし、この手法では判断基準の修正の為の学習に時間がかかる、専用回路の複雑化と資源の増大が予想されるため、本研究では、より簡単な方策の一つとしてモンテカルロ木探索法を用いることで精度の良い AI を設計することにした。モンテカルロ木探索は、各ノードの手の選択をランダムにゲーム終了まで試行（プレイアウト）し、その結果がどうなったかを記録しておく。このプレイアウトを何度も行うことによって、より自分に有利な展開になった手を選択するという手法である。このモンテカルロ木探索において、各ノードは基本的なゲームの処理しか行わない。また、ポーカーのゲーム上、チップ操作や役の判定などはオセロの盤面の評価ほど計算量が多くない。よって、本研究のオセロのような、ゲーム木の構築をソフトウェアで、評価の一部を専用回路で行うという方法は高速化に得策ではないといえる。モンテカルロ木探索による計算時間はほとんどが多大な試行数のプレイアウトを占めることが多く、プレイアウトそのものを専用回路化して、何倍にも高速化する必要がある。しかし、プレイアウトの設計はオセロのパターンチェック回路のように簡単ではなく、小規模な専用回路の実装で、ある程度の高速化が見込める FPGA タブレットのシステムの利点が損なわれる。

以上の考察から、FPGA タブレットではオセロのように高速化しやすい計算と、ポーカーのようにそれが困難なものが存在し、それを考慮した上で専用回路の設計をする必要がある。

4 DeepLearning アプリケーションの高速化

この章では、FPGA タブレットで DeepLearning アプリケーションとして、畳み込みニューラルネットワークのアルゴリズムの高速化に関する手法の提案を行う。

4.1 先行研究

第 1 章でも述べたとおり、近年は計算機の発達により Deep Learning の研究が盛んに行われている。ここでは次の先行研究を取り上げる。

4.1.1 Accelerating Deep Convolutional Neural Networks Using Specialized Hardware

Deep Learning を専用ハードウェアで高速化した例として、Microsoft 社の FPGA による畳み込みニューラルネットワーク [7] が挙げられる。この研究では、カタパルトプロジェクトの一部として、FPGA を大量に用いたデータセンターのサーバアクセラレータの開発に成功している。このアクセラレータは畳み込みニューラルネットの前伝播の演算の高速化を行うものである。図 18 にその構成図を示す。このサーバアクセラレータは次のような特徴を持つ。

1. ハードウェアのリコンパイル不要
2. データバッファリングによる循環利用とオフチップとの低トラフィック
3. 乗算加算等を行う PE を沢山用意することで大規模なネットワークに対応可能

本研究の類似点としては、ハードウェアのリコンパイルが不要という点があげられる。本研究は、サーバアクセラレータとしてではなく、あくまでローカルのタブレット内で処理が完結する点が異なる。また、本研究で複数のニューラルネットワークの高速化に対応するには、複数の部分専用回路を作成する必要がある。しかし、部分専用回路はニューラルネットワーク以外の様々なアクセラレータとしても利用できるという利点がある。

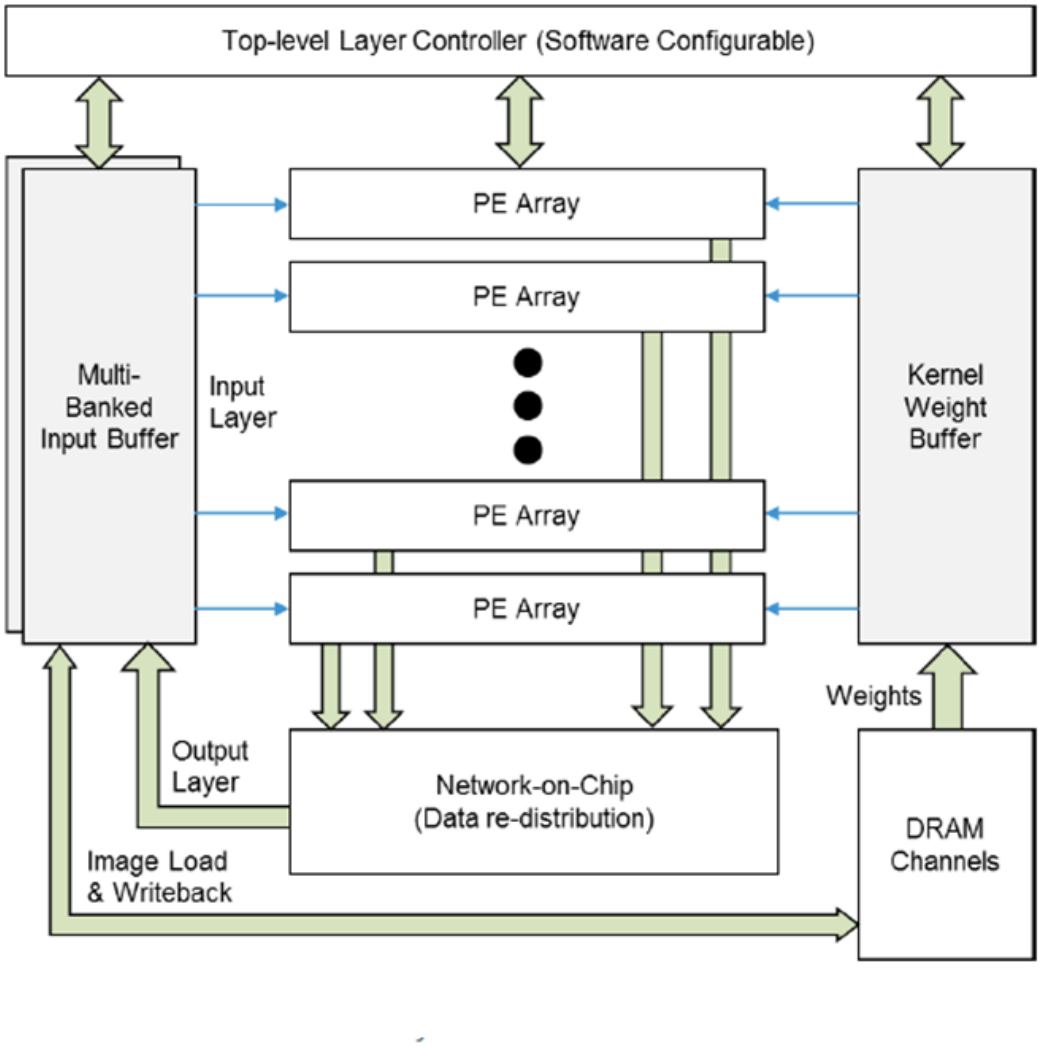

図 18　FPGA アクセラレータ（参考文献 [7] の 3P 図 3）

4.2　想定するニューラルネットワーク

DeepLearning とは、一般的に多層のニューラルネットワークによる機械学習のことを指す。本研究においても、多層のニューラルネットワークを構築しその一部の処理に対して高速化を行う。

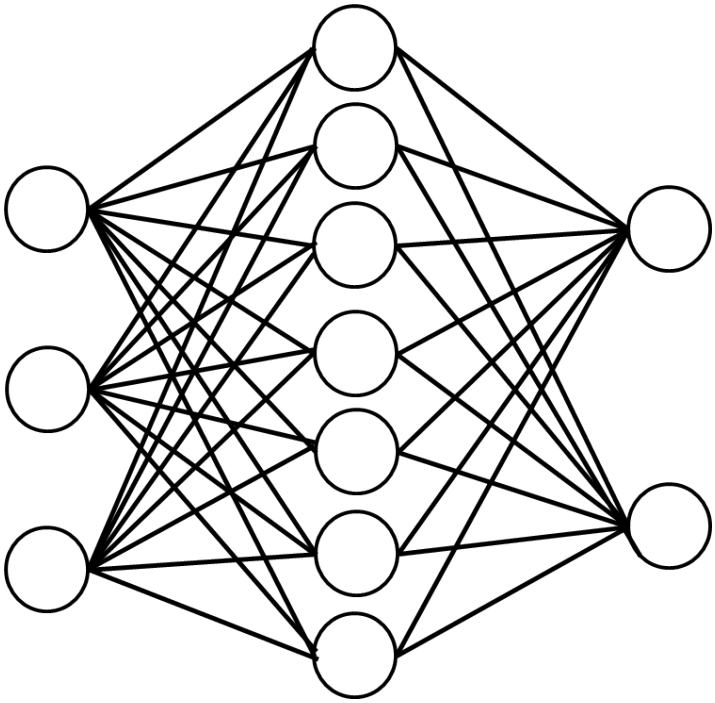

図 19　ニューラルネットワーク

4.2.1　ニューラルネットワーク

ニューラルネットワークとは人間の脳の構成を模倣した仮想的なネットワークのことである。これらのニューラルネットワークは図 19 のように無向グラフで表され、それぞれのノードはニューロンとして値を持っている。一般的に、このニューラルネットワークは層状に並んでおり、一番左のデータを受け取る層のことを「入力層」と呼ぶ。逆に、このニューラルネットワークの処理結果である最も右の層のことを「出力層」と呼ぶ。入力層と出力層の中間にある層のことを「隠れ層」や「中間層」と呼ぶ。この中間層にあるノードのひとつ一つは図 20 にあるように前の層のデータを受け継いで一つの値を出し、次の層へと受け渡す。この例では、ノードへの入力は 3 つで、これに対して、このノードが固有で持っている重み w_n を掛けて合計したものを活性化関数 f を作用させた値 z を出力とする。

$$d = a * w_1 + b * w_2 + c * w_3$$

$$z = f(d)$$

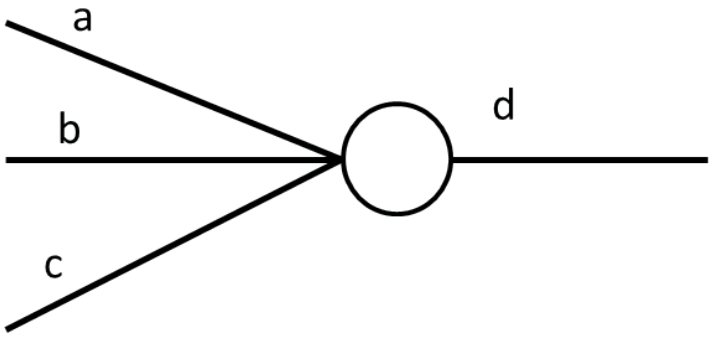

図 20　ノードにおける計算

この重み w_n を適切に設定することで、ある入力に対する値を出力する。つまり、ニューラルネットワーク内のそれぞれのノードの重みを上手く設定することで、任意の関数に対する近似が可能である。一般的に、中間層のノード数や層の厚さ（深さ）が増すと、表現の自由度も大きくなるとされる。しかし、あまりに大きすぎると、重みを学習する際の教師データの入力と出力自体を模倣してしまうため、本来の関数が持つべき性質が損なわれる可能性がある。

4.2.2　重みの更新

ニューラルネットワークは重みを上手く設定することで任意の関数近似が得られるが、どのように重みを設定するかが問題となる。そこでは、ある入力 x に対して得られる望ましい出力 d のペアを考え、これらを訓練データとして用いられる。つまり、ニューラルネットワークはこの訓練データを用いて、入力 x に対して d に近くなるように重み w_n を設定する。この近さは二乗誤差を用いて表現することが一般的である。ある重み $\boldsymbol{w}$ に対して、各訓練データ $\boldsymbol{d_n}$ と出力 $\boldsymbol{y}(\boldsymbol{x}_n; \boldsymbol{w})$ の二乗誤差を足して係数倍したものを誤差関数と呼ぶ。

$$E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} ||\boldsymbol{d_n} - \boldsymbol{y}(\boldsymbol{x}_n; \boldsymbol{w})||^2$$

この $E(\boldsymbol{w})$ が最小となるように w を選ぶが、勾配降下法を用いることで簡単に実行できる。ここでいう勾配は以下になる。

$$\nabla E \equiv \frac{\partial E}{\partial \boldsymbol{w}}$$

そして、この勾配を用いて次の式で $\boldsymbol{w}$ を更新していく。

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \epsilon \nabla E$$

ϵ はある十分小さい値である。この式から、負の勾配方向に対して少しだけ重み $\boldsymbol{w}^{(t)}$ を動かすことで $\boldsymbol{w}^{(t+1)}$ が得られる。ここで t は更新回数である。更新の度に、重みは極小値に収束していくはずであるが、ϵ が大きすぎると増大してしまうし、小さすぎると一回の学習で更新される値も小さくなってしまい学習に時間がかかってしまう。そのため、この ϵ は慎重に決定する必要がある。

4.2.3 誤差逆伝播法

各層の ∇E を求めるには誤差関数の計算が必要となるが、誤差関数にはニューラルネットワークの出力 y を含んでおり、この y は

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{u})$$

となる。ここで $\boldsymbol{u}$ は各ノードの重みと入力とバイアスの和である。ニューラルネットの構成上、入力層以外の層では前の層の出力がそのまま入力となる。つまり、ある l 層の $\boldsymbol{u}$ はさらにその前の $l-1$ 層の入力が必要となり、結果的に活性化関数 $\boldsymbol{f}$ の入れ子状態となる。

$$\boldsymbol{y}(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{u}^{(l)}) = \boldsymbol{f}(\sum \boldsymbol{w}\boldsymbol{f}(\boldsymbol{u}^{(l-1)}) + \boldsymbol{b})$$

このままプログラミングを行うと計算量が膨大となり非効率であるため、誤差逆伝播法が一般に用いられる。

誤差逆伝播法では、一度とある入力を行いネットワークを伝播させ、その出力層の誤差を初期値として入力層へ向かって誤差値を逆伝播する手法である。

4.2.4 畳み込みニューラルネットワーク

畳み込みニューラルネットワークは、通常のニューラルネットワークに「畳み込み層」や「プーリング層」などの特に画像に対して有効な特殊な操作を行う層を多数追加したネットワークのことである。畳み込み層とは、与えられた入力に対し、ある一定の領域ごとに重みであるフィルタとの畳み込みを行う層のことである。また、各ノードが持つ重みにはフィルタとして共通の値を採用している。学習の際には、このフィルタ値を更新することによって、学習データの特徴を分

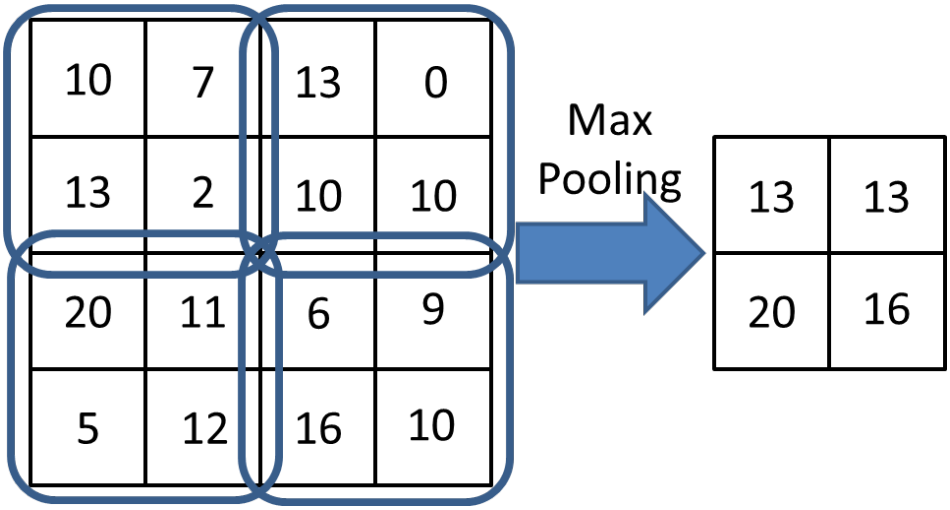

図 21　最大プーリングの例

類するためのフィルタの精度を向上させていくことになる。

プーリング層は、主に与えられた入力の抽象化を行う層である。同種の入力は同じようなデータとして扱いたいことがしばしばあるが、ノイズやズレなどの悪要因によって正しい認識を阻害してしまう可能性がある。そこで、このプーリング層で、ある程度の抽象化を行いノイズやズレによるデータの変化を吸収する。プーリング層は、畳み込み層のような窓を用いてデータの抽出を行う。一般的に用いられる抽出方法は、窓内の最大値を抽出する最大プーリング (図 21)、窓内の平均を取る平均プーリングなどがあげられる。

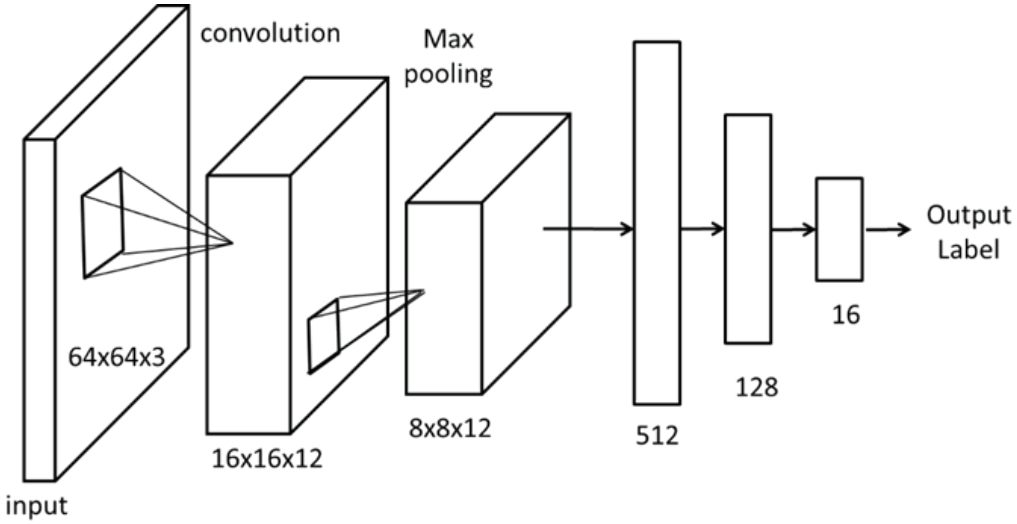

図 22　実装する畳み込みニューラルネットワーク

4.3　実装するニューラルネットワークの構成

本研究で実装する畳み込みニューラルネットワークを図 22 に示す。Krizhevsky らは、2012 年に 14 層の畳み込みニューラルネットワーク [8] を用いて ILSVRC で優勝した。このニューラルネットワークのように、通常であれば、畳み込み層やプーリング層を多層に配置したより深いネットワークを構築するいが、FPGA タブレット上の Android OS が管理するメモリの都合を考慮すると深いネットワークは構築しづらいと考えた。そこで、本研究では畳み込み層とプーリング層を一つだけにした簡易的な畳み込みニューラルネットワークを構築することにした。加えて、それぞれの層のサイズにも制限を掛けた。一つの畳み込み層であったとしてもフィルタサイズを大きくするとそれだけ膨大な資源を利用することになるため、なるべく小さなフィルタを用いたい。しかし、フィルタが小さくなるとその分、フィルタが持つ特徴抽出を行う性質も弱まってしまう。そこで、扱う画像のサイズそのものを小さくすることで、計算途中のパラメータ数の増大を防いでいる。各層の活性化関数には出力層を除いて正規化線形関数を選択している。出力層にはソフトマックス関数を用い、出力結果は各画像カテゴリへの分類結果である。また、多クラス分類問題を想定し、誤差関数には交差エントロピーを用

いた。プーリング層では最大プーリングを採用した。また、可適合の緩和のために、ドロップアウト [9] を用いることにした。ドロップアウトはニューラルネットのユニットを確率的に選別して学習する手法で、学習時にネットワークを強制的に小さくすることによって過適合を防ぐことができる。

4.4 ニューラルネットワークの計算時間のモデル化

構築したネットワークでは、畳み込み層の演算のウェイトが大きい。そのため、この部分に関して専用回路化を行い、アプリケーションの高速化を目指す。一般的に、処理の専用回路化の規模が大きくなるほど高速になるが、資源や時間的コストが増大するというトレードオフが存在する。そこで、本研究では専用回路化にあたり、どの程度が妥当か検討を行う。

4.4.1 モデルの構築

畳み込み層に関する処理時間は次の式となる。

$$T = T_{CPU} + T_{FPGA} + T_{COM}$$

ここで、$T_{CPU}, T_{FPGA}, T_{COM}$ はそれぞれ、CPU での処理時間、FPGA での処理時間、CPU-FPGA 間のデータ転送時間を指す。T_{CPU} は畳み込み関数内で FPGA に計算させる箇所をコメントアウトしてそれ以外の処理時間を測定することで求める。具体的には、活性化関数の適用の処理がある。T_{COM} は要素数 N の float 型配列を FPGA に転送する時間を測定することで求める。T_{FPGA} は次のように見積もる。畳み込み処理は出力データのパラメータに関するループ、フィルタに関するループの入れ子構造で処理されており、実際の算術演算は乗算と加算である。ここで、FPGA タブレットのアプリケーションから利用できる回路書き換え可能領域には 120 個の DSP が存在するので、積和演算が 1 クロックで完了し、かつ 120 個の乗算器を用いて並列化を行う場合を仮定する。フィルタとの畳み込みとバイアスの加算を含めた全体の計算時間は次の式となる。

$$T_{FPGA} = \frac{T_c}{120} * N_f * W_{out} * H_{out}(C * W_{fil} * H_{fil} + 1)$$

但し、T_c はクロック周期、N_f は畳み込むフィルタ数、W_{out},H_{out} は出力されるデータの幅と高さ、C は入力チャネル数、W_{fil},H_{fil} はフィルタの幅と高さであ

表 5　モデル式の各パラメータの測定結果

パラネータ	時間 (ms)
T_{CPU}	0.862
T_{COM}	0.725
T_{FPGA}	0.028
T	1.615

る。扱う画像及びフィルタが正方形の場合、それぞれの辺の大きさを L_{out},L_{fil} と置くとさらに簡略化することができる。

$$T_{FPGA} = \frac{T_c}{120} * N_f * {L_{out}}^2 (C * L_{fil}^2 + 1)$$

4.4.2　測定結果

T_{CPU}, T_{COM} について、FPGA タブレット上で測定した結果を表 5 に示す。また、今回実装する畳み込み層のパラメータより、6 × 6 フィルタ 12 枚、入力画像は 64 × 64 × 3 とし、FPGA の動作クロック 100MHz から FPGA の処理時間 T_{FPGA} も求めることができる。以上より、モデル式から畳み込み層の順伝播処理の計算時間の推定値が求まり、その値は 1.615ms であった。また、CPU のみでの畳み込み層の順伝播の計算時間は 52.367ms であった。このことから FPGA による部分専用回路化を行うことで約 32 倍のスピードアップが期待される。また、活性化関数の正規化線形関数も回路に組み込むと高々 3072 要素の符号検査で済むため、約 72 倍の高速化が期待できる。ただし、実際の回路の設計に依存する部分が大きいため、この見積もり値よりも効率が悪くなることが予想される。

表 6　単純な高位合成結果 (レイテンシ)

	min	max
レイテンシ (clock sycles)	1,623,577	60,900,889
インターバル (clock sycles)	1,623,578	60,900,890

4.5　専用回路の作成

部分専用回路の作成にあたって、本研究では Vivado HLS[10] を用いた。

4.5.1　Vivado HLS

Vivado HLS とは、Xilinx 社による高位合成ツールである。近年、専用回路化が望まれるアルゴリズムは高機能なものが多く、開発コストが多くかかってしまう。このツールでは、C や C++ などの高級言語で実装されたアルゴリズムから IP を自動で生成することが可能であり、手動で RLT を設計せずともそれと同等の機能を迅速に実装することができる。また、C や C++ のテストベンチやシミュレーション、VHDL や Verilog-HDL のロジックシミュレーションも可能であるため、低コストでの開発が可能となる。

4.5.2　回路の設計

本研究では Vivado HLS を用いて、畳み込み層の順伝播の処理を専用回路化した。まず、Java で実装された順伝播のプログラムを C++ の形式に書き直し、回路の生成を行った。この際にツールが吐き出した回路のレイテンシと資源データに関するデータを表 6 と表 7 に示す。 畳み込み層の順伝播に必要なデータは、層への入力、フィルタの値、バイアス値となる。高位合成された回路モジュールへの I/O は表 8 の通りである。この回路のシミュレーションの波形を図 23 に示す。この図の通り、各入力データはアプリケーションからは単精度の浮動小数点数の配列データとして受け取るが、モジュールへは値を一つずつ送信していく

表 7　単純な高位合成結果 (回路資源)

	DSP48E	FF	LUT
Expression	-	0	213
Instance	5	348	711
Multiplexer	-	-	197
Register	-	358	-
Total	5	706	1,121

ことになる。その際に、モジュール側のデータ受付準備ができているかどうかを `xxx_ce0` のイネーブル信号によって判断する。計算が完了した場合は、`ap_done` が 1 となり、計算結果が返却されるが、このときは `res_we0` のイネーブル信号毎に `res_d0` からデータが送られてくる。気を付けなければいけないことは、モジュール側が要求するデータは送信した配列と同じ順番になっているわけではなく、必要とするデータは `xxx_address0` で指定された順番である点である。

この回路とアプリ側のデータを保持しておく BRAM を合わせた回路が部分専用回路となる。この部分専用回路のブロック図を図 24 に示す。

表 8　合成された順伝播回路への入出力

ピン	IN/OUT
ap_clk	input
ap_rst	input
ap_start	input
ap_done	output
ap_idle	output
ap_ready	output
data_address0	output 4bit
data_ce0	output
data_q0	input
filter_address0	output 4bit
filter_ce0	output
filter_q0	input
res_address0	output 4bit
res_ce0	output
res_we0	output
res_d0	input
bias_address0	output 2bit
bias_ce0	output
bias_q0	input

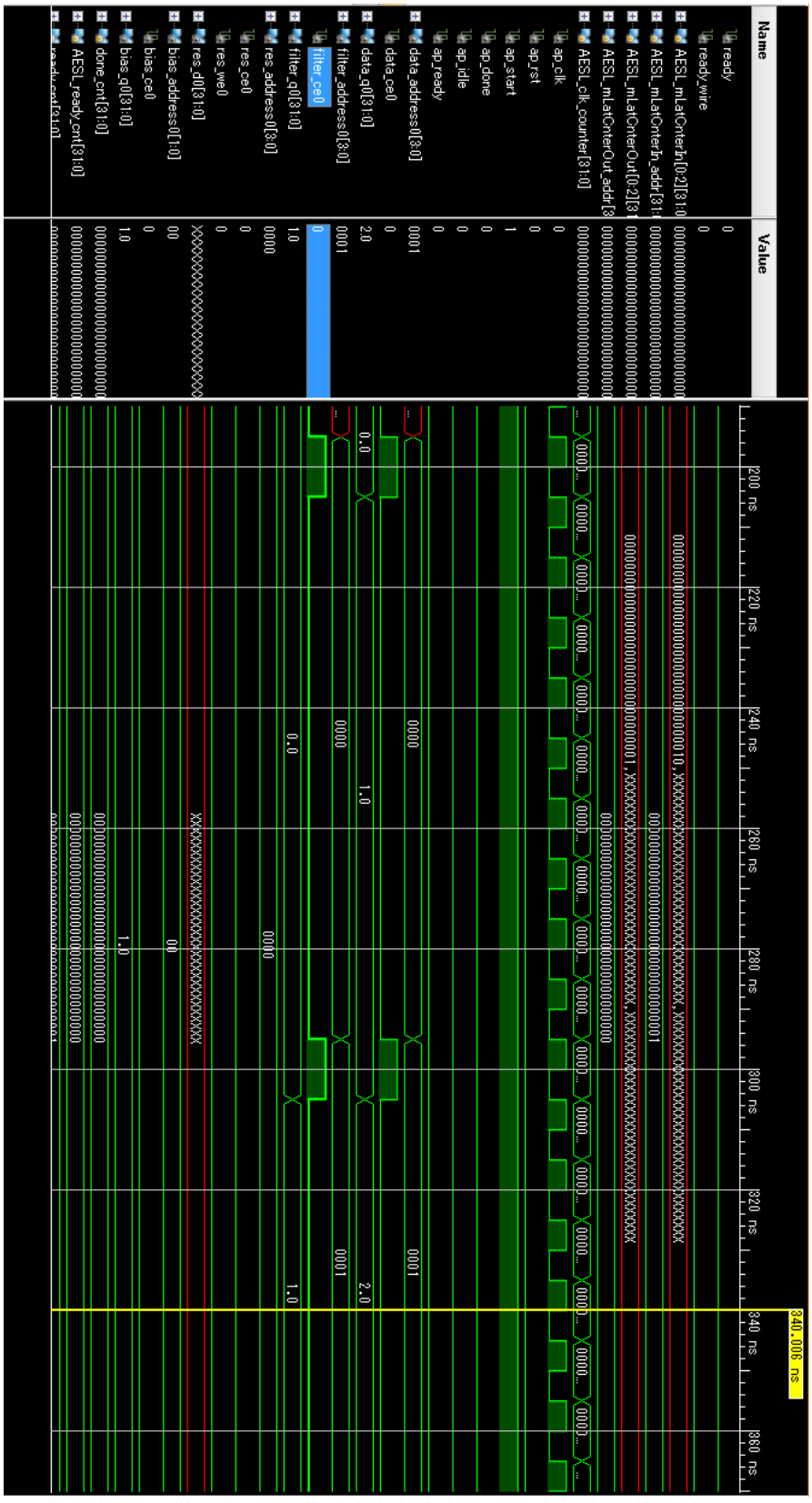

図 23　畳み込み層順伝播処理の部分専用回路の波形

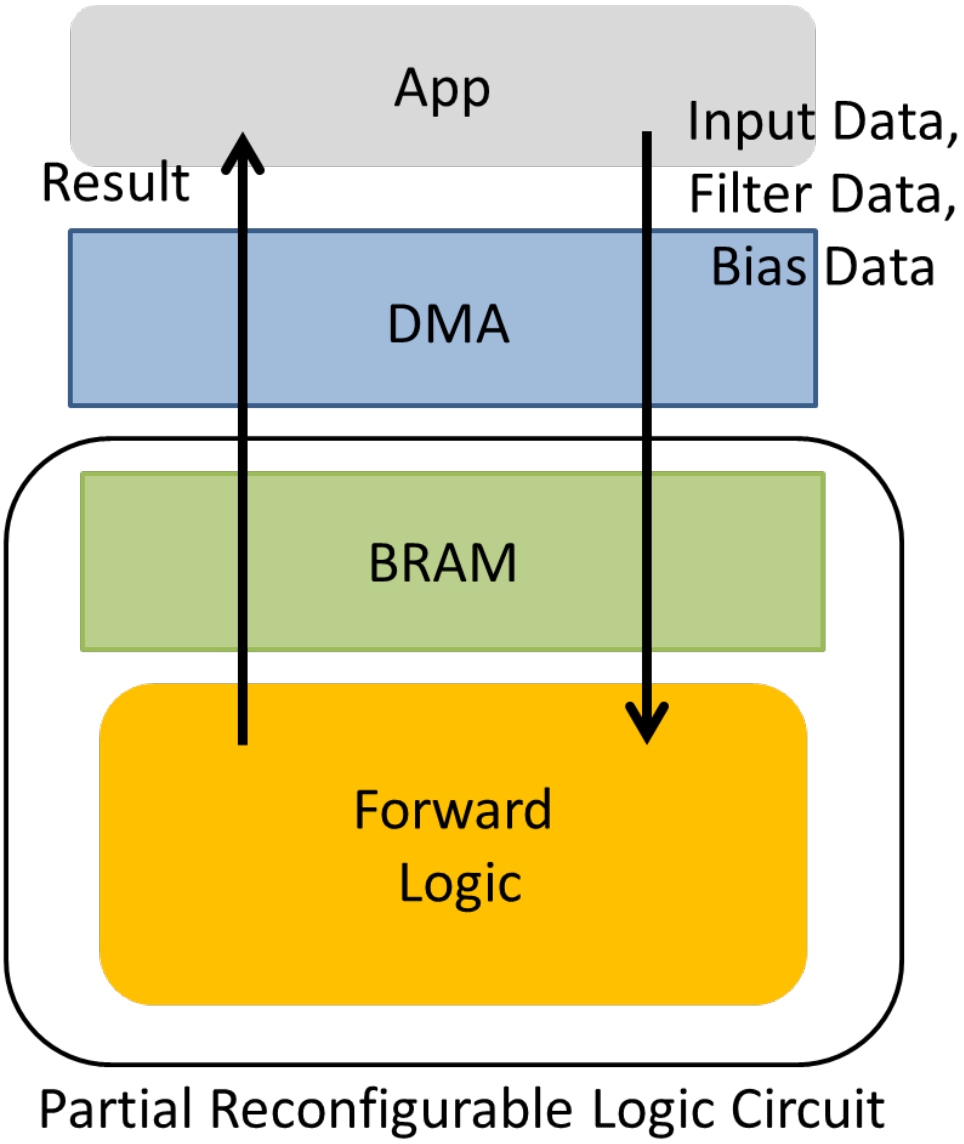

図 24　畳み込み層順伝播処理の部分専用回路のブロック図

表 9　回路化した畳み込みニューラルネットの順伝播処理にかかる計算時間

	min	max
レイテンシ	16.236ms	609.01ms
インターバル	16.24ms	609.01ms

表 10　畳み込みニューラルネットの順伝播処理全体でかかる計算時間

	min	max	Ave
T_{CPU}			0.862ms
T_{COM}			0.725ms
回路のレイテンシ	16.236ms	609.01ms	
T	17.827ms	610.597ms	

4.6　評価・考察

4.6.1　評価

Vivado HLS で作成した畳み込みニューラルネットの順伝播処理に掛かる時間を計算する。表 6 から処理に掛かるレイテンシと、FPGA タブレットの動作周波数 100MHz を考慮したおおよその計算時間を表 9 に示す。また、部分専用回路への転送時間、CPU 側での処理時間等を含めた畳み込みニューラルネットの順伝播全体でかかる計算時間を表 10 に示す。

4.6.2 考察

前述の計算結果から、Vivado HLS で作成した部分専用回路は最小で 17.827ms の処理時間がかかることが分かった。また、CPU のみの畳み込み層の順伝播の計算時間は 52.367ms であるから、2.938 倍の高速化になる。但し、最大で 610ms の処理時間になるため、この場合は CPU の演算性能よりも劣ることになる。今回の場合、Vivado HLS で自動的に生成させた為、開発者はほとんど順伝播処理の HDL 設計を行わなくて済んだ。開発者は生成された回路と CPU 側との通信を行う為の回路の設計程度でよく、FPGA タブレットのインタフェースが決まっていることを考慮すると、大抵の場合一度設計してしまえば、Vivado HLS で作成した回路を取り換えるだけで別の部分専用回路を作成することが可能となる。ただし、厳密には Vivado HLS で回路化を行うモジュールの I/O は高位合成を行う際に定義した関数の引数に依存するため、モジュールへの値渡しと受け取りは毎回設計しなおさなくてはいけない。

Vivado HLS で高位合成した回路は基本的にハードウェアを意識して記述したものではなく、CPU 上で動作することを前提として書かれたプログラムである。さらなる高速化を行う場合は、高位合成する元の C++ のソースコードをハードウェアを意識して書く必要があると思われる。また、Vivado HLS の機能として、ディレクティブを指定することが可能である。ディレクティブの中にはループアンローリングを指定したり、パイプライン化を行うこともできる。ここで、全てのループをアンローリングするように指定した場合の高位合成結果を表 11 と表 12 に示す。ループの状態遷移を考えなくてよくなった為か、レイテンシの最大と最小の差がなくなったことが確認できた。しかし、最小の場合のレイテンシが 1.67 倍となってしまった。また、アンローリングしたことによって組み合わせ回路が増えたことにより LUT の消費が増えたことも確認できた。単純に高位合成した場合、最大のレイテンシがかなり大きいこと、アンローリングすることによって最小レイテンシが増えるというデメリットを考慮した上で適切なディレクティブの指定によって、より高い性能を示す回路の作成が期待できる。

表 11　全ループをアンローリングした高位合成結果 (レイテンシ)

	min	max
レイテンシ (clock sycles)	27,084,313	27,084,313
インターバル (clock sycles)	27,084,314	27,084,314

表 12　単純な高位合成結果 (回路資源)

	DSP48E	FF	LUT
Expression	-	0	6,560
Instance	5	348	711
Multiplexer	-	-	1,839
Register	-	1,653	-
Total	5	5,001	9,110

5　まとめ

5.1　本研究で行ったこと

本研究では、FPGA タブレットを用いて二種類の人工知能アプリケーションの高速化を行った。

オセロアプリでは、AI の一部の処理を部分回路化することにより CPU と比較して平均約 2 倍程度の高速化に成功した。

Deep Learning を用いた画像分類アプリでは、畳み込み層の順伝播の処理を部分回路化することにより、FPGA タブレットの部分専用回路領域に用いられている資源やデータ転送時間等を考慮したモデル式では約 32 倍、活性化関数も回路化すると約 72 倍の高速化が望める。VivadoHLS を用いた高位合成によって生成された部分専用回路の場合、CPU と比較して平均 2.938 倍程度の高速化が望めることを示した。これにより、HDL を自作せずとも C や C++ などの高級言語から回路を自動生成することによってある程度の高速化が望める。

5.2　今後の課題

本研究で用いられる部分専用回路の設計に関しては、3 章の考察にもあった通り、効率的な高速化が望めるものと望めないものが存在し、対象の処理が多重ループになっているものや、少ないデータで大量の計算を行うような処理に関しては有用である。この点について具体的な評価の指標が示せれば、FPGA タブレットの利用場面の選別が可能になると考えられる。

また、これまでの課題であった HDL の自作による開発者への負担は、高位合成を用いることで軽減できる可能性を示したが、実際には自動生成したモジュールを基に部分専用回路を設計するため、まだまだ開発者への負担は大きいと言わざるを得ない。

FPGA タブレット自体の改良は今回は行っていないが、モバイル端末としてはかなり大きい端末であるため、できるだけ小型化することも今後の課題である。

参考文献

[1] 第 2 回 : FPGA の基礎その 2, http://www.fpga-net.jp/basic-lesson/fpga02-2.html

[2] 塩谷丈史, 成見哲, “FPGA タブレットによるモバイルアクセラレータ”, 情報処理学会研究報告,Vol.2015-HPC-148, No.18, 2015.

[3] ZedBoard, http://zedboard.org/product/zedboard

[4] Wong, C.K., Lo, K.K. ; Leong, P.H.W., *”An FPGA-based Othello endgame solver”*,Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on, PP81-88, 2004.

[5] テキサス・ホールデム達人, https://play.google.com/store/apps/details?id=com.droidhen.game.poker

[6] M. Zinkevich, M. Johanson, M. Bowling, C. Piccione, *”Regret Minimization in Games with Incomplete Information”*, NIPS 2007.

[7] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, Eric Chung, *”Accelerating Deep Convolutional Neural Networks Using Specialized Hardware”*, Microsoft Research, February 23, 2015.

[8] A. Krizhevsky, I. Sutskever, GE. Hinton, *”Imagenet classification with deep convolutional neural networks”*, Advances in neural information processing systems, PP1097-1105, 2012.

[9] N. Srivastava, GE.. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, *”Dropout: a simple way to prevent neural networks from overfitting.”*,Jounal of Machine Learning Research, PP1929-1958, 2014.

[10] Vivado HLS, https://japan.xilinx.com/products/design-tools/vivado/integration/esl-design.html

謝辞

本研究を進めるにあたり、ご指導いただいた主任指導教員の成見哲教授、また副指導教員の佐藤証教授に感謝いたします。