目次

1. 研究動機

私は FPGA（リコンフィギャラブルシステム）と高位合成技術の研究に興味がある。特に、抽象度が高い言語からハードウェアを意識せずに、最適化された回路を生成する技術に興味がある。高位合成について研究するには FPGA やハードウェアの知識が不可欠となる。しかし、私はそれらの知識や開発経験に乏しい。そのため、卒業研究として FPGA 上のプログラム・回路実装をテーマとして、大学院でその高位合成の研究を行おうと決めた。

今回の研究で実装するプログラムは、モルフォロジー画像処理を採用した。モルフォロジー画像処理を実装するプログラムとして採用した理由は、一要素あたりの処理内容が単純でスケールアップしやすいと考えたからだ。

私が高位合成を研究したい理由としては、FPGA によるハードウェアロジックによる処理が、近年の計算資源の枯渇の解決策の一つになり得ると私が期待しているからだ。

数年前まで、CMOS プロセスが順調に向上し CPU のクロックはそれに比例して向上していた。だが、近年で CPU クロックの向上は鈍化、もしくは頭打ちとなっている。[24] 民生用 CPU のクロックは３～４Ghz で落ちついている。[24] すなわちこれは、シーケンシャル性能の頭打ちとも理解出来る。そして、ここ数年クロックの代わりに CPU のコア数が増加している。しかし、ビッグデータ処理や高速ネットワーク等の影響で計算資源の需要の伸びが CPU の性能向上を超えてしまっている。計算需要に答えるために、CPU の処理を肩代わりするものとして GPU が汎用計算分野でも使用されるようになった。GPU は、クロックや１コアあたりの性能が CPU に比べ低い。そのかわり、膨大なコア数と、そのコアが同時にデータにアクセスするための広いメモリバンド幅をもっている。そのため、疎結合な処理やデータにおいては GPU が CPU 計算速度で秀でることが出来る。しかし、GPU は本来画像描画用に設計されてきた経緯上 SIMD 演算に最適化されている。MIMD 演算を実行使用とすると処理時に SIMD 演算に変換され大きなオーバヘッドが発生する。このことから、密結合な処理やデータ、細粒度な処理は GPU では困難である事が分かる。

GPU のように超並列演算がおこなえ、MIMD 処理もおこなえる計算リソースとして FPGA が注目されている。[5] FPGA は、ハードウェアロジック・アーキテクチャから設計するため ISA や既存のプロセッサアーキテクチャによる制約がない。そのため、データの量、粒度に合わせて演算ユニットの数を調整、データの性質に併せて演算ユニットを設計可能だ。だが、そのような設計の自由度の高さとハードウェアレベル(RTL)からの設計が必要であるために、プログラムの実装・設計が CPU や GPU と比べ困難である。

開発コスト、開発期間もそれに伴って CPU や GPU と比べ大きい。この開発難易度という短所を補うために、近年注目されているのが高位合成だ。高位合成は動作記述から回路データを生成する技術である。しかし、高位合成フレームワークがベースとして採用している言語は決してモダン言語とは言えない C や Java である。また、ハードウェアを意識してコード記述する必要がある。RTL 記述に比べ高位合成の開発難易度は低くなっているが、CPU や GPU 向けのプログラムに比べると依然として開発難易度は高いままである。そのため、私は FPGA の開発難易度を下げるであろう、より抽象度が高くハードウェアを意識する必要がない高位合成技術に興味がある。

2. 基盤技術

2-1. FPGA

FPGA とは field-programmable gate array の略である。
プログラムすることで後からハードウェアロジックを実装出来る論理デバイスのことである。FPGA は、広義のプログラマブルロジックデバイス（PLD）の一種である。しかし、これまでの PLD とは設計の自由度の高さ、ゲートアレイに近い構造を持つ点で異なり。そのため、FPGA と称される。ほとんどの FPGA は何度もハードウェアロジックを書き換えることが出来る。その再書き換えを高速化することによって、動的に再構成可能な FPGA も実現している。

これまで、FPGA・PLD を 40 社以上が製造していた。しかし、現在 FPGA のほとんどが２社によって生産されている。Xilinx と Intel（Altera)。Intel は 2015 年に約 167 億ドルで Altera を買収した。

現在の FPGA はスタティックメモリによって構成されているものが主流だ。スタティックメモリタイプの FPGA は CMOS プロセスの恩恵を受けることが可能で。また、無制限に書き換えが可能だ。短所としては、揮発性であること、リーバースエンジニアリングに弱いこと等が挙げられる。

他に、アンチヒューズタイプとフラッシュメモリタイプがある。

アンチヒューズタイプは、高密度で実装可能、不揮発であること、リーバースエンジニアリングに強いこと、そしてソフトエラー耐性があることが利点だ。短所としては、書き換え不可能、書き込みに欠陥が存在しても再書き換え出来ない故に歩留まりが 100％にならない点。

フラッシュメモリタイプは、不揮発であること、LAPU（Live At Power-UP;電源投入後に即動作）可能であることが利点だ。短所としては、書き換えに高電圧が必要なこと、最先端 CMOS の微細化プロセスが使えない点である。

「CMOS の最先端微細化テクノロジー」使用できる点で、他のプログラミングテクノロジーを圧倒している為スタティックメモリベースの FPGA が主流となっている。

ここでは、大手 2 社の FPGA について説明する。アイランドスタイルの FPGA は、主として論理ブロック、乗算ブロック、I/O ブロック、メモリブロック、そしてそれら

を接続する、スイッチブロック、コネクションブロック、配線チャネルからなる。論理ブロックと乗算ブロックは論理回路を表現する為の演算回路。メモリブロックはメモリである。乗算ブロックとメモリブロックは製造時に作り込まれたものである。論理ブロックは 4 入力ルックアップ・テーブル(LUT:　Look-Up Table)の組み合わせによって実装されている。

FPGA のハードウェアロジック設計は、ハードウェア記述言語(HDL)によるレジスタ転送レベル(RTL)のソースコードによって開発されることがほとんどだ。もともとは、論理回路の記述によって設計されていたが回路規模の増大によって RTL による開発が主流となった。

HDL からの設計は、RTL 記述、論理合成・テクノロジーマッピング、配置配線、コンフィギュレーションデータ生成、FPGA に書き込みによってなされる。このとき、FPGA の型番、ボード情報、クロック等の物理的な制約の情報も必要となる。

論理合成とは、RTL 記述から論理回路・順序回路を生成する工程である。論理合成の結果はネットリストとして出力される。ネットリストは、論理ゲートやフリップフロップなどの論理素子の集合とその接続関係を表すモノである。

テクノロジーマッピングは、ネットリストが表す論理を実際の FPGA の論理素子に割り当てる工程である。

配置配線は、ネットリストをチップ上の論理資源や配線資源に割り当てる工程である。配置配線は非常に時間のかかる作業である。特に、回路規模が大きくなり論理資源の使用率が高くなると、計算時間が膨大になりまた配線が失敗する可能性も出てくる。

完成した回路のデータは、コンフィギュレーションデータ、ビットストリーム、プログラムファイルと呼ばれる。回路データを FPGA に書き込むことによって、FPGA にハードウェアロジックが実装される。デバイスにコンフィギュレーションデータを書き込むにはプログラマと呼ばれるツールが用いられる。

RTL による設計は、人手による最適化が可能だが、設計難易度が高く、人為的ミスが混入しやすい。そのため、抽象度の高い動作記述から回路設計する技術が研究されている。また、部分的に実用化もされている。この技術を高位合成（HLS: High Level Synthesis）あるいは動作合成(Behavioral Synthesis)と呼ぶ。現在多くの高位合成フレ

ームワークは、C 言語もしくは C 言語をベースとしたものを採用している。ソフトウェアとしてのプログラムがそのままハードウェアに合成されるわけではない。FPGA 上でプログラム実装する際に、ハードウェアアルゴリズムを意識する必要がある。ハードウェアアルゴリズムで特にしようされるのは、パイプラインと並列化である。パイプライン処理は、連続して行われる多数の処理を高速化する、つまりスループット向上の為に用いられる手法である。パイプライン処理は、逐次的な処理を多段階に分割し、先行する処理全体の完了を待たずに、段階ごとに処理を実行することによってスループットを向上させる。並列化はその名の通り、処理やデータにあった演算器の数を用意することによって率直にスループットを向上させる。これら以外にも、ベクトルや行列に最適化したアルゴリズムであるシストリックアレイ、ノイマン型計算機が本質的にもつ命令メモリフェッチによるボトルネックを解消するデータフローマシン、ハードウェアに最適化したソーティングアルゴリズやパターンマッチングアルゴリズムなどが存在する。

FPGA は､PLD や ASIC の代替デバイスとしてのみならず､スーパーコンピュータ、ネットワーク、ビックデータ処理、ゲノム科学、金融市場、人工知能、画像処理といった様々な分野で用いられている。

2-2. 高位合成

高位合成とは、高級言語で書かれた動作記述から回路を記述可能なコード(Verilog HDL, VHDL)を生成するツール・仕組みである。FPGA のゲート数の向上にともない、RTL による設計が困難になっている。RTL は、クロックや論理回路等のハードウェアレベルの要素を設計者に意識させる。また、記述は冗長で膨大な量になりがちだ。そのため、設計にかかる時間や混入する人為的設計ミスも多くなる。これらが、設計すべき回路の増大によって顕著になってきている。

そのため、抽象度の高い動作記述から回路を生成する高位合成が注目されている。ほとんどの、高位合成フレームワークが、C や Java といった高級言語の中でも抽象度の低い言語、もしくはそれらをベースとした言語を採用している。たとえば、Xilinx 社の Vivado HLS は C 言語から HDL を生成する。また、Altera 社の SDK for OpenCl は C をベースとしている。

ソフトウェアとしてのプログラムがそのままハードウェアに合成されるわけではない。記述によっては、ハードウェアの性能が大きく異なることもある。このため、高位合成においても、生成されるハードウェアを意識した記述が開発者には求められる。高位合成では、ソフトウェアプログラミングと同様に変数、演算子、代入文、制御文等が使用できる。多くの場合、変数はレジスト、配列はメモリ、回路は回路モジュールとしてインスタンス化される。また、ステートマシンによって、逐次実行、分岐、ループ、関数呼出などの制御が実現される。

高位合成には、ハードウェアロジックを生成する為の特有の制限が存在する。その中でも、ほとんどの高位合成フレームワークに共通する制約として、再呼出しの禁止と動的ポインタの禁止が存在する。関数の再帰呼出しは、関数すなわち回路モジュールのインスタンスが実行時に動的に生成される、そのため高位合成では禁止されている。動的ポインタも同様に、実行時にポインタ値が変化し、アクセス対象のメモリインスタンスが実行時に変化するため、高位合成では禁止されている。

高位合成では、多くの場合任意のビット幅の型を使うことが出来る。これにより、設計者は、回路規模、動作速度、消費電力と演算精度を最適化出来る。

現在のところ、プログラムやデータから依存関係を解析し最適な並列化を行うことは困難である。そのため、設計者が pragma や directive、あるいは拡張命令でループのパイプライン化やブロックの並列処理化を指示する必要がある。

以上のことから分かるように、高位合成を用いて FPGA を設計する場合でもハードウェアの知識や、抽象度の低い記述が求められる。しかし、RTL に比べ高位合成は設計コストが圧倒的に低いため現在多くの分野で使用されている。

2-3. モルフォロジー

モルフォロジー処理とは、画像に対して膨張処理と収縮処理を行い、画像の持つ構造を抽出する処理である[15]。画像の平滑化、孤立点除去、輪郭抽出やノイズ除去等に利用されている。この論文では、二値化された白黒画像に対してのモルフォロジー処理の具体的説明を行う。

すでに述べたように、モルフォロジー処理は膨張・収縮処理で構成される。膨張処理(Dilation)とは、注目した画素とその近傍に対して論理和(OR)をとる処理である。収縮処理(Erosion)とは、注目した画素とその近傍に対して論理積(AND)をとる処理である。

モルフォロジー処理では、多くの場合膨張・収縮処理を繰り返し行う。とくに、収縮を繰り返し行った後、同様の回数だけ膨張する処理をオープニング(Opening)と呼ぶ。オープニングは、図形の突起部分の除去や結合部分の分離等に利用される。オープニング処理とは逆に、膨張を繰り返し行った後、同様の回数だけ収縮する処理をクロージング(Closing)と呼ぶ。クロージングは、図形の穴埋めや切断部分の結合等に利用される。オープニング、クロージング両処理とも、小さいパターンや細かいパターン(ノイズ)を除去する効果を持つ。

また、元画像とクロージング後の画像の差分をとる処理をブラックハット(Black-Hat)と呼び、元画像とオープニング後の画像の差分をとる処理をトップハット(Top-Hat)と呼ぶ。

図 0 モルフォロジー処理画像

一番右の画像が元画像、順に、オープニング処理を行った画像、クロージング処理を行った画像。

3. モルフォロジー処理回路の FPGA への実装

3-1. 研究環境

3-1-1. 使用したものリスト

今回研究、モルフォロジー処理回路の実装に使用した機材をリストにする。そして、以降の章にて説明する。

リスト

1. ZYBO
2. Vivado 開発ツール
3. USB ケーブル
4. コンピュータ（開発・デバック等に使用）

3-1-2. ZYBO

今回､実装と実験に Digilent 社の ZYBO を使用した｡Zybo を利用した理由としては、大きく４つある｡一つは､秋月電子ですぐに購入できる点｡税込み 24.700 円に収まる、すなわち実験費に収まる点。Xilinx 社が高位合成ツールを無償でユーザに提供している点。最後に、ユーザ・デベロッパーが他のボードに比べて多くの情報が手に入りやすいからだ。また、上記の４つほどではないが、Xilinx 社の統合開発環境が Altera 社の統合開発環境よりも個人的に使い安いと感じた点も理由の一つとなっている。

Zybo は、初心者向けの開発ボードで、ザイリンクス Zynq-7000 ファミリの最小デバイス Z-7010(XC7ZZ010)を搭載した FPGA ボードである。Z-7010(XC7ZZ010)チップは、ARM 社の Coretex－A9 プロセッサデュアルコアをハードマクロとして搭載している。そのため、PC との USB 接続による書き込みや通信等の処理をプロセッサとハードマクロを介して行うことが出来る。ハードマクロとは、FPGA チップ上にあらかじめ C 組み込まれた機能の事である。メリットはソフトマクロと比べ動作周波数・性能が高い点で、デメリットはそれらを使用しない場合回路面積が無駄になる点である。

ＺＹＮＱ　ＸＣ７Ｚ０１０－１ＣＬＧ４００Ｃの主要な機能：XILINX のレファレンスページより

1. 650MHz デュアルコア Cortex™-A9 プロセッサ
2. 8 DMA チャネルの DDR3 メモリ コントローラー
3. 28,000 ロジック セル (FPGA の回路規模を比較する為の値)
4. 17,600 ルックアップテーブル（任意の論理の組み合わせ回路を作成するブロック）
5. 240KB ブロック RAM (FPGA 内蔵されているメモリ｡初期値を用意してＲＯＭとしたり、FIFO として使用したりする。任意のビット幅、任意のワード数を設定できるが、内部的には１８K ビット単位で処理される。)
6. 80 DSP スライス(加算器と乗算器が組み合わさったもの)

ボード（Zynq チップに搭載されたものを除く）の主要な機能：XILINX のレファレンスページより

1. 128Mb シリアル フラッシュ (QSPI インターフェイス付き)
2. 16 ビット/ピクセルの VGA 出力ポート
3. 512MB x32 DDR3 (1050Mbps 帯域幅)
4. オーディオ コーデック (ヘッドフォン出力、マイクロフォン/ライン入力ジャック付き)
5. デュアルロール (ソース/シンク) HDMI ポート
6. 外付け I EEPROM (48 ビットのグローバルな固有 EUI-48/64™ 互換識別子でプログラム済み)
7. GPIO : 6 プッシュボタン式、4 スライド スイッチ、5 LED
8. 高帯域幅ペリフェラル コントローラー : 1G Ethernet、USB 2.0、SDIO
9. 低帯域幅ペリフェラル コントローラー : SPI、UART、I2C
10. OTG USB 2.0 PHY (ホストおよびデバイスをサポート)
11. オンボード JTAG プログラミングおよび UART-USB コンバーター
12. オンチップ デュアル チャネル、12 ビット、1 MSPS アナログ/デジタル変換器 (XADC)
13. 6 Pmod ポート (1 プロセッサ専用、1 デュアルアナログ/デジタル)
14. トライモード (1Gbit/100Mbit/10Mbit) イーサネット PHY
15. microSD スロット (Linux ファイル システムをサポート)

図 1　Digilent 社製 ZYBO 評価ボードの全体写真

3-1-3. FPGA 開発ツール

FPGA 上に回路を構築する為には、そのための回路データが必要となる。その回路データを作成するには、専用の開発ツールが必要である。開発ツールは FPGA ベンダーそれぞれが、独自に開発している。Altera は Quartus® Prime、Xilinx は Vivado® Design Suite と SDSoC だ。

今回の研究では Xilinx 社 FPGA が搭載された ZYBO を使用しているので、開発環境は Vivado Design Suite HL WebPACK Edition を使用した。Vivado Design Suite HL WebPACK Edition は小規模 FPGA しか開発が出来ないが、無償で提供されている。Xilinx SDSoC は FPGA と CPU のヘテロジニアスコンピューティングアプリケーション開発ツールなのだが、ライセンスが有料かつ高額なので使用できなかった。

この Vivado Design Suite は、FPGA 開発に必要なツール群の総称である。Vivado Design Suite には統合開発環境である Vivado、ソフトウェア開発環境である Xilinx SDK、高位合成ツールである Vivado HLS などが含まれる。

統合開発環境である Vivado には、IP インテグレータ、論理合成、bit ファイル生成、配置・配線、ロジックアナライザ、シミュレータ、各種解析機能が含まれている。

IP インテグレータについて説明する。IP インテグレータとは、Xilinx 社およびサードパーティーが提供する IP コアを管理する機能である。IP コアとは、再利用可能なブロック、すなわちソフトウェアにおけるライブラリのようなものである。

ソフトウェア開発環境である Xilinx SDK は、Xilinx のソフトマクロ CPU である MicroBlase、ハードマクロの CPU 上でのソフトウェア開発に使用する。Xilinx SDK では、C/C++コンパイル・リンク、FPGA のコンフィギュレーション、プログラムのダウンロード、ソフトウェアのデバック、FPGA で実装したロジック・IP のドライバ管理、プロファイリング、Zynq 用ブートローダなどが可能である。

高位合成ツールである Vivado HLS はその名の通り、C/C++で記述されたソースコードから HDL を生成する高位合成フレームワークである。Vivado HLS は他の Vivado Design Suite ツール群との整合性が高いため、既存のツールのような他のツールとの連携するための手直しが必要ない。また、Vivado HLS では、高位合成、C/C++で記述した回路のテスト、インタフェースの生成、IP 化・ドライバソフト類(API)の自動生成が可能である。

図 2 Vivado の Project Summary

3-1-4. コンパイルや論理合成に使用したコンピュータ

今回の研究で、開発および実験時に使ったコンピュータのスペック等をリストにする。

1. CPU: Intel Core i7-4790k(4.00GHz,4Core,8Thred)
2. Memory: DDR3 16.0GB
3. Storage: Samsung SSD 850 EVO 250GB
4. OS: Windows 10 Home
5. コンパイラ、開発環境: Visual Studio Community2015、MSYS2/GCC6.2.0
6. 使用ライブラリ: OpenCV3.1

3-2. 実装

3-2-1. 回路・システム概要

次のページの図が ZYBO に実装された回路の概略図である。図に示された回路の簡略な説明を記述する。

1. processing_system7_0:
   プロセッサとその周辺回路、ソフトウェア処理を行う
2. morphology_0:
   モルフォロジー処理を行う回路、以降の章で説明
3. axi_gpio_0:
   led を制御する為の回路、デバッグ用に使用
4. rst_ps_7_100M:
   リセット信号を生成する回路
5. axi_mem_intercon:
   AXI HP ポート(High-Performance Port:高速データ通信用ポート)をかいして、PL(プログラマブルロジック)から PS(プロセッサ)へ接続する為の回路
6. ps7_0_axi_periph:
   AXI GP ポート(General-Purpose Ports:低速ポート、レジスタ設定用)をかいして、PS(プロセッサ)から PL(プログラマブルロジック)へ接続する為の回路。

AXI とは ARM 社が策定および公開しているインタフェースの規格である。

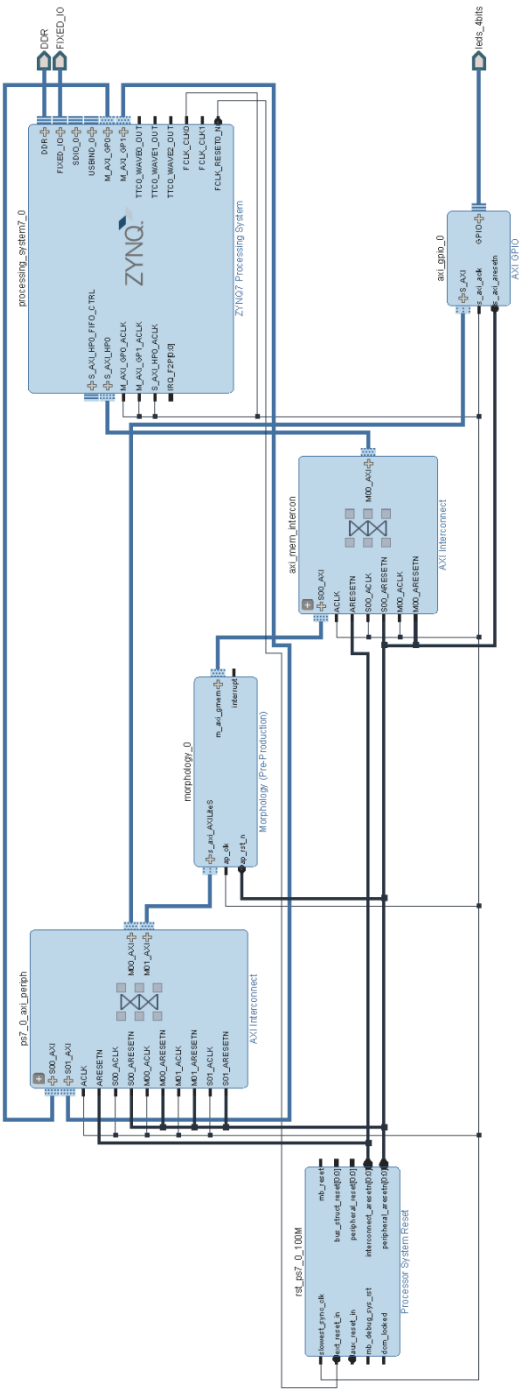
DDR
FIXED_IO
leds_4bits
processing_system7_0
ZYNQ7 Processing System
axi_gpio_0
AXI GPIO
axi_mem_intercon
AXI Interconnect
morphology_0
Morphology (Pre-Production)
ps7_0_axi_periph
AXI Interconnect
rst_ps7_0_100M
Processor System Reset

図 3　システム図

3-2-2. ハードウェア部とソフトウェア部の分割戦略

今回モルフォロジー画像処理アプリケーションを FPGA 上に実装するに当たって、モルフォロジー処理そのものをハードウェアで、その処理の制御や入出力等の部分をソフトウェアで分けて実装した。なぜなら、モルフォロジー画像処理のハードウェア処理化が今回の研究のテーマであり、そのテーマと関連が薄い制御・入出力処理の実装の手間を省く為である。

次の章でソフトウェア実装部、その次の章でハードウェア実装部について説明する。

3-2-3. ソフトウェア部の実装

今回の実装では、モルフォロジー画像処理回路の制御、入出力、時間計測等の処理をソフトウェアで実装した。これらの処理をソフトウェアで実装した理由の一つは前述したように、今回の研究のテーマと関連が薄いというものである。そして、これは前述していないが、上で述べた処理がハードウェアで実装しづらい処理だというのも理由の一つである。

モルフォロジー画像処理回路の制御は、Vivado HLS が IP コア生成時に同時に自動的に生成される API を使うことで行える。

入出力処理は、Xilinx 社が提供する xil_io ライブラリを使用した。
xil_io ライブラリの関数は、C の標準入出力関数に比べて少ない計算リソースで入出力処理を行うことが出来る。

時間計測は、Xilinx 社が提供する xtime ライブラリの XTime_GetTime 関数を使用した。XTime_GetTime 関数は、クロック数を計測することで経過時間を計測することが出来る関数、すなわちベアメタル(OS レス)で時間を計測することが出来る関数である。

今回のモルフォロジー処理の実装では ARM コアをベアメタルで使用した為、Xilinx 社の ARM コアをベアメタルかつ少ないリソースで使用できる関数群がとても有用であった。

3-2-3. ハードウェア部の実装: 高位合成の使用

今回モルフォロジー画像処理回路を実装するにあたって、高位合成ツール(Vivado HLS)を使用した。なぜなら、HDL を使ってその回路を実装することが現在の私の知識、技術では困難であるからだ。高位合成を使わずに回路設計をする場合 HDL(ハードウェア記述言語:ハードウェア回路を記述する為の言語 Verilog HDL、VHDL などがある)で回路記述する必要がある。HDL で回路設計をする場合、クロックの記述、手動での最適化等複雑なハードウェア周りの設計が必要となるが、現在の私にはその周辺を効率良くかつ人為的ミスによるバグなしに実装することが難しい。そのため、今回の研究では高位合成を使うことになった。

また、高位合成を使うことによって、デメリットも存在したが、多くのメリットを享受することも出来た。

高位合成によるメリットは大きく５つある。一つは、C/C++記述でバグを調べることが出来る点、これによってソフトウェアの知見を使いハードウェアをデバッグすることが出来た、以降の章で詳しく記述する。HDL に比べて C/C++記述はソースコード短く簡潔になる点、ソースコードの大局的理解に役に立った。HDL に比べて C/C++はソースコードの記述にかかる時間が短い点、短い時間でプログラムが記述可能だったので何度もプログラムを書き直すことが出来た。高位合成後に性能予想レポートを自動的に生成する点、これによって何が最適な記述や適切な最適化かを判断することが出来た。そして、プラグマ、ディレクティブ(最適化の為の指示子)を使用することでソースコードを書き直す必要なく最適化(パイプライン、並列化等)、生成する回路を変更することが可能な点、これによって何種類ものパターンの回路試すことが出来た、以降の章で詳しく記述する。

高位合成を使用したことによるデメリットは大きく 2 つある。複雑なソースコードの高位合成には長い時間がかかる点、今回の実験では数時間の高位合成処理がざらにあった。また、複雑なソースコードの高位合成に多くのメモリが必要な点、今回の研究でも何度かメモリが不足し高位合成ツールが停止したことがある。

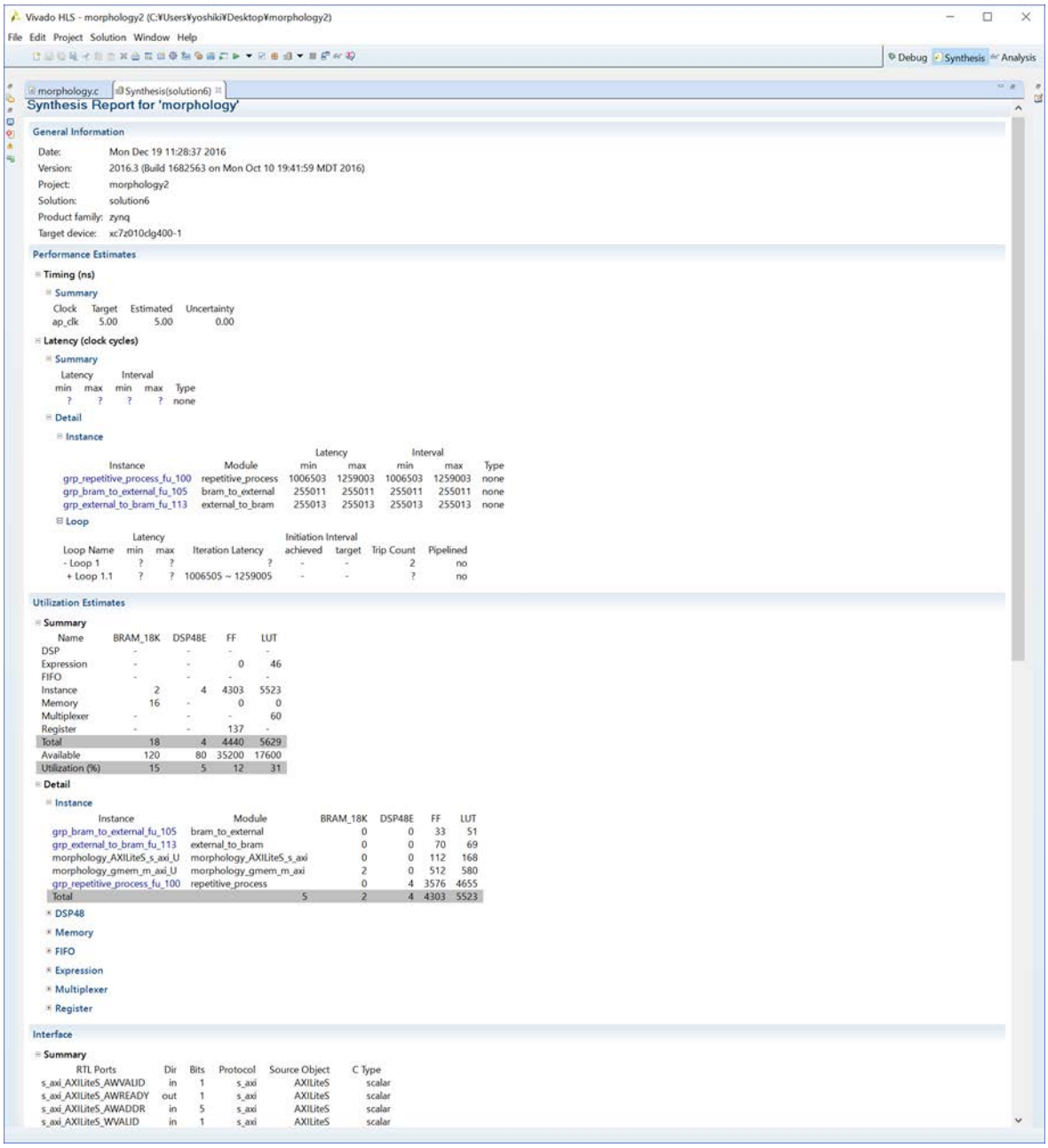

図 4 Vivado HLS　のシステムレポート(性能予想レポート)

3-2-4. ハードウェア部の実装: DMA の実装

DMA は今回の実装において最も行き詰まった部分である。

まず、DMA について説明する。DMA とは Direct Memory Access の略で、メモリに CPU を介さずにアクセスする機能である。DMA と対になるものとして、CPU を介して各デバイスとアクセスする PIO(Programmed Input/Output)が存在する。転送速度は、DMA の方が PIO に比べて高速である。

モルフォロジー処理回路の入出力データはデータ量が多くなるので DMA のバースト転送が必要となった。バースト転送とは、メモリのアドレス指定等の手順を一部省略することでデータ転送を高速化するものである。バースト転送では、あらかじめデータ幅を設定することにより、指定アドレスを先頭としてデータ幅分だけ一気にデータを転送する。

最初に DMA を実装するにあたって、IP コアを使い DMA コントローラを実装し、それとモルフォロジー処理回路を接続しようとした。しかし、それを行うにはクロックレイテンシやプロトコル等のハードウェアレベルの知識が必要になり、また、DMA コントローラとモルフォロジー処理回路の接続部分を HDL で記述する必要があった。これらは、私にとって困難なものであり、結果的にその手法で実装することが出来なかった。

そのため、より簡単に DMA バースト転送を使用する方法はないか Xilinx 社のレファレンスを調べると、Vivado HLS すなわち高位合成を使うことで DMA バースト転送が C/C++記述で実装出来ることが分かった。その方法は、Vivado HLS 上でメモリを配列として扱うようにディレクティブ(HLS INTERFACE m_axi)で指定して、その配列を for ループで愚直にコピーするコード(memcpy 関数でも可能だが、今回 64bit から 1bit にキャストする必要があったため for ループを使用した。) を記述するものだ。for ループ中に if 文等を挿入し処理を煩雑化するとバースト転送回路が生成されなくなると、レファレンスに記述があった。そのコードを高位合成すると自動的に DMA バースト転送回路とそのドライバ・API が生成される。生成された回路の m_axi ポートを s_axi ポート(今回は 64bit 幅の s_axi_hp ポート) と接続することで DMA が可能になる。

ディレクティブ(HLS INTERFACE m_axi)について図とともに説明する。

INTERFACE 指示句は、配列や変数を外部回路の入出力と対応させるものである。m_axi 指示句は、その配列や変数が axi バス上のデバイス(メモリ)と対応させるものである。

depth 指示句は、配列の要素数を指定するものである。(配列の要素ひとつひとつのビット幅は、配列の定義に従う。)

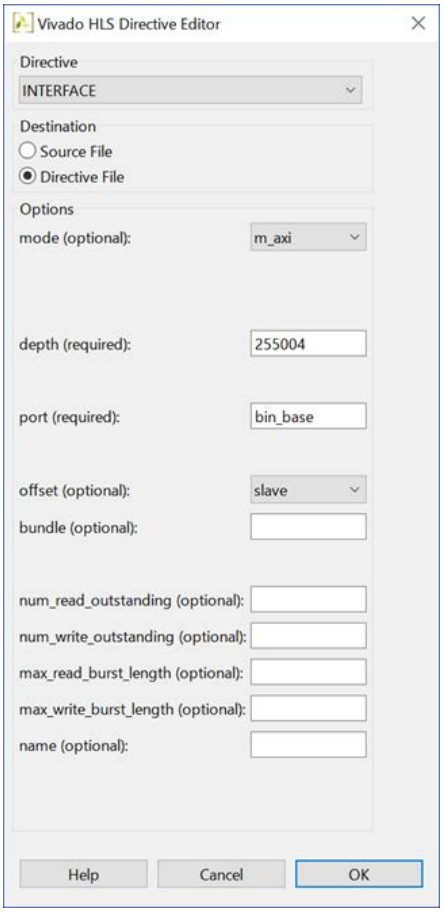

図 5　ディレクティブ設定ウインドウ

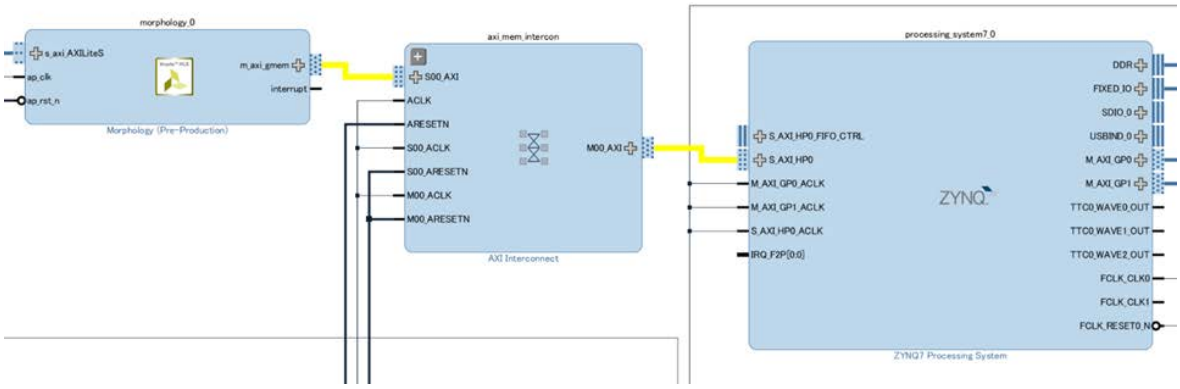

図 6　AXI バス接続概観

3-2-5. ハードウェア部の実装: ハードウェアを意識した記述

高位合成を使えばHDLを使わずにC/C++で回路を生成することが出来る。しかし、愚直にCPU上で動くソフトウェアと同じソースコードを高位合成すると、思うような性能が出ない場合や、回路規模が大きくなりすぎてFPGAチップに収まらないことがある。そのため、今回の実装でもハードウェアを意識したソースコードを記述する必要があった。

今回、まず愚直にCPU上で動くコードを高位合成したところ配列へのアクセス多くオーバーヘッドとなっていることが分かった。そのオーバーヘッドの理由は、簡潔に言うとブロック RAM へのアクセス数が多すぎるということである。以下、詳しく説明する。そもそも、ブロックRAM(BRAM)とはFPGA上に組み込まれているメモリの事である。Vivado HLSで配列を定義すると、ディレクティブ等で指示しないかぎり、基本的に BRAM が対応する。BRAM は、レジスタではなくメモリなのでクロックあたりに転送できるデータ量が限られている。そのため、配列へのアクセスが多いと、その配列に依存する処理が BRAM の帯域とクロックの律速となりスループット向上が難しくなる。

モルフォロジー処理において配列へのアクセスが多くなった理由について述べる。モルフォロジー処理のプリミティブな処理要素は、膨張と収縮処理である。膨張と収縮処理は、注目した画素とその周辺の画素に論理演算を行うものである。今回の実装では、注目要素の9近傍の論理演算を行う。すなわち、1要素あたりに9つの配列要素を入力としてアクセスし、1 つの配列要素を出力としてアクセスする必要があるということである。そのため、論理演算は 1 クロックで行えるはずだが、配列へのアクセスが多く、結果としてオーバーヘッドが大きく性能が出なかった。

このオーバーヘッドを解決するために、まとまった量のデータを配列から分散RAMへバッファリングして論理演算を行うようにソースコードを改良した。分散RAMは、ARRAY_PARTITION ディレクティブとオプショナル指示句の complete を配列に使うことで使用できる。分散RAMはCLB(Configurable Logic Block)、すなわちフリップフロップやロジックセル等を消費して実装する。分散RAMは非同期アクセス(任意データに即時にアクセス)が可能だ。今回のモルフォロジー処理では、1ラインごとに、注目ラインとその上下1ラインずつバッファリングすることで BRAM へのアクセス

数を抑えている。このバッファリング処理によってBRAMへのアクセスは約1/3に抑えることが出来た

3-2-6. ハードウェア部の実装: ディレクティブを使った最適化

Vivado HLS はディレクティブを使うことで最適化を行ったり、生成する回路を指定したり出来る。

今回使用したディレクティブをリスト化する。

HLS_PIPELINE	
HLS_ARRAY_PARTITION	前章で説明済み
HLS_INTERFACE	DMA 実装で説明済み

この章では、HLS_PIPLINE と使用していないが HLS_UNROLL についても説明する。

HLS_PIPLINE は指示したブロックの処理をパイプラインかするディレクティブである。このディレクティブをループ処理中の全てのブロックに使用した。このディレクティブを使った場合、使わなかった場合に比べ、Vivado HLS の Performance Estimates によるとスループットが最大約 4 倍向上した。

HLS_UNROLL は指示したループの処理をアンロールして並列化するディレクティブである。今回の実装では HLS_PIPLINE しか最適化に使用していない。本来ならば、HLS_UNROLL も同時に使い並列化した方がスループットの向上が見込まれた。

しかし、高位合成時にメモリが不足し途中で高位合成が停止して回路が生成出来ず HLS_UNROLL を使った最適化が使用できなかった。

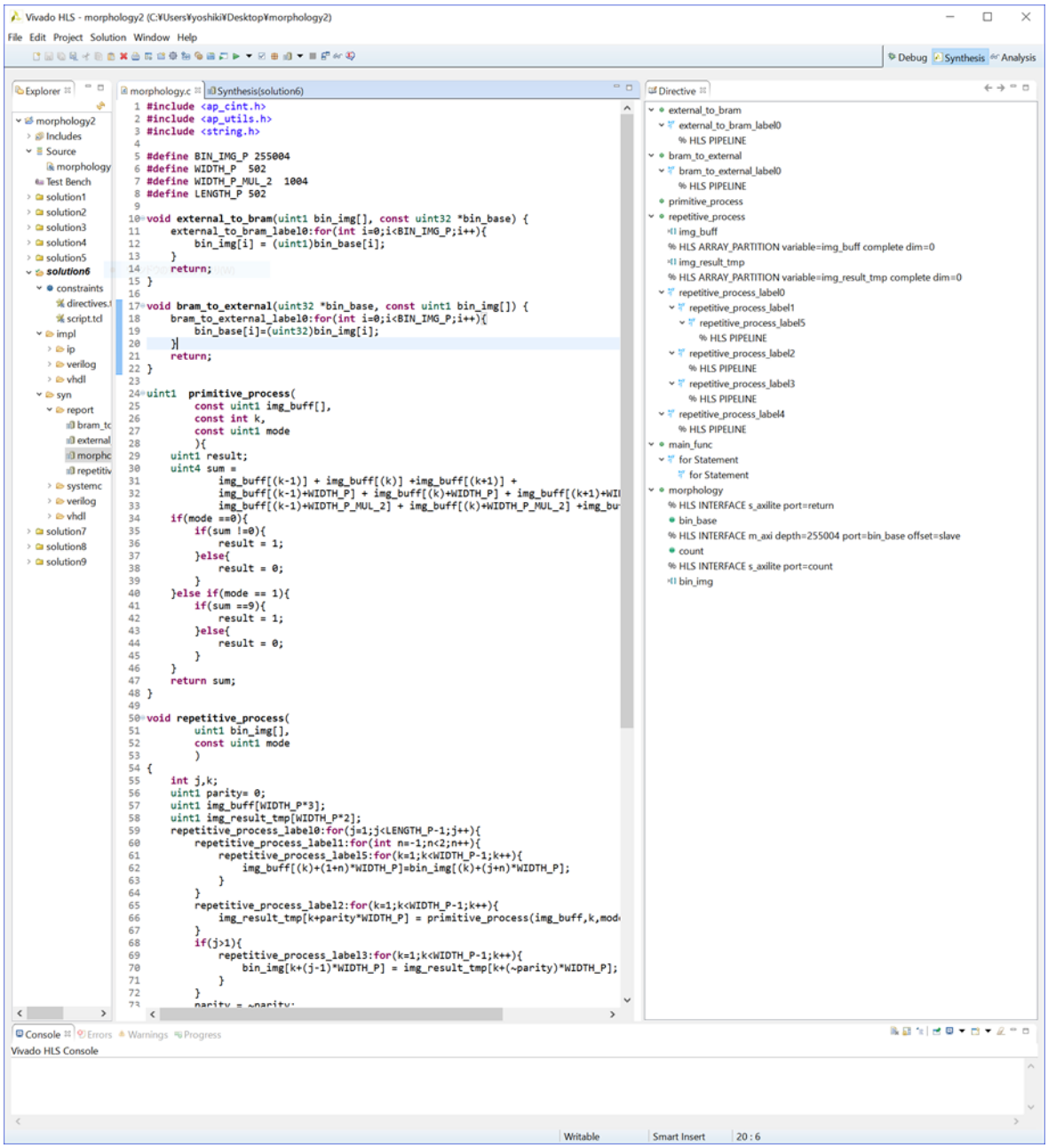

図 7　Vivado HLS　右部分がディレクティブ一覧

3-2-7. ハードウェア部の実装: 動作・振る舞いの確認

モルフォロジー処理回路が適切に動作しているかを確認する為に２つのテストを行った。

１つは、愚直なモルフォロジー処理プログラムと、ハードウェアを意識したモルフォロジー処理のコードを同時に実行して出力が一致するかどうか確認するテストだ。これは、高位合成によって C/C++によって記述できる利点を使用したテストで、高位合成器の正しさを信頼して行ったものである。

もう１つは、モルフォロジー処理の入力データを、本来は 500x500 だが、5x5 と小さいにものにして実際に回路を生成し、それを FPGA 上で動かして結果を Vivado SDK のデバッガで確認するものだ。

この二つのテストで回路が意図した通りの動作をするかを確認した。

3-2-8. 実装された回路図

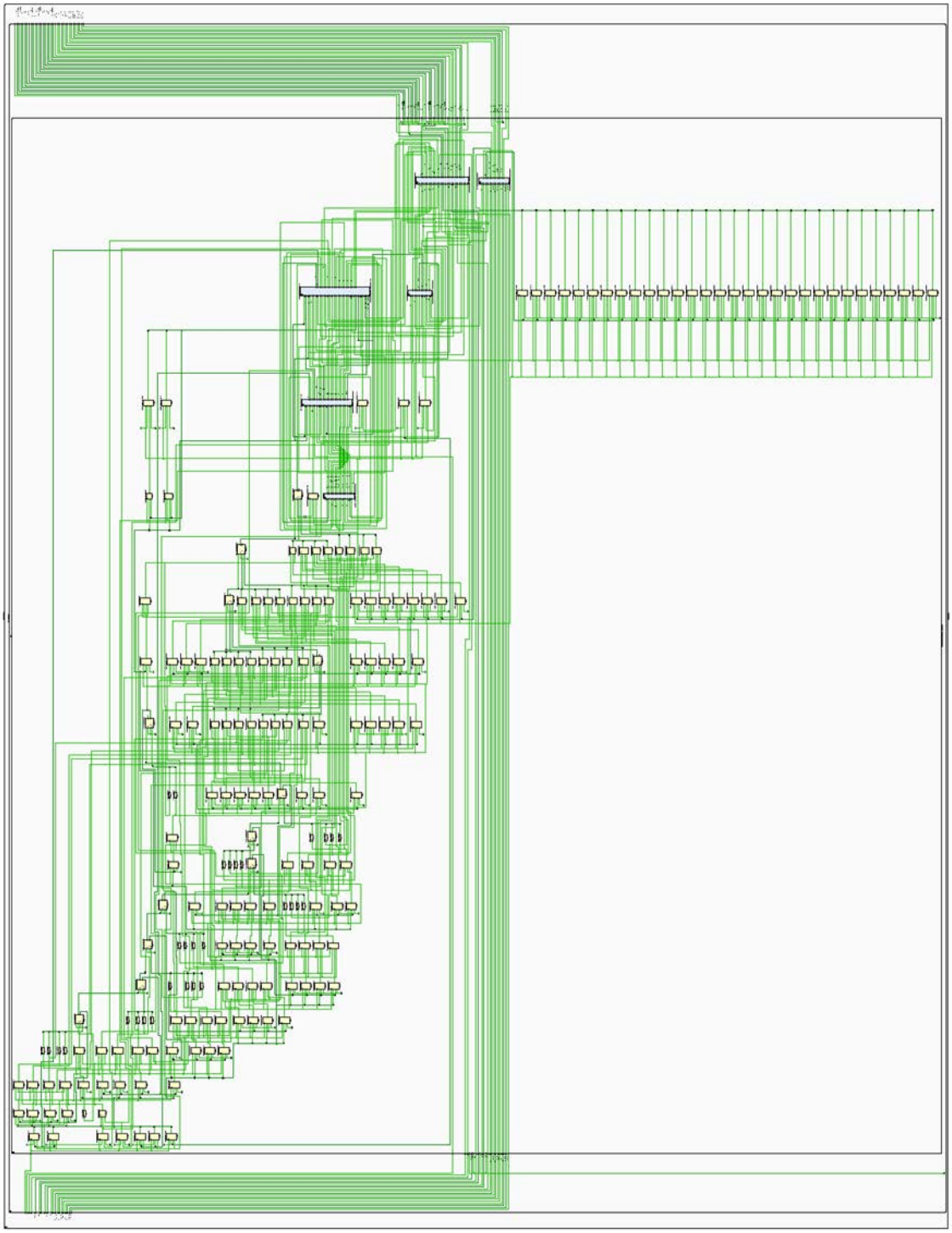

図 8　モルフォロジー処理回路

4. 実験結果と考察

4-1. 実験・計測環境

モルフォロジー処理回路の性能を比較対象として、ZYBO 上の ARM コア、Intel CPU でモルフォロジー処理プログラムを実行し時間を計測した。

4-1-1. FPGA

FPGA 上のモルフォロジー処理回路の処理時間は、処理開始 API を呼び出したとき、そして処理完了フラグが立ったときに、それぞれ XTime_GetTime 関数を呼びベアメタルで処理時間を計測した。また、モルフォロジー処理回路のクロック数は 200MHz である。

4-1-2. ARM

、ZYBO 上の ARM コアの処理時間は、モルフォロジー処理を関数にし、その関数を呼び出したとき、そして返り値が返ったときに、それぞれ XTime_GetTime 関数を呼びベアメタルで、モルフォロジー処理回路と同じように、処理時間を計測した。また、プログラムは-O2 指示句(最適化指示句)をつけてコンパイルした。ARM コアのクロック数は 600MHz で、コアのモデルは Cortex-A9 でスマートフォン等(iPhone4S,PlayStationVita)によく使われるモデルである。

4-1-1. Intel

Intel CPU での処理時間計測は、上の 2 つの場合と異なり、ベアメタルではなく、Windows 上で計測した。そのため、処理時間計測には std::chrono ライブラリの関数を使用した。また、プログラムのコンパイルには、ARM コアと同じように、-O2 オプションを使用した。Intel CPU のモデルは Intel Core i7-4790K でコンシューマ向けのハイエンドなグループに分類される製品である。クロック数は、スペック上は 4.00GHz(最大 4.4Ghz)である。Core i7 は、自動クロックアップ機能である Intel Turbo Boost Technology によって動的にクロックが可変する。

4-2. 実行・計測

計測時のスクリーンショットを示す。

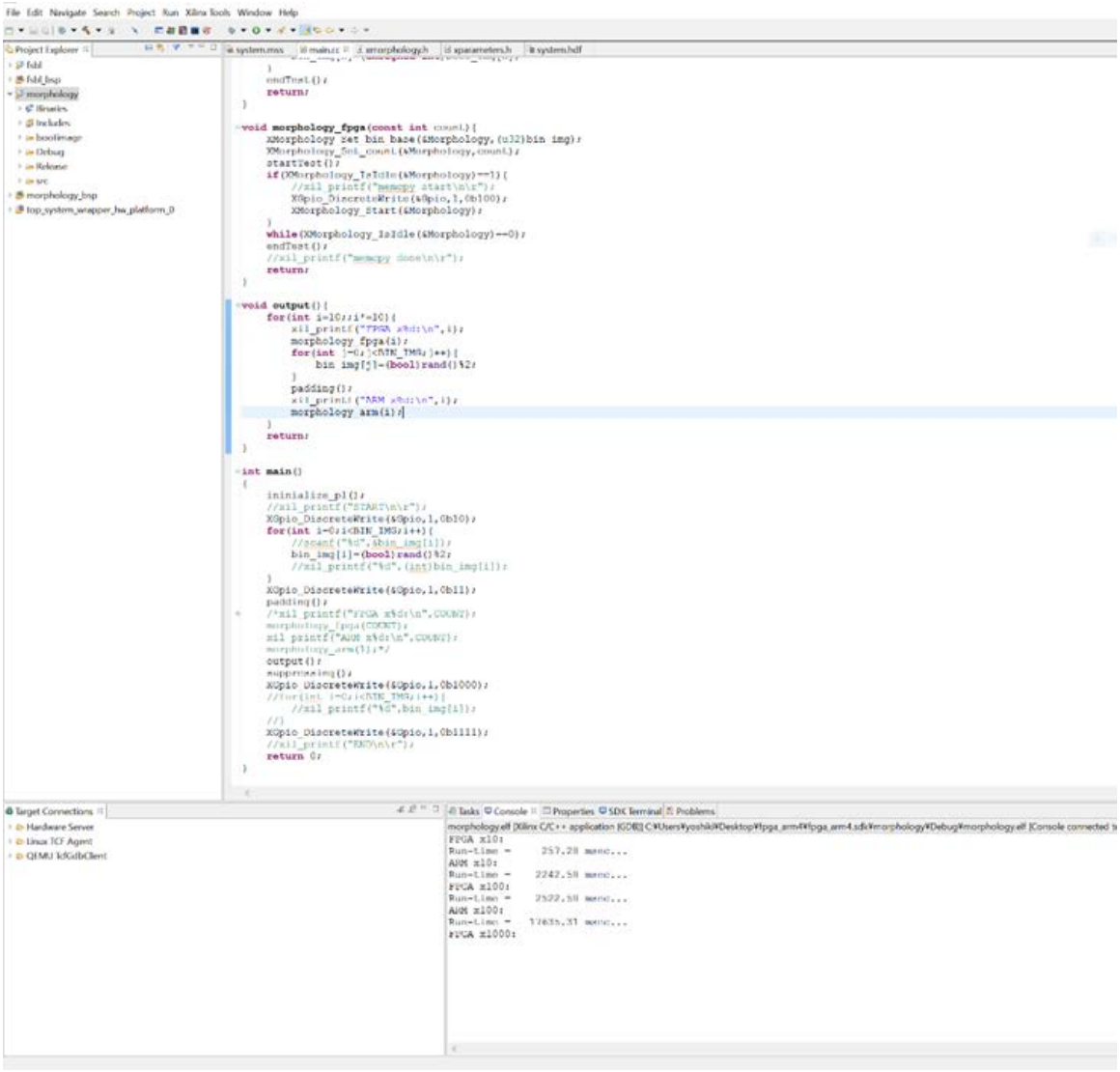

図 9 FPGA と ARM の実行時間計測画面

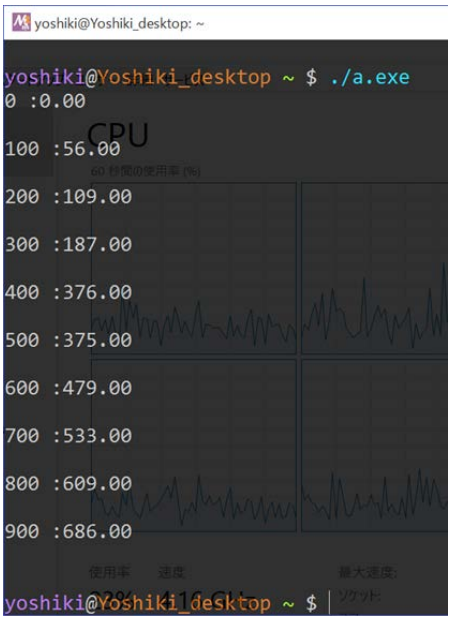

図 10 Intel(Windows MSYS2) の実行時間計測画面

4-3. 結果

処理時間計測の結果をグラフとして示す。測定値は計測データ 100 回分の平均値となっている。

結果・グラフから分かるように、処理速度は Intel、FPGA、ARM の順になった。詳しく比較すると、500x500 配列に対して 1000 回膨張・収縮処理を行った場合、Intel は約 875msecFPGA は約 12,588msec、ARM は約 98,324msec かかった。FPGA を基準に考えると、ARM は FPGA の 7.8 倍遅く、Intel は FPGA の 14 倍速いことが分かる。

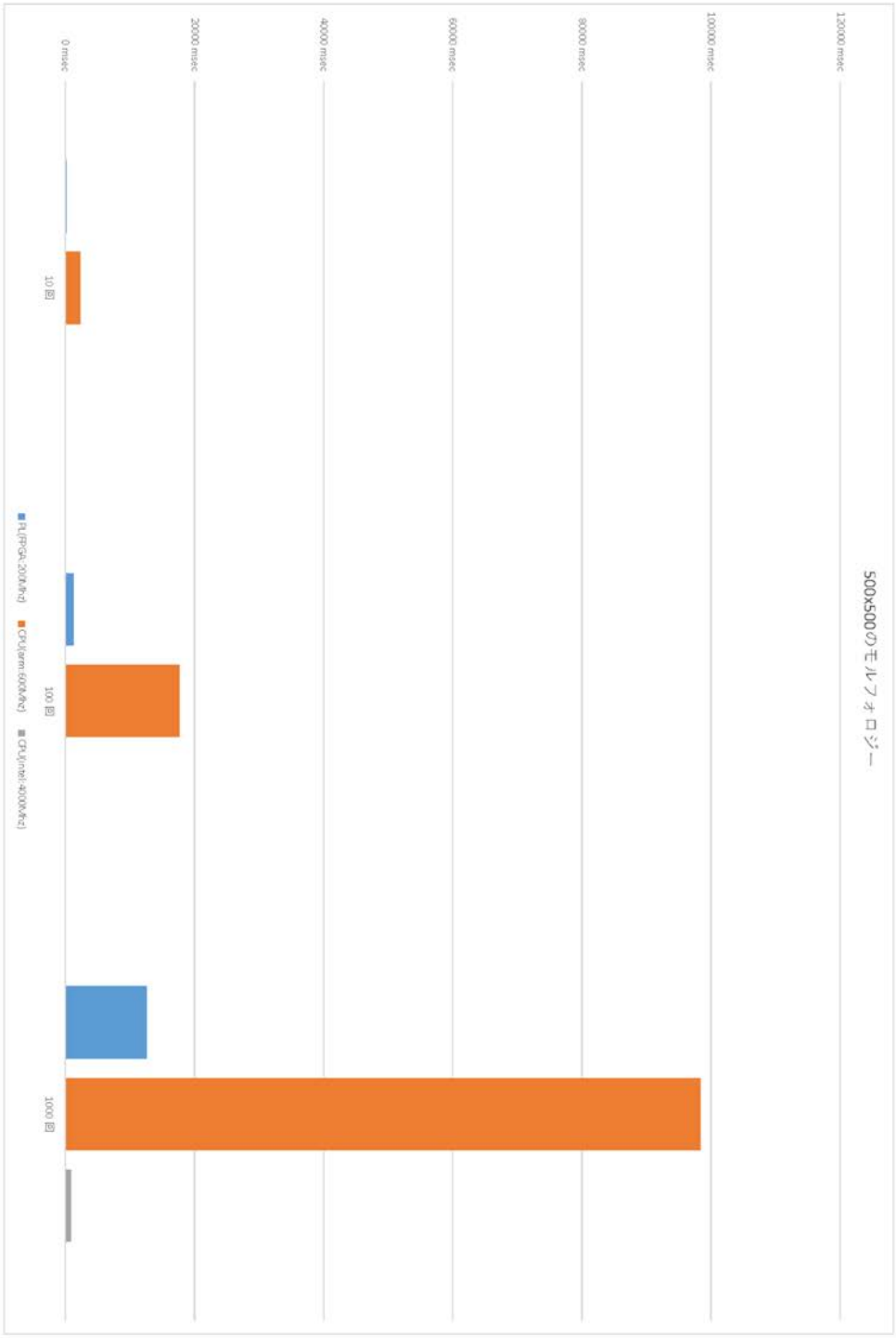

図 11　処理時間の比較グラブ

5. 結論

5-1. 考察

この章では、結果・性能の評価と性能差に対する考察を行う。

まず、計測結果とそれから分かる性能について考察する。実験結果から分かるように、Intel CPU である Intel i7 は、FPGA 上のモルフォロジー処理回路と 10 倍以上の圧倒的な性能差がある。ただ、一方的に Intel i7 が優れているとは断定出来ない。なぜなら、ZYBO は USB 電源、すなわち 5V1A の電力で動くのだ。＊Vivado 統合開発環境の消費電力見積もりツールによると、このシステム全体の最大消費電力は約 1.8W と見積もられていた。Intel i7 と FPGA の消費電力の差を比較するのは難しいが、確証性は低いがワットパフォーマンスでは性能差が少ないもしくは勝っているとも考えることが出来る。

次に、Intel i7 とモルフォロジー処理回路でなぜこのような性能差がでたのかについて考察する。性能差の理由として考えられるのは、第 1 に今回のプログラムがハードウェア化に向かなかった点だ。今回のモルフォロジー処理では、周辺 9 要素にたいする論理演算で、演算処理よりもデータアクセスにかかる時間が多く、結果的に性能がメモリ帯域に依存してしまった。そのため、演算処理のスループットを並列化によって向上しても、メモリ帯域がボトルネックとなり、結果的にシーケンシャル性能が高い Intel i7 に性能差で圧倒されてしまったと考えられる。このことから、モルフォロジー処理の論理演算がより複雑なものであれば、すなわち配列アクセスよりも演算で消費される実行時間の比率が高くなれば、FPGA の方がスループット高くなる可能性も考えることが出来る。

次に、Intel i7 の圧倒的な性能の理由として考えられるのは、最適化の練度の高さと CMOS テクノロジーの世代差が大きい点である。なぜなら、Intel や NVIDIA 等のプロセッサベンダーは、製品の性能を向上させる為に膨大な人的リソース、時間的リソース、そして金銭的リソースをつぎ混んでいる。そのため、コンパイラ、アウトオブオーダ、ハードウェアスケジューラ等の最適化技術によって、愚直なコードでも一定以上の速度が出てしまう。

それ故に、ハードウェア開発初心者が独自回路で CPU 性能を圧倒するアプリケーシ

ョンを開発するのは困難で、それを成し遂げるにはハードウェア、FPGA 開発の知識と経験が必要となる。

5-2. まとめ・将来展望

考察と結果からわかるように、FPGA 上でのアプリケーション開発にはそれなりの知識や経験が求められ、難しい。このことが、今回の卒業研究でのもっともな成果だと思う。今回の実装を行うまで、知識としては、ハードウェア開発とソフトウェア開発では流儀が異なり、ソフトウェア的感覚での予想は一致しづらいと知っていたが、意識下では Raspberry Pi や Arduino と同じように感覚的に捉えていた。しかし、今回実装を行うことで、ハードウェアロジック開発の難しさを理解し、研究動機でも述べたが、FPGA の開発難易度を下げるであろう高位合成技術の向上の必要性をあらためて実感した。

次の研究では、今回の卒業研究での困難を生かし、より多くの人が FPGA の恩恵を授かれるように、高位合成技術の開発を行いたい。

6. 参考文献

[1] Xilinx, “ Xilinx User Guides, Tutotials, Product Guides, Application Notes, WhitePapers”.

[2] S. Churiwala, Designing with Xilinx® FPGAs, Springer, 2017.

[3] 天野秀明, FPGA の原理と構成, オーム社, 2016.

[4] 藤田昌宏, VLSI 設計工学―SoC における設計からハードウェアまで (新・電子システム工学), 数理工学社, 2009.

[5] W. B. K. Vanderbauwhede, High-Performance Computing Using FPGAs, Springer, 2013.

[6] R. a. E. M. a. E. D. N. Louise H Crocket, The Zynq Book Tutorials for Zybo and Zedboard, Strathclyde Academic Media, 2015.

[7] 小林優, FPGA ボードで学ぶ組込みシステム開発入門［Xilinx 編］, 技術評論社, 2013.

[8] FPGA マガジン編集部, FPGA マガジン No.14, CQ 出版, 2016.

[9] FPGA マガジン編集部, FPGA マガジン No.16, CQ 出版, 2017.

[10] FPGA マガジン編集部, FPGA マガジン No.1, CQ 出版, 2013.

[11] FPGA マガジン編集部, FPGA マガジン No.12, CQ 出版, 2016.

[12] FPGA マガジン編集部, FPGA マガジン NO.15, CQ 出版, 2016.

[13] FPGA マガジン編集部, FPGA マガジン NO.6, CQ 出版, 2014.

[14] 坂井修一, 論理回路入門, 培風館, 2003.

[15] 晃. 浅野 , 肇. 延原, “マセマティカルモルフォロジーの基礎と新展開,” *電子情報通信学会誌,* 第 巻 92, 第 10, p. 876, 2009.

[16] 横. 岩田利王, FPGA パソコン ZYBO で作る Linux I/O ミニコンピュータ, CQ 出版, 2016.

[17] 小林優, FPGA プログラミング大全 Xilinx 編, 秀和システム, 2016.

[18] 森岡澄夫, HDL による高性能ディジタル回路設計―ソフトウェア感覚を離れてハードウェアを意識する, CQ 出版, 2002.

[19] 森岡澄夫, LSI/FPGA の回路アーキテクチャ設計法, CQ 出版, 2012.

[20] 坂井修一, コンピュータアーキテクチャ (電子情報通信レクチャーシリーズ), コロナ社, 2004.

[21] 天. 末吉敏則, リコンフィギャラブルシステム, オーム社, 2005.

[22] 伊. 萩. 前野滝授, “CUDA を用いた高速なモルフォロジー演算,” *情報処理学会研究報告,* 第 巻 2008, 第 30, p. 43, 2008.

[23] ス. 天. 宮. デビッドペレリン, C 言語による実践的 FPGA プログラミング, エスアイビーアクセス, 2011.

[24] H. Ando, コンピュータアーキテクチャ技術入門 ~高速化の追求×消費電力の壁, 技術評論社, 2014.

[25] H. Ando, プロセッサを支える技術　　－－果てしなくスピードを追求する世界, 技術評論社, 2011.

[26] 三好健文, “FPGA向けの高位合成言語と処理系の研究動向,” *コンピュータソフトウェア,* 第 巻 30, 第 1, p. 76, 2013.

[27] 若林一敏, “ソフトウェアプログラムからハードウェア記述を合成する高位合成技術－ プロセッサ以外の汎用プログラム実行機構 －,” *電子情報通信学会 基礎・境界ソサイエティ,* 第 巻 6, 第 1, p. 37, 2012.

7. English Abstract

This paper presents a research about the implementation of mathematical morphological operations, such as dilations and erosions, using FPGA (Field Processing Gate Array) on digital images. FPGA has been selected for computation as it is one of the newest computation resources and the understanding of the usage of FPGAs is necessary for our future research purposes. In this research we partially replaced the CPU with an FPGA for the computation of binary morphological operations (dilations and erosions) on images and we measured the computational speed of the process'. We investigated how complicated it is to design the hardware logic on an FPGA device as well as how much digital and arithmetic circuit design knowledge is necessary for a developer to implement his/her software on an FPGA device. Our results proved that FPGA is a very promising computation resource for morphological image processing. We found that even a low specification FPGA device can outperform the ARM processor when computing low level binary morphological operations.

In our future studies we are going to continue our investigation how well FPGAs can be used for other computational purposes, such as its usage for HLS systems.