メモリ帯域の高効率利用に基づくFPGAアクセラレータの設計に関する研究

博士学位論文

メモリ帯域の高効率利用に基づく FPGA アクセラレータの設計に関する研究

武井　康浩

東北大学大学院情報科学研究科
情報基礎科学専攻

目 次

第 1 章　緒言　7

第 2 章　データ転送を考慮したマルチコア MIMD アクセラレータの最適設計　13

2.1　まえがき 13
2.2　MIMD アクセラレータのアーキテクチャモデル 14
2.3　Window 画像処理 19
2.4　Window 画像処理の処理時間見積もり 26
2.4.1　外部メモリからアクセラレータコアへのデータ転送時間 (Phase 1) 26
2.4.2　アクセラレータコアにおける演算処理時間　(Phase 2) 27
2.4.3　アクセラレータコアから外部メモリへのデータ転送時間　(Phase 3) 28
2.4.4　オーバーラップを考慮した処理時間の見積もり　. 29
2.5　評価 35
2.5.1　処理時間の見積もり式の精度の評価 35
2.5.2　最適な設計自由度の評価 36
2.5.3　異なるウィンドウサイズに対する設計自由度の最適化 37
2.5.4　カスタムプロセッサとの比較評価 42
2.6　結言 44

第 3 章　内部メモリの効率的利用を考慮したステンシル計算アクセラレータの

最適設計 **45**
3.1 まえがき 45
3.2 ステンシル計算 46
3.3 ステンシル計算アクセラレータの設計 50
3.3.1 ステンシル計算アクセラレータのアーキテクチャモデル . . . 50
3.3.2 OpenCL を用いた FPGA アクセラレータの設計 54
3.4 ステンシル計算アクセラレータの最適設計法 59
3.4.1 制約条件 59
3.4.2 計算時間見積もり式 60
3.5 評価 63
3.5.1 見積もり式による処理時間最小化の評価 63
3.5.2 設計自由度ごとの消費電力の評価 70
3.6 結言 72

第 4 章 簡潔グラフ表現に基づく最短経路探索アクセラレータ **75**
4.1 まえがき 75
4.2 簡潔グラフ表現に基づくデータ圧縮と FPGA による簡潔グラフ表現の処理 77
4.2.1 簡潔データ構造 77
4.2.2 簡潔グラフ表現 78
4.2.3 FPGA による簡潔グラフ表現の処理 82
4.2.4 簡潔グラフ表現によるデータ圧縮の評価 87
4.3 中間データ記憶量を削減した最短経路問題処理ユニット 89
4.3.1 ダイクストラ法 89
4.3.2 最短経路検索ユニットのアーキテクチャ 93
4.3.3 評価 98

4.4 結言 100

第 5 章 結言 **103**

参考文献 **107**

謝辞 **115**

第1章　緒言

近年，医用画像処理・データマイニング等の高度な画像処理，流体解析・電磁解解析などに代表される数値シミュレーション，道路ネットワーク・ソーシャルネットワークなどに代表される大規模グラフ構造の解析など，ビックデータ/高性能計算が様々なアプリケーションで応用されている．そのため，ビックデータ/高性能計算のアプリケーションを高速かつ低消費電力で処理することができるような計算機システムが強く求められている．

ビッグデータ/高性能計算を高速に処理する計算機システムとして，スーパーコンピュータ「京」[1] に代表されるような，マルチコア CPU を搭載した多数の計算機を並列動作させる PC クラスタが主に使用されている．しかしながら，大規模な PC クラスタの問題点として，計算機システムの動作および冷却にかかる消費電力が大きいことがあげられる．

一方で，グラフィックを処理するためのアクセラレータである GPU を用いて，ビッグデータ/高性能計算の処理を高速化する研究も注目されている．GPU には SIMD 演算器が多数搭載されており，単純な演算を高い並列度で実行することができる．そのため，CUDA などの GPU による高性能計算向けのプログラミング言語が用意されており [2]，GPU を使用したビッグデータ/高性能計算の並列処理のためのプログラミングが可能となっている．しかしながら，GPU による並列処理の問題点として，分岐命令が多く含まれるような複雑な演算を効率よく処理できないこと，GPU 単体の消費電力が CPU と比べても大きいことがあげられる．

本研究ではビッグデータ/高性能計算の高速かつ低消費電力な演算処理を実現でき

る計算機システムとして，FPGA(Field Programable Gate Array) に応用に特化したビッグデータ/高性能計算向けアーキテクチャを実装したカスタムアクセラレータに注目している．FPGA は再構成可能な VLSI であり，プログラム可能な論理回路や DSP ユニット，メモリモジュールが切り替え可能な配線網によって接続されている．そのため，FPGA 上に実装するアクセラレータのアーキテクチャを応用に合わせて自由に再構成することができる．また，近年の集積回路の微細化技術の発展により，FPGA 上にビッグデータ/高性能計算向けの大規模なアーキテクチャを実装して処理を高速化することが可能になっている．さらに，FPGA の消費電力は CPU の約 1/5，GPU の約 1/10 で非常に小さいことも利点としてあげられる．FPGA を採用しているッグデータ/高性能計算向けの計算機システムの例として，Microsoft が運用している検索エンジン向けデータセンタがあげられる [3]．FPGA に検索処理に特化したカスタムアクセラレータを実装することにより，CPU のみによる実装と比較して検索処理のスループットが約 2 倍，レイテンシが約 30%削減されるような性能向上が得られている．また，消費電力のオーバーヘッドは 25W 以下で非常に小さな値に収まっている．

FPGA によるビッグデータ/高性能計算向けカスタムアクセラレータの問題点として，外部メモリ・ストレージとのデータ転送がボトルネックになることが挙げられる．図 1.1 に FPGA アクセラレータのメモリ階層の概要を示す．FPGA アクセラレータのメモリ階層は，FPGA 内部に搭載されているオンチップメモリモジュール，DDR SDRAM など FPGA ボードに搭載される外部メモリ，SSD などの外部ストレージに階層化される．FPGA 内部に搭載されているメモリモジュールは，FPGA 上に実装されている演算部とのデータ転送を最も高速に行うことができる．しかしながら，内部メモリの容量は高性能な FPGA を使用しても最大数十 MB 程度しか確保できないため，ビッグデータ/高性能計算で扱う全データを内部メモリで記憶することは不可能である．DDR SDRAM などの FPGA ボードに搭載されている外部メモリの容量は，数 GB から数十 GB であるため，内部メモリで記憶出来ない容量のデータを

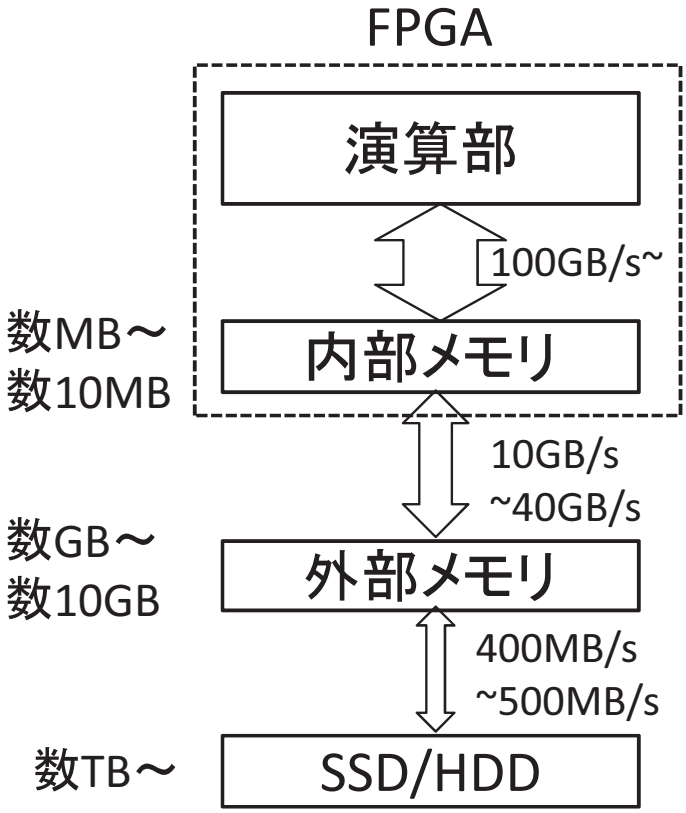


図 1.1: FPGA アクセラレータのメモリ階層

記憶する事ができる．しかしながら，外部メモリのデータ転送帯域は 10GB/s から 40GB/s 程度であり，GPU における外部メモリの転送帯域と比較すると約 1/10 程度である．そのため，FPGA カスタムアクセラレータによるアプリケーションの処理性能において，外部メモリとのデータ転送が性能ボトルネックになりやすい．さらに，ビッグデータのアプリケーションによっては，外部メモリを使用しても記憶できない容量のデータを処理する必要がある．このような膨大なデータを記憶するためには，SSD/HDD などの大容量な外部ストレージを FPGA のボードに接続する必要がある．しかしながら，外部ストレージのデータ転送帯域は 400MB/s から 500MB/s 程度であり，外部メモリとの転送帯域の 1/20 以下である．

本論文では，外部メモリ・ストレージとのデータ転送時間のボトルネックの削減に着目した 3 種類の FPGA カスタムアクセラレータの設計手法を提案する．

第 1 章は緒言であり，本研究の背景，目的および概要について述べている．

2 章では，データ転送を考慮した MIMD アクセラレータの最適設計法を示す．演算依存性がある並列処理を効率的に処理するための FPGA ベース MIMD アクセラレータモデルについて，アクセラレータコアと外部メモリ間のデータ転送が Window 画像処理における性能向上のボトルネックになっている．この問題の解決法として，複数コアを実装してデータ転送と演算処理をオーバーラップさせて全体の処理時間を削減する方法があげられる．オーバーラップを考慮した場合の処理時間について，データ転送時間と演算処理時間は演算並列度・コア数に依存するため，コア数，並列度によってオーバーラップが変化して全体の処理時間に大きく影響する．そのため，データ転送と演算処理のオーバーラップの変化を考慮してコア数，並列度などの設計自由度を最適化する MIMD アクセラレータの最適設計手法を提案する．

3 章では，熱力学，電磁界解析などで用いられるステンシル計算について，内部メモリの効率的利用によって外部メモリアクセス回数を削減したステンシル計算のアクセラレータにおける最適設計を示す．ステンシル計算では，計算領域分の外部メモリアクセスがタイムステップごとに発生するため，データ転送オーバーヘッドが大きくなる．外部メモリアクセスの回数を削減するために，タイムステップごとに中間結果データを記憶するバッファとステンシル計算を処理するための演算器を搭載したステンシル計算アクセラレータのアーキテクチャを提案する内部メモリで処理するタイムステップ数とタイムステップごとのステンシル演算の並列度にはトレードオフの関係があるため，これらの 2 つの設計自由度を最適化するアクセラレータの最適設計手法を提案する

4 章では，最短経路問題を処理する FPGA アクセラレータについて，簡潔データ構造に基づくグラフデータ圧縮による転送時間の削減手法を示す．道路ネットワークなどの大規模グラフを処理する場合について，データ量が外部メモリの容量以上の

グラフを処理する場合，帯域の小さい外部ストレージにデータを記憶する必要がある．そのため，高速なデータ処理とコンパクトな記憶容量を両立して実現できる簡潔データ表現に基づくグラフデータ圧縮により，外部メモリに圧縮したグラフデータを記憶してデータ転送時間を削減する手法を提案する．さらに，最短経路問題の処理における中間結果の記憶量を削減することにより，最短経路検索の処理速度を向上させたアーキテクチャを提案する．

第 5 章は結言である．

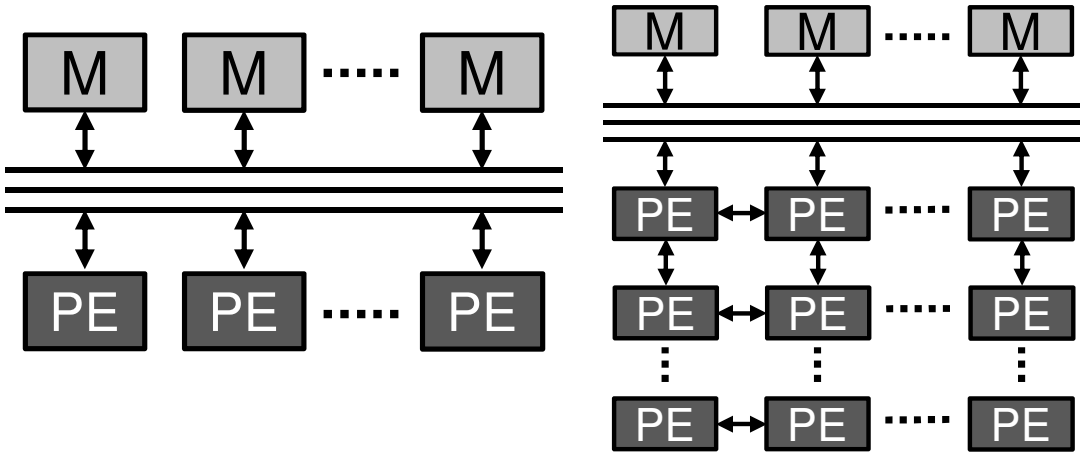
以上に本論文の企図するところを概説した．

第2章　データ転送を考慮したマルチコア MIMD アクセラレータの最適設計

2.1　まえがき

本章では，FPGA ベースの MIMD アクセラレータのアーキテクチャについて，データ転送と演算処理のオーバーラップを考慮した最適設計手法を提案する．MIMD アクセラータを用いた Window 画像処理は，複数のアクセラレータコアを実装することにより，外部メモリからのデータ転送時間と Window 演算処理時間をオーバーラップさせて全体の処理時間を削減できる．オーバーラップを考慮して処理時間を最小化するためには，コア数，コアあたりのアーキテクチャ，演算の並列度に関する設計自由度からデータ転送時間および Window 演算処理時間の見積もりを導出して，オーバーラップをモデル化する必要がある．

本章では，最初に本研究で対象とする FPGA ベースの MIMD アクセラレータのアーキテクチャモデルの概要について説明する．次に，Window 画像処理の並列度，コア数などの設計自由度およびハードウェア制約条件について説明して，最適設計問題の目的関数であるデータ転送と演算処理のオーバーラップを考慮した処理時間の見積もり式を導出する．そのために，シーケンスごとのデータ転送時間，演算時間，コア数に注目してオーバーラップの場合分けを行い，それぞれのケースにおける処理時間の見積もりの算出を行う．



(a) SIMD-1D (b) MIMD-2D

図 2.1: アクセラレータコアのアーキテクチャモデルの分類

2.2 MIMD アクセラレータのアーキテクチャモデル

演算器，メモリモジュールを並列的に搭載しているアクセラレータのアーキテクチャモデルについて，命令実行の形式およびネットワークの形状から，図2.1(a) に示されるような SIMD-1D(Single-Instruction Multiple-Data 1-Dimensional)，または図 2.1(b) に示されるような MIMD-2D(Multiple-Instruction Multiple-Data 2-Dimensional) の 2 種類のアーキテクチャに分類することができる．

SIMD-1D は GPU および MX コア [4] などに採用されているアーキテクチャであり，1 次元的に接続された PE アレイによって同一命令の演算が処理される．回路構成が単純で制御オーバーヘッドが小さいため，シンプルで高並列な処理が求められるアプリケーションの演算処理に適している．MIMD-2D は FE-GA(Flexible Engine/Generic ALU Array) [5] および DRP[6] などに採用されているアーキテクチャであり，2 次元的に接続された PE では，PE ごとに異なる命令の演算を処理することができる．制御オーバーヘッドは SIMD-1D よりも大きくなるが，データパスの構成の自由度は大きくなる．そのため，演算の依存性について考慮する必要があるような複雑な並

列処理に適している．本研究ではアクセラレータの最適化に関して様々なデータパスを実装する必要があるため，演算実装の自由度が大きい MIMD-2D アクセラレータに注目している．

本研究で対象とする FPGA ベース MIMD アクセラレータのアーキテクチャモデルは，先行研究 [7] を元に構成している．図 2.2 にアクセラレータ全体のアーキテクチャモデルの概要を示す．このアーキテクチャモデルについて，FPGA 上には MIMD アクセラレータコアが複数搭載されている．また，MIMD アクセラレータコアの他に，アクセラレータごとの演算開始/終了の制御，外部メモリとのデータ転送を実行するために，プログラムによる複雑な処理を実装するための CPU コアが搭載されている．

図 2.3 に，図 2.2 に実装されている MIMD アクセラレータコアのアーキテクチャモデルの概要を示す．この MIMD アクセラレータコアは， 2 次元状に接続された PE アレイ，メモリモジュール，各メモリモジュールに搭載されているアドレス生成ユニットによって構成されている．PE 間の接続は，画像処理で用いられるようなシンプルなデータフローを処理しやすい構成になっている．メモリモジュールから PE へのデータ転送は，PE アレイの左端に位置する PE へのデータ転送のみに限定されている．また，データフローの方向は 1 方向に限定されており，PE 間の接続に必要なネットワークのリソース量を小さくするような構成となっている．このアーキテクチャモデルについて，MIMD-2D に基づく代表的なアクセラレータである FE-GA[5] を元に構成されている．

図 2.4 に MIMD アクセラレータコアの PE のアーキテクチャモデルを示す．この PE は，16bit 固定小数点演算を処理する ALU および乗算器，レジスタなどによって構成されている．PE で実行できる演算命令について，加算・減算，絶対値差分演算，乗算，累積加算，積和演算をパイプライン処理することができる．また，PE をネットワークによって接続することにより，異なる演算命令の組み合わせによる複雑な処理を，PE アレイ上でパイプライン的に処理することができる．

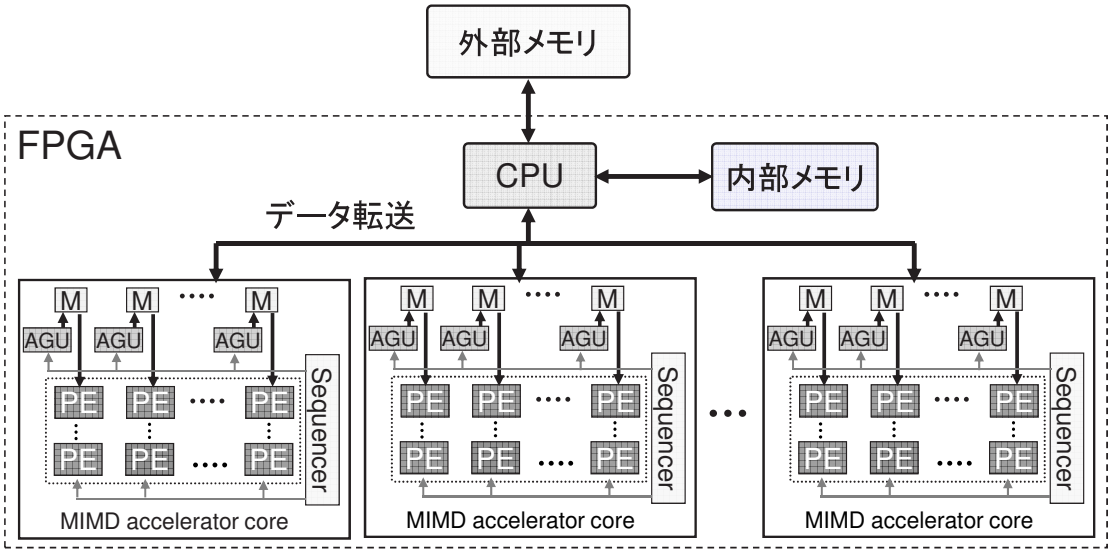


図 2.2: アクセラレータ全体のアーキテクチャモデル

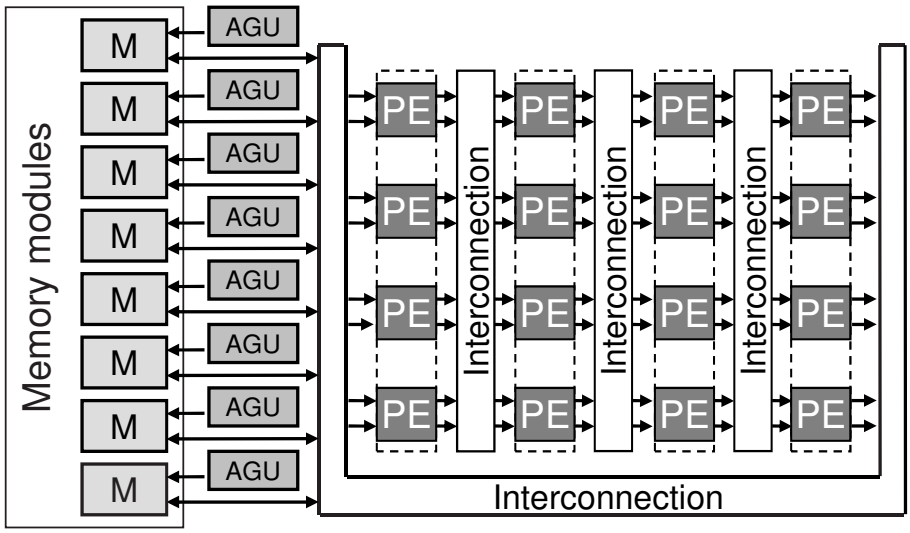


図 2.3: MIMD アクセラレータコア

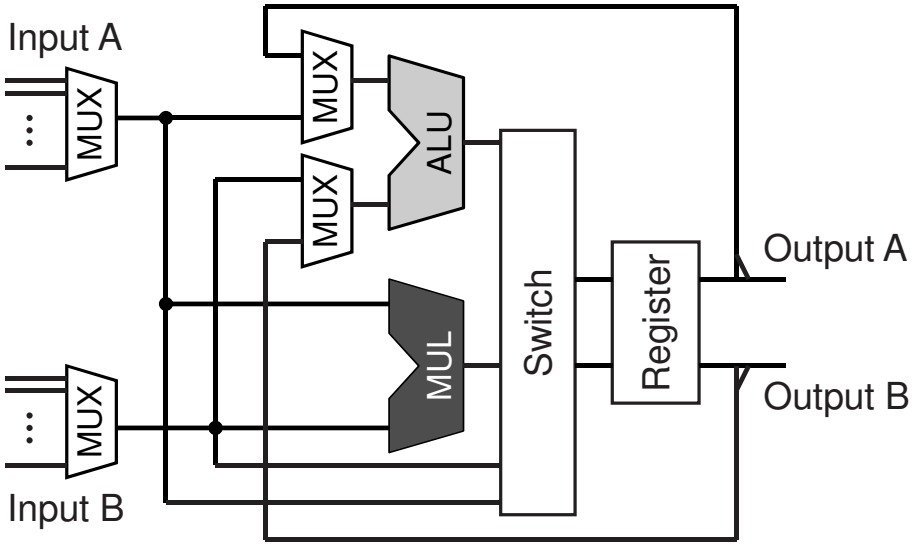
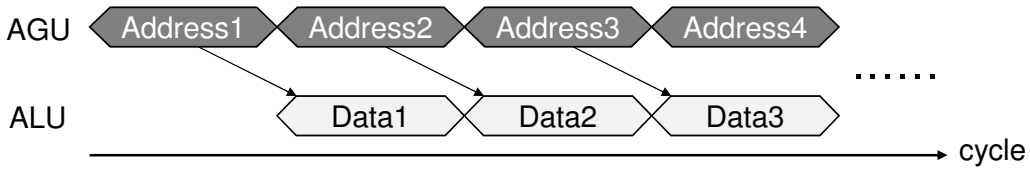


図 2.4: PE のアーキテクチャ

画像処理などの複数のデータを処理する場合について，メモリモジュール内のデータアクセスのためのアドレス計算が必要になる．MIMD アクセラレータコアにおけるメモリモジュールのアドレス計算は，図 2.3 のように各メモリモジュールに搭載されているアドレス生成ユニット (AGU) によって行われる．CPU および GPU においては，図 2.5(a) のようにアドレス計算とデータ処理が同じ ALU によって逐次実行される．一方で，MIMD アクセラレータコアにはメモリモジュールごとに AGU が搭載されているため，図 2.5(b) のようにアドレス計算とデータ処理を並列的に実行することができる．そのため，アドレス計算とデータ処理を同じ ALU で処理する場合比べて高速に処理することができる．

AGU で生成するアドレス関数のパターンについて，複数の先行研究において画像処理におけるアドレス関数がそれぞれ提案されている [8, 9, 10]．AGU は各メモリモジュールごとに搭載されるため，小さいリソース使用量で実装される AGU で生成可能なアドレス関数が好ましいと考えられる．そのため，本研究では，先行研究 [10, 11] で提案されているアドレス関数を採用する．図 2.6 にアドレス関数の概形を

(a) ALU におけるアドレス計算



(b) AGU におけるアドレス計算

図 2.5: アドレス計算のタイムチャート

示す．このアドレス関数は線形関数がベースになっており，カウンタおよび加算器など，シンプルかつ小面積な回路によって構成される AGU を用いて生成することができる．また，このアドレス関数は，画像処理に対するピクセルデータの再利用を考慮した外部メモリとのデータ転送量の最小化手法に用いられている [10]．そのため，先行研究で提案されているデータ転送の最適化手法を，本研究における画像処理の実装に応用にすることが可能である．

MIMD アクセラレータの性能向上のためには，アクセラレータコアのアーキテクチャモデルの他にアクセラレータコアの数も重要になる．図 2.7(a) のようにコア数 1 のアクセラレータでデータ処理を実行する場合，外部メモリとのデータ転送と演算処理が逐次実行される．一方で，図 2.7(b) のようにコア数が 2 のアクセラレータで処理する場合，図 2.7(b) のように 1 つのコアでデータ転送を，もう 1 つのコアで演算処理を同時に実行することにより，全体のリソース量を変えずに処理時間を削減することができる．

このようなデータ転送と演算処理のオーバーラップは先行研究において，GPU[12, 13] および Cell.B.E processor [14] 等のマルチコアアクセラレータにおける処理時間を削減する手法として実装されている．しかしながら，搭載されているコア数，コア

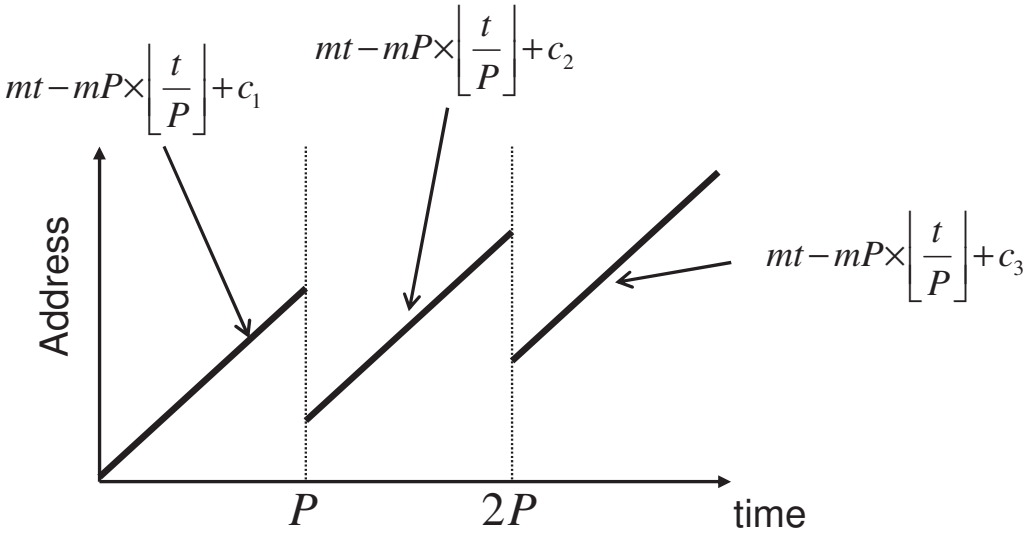
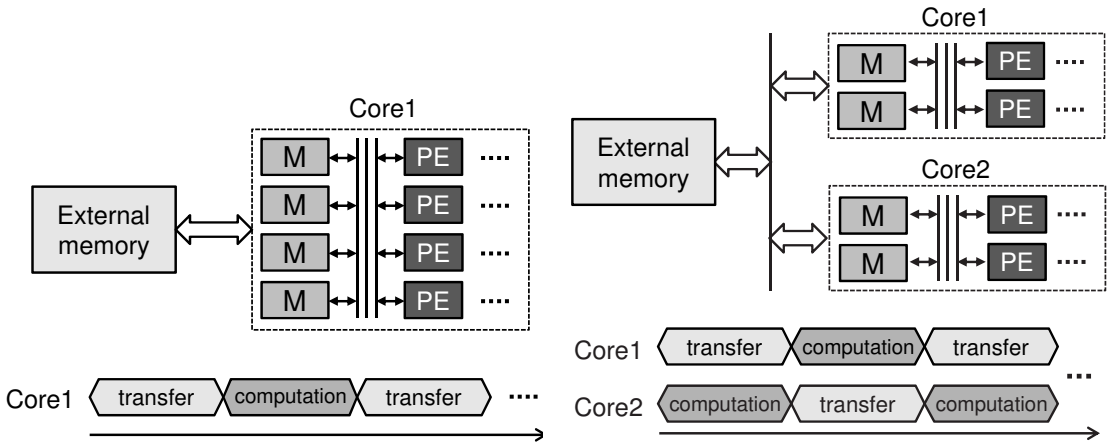


図 2.6: アドレス関数

あたりのリソース量については従来のマルチコアアーキテクチャでは変更することができないため，データ転送時間と演算処理時間の割合を変更するようなオーバーラップを考慮に入れた計算時間の最小化をすることは不可能である．一方で，本研究で使用する FPGA ベースのアーキテクチャは，コア数，コアあたりのリソース量をアプリケーションに合わせて再構成させることができるため，データ転送時間と演算処理時間のオーバーラップを考慮して処理時間を最小化するアーキテクチャを実装することができる．

2.3 Window 画像処理

Window 領域内に含まれるデータに対して演算処理を行う Window 演算は，行列計算，フィルタ演算，ステレオビジョン [8] およびオプティカルフロー抽出 [11] に用いられる SAD マッチング，HOG[15] および SIFT[16] などの画像特徴量抽出など，多くの画像処理で用いられている．Window 画像処理に対するデータ転送量を最小化したメモリアロケーション，および 1 コアに対するデータ転送と Window 演算の



(a) 1 コア　　(b) 2 コア

図 2.7: データ転送と演算処理のタイムチャート

スケジューリングについては，先行研究 [10] で提案されている手法を使用する．

図 2.8 にウィンドウごとの並列処理のモデルを示す．入力画像は並列処理するために $N_{partial}$ 個のパーシャルイメージに分割される．縦方向の画像分割数を N_V，横方向の画像分割数を N_H とすると $N_{partial} = N_H \times N_V$ となり，パーシャルイメージの縦方向の大きさ P_H と横方向の大きさ P_W はそれぞれ式 (2.1),(2.2) で表される．

$$P_H = \left\lceil \frac{I_H + (W_H - 1) \times (N_V - 1))}{N_V} \right\rceil \tag{2.1}$$

$$P_W = \left\lceil \frac{I_W + (W_W - 1) \times (N_H - 1))}{N_H} \right\rceil \tag{2.2}$$

パーシャルイメージ内の Window 演算処理のスケジューリングについて，パーシャルイメージごとに Window 領域分のラスタスキャンが実行される．水平方向のスキャンエリア内の Window 演算処理について，図 2.9 のように，内部のピクセルの演算処理が終了するたびに Window 領域がスキャンエリア内を 1 ピクセルずつ右に移動

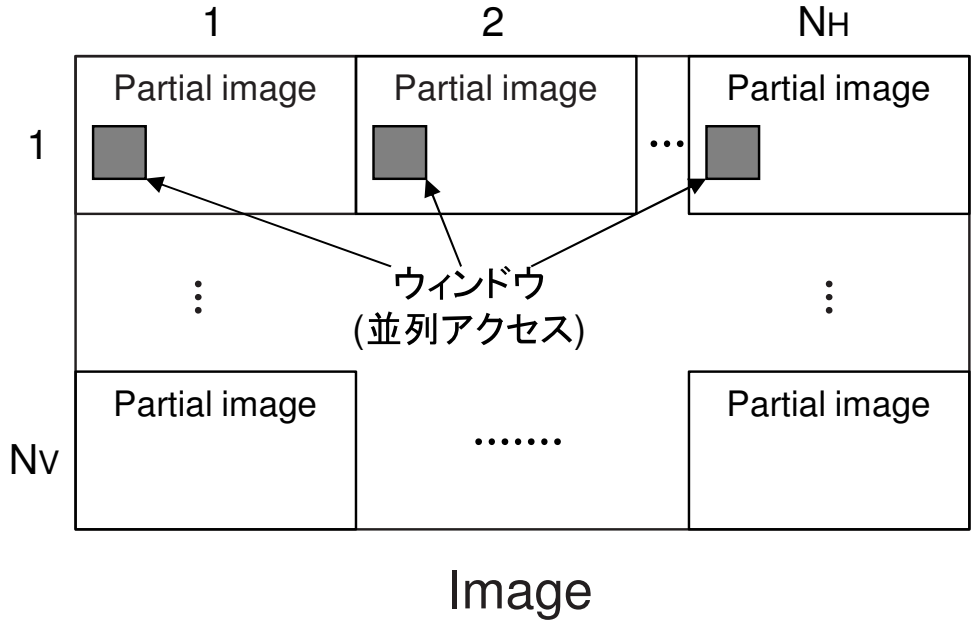


図 2.8: 画像分割と Window 並列度

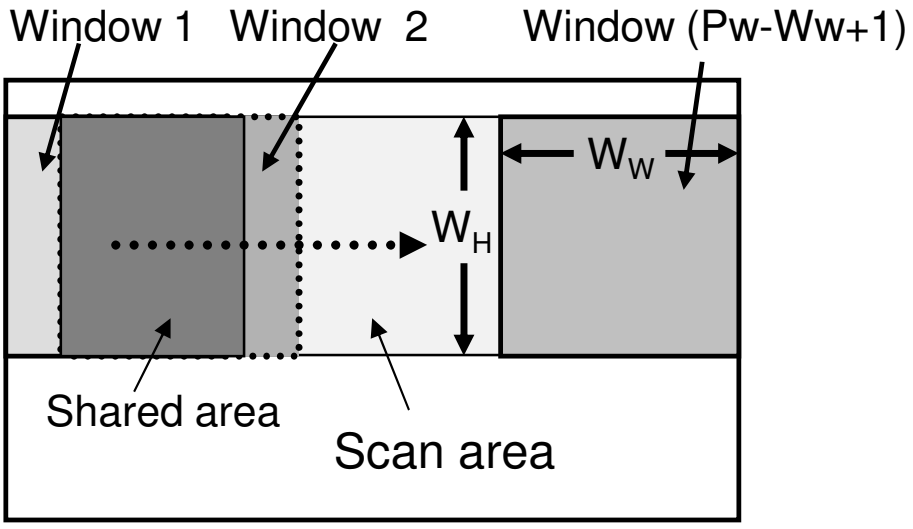


図 2.9: スキャンエリアにおける Window 領域のアクセス

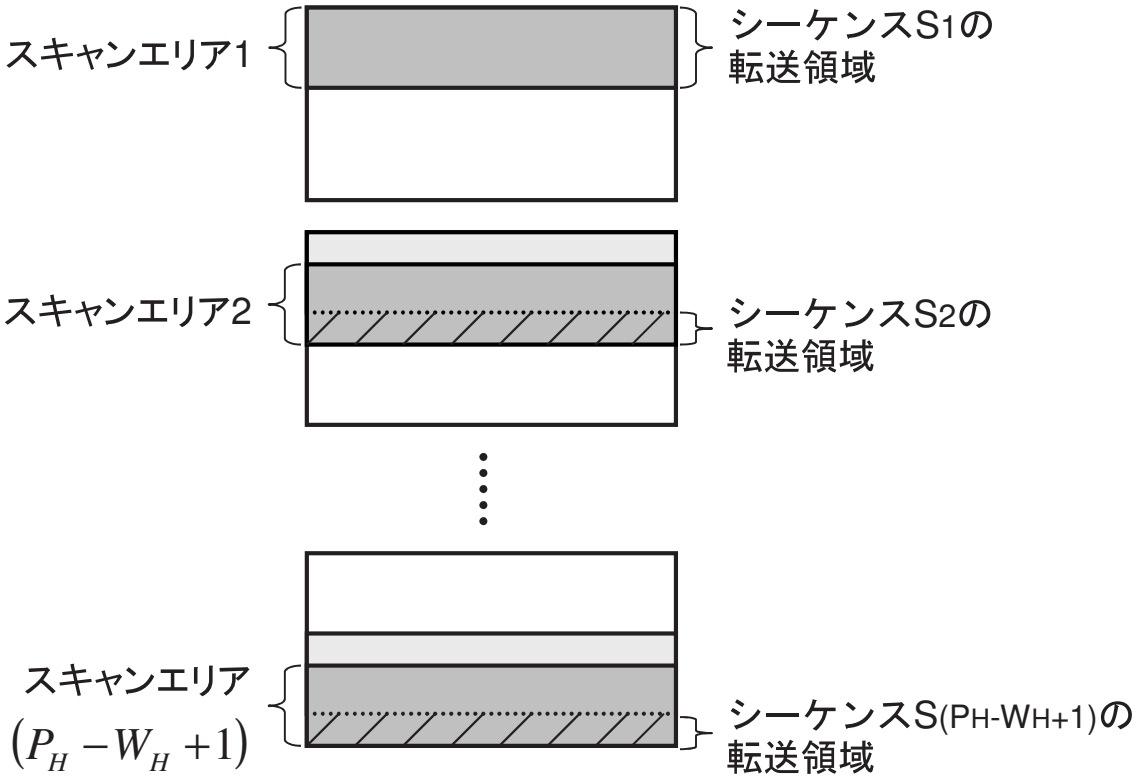


図 2.10: スキャンエリアのシーケンシャルなデータ転送

する．スキャンエリア内の全ての Window 演算が終了したときには，スキャンエリアを下方向に 1 ピクセル移動するために，図 2.10 のようなパーシャルイメージ 1 行分のデータ転送が実行される．このようなデータ転送はシーケンシャルに実行され，それぞれのスキャンエリアについてシーケンス番号が割り当てられている．シーケンス 1 のデータ転送ではスキャンエリア 1 の全ピクセルデータが転送される．シーケンス 2 以降のデータ転送ではスキャンエリアを下方向に移動するために，差分領域のパーシャルイメージ 1 行分のデータ転送が行われる．スキャンエリアの高さは Window の高さ W_H に，幅はパーシャルイメージの幅 P_W に等しい．そのため，パーシャルイメージに含まれるスキャンエリアの数は $(P_H - W_H + 1)$ となる．

Window 画像処理の並列度について，ピクセル並列度とウィンドウ並列度の 2 つ

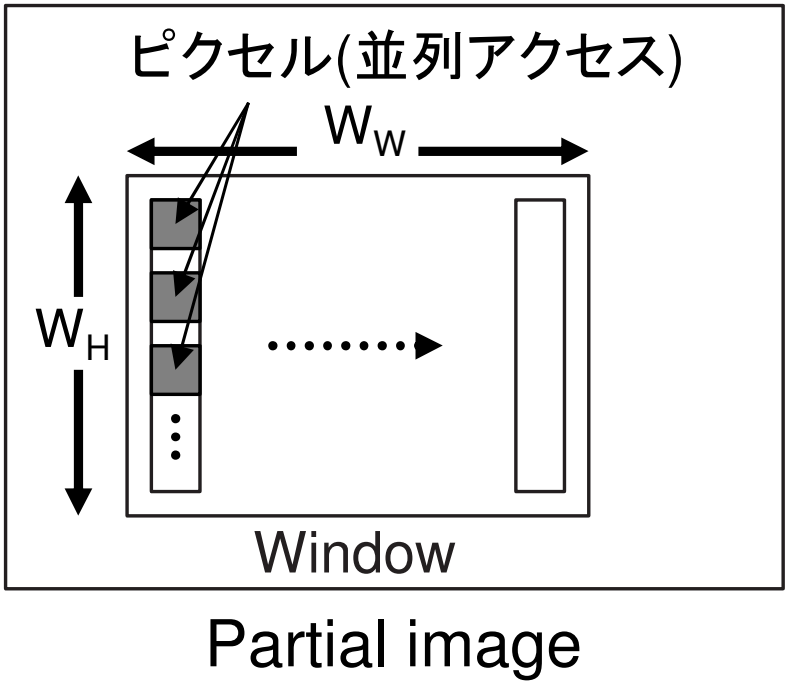


図 2.11: ピクセル並列度

の並列度が定義される.

ピクセル並列度 (P_P) は図 2.11 のように 1 つのウィンドウ内で同時に処理されるピクセルの数で定義される．ピクセル並列度はウィンドウの大きさとの関係式 (2.3) を満たす必要がある．ここで，C_M は自然数である.

$$P_P \times C_M = W_H \tag{2.3}$$

ウィンドウ並列度 (W_P) は図 2.8 のように画像内で同時に処理されるウィンドウの数で定義される．パーシャルイメージあたりの処理される window の数は 1 であるため，画像分割数によってウィンドウ並列度の上限が定義される．($N_{partial} \geq W_P$) さらに，コア数 (N_C) とウィンドウ並列度は関係式 (2.4) を満たす必要がある．ここ

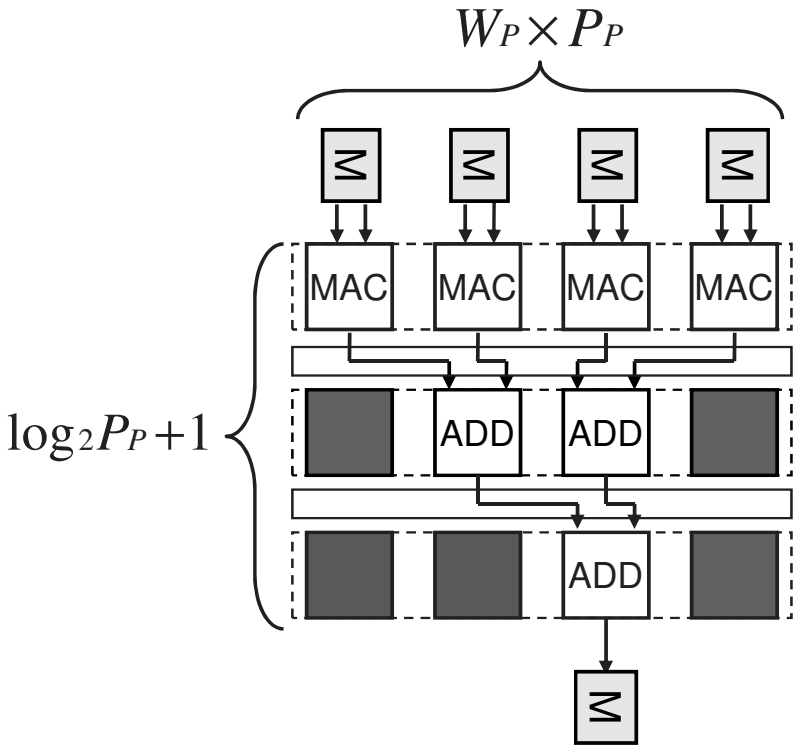


図 2.12: MIMD アクセラレータにおける DFG マッピング

で， N_W はコアあたりのウィンドウ処理数を示す自然数である．

$$W_P = N_C \times N_W \tag{2.4}$$

ピクセル並列度，ウィンドウ並列度の制約条件として，メモリモジュール数，PE 数によるハードウェア量の上限を考慮する必要がある．図 2.3 で説明される MIMD アクセラレータコアのアーキテクチャモデルについて，列ごとに実装される PE 数を n とする．図 2.12 にフィルタ演算のデータフローグラフ（DFG）を MIMD アクセラレータコアにマッピングした例を示す．最初の PE の列のみメモリモジュールに接続されるため，Window 画像処理に用いられるような木構造の DFG のダイレクトマッピングを考慮すると，$n \times (log_2 n + 1)$ 個の PE がマッピングに必要になる．

ピクセル並列度 (P_P) は木構造の DFG の最初の入力数に等しくなるため，ウィンドウ並列度 (W_P) の個数の DFG をマッピングする Window 演算に必要な PE 数 (N_{PE}) は式 (2.5) で導出される．

$$W_P \times P_P \times (log_2 P_P + 1) \leq N_{PE} \tag{2.5}$$

PE 数の制約について，複雑な Window 画像処理を処理する場合は必要な PE 数が大きくなることを考慮に入れる必要がある．

また，必要なメモリモジュール数 (N_{LM}) については Window 演算の DFG の入力数だけ必要になるため，式 (2.6) で導出される．

$$W_P \times P_W \leq N_{LM} \tag{2.6}$$

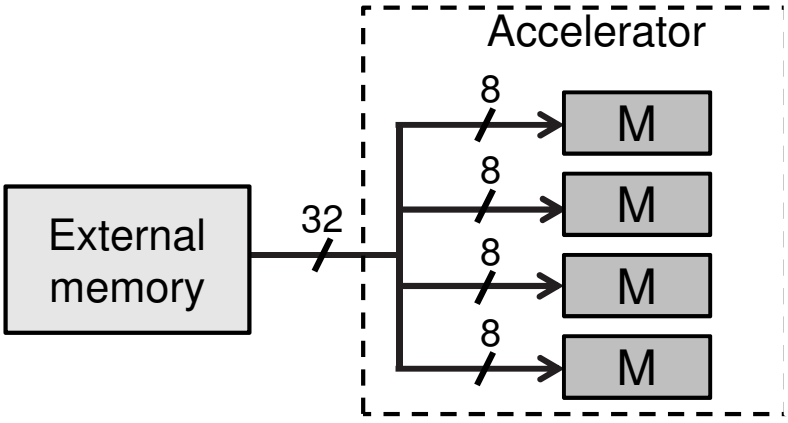


図 2.13: 複数データの並列転送 ($B_{CM} = 8$, $B_B = 32$)

2.4 Window 画像処理の処理時間見積もり

本節では，MIMD アクセラレータのアーキテクチャを最適化するための計算時間見積もり式の導出について説明する．MIMD アクセラレータにおける Window 画像処理について，図 2.10 に示されるようなシーケンスごとの処理は，次のような 3 つのフェーズに分類することができる．

Phase 1: 外部メモリからアクセラレータコアへのデータ転送

Phase 2: アクセラレータコアにおける Window 演算処理

Phase 3: アクセラレータコアから外部メモリへのデータ転送

2.4.1 外部メモリからアクセラレータコアへのデータ転送時間 (Phase 1)

外部メモリからアクセラレータコアへのデータ転送については，データ長とバス幅の関係を考慮する必要がある．例として，データ長が 8bit, バス幅が 32bit の場合，図 2.13 のように一回のデータ転送で 4 データを同時に並列転送することができる．アクセラレータコアのメモリモジュールに入力されるデータ長を B_{MA}，バス幅を B_B

とする．$B_B \geq B_{MA}$ の場合，図 2.13 のように一回のデータ転送で複数データを同時に並列転送することができる．一方で，$B_B < B_{MA}$ の場合，1 つのデータを複数回に分けて転送する必要がある．そのため，データの並列転送数 (N_{MA}) は式 (2.7) で与えられる．

$$N_{MA} = \begin{cases} \left\lfloor \frac{B_B}{B_{MA}} \right\rfloor & \text{when} B_B \geq B_{MA} \\ 1/\left\lceil \frac{B_{MA}}{B_B} \right\rceil & \text{when} B_B < B_{MA} \end{cases} \tag{2.7}$$

シーケンスごとのデータ転送量について，シーケンス S_1 と他のシーケンスでは異なる．シーケンス S_1 では，図 2.10 のように $P_W \times W_H$ ピクセルのデータがアクセラレータコアに転送される．そのため，シーケンス S_1 におけるデータ転送時間 t_{MA1} は式 (2.8) で与えられる．

$$t_{MA1} = \alpha \times \left\lceil \frac{N_W}{N_{MA}} \right\rceil \times P_W \times W_H \tag{2.8}$$

ここで，N_W は式 (2.4) で定義されるコアあたりのウィンドウ処理数，α は外部メモリからアクセラレータコアへの平均データ転送時間である．

シーケンス $S_2 \sim S_{(P_H - W_H + 1)}$ では，図 2.10 のように P_W ピクセルのデータがアクセラレータコアに転送される．そのため，シーケンスあたりのデータ転送時間 t_{MA2} は式 (2.9) で与えられる．

$$t_{MA2} = \alpha \times \left\lceil \frac{N_W}{N_{MA}} \right\rceil \times P_W. \tag{2.9}$$

2.4.2 アクセラレータコアにおける演算処理時間 (Phase 2)

アクセラレータコアにおけるシーケンスあたりの演算処理時間 (t_{comp}) について，MIMD アクセラレータコアのデータパスはパイプライン化されており，パイプライ

ンが満たされた後に計算結果が毎サイクルごとに出力される．図 2.9 より，1 つのスキャンエリアに含まれる Window 領域数は $(P_W - W_W + 1)$ である．また，図 2.11 より，Window 領域の処理では P_P ピクセルの並列処理が行われるため，シーケンスあたりの計算時間 t_{comp} は式 (2.10) で与えられる．

$$t_{comp} = \frac{1}{f_A} \times \frac{W_H \times W_W}{P_P} \times (P_W - W_W + 1) + t_{pipe} \tag{2.10}$$

ここで，f_A はアクセラレータの動作周波数，t_{pipe} は MIMD アクセラレータのパイプラインレイテンシである．アドレス計算時間については，図 2.5(b) のように AGU を用いてアドレス計算を処理しているため，PE における Window 演算処理時間とオーバーラップされている．そのため，アクセラレータコアにおける演算処理時間の見積もりについては Window 処理時間のみを考慮すれば良い．

2.4.3　アクセラレータコアから外部メモリへのデータ転送時間　(Phase 3)

アクセラレータコアから外部メモリへのデータ転送について，外部メモリに入力されるデータ長を B_{AM} とする．データ転送数 (N_{AM}) は章の Phase 1 と同様の方法で求められる．

$$N_{AM} = \begin{cases} \left\lfloor \frac{B_B}{B_{AM}} \right\rfloor & \text{when} B_B \geq B_{AM} \\ 1/\left\lceil \frac{B_{AM}}{B_B} \right\rceil & \text{when} B_B < B_{AM} \end{cases} \tag{2.11}$$

データ転送量について，1 つのスキャンエリアに含まれる Window 領域数は $(P_W - W_W + 1)$ である．それぞれの Window 演算の結果のデータが外部メモリに転送されるため，シーケンスあたりのデータ転送時間 t_{AM} は式 (2.12) で与えられる．

$$t_{AM} = \beta \times \left\lceil \frac{N_W}{N_{AM}} \right\rceil \times (P_W - W_W + 1) \tag{2.12}$$

ここで，β はアクセラレータコアから外部メモリへの平均データ転送時間である．

2.4.4 オーバーラップを考慮した処理時間の見積もり

Window 演算処理は W_P 個のパーシャルイメージで同時に実行されるため，全体のタイムチャートは図 2.14 で表される．

t_{init} はシーケンス S_1 における外部メモリからアクセラレータコアへのデータ転送時間とアクセラレータコアにおける演算処理時間を含んでいる．MIMD アクセラレータコアと外部メモリ間は図 2.2 に示されるように共有バスで接続されている．そのため，データ転送は複数のアクセラレータのコアに対して同時に実行することができない．t_{init} 内ではデータ転送と演算処理のオーバーラップは発生しないため，t_{init} は式 (2.13) のように与えられる．

$$t_{init} = t_{MA1} \times N_C + t_{comp} \tag{2.13}$$

t_{mid} では， t_{trans} と t_{comp} が図 2.14 のように繰り返される． ここで，t_{trans} は式 (2.14) で定義される．

$$t_{trans} = t_{AM} + t_{MA2} + t_{ctrl} \tag{2.14}$$

ここで， t_{ctrl} はアクセラレータの開始/終了の制御にかかる時間である．

t_{mid} の導出について，データ転送と Window 演算のオーバーラップは図 2.15 のような 2 通りに分類できる．

Case A1: 図 2.15(a) のようにコア 1 における t_{comp} がコア 2〜Nc における t_{trans} に完全にオーバーラップされる場合 ($t_{comp} < (N_C - 1) \times t_{trans}$) について，2.3 章で説明したように，Window 画像処理に含まれるシーケンス数は $P_H - W_H + 1$ である．そのうち，t_{mid} では $(N_C \times t_{trans})$ のタイムピリオドが $(P_H - W_H)$ 回繰り返される．

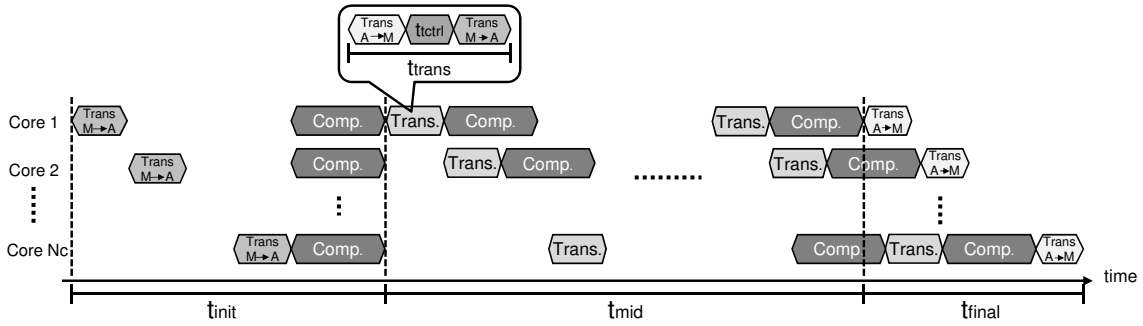


図 2.14: タイムチャート

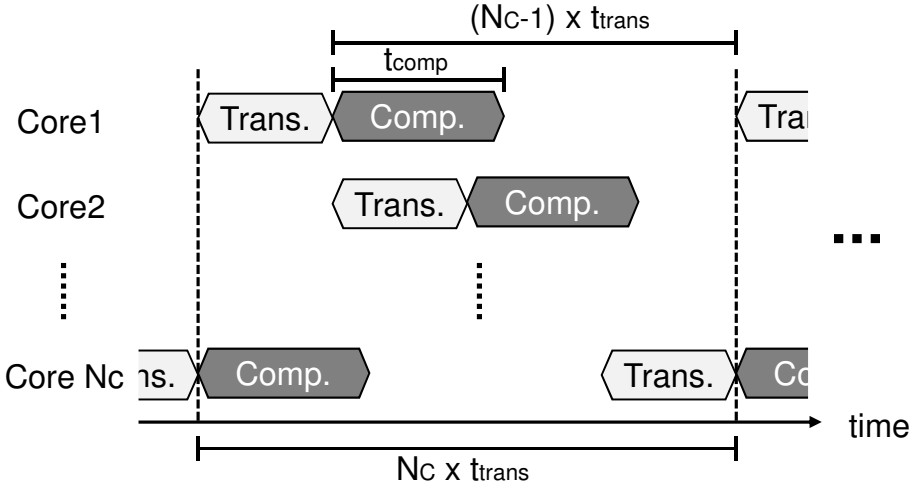
Case A2: 図 2.15(b) のようにコア 1 における t_{comp} がコア 2〜Nc における t_{trans} に完全にオーバーラップされない場合 $(t_{comp} \geq (N_C - 1) \times t_{trans})$ について，Case A1 と同様に t_{mid} では $(t_{comp} + t_{trans})$ のタイムピリオドが $(P_H - W_H)$ 回繰り返される．

以上のことから，t_{mid} は式 (2.15) のように与えられる．

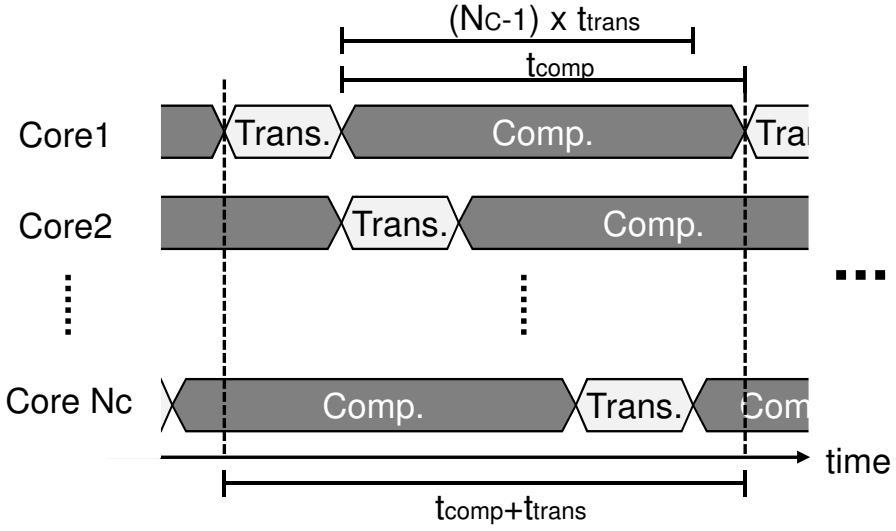
$$t_{mid} = \begin{cases} N_C \times t_{trans} \times (P_H - W_H) \\ \text{when } t_{comp} < (N_C - 1) \times t_{trans} \\ \qquad\qquad \text{(Case A1)} \\ \\ (t_{trans} + t_{comp}) \times (P_H - W_H) \\ \text{when } t_{comp} \geq (N_C - 1) \times t_{trans} \\ \qquad\qquad \text{(Case A2)} \end{cases} \tag{2.15}$$

t_{final} はシーケンス $S_{(P_H - W_H + 1)}$ におけるアクセラレータコアから外部メモリへのデータ転送時間を含んでいる．

t_{final} における t_{AM} と Window 演算のオーバーラップは図 2.16 のように次の 3 通りに分類できる．



(a) Case A1 $(t_{comp} < (N_C - 1) \times t_{trans})$



(b) Case A2 $(t_{comp} \geq (N_C - 1) \times t_{trans})$

図 2.15: t_{mid} におけるデータ転送と演算処理のオーバーラップ

Case B1:　$t_{comp} \geq (N_C - 1) \times t_{trans}$

Case B2:　$(N_C - 1) \times t_{AM} \leq t_{comp} < (N_C - 1) \times t_{trans}$

Case B3:　$t_{comp} < (N_C - 1) \times t_{AM}$

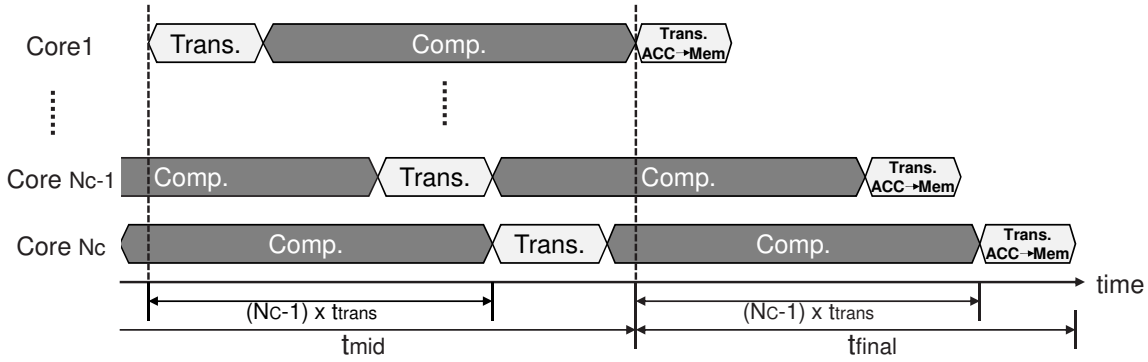
Case B1 では t_{AM} は t_{trans} より小さいため，図 2.16(a) のように 1 コアの t_{comp} が他の全コアの t_{AM} をオーバーラップしている．Case B2 と Case B3 では，t_{AM} と t_{comp} の大小関係について，図 2.16(a) と図 2.16(c) のように t_{mid} と同様のオーバーラップを考慮する必要がある

そのため，t_{final} は式 (2.16) のように与えられる．

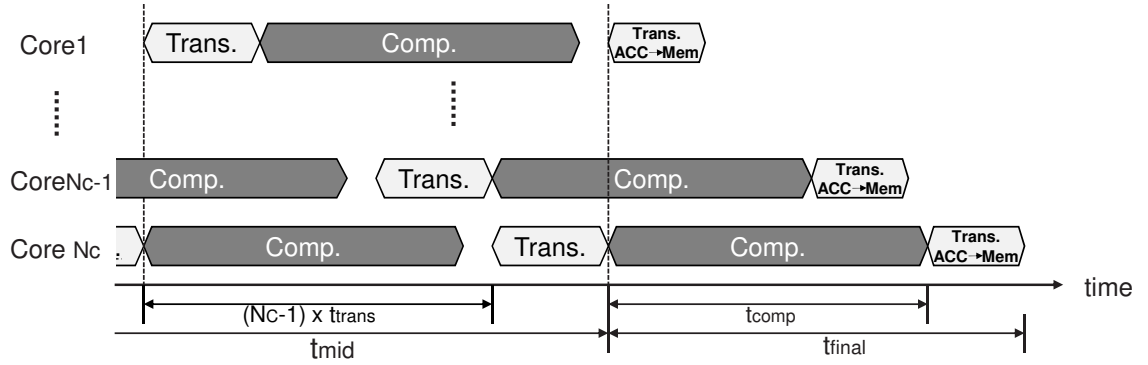
$$t_{final} = \begin{cases} (N_C - 1) \times t_{trans} + t_{AM} \\ \text{when } t_{comp} \geq (N_C - 1) \times t_{trans} \\ \qquad\qquad\qquad\qquad \text{(Case B1)} \\ \\ t_{AM} + t_{comp} \\ \text{when } (N_C - 1) \times t_{AM} \leq t_{comp} \\ \qquad < (N_C - 1) \times t_{trans} \\ \qquad\qquad\qquad\qquad \text{(Case B2)} \\ \\ N_C \times t_{AM} \\ \text{when } t_{comp} < (N_C - 1) \times t_{AM} \\ \qquad\qquad\qquad\qquad \text{(Case B3)} \end{cases} \tag{2.16}$$

式 (2.13),(2.15),(2.16) より，パーシャルイメージ W_P 個あたりの処理時間は式 (2.17) のように与えられる．

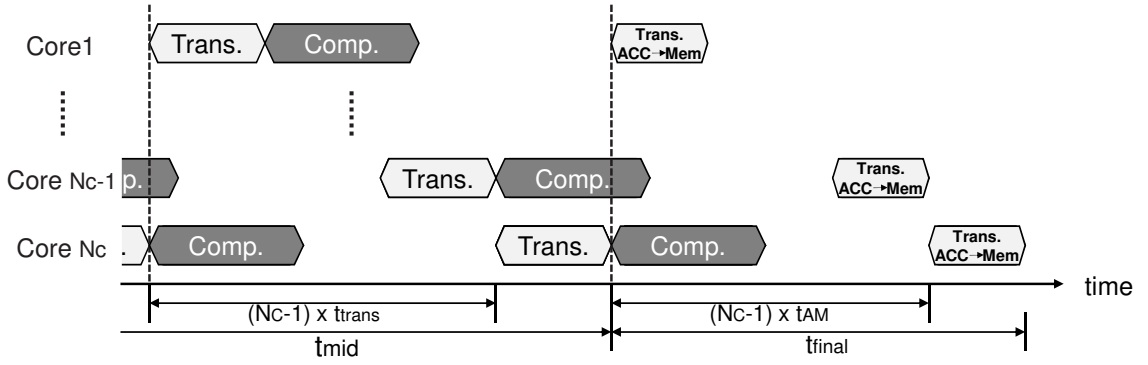
$$t_{partial} = t_{init} + t_{mid} + t_{final} \tag{2.17}$$



(a) Case B1: $t_{comp} \geq (N_C - 1) \times t_{trans}$



(b) Case B2: $(N_C - 1) \times t_{AC} \leq t_{comp} < (N_C - 1) \times t_{trans}$



(c) Case B3: $t_{comp} \leq (N_C - 1) \times t_{AC}$

図 2.16: t_{final} のオーバーラップ

画像処理全体の処理時間 T_{image} は，$N_{partial}$ 個のパーシャルイメージを W_P 並列で処理するため，式 (2.18) のように与えられる．

$$T_{image} = t_{partial} \times \left\lceil \frac{N_{partial}}{W_P} \right\rceil \tag{2.18}$$

式 (2.13), (2.15), (2.16) および (2.17) より，全体の処理時間 T_{image} は設計自由度 W_P, P_P, N_C, N_W, N_H，N_V の値から計算することができる．そのため，Window 画像処理の処理時間の最小化は設計自由度の全探索を実行することにより実現することができる．

処理時間の見積もり式の精度，および設計自由度の最適化の評価結果については 2.5 章で説明する．

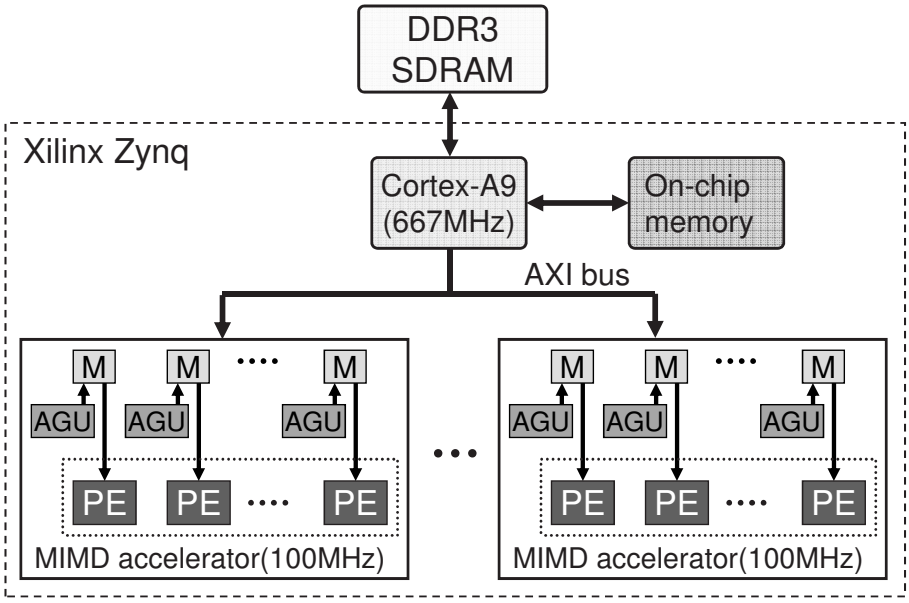


図 2.17: 実装したアーキテクチャ

2.5 評価

2.5.1 処理時間の見積もり式の精度の評価

本研究では Xilinx Zynq-7000 EPP ZC702 board [17] を用いて MIMD アクセラレータの実装評価を行った．Xilinx Zynq-7000 は高性能な組み込み向けデュアルコア CPU である Cortex-A9 を搭載した FPGA である．また，この評価ボードには外部メモリとして DDR3 SDRAM(1GB) が搭載されている．処理時間の見積もり値と実測値の比較評価のために，図 2.17 に示されるようなアーキテクチャを Xilinx PlanAhead 14.2 を用いて実装した．CPU コアの動作周波数は 667MHz, MIMD アクセラレータコアおよび AXI バスの動作周波数は 100MHz である．

t_{trans} を求めるために， 式 (2.8)，(2.12) ， (2.14) の α, β , t_{ctrl} をアクセラレータコアと外部メモリとのデータ転送時間から計測した．その結果，アクセラレータコアの周波数が 100MHz の場合について，α, β，t_{ctrl} の値はそれぞれ 186.06 (ns),

表 2.1: 処理時間の見積もり値と計測値の違い

ウィンドウサイズ	処理時間 見積もり値 (ms)	処理時間 計測値 (ms)	誤差率 (%)
12×12	46.93	46.51	0.51
16×16	60.38	60.07	0.91
18×18	70.50	70.51	0.02
24×24	115.89	115.87	0.01

213.02 (ns)，430.00 (ns) となった．2.4 章で求めた処理時間の見積もり式の精度を測るため，FPGA に実装した MIMD アクセラレータにおける画像フィルタ処理の処理時間を AXI timer IP を用いて計測した．見積もり値と実測値との誤差率を式 (2.19) で評価した結果，表 2.1 のように誤差率が 1%以下に抑えられていることが確認された．そのため，処理時間の見積もり式により最適な設計自由度を導出して，高性能な MIMD アクセラレータを設計するための指標を得るためには十分な精度が得られていることが確認された．

$$\mathrm{Error} = \frac{|\mathrm{Measured} - \mathrm{Estimated}|}{\mathrm{Measured}} \times 100(\%) \tag{2.19}$$

2.5.2 最適な設計自由度の評価

本節では 2.4 章で導出した処理時間の見積もり式を用いて，Window 画像処理における処理時間を最小化するための最適な設計自由度の導出を行う．画像の大きさを 640×480，フィルタの大きさを 16×16，メモリモジュール数の制約 $(W_P \times P_P)$ を 16 とした場合，設計自由度に対する処理時間は表 2.2 のようになる．処理時間が最小値になる場合の設計自由度を全探索で探索した結果，$W_P = 16$, $P_P = 1$, $N_C = 4$, $N_W = 4$, $N_H = 8$, $N_V = 2$ の場合の処理時間が最小値をとることが確認された．

表 2.2: 設計自由度に対する処理時間の最適化

設計自由度					オーバーラップ			
ウィンドウ並列度 W_P	ピクセル並列度 P_P	コア数 N_C	縦方向の画像分割 N_V	横方向の画像分割 N_H	t_{comp} (ms)	t_{trans} (ms)	t_{mid} の case	全体の計算時間 (ms)
1	4	1	1	1	0.4001	0.2531	Case A2	305.80
4	2	1	2	2	0.4007	0.1868	Case A2	137.98
4	2	4	4	1	0.2010	0.0663	Case A2	127.02
4	2	2	4	1	0.2010	0.0663	Case A2	125.61
16	1	1	8	2	0.2024	0.1981	Case A2	94.70
16	1	2	8	2	0.2023	0.0993	Case A2	71.83
16	1	4	4	4	0.4020	0.0955	Case A2	61.02
16	1	4	8	2	0.2023	0.0499	Case A2	60.38*
16	1	8	8	2	0.4020	0.0663	Case A1	67.47
16	1	16	4	4	0.4020	0.0663	Case A1	134.14

*処理時間最小値.

設計自由度の探索範囲は関係式 (2.3),(2.4)，およびハードウェア制約式 (2.5),(2.6) により限定されるため，全探索の計算時間は汎用 PC においても短時間で処理することができる．処理時間が最小値になる場合のデータ転送と Window 演算処理のオーバーラップについて，図 2.15 で定義した定常状態におけるオーバーラップは，$t_{comp} > (N_C - 1) \times t_{trans}$ が成り立つことから，図 2.15(b) で示される Case A2 になる．

2.5.3 異なるウィンドウサイズに対する設計自由度の最適化

本節では 2.4 章で導出した処理時間の見積もり式を用いて，ウィンドウサイズを変化させた場合についての最適な設計自由度の変化について考察する．

表 2.3 に異なるフィルタサイズに対する最適化された設計自由度の値の表を示す.

表 2.3: 異なるフィルタサイズに対する自由度の最適化 (メモリモジュール数 16)

ウィンドウサイズ $W_H = W_W$	設計自由度					全体の処理時間 (ms)	オーバーラップ
	ウィンドウ並列度 W_P	ピクセル並列度 P_P	コア数 N_C	画像分割数			
				横 N_H	縦 N_V		
8	16	1	2	4	4	46.24	Case A1
9	16	1	2	4	4	45.88	Case A1
10	16	1	2	4	4	46.20	Case A1
12	16	1	4	4	4	46.93	Case A1
13	16	1	4	4	4	46.62	Case A1
15	16	1	4	8	2	54.50	Case A2
16	16	1	4	8	2	60.38	Case A2
17	16	1	4	8	2	65.53	Case A2
18	16	1	8	8	2	70.50	Case A2
20	16	1	8	16	1	83.97	Case A2
24	16	1	8	16	1	115.89	Case A2

この結果から，最適化された設計自由度の値はフィルタサイズに依存して変化していることが確認できる．従来のカスタムプロセッサのアクセラレータコア [5] については，コア数，コアあたりの PE 数が固定されている．そのため，本研究のように，コア数の変化によるデータ転送と演算処理のオーバーラップの変化を考慮したアーキテクチャの最適化が不可能である．一方で，本研究のような FPGA ベースのマルチコアアーキテクチャについては，画像の大きさ，ウィンドウの大きさなどアプリケーションの仕様に合わせて最適なコア数を選択したアーキテクチャを FPGA に実装できる．

最適化されたアーキテクチャにおけるデータ転送と演算処理のオーバーラップに

表 2.4: 異なるフィルタサイズに対する自由度の最適化 (メモリモジュール数 32)

ウィンドウサイズ $W_H = W_W$	設計自由度					全体の処理時間 (ms)	オーバーラップ
	ウィンドウ並列度 W_P	ピクセル並列度 P_P	コア数 N_C	画像分割数			
				横 N_H	縦 N_V		
8	8	2	2	4	2	45.58	Case A1
9	16	1	2	4	4	45.88	Case A1
10	8	2	2	4	2	45.53	Case A1
12	16	1	4	4	4	46.93	Case A1
13	16	1	4	4	4	46.62	Case A1
15	32	1	2	8	4	49.00	Case A1
16	32	1	4	8	4	49.53	Case A1
17	32	1	4	8	4	49.05	Case A1
18	32	1	4	8	4	49.55	Case A1
20	32	1	8	8	4	50.75	Case A1
24	32	1	8	16	2	61.60	Case A2

ついて，ウィンドウサイズが 15×15 よりも小さい場合，定常状態におけるデータ転送と演算処理のオーバーラップは図 2.15 の分類で CaseA1 になる．この理由について，1 コアにおける演算時間が他コアのデータ転送のオーバーヘッドをオーバーラップするほど長くないためであると考えられる．一方で，ウィンドウサイズが 15×15 よりも大きい場合，定常状態におけるデータ転送と演算処理のオーバーラップは図 2.15 の分類で CaseA2 になる． この理由について，1 コアにおける演算時間が他コアのデータ転送のオーバーヘッドをオーバーラップするほうが，全体の処理時間を短くすることができるためであると考えられる

表 2.4，表 2.5 にメモリモジュール数制約が 32 と 64 の場合についての最適化され

表 2.5: 異なるフィルタサイズに対する自由度の最適化 (メモリモジュール数　64)

ウィンドウサイズ $W_H = W_W$	設計自由度					全体の処理時間 (ms)	オーバーラップ
	ウィンドウ並列度 W_P	ピクセル並列度 P_P	コア数 N_C	画像分割数 横 N_H	画像分割数 縦 N_V		
8	8	2	2	4	2	45.58	Case A1
9	16	1	2	4	4	45.88	Case A1
10	8	2	2	4	2	45.53	Case A1
12	16	2	2	4	4	46.93	Case A1
13	16	1	4	4	4	46.62	Case A1
15	32	1	2	8	4	49.00	Case A1
16	16	2	4	4	4	47.33	Case A1
17	32	1	4	8	4	49.05	Case A1
18	16	2	4	4	4	47.35	Case A1
20	16	2	4	8	2	49.75	Case A2
24	64	1	4	16	1	57.43	Case A1

た設計自由度の値を示す．表 2.3 と比較すると，定常状態におけるデータ転送と演算処理のオーバーラップは図 2.15 の分類で CaseA1 になる設計自由度の組み合わせが多くなっている．この理由について，ハードウェア制約の上限が増加して演算並列度が大きくなった場合は，Window 演算時間が短縮されるため，他コアにおけるデータ転送のオーバーヘッドを完全にオーバーラップできなくなるためであると考えられる．

MIMD アクセラレータにおける Window 演算の最適実装に関する先行研究について，[10] のようなデータ転送・演算時間の合計を最小化する手法が提案されている．先行研究 [10] におけるアクセラレータの最適化手法との計算時間の比較を表 2.6 に

表 2.6: 先行研究 [10] と本研究の最適化手法の比較

メモリモジュール数の上限 $W_P \times P_P$	ウィンドウサイズ $W_H = W_W$	先行研究 [10] 処理時間 (ms)	本研究 処理時間 (ms)	削減率 (%)
16	8	56.42	46.24	18.04
	10	62.81	46.20	26.45
	15	84.75	54.50	35.69
	20	114.79	83.97	26.85
	24	144.32	115.89	19.70
32	8	50.43	45.58	9.63
	10	53.53	45.53	14.94
	15	64.24	49.00	23.72
	20	79.00	50.73	35.79
	22	85.99	54.10	37.09
	24	93.57	61.60	34.17
64	8	47.44	45.58	3.92
	10	48.88	45.53	6.86
	15	53.98	49.00	9.23
	20	61.11	49.75	18.59
	24	68.19	57.43	15.78

示す．先行研究では 1 コアにおけるデータ転送・演算時間の合計の最適化のみ考慮されているため，データ転送・演算時間のオーバーラップは実装されていない．本研究では、複数コアにおけるデータ転送・演算時間の同時実行によるオーバーラップを考慮しているため，先行研究と比べて 3%から 35%の処理時間の削減を実現している．

2.5.4　カスタムプロセッサとの比較評価

従来のヘテロジニアスマルチコアプロセッサと本研究で提案する FPGA ベースのアクセラレータとの性能比較について，日立製の RP1 プロセッサ [5] における Window 演算の処理時間と消費電力を比較評価した．RP1 プロセッサは，CPU コアとして SH-4A プロセッサを 4 コア，アクセラレータコアとして FE-GA を 2 コア搭載したヘテロジニアスマルチコアプロセッサである．SH-4A の動作周波数は 600MHz，FE-GA の動作周波数は 300MHz である．FE-GA コアに搭載されている PE 数は 32，メモリモジュールに接続される PE 数は 8 である．今回の比較評価では，FPGA に実装する MIMD アクセラレータの PE 数，メモリモジュール数については，FE-GA2 コアに相当するように設定している．

表 2.7 に Window 画像処理の処理時間の比較結果を示す．ウィンドウサイズが 18×18 の場合については，FE-GA を 2 コア使用した RP1 における処理時間よりも FPGA ベースの提案アクセラレータにおける処理時間のほうが小さいことが確認できる．その他のウィンドウサイズの場合についても，FPGA ベースの MIMD アクセラレータにおける処理時間の増加量は，RP1 における処理時間の約 20%から 30%程度である．

表 2.8 に消費電力の比較結果を示す． Zynq の消費電力は Xilinx Power Estimator 14.3. を用いて計測している．FPGA ベースの提案アーキテクチャはカスタムプロセッサである RP1 と比べても小さい消費電力で演算処理を実行できることが確認できる．FPGA の消費電力について考察すると，FPGA はプログラムによる回路構成の変更を実現するためのスイッチなどを搭載している冗長な回路構成となっている．

表 2.7: RP1 との計算時間比較 (ms)

ウィンドウサイズ	Zynq		RP1 [10]	
	1xCortex-A9 +FPGA	2xCortex-A9	1xSH-4A +1xFE-GA	1xSH-4A +2xFE-GA
12×12	46.51	1322.19	47.72	36.24
18×18	70.51	2920.37	102.67	72.94
24×24	115.87	5145.57	137.07	96.55

表 2.8: RP1 との消費電力比較

	Zynq		RP1 [10]	
	1xCortex-A9 +FPGA	2xCortex-A9	1xSH-4A +1xFE-GA	1xSH-4A +2xFE-GA
消費電力	1.03W	1.13W	1.30W	1.36W
プロセスルール	28nm [18]		90nm [5]	

そのため，従来はFPGAベースのアーキテクチャの消費電力は，専用回路によるアクセラレータに比べて大きくなりやすと考えられていた．しかしながら，集積回路の微細化による製造コストの増大により，大量生産できるFPGAのプロセスルールについては，少数生産が必要な専用回路のプロセスルールと比べて微細化したものを採用することが容易である．本研究で使用しているFPGAのプロセスルールは28nmであるが，同等のプロセスルールで専用回路を製造するためには，膨大な製造コストが必要になることが予想される．そのため，微細化技術による消費電力削減効果を受けやすいFPGAベースのアーキテクチャは，製造コストを抑えつつ専用回路の消費電力と比べても遜色が無いアーキテクチャを作成できるようになりつつあると考えられる．

2.6　結言

本章では，Window 画像処理におけるデータ転送と演算処理をオーバーラップを考慮した MIMD アクセラレータの最適設計法を提案した．FPGA ベース MIMD アクセラレータのアーキテクチャモデルについて，データ転送と演算処理をオーバーラップを考慮した処理時間の見積もり式を導出した．見積もり式を用いた Window 画像処理時間の評価結果より，オーバーラップを考慮した Window 画像処理時間を最小化するための設計自由度が導出されていることを確認した．また，従来のカスタム化されたヘテロジニアスマルチコアプロセッサとの比較評価により，本研究で提案する FPGA ベースのアクセラレータは処理速度，消費電力においてカスタムプロセッサと遜色ない性能を実現していることを確認した．

第3章　内部メモリの効率的利用を考慮したステンシル計算アクセラレータの最適設計

3.1　まえがき

本章では熱力学，流体力学，電磁界解析などの計算で用いられるステンシル計算について，内部メモリによる中間結果の再利用に注目したFPGAアクセラレータの最適設計法を提案する．ステンシル計算の処理について，タイムステップごとに外部メモリアクセスが発生するため，FPGA内部メモリにタイムステップごとの計算結果を効率的に格納することが重要になる．そのため，内部メモリに中間計算結果を格納して，　外部メモリアクセス回数を削減に注目したステンシル計算アクセラレータのアーキテクチャモデルについて説明する．また，アクセラレータのアーキテクチャの最適化のために，FPGA内部で処理するタイムステップ回数とタイムステップあたりのステンシル演算の並列度を考慮した処理時間の見積もり式を導出する．アクセラレータの最適化の評価について，設計自由度ごとの計算時間，および消費電力の比較を行い，最適化の方法について考察する．

3.2　ステンシル計算

ステンシル計算は，線形方程式を反復法で解く場合に用いられる計算法で，熱力学，流体力学，FDTD 法による電磁界解析などの数値シミュレーションの分野で幅広く使用されている．ステンシル計算を用いて数値シミュレーションを実行する場合について，シュミレーション空間の計算領域は，離散化された座標におけるグリッドごとの数値データの集合として扱われる．また，数値シミュレーションを実行する時系列について，計算する時間を短いタイムステップに区切り，タイムステップごとのグリッドごとの値を更新する演算を実行する．時系列ごとのグリッドごとの数値の変化を計算することにより，現実世界の熱，電磁波などの挙動の時間変化をコンピュータ内でシミュレーションすることができる．タイムステップごとのグリッドデータを更新する計算について，近傍グリッドの値からグリッドごとの数値データを計算するステンシル演算が行われる．このような近傍グリッドを含むグリッド領域について，本研究ではステンシル領域と定義する．図 3.1 に 2 次元の計算領域におけて，1 次差分方程式を離散化したステンシル計算で用いられるステンシル領域の例を示す．このステンシル領域には，ステンシル演算の対象グリッド，および対象グリッドに隣接するグリッドが含まれている．ステンシル領域内のタイムステップ t における座標 (i,j) に位置するグリッド $v^t_{j,i}$ を更新するためのステンシル演算は式 (3.1) のように定義される．

$$v^{t+1}_{j,i} = c_0 v^t_{j,i} + c_1 v^t_{j-1,i} + c_2 v^t_{j+1,i} + c_3 v^t_{j,i-1} + c_4 v^t_{j,i+1} \tag{3.1}$$

ここで，c_0，c_1，c_2，c_3，c_4 は計算定数で，アプリケーションによって決定される．ステンシル領域の形状は，高次差分方程式などステンシル計算のアプリケーションによっては図 3.1 とは異なる形状をしている場合もある．また，境界に接するグリッド領域においては，境界条件を処理するための計算式によってグリッドデータが更新される．ステンシル計算全体の処理では，ステンシル演算を計算領域内の全グリッ

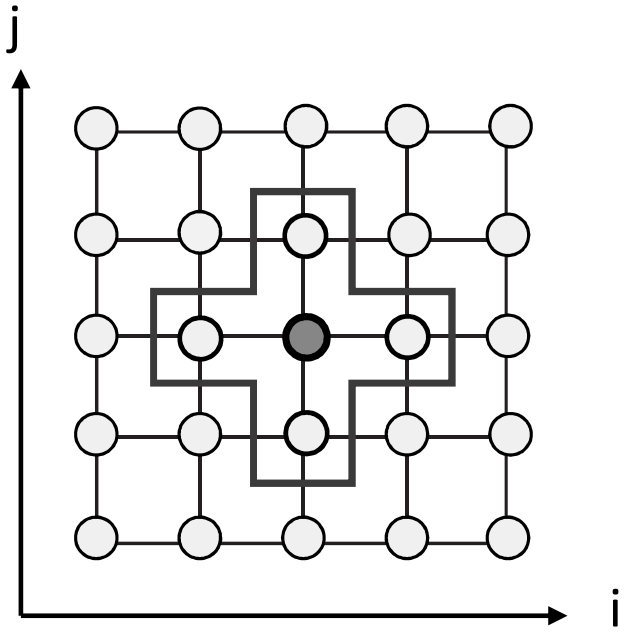


図 3.1: ステンシル領域

ドに実行する処理を，図 3.2 のように与えられたタイムステップ回数繰り返し実行する．

ステンシル計算は数値シミュレーションに関する幅広い分野のアプリケーションに使用されているため，ステンシル計算の処理を高速化するための先行研究が多く報告されている．1 つのタイムステップごとに処理する計算領域分のステンシル計算については，座標ごとのステンシル演算を並列化して処理することができる．そのため，マルチコア CPU[19, 20]，GPU[21, 22] などのマルチコアアーキテクチャを使用したステンシル計算の高速化に関する先行研究が報告されている．さらに，FPGA を使用したステンシル計算アクセラレータの設計についての先行研究も報告されている [23, 24, 25, 26]．FPGA を使用したステンシル計算アクセラレータに関する先行研究の中には，複数の FPGA ボードを連結した大規模なアーキテクチャを実装したものも存在している [24, 25, 26]．

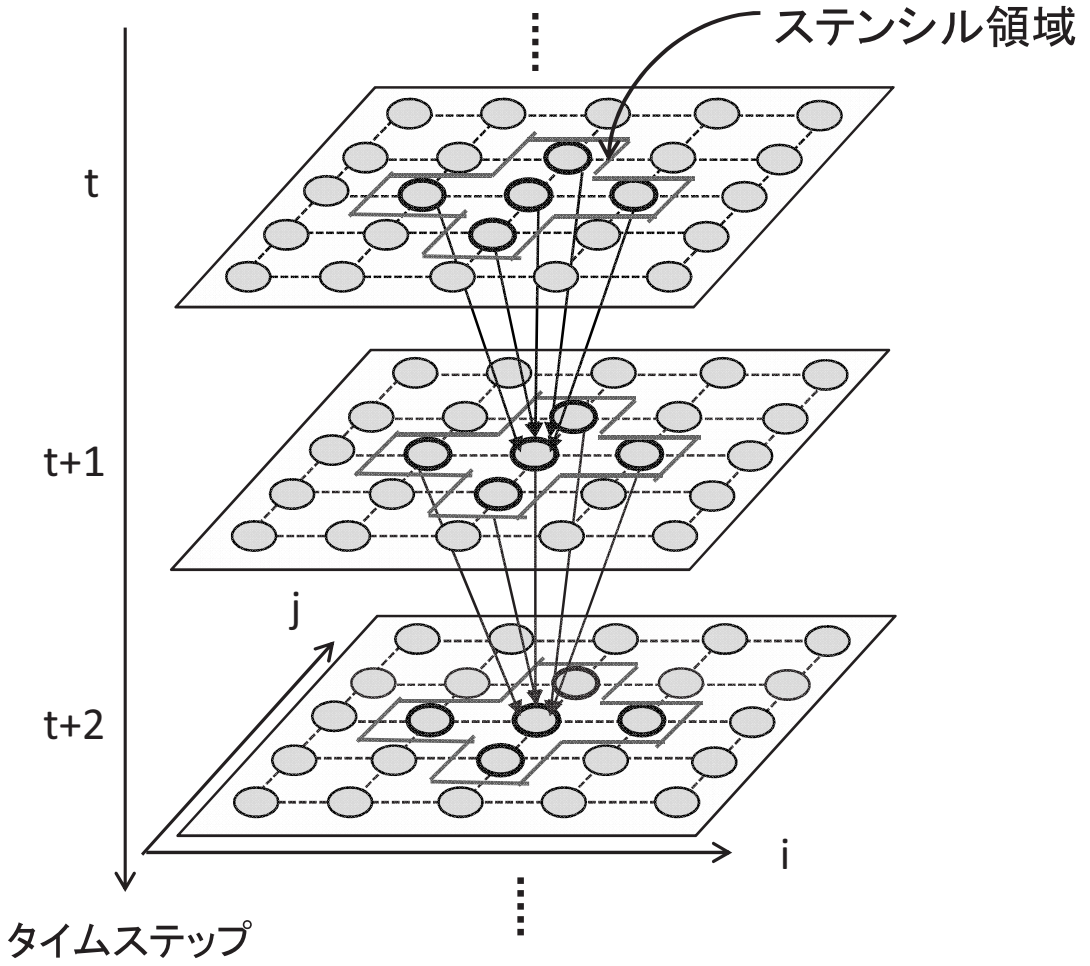


図 3.2: ステンシル計算の時間方向の処理

ステンシル計算の並列化については，これらの先行研究のようにマルチコアアーキテクチャを用いることにより比較的容易に実現することができる．しかしながら，ステンシル計算の処理の高速化のためには，メモリからのデータ転送時間も考慮する必要がある．図 3.2 のようなステンシル計算を処理するためには，計算領域全体のグリッドデータへのアクセスがタイムステップごとの処理に必要になる．そのため，ステンシル計算を用いた数値シミュレーションにおいては，計算領域のグリッドデータ量が内部メモリの記憶容量よりも大きなデータを扱う必要があるため，外

部メモリへのアクセスがタイムステップごとに発生している．結果として，ステンシル計算の処理性能は主に外部メモリとの帯域がボトルネックとなっており，メモリ内部のグリッドデータへのアクセスの効率化が重要になる．

メモリアクセスを効率化する方法として，プロセッサの内部メモリにステンシル計算の計算結果を次のタイムステップに使用される計算領域の分だけ格納して，外部メモリとのアクセス回数を減少させる手法が提案されている．例として，GPU などのアクセラレータのローカルメモリに，分割した計算領域（タイル）をタイルの周辺領域とともに格納して複数タイムステップを外部メモリアクセス無しで処理する Overlapped tiling [22] があげられる．この方法の問題点として，内部メモリのアクセスで処理するタイムステップ数を増やした場合について，計算に必要なタイルの周辺領域が増加していくため，データの記憶量や計算量が増加してしまうことがあげられる．その他の手法として，タイムステップごとの空間的なステンシル演算のラスタスキャンとタイムステップ方向の時間的なステンシル演算の繰り返しをパイプライン化した時空間的なパイプラインスケジューリング [26] が提案されている．このスケジューリングでは，タイムステップごとの処理に必要な中間結果の記憶データ量を小さくすることができるため，FPGA 内部のメモリで処理するタイムステップを増加させるためには非常に有用である．

本研究では，FPGA の内部メモリアクセスのみで処理するタイムステップ回数を増やすことにより，中間結果の記憶外部メモリアクセス回数の削減しているステンシル計算アクセラレータに注目している．ステンシル計算アクセラレータの具体的なアーキテクチャモデルについては 3.3 章で説明する．

3.3　ステンシル計算アクセラレータの設計

本章では，ステンシル計算アクセラレータのアーキテクチャモデルの概要，および OpenCL を使用した FPGA ベースのステンシル計算アクセラレータの設計手法について説明する．

3.3.1　ステンシル計算アクセラレータのアーキテクチャモデル

本研究で対象とするステンシル計算アクセラレータのアーキテクチャモデルを図 3.3 に示す．このアーキテクチャは，中間データを記憶するバッファとステンシル演算を処理するための PE アレイで構成されているタイムステップごとの計算モジュールが，FPGA 内部で処理するタイムステップの分だけ搭載されている．このようなアーキテクチャによって，外部メモリからの 1 回のデータ読み出しに対して複数のタイムステップのステンシル演算を処理するようにしている．本研究では，FPGA 内部で処理するタイムステップの集合を depth と定義している．

バッファに記憶する中間データの量を小さくするために，ステンシル計算のスケジューリングは 3.1 章で説明した時空間的なパイプラインスケジューリングを採用している．図 3.4 に時空間的なパイプラインスケジューリングの概要を示す．このスケジューリングでは，サイクルごとに発生するステンシル計算の結果を次のタイムステップのバッファに入力していくストリーミング処理を行っている．

タイムステップごとのステンシル計算に必要なバッファ構造について，ステンシル計算を処理する最低条件として，ステンシル領域内のグリッドデータが全て格納されている必要がある．また，図 3.4 に示されるような時空間的なパイプラインスケジューリングでは，サイクルごとに計算領域から，または前のタイムステップにおけるステンシル計算の計算結果が 1 グリッドデータずつバッファに入力される．サイクルごとに入力されるグリッドデータ，および計算するステンシル領域は，右方向に 1 グリッドずつシフトしている．本研究では，タイムステップごとの処理に使

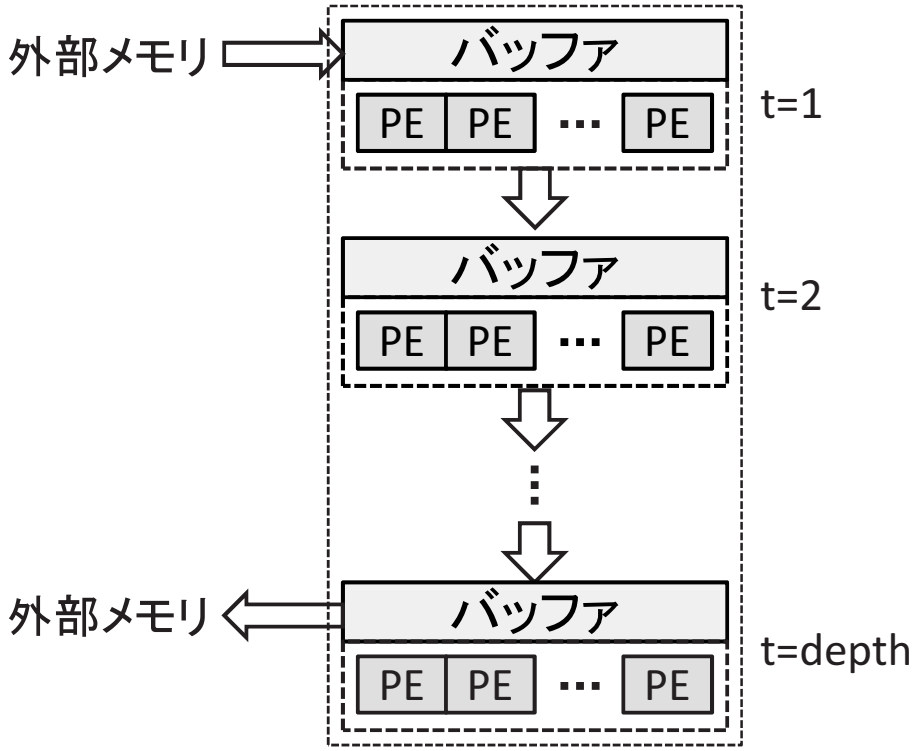


図 3.3: ステンシル計算アクセラレータのアーキテクチャモデル

用するバッファについて，図 3.5 のようなステンシル領域を含有するシフトレジスタベースの構造を採用している．このバッファは，計算領域 1 行分のシフトレジスタをステンシル領域内のグリッドデータを記憶できるように連結している．計算領域 1 行分のシフトレジスタの連結数は，ステンシル領域の縦方向の大きさから 1 を引いた数に等しくなる．さらに，ステンシル領域をカバーできるように，ステンシル領域の横方向のグリッド数分のシフトレジスタ領域を追加する．このシフトレジスタの記憶領域は，計算領域に対してして小面積であるため，少ないリソース量で実装することができる．そのため，FPGA 上に実装する depth を大きくして，外部メモリアクセス回数を削減したステンシル計算アクセラレータを実装することができる．

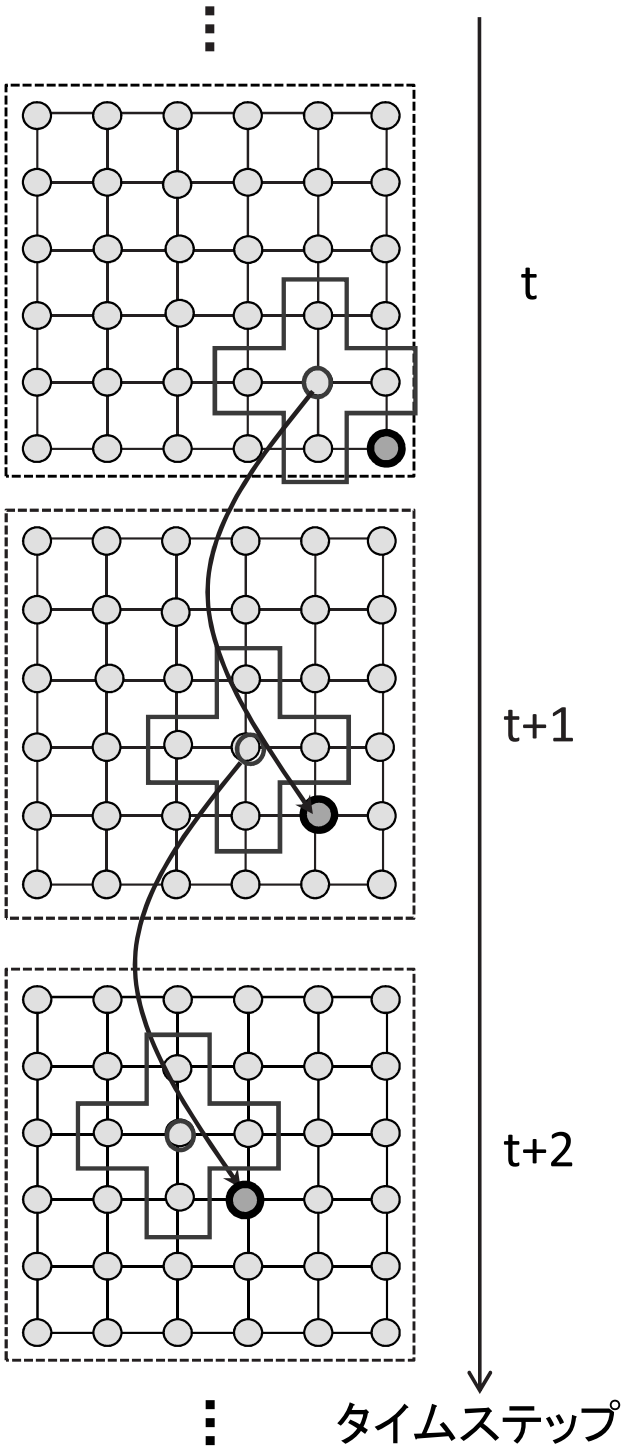


図 3.4: ステンシル計算の時空間的なパイプラインスケジューリング

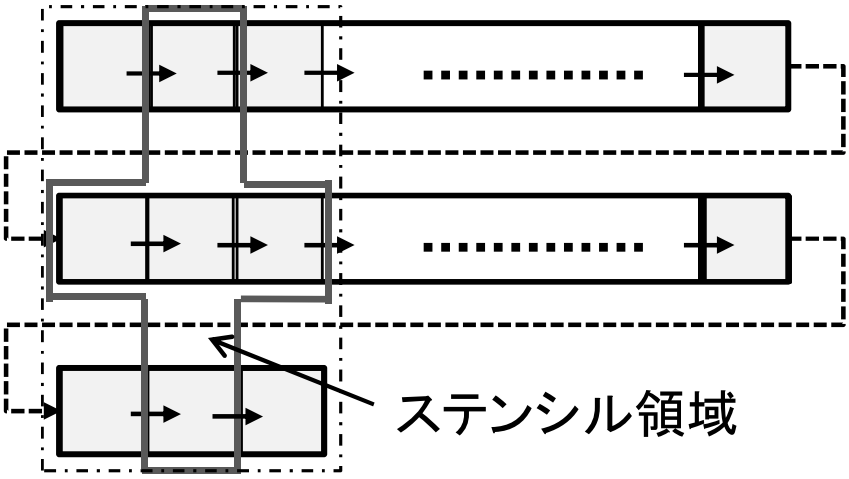


図 3.5: シフトレジスタによるバッファ構造

図 3.3 のようなステンシル計算アクセラレータの設計自由度について，1 タイムステップあたりのステンシル計算を処理する計算モジュールの数 (depth) と計算モジュールの内の PE 数 (タイムステップごとのステンシル演算の並列度) の 2 つが定義される．外部メモリのアクセス回数の削減に注目すると，FPGA のリソース量の限界まで depth を大きく取るようなアーキテクチャが好ましいと考えられる．一方で，タイムステップごとのステンシル演算の並列度を上げることについても，ステンシル計算の処理時間の短縮に有効である．しかしながら，タイムステップごとのステンシル演算の並列度を上げる場合は，並列演算のための PE と 1 グリッドデータ分のシフトレジスタがタイムステップあたりのステンシル計算回路に追加される．そのため，リソース制約下においてはステンシル演算の並列度を上げると depth が小さくなり，外部メモリとのアクセス回数が増加してしまう．ステンシル計算アクセラレータの最大性能を得るためには，このような 2 つの設計自由度のトレードオフの関係について考慮する必要がある．ステンシル計算アクセラレータの設計自由度の最適化手法については，3.4 章で詳しく説明する．

3.3.2　OpenCL を用いた FPGA アクセラレータの設計

FPGA にステンシル計算アクセラレータを実装するためには，従来は VHDL，Verilog HDL に代表されるようなハードウェア記述言語によってアクセラレータの回路構成を記述する必要があった．しかしながら，CPU，GPU 向けソフトウェアによるアプリケーション実装と比較すると，FPGA によるハードウェア実装は演算部，制御部，外部メモリおよびホスト PC との接続部，モジュール間のインターコネクションなど，必要な回路をハードウェア記述言語を用いてすべて設計しなければならない．そのため，高性能計算を効率的に処理するための大規模な FPGA アクセラレータを設計するためには，長期間の設計時間と回路設計に関する高度な知識が要求される．

本研究では，ステンシル計算アクセラレータの設計を簡単化するために，データパスの設計に GPU などのマルチコア向けプログラミング言語である OpenCL を使用して，図 3.3 に示されるようなステンシル計算アクセラレータを FPGA に実装している．OpenCL は，C 言語をベースにした並列処理を実行するためのヘテロジニアスアーキテクチャ向けフレームワークであり，ホスト PC で実行するためのホストコード，アクセラレータ側で並列処理を実行するためのカーネルによって構成されている．OpenCL は，Apple,Intel,Google などの多数の企業が参画している非営利団体の Khronos グループ [28] により標準化されている．OpenCL の特徴として，デバイスに依存しないプログラミングが可能なことがあげられる．マルチコア CPU，GPU，CELL プロセッサなどそれぞれのアーキテクチャに対応したコンパイラを使用することにより，共通の OpenCL コードを異なるアーキテクチャ上で実行させることができる．また，FPGA 上に OpenCL で記述されたプログラムを実行させるために，FPGA 向けの OpenCL コンパイラである Altera SDK for OpenCL[27] がリリースされている．このコンパイラを使用することにより，CPU，GPU 向けプログラミング言語による抽象的な記述から FPGA 向けの回路構成を自動的に合成することが可能になり，FPGA 実装に必要な設計時間を大幅に削減することができるよ

うになった．しかしながら，GPU 向けの OpenCL コードを FPGA に実装しても高性能な処理を実現できないため，FPGA のアーキテクチャに適するような OpenCL 記述を実装する必要がある．OpenCL コンパイラを用いた FPGA アクセラレータの先行研究として，フラクタル画像処理 [29]，AES 暗号処理 [30] などが報告されている．いずれの実装例においても，GPU などの他のアーキテクチャと比較して高速かつ低消費電力な処理を実現している．OpenCL コンパイラと HDL 設計の比較については，画像処理アクセラレータ設計における比較評価が先行研究 [31] で報告されている．この先行研究では，OpenCL コンパイラを用いることにより，HDL による設計と同等の処理性能を持つ画像処理アクセラレータを，約 1/6 の設計時間で FPGA に実装できることが報告されている．

図 3.6 に OpenCL により生成される FPGA アクセラレータのアーキテクチャモデルの概要を示す．カーネルパイプライン，ローカルメモリ間のインターコネクションなどの回路構成を FPGA 実装するためのデータは，演算処理を記述したカーネルから OpenCL コンパイラによって自動的に合成される．さらに，外部メモリの制御回路，PCI-Express によるホスト PC との通信部の回路についても，OpenCL コンパイラによって自動的に合成される．

本研究で対象とするステンシル計算アクセラレータを OpenCL を用いて実装するためには，タイムステップごとのステンシル計算モジュールに搭載されるシフトレジスタ，ステンシル計算を処理する PE を OpenCL で記述する必要がある．しかしながら，効率的にステンシル計算を処理するアーキテクチャを FPGA に実装するためには，FPGA に適するように OpenCL 記述をする必要がある．そのため，本研究では OpenCL コンパイラの製造元である Altera 社が推奨するスライディングウィンドウデザインパターン [32, 33] に基づいて，ステンシル計算アクセラレータのデータパスを記述している．

図 3.7 に図 3.5 のようなシフトレジスタ構造の OpenCL 記述例を示す．5 行目の記述により，バッファ内のデータシフトを表現している．このような記述を用いるこ

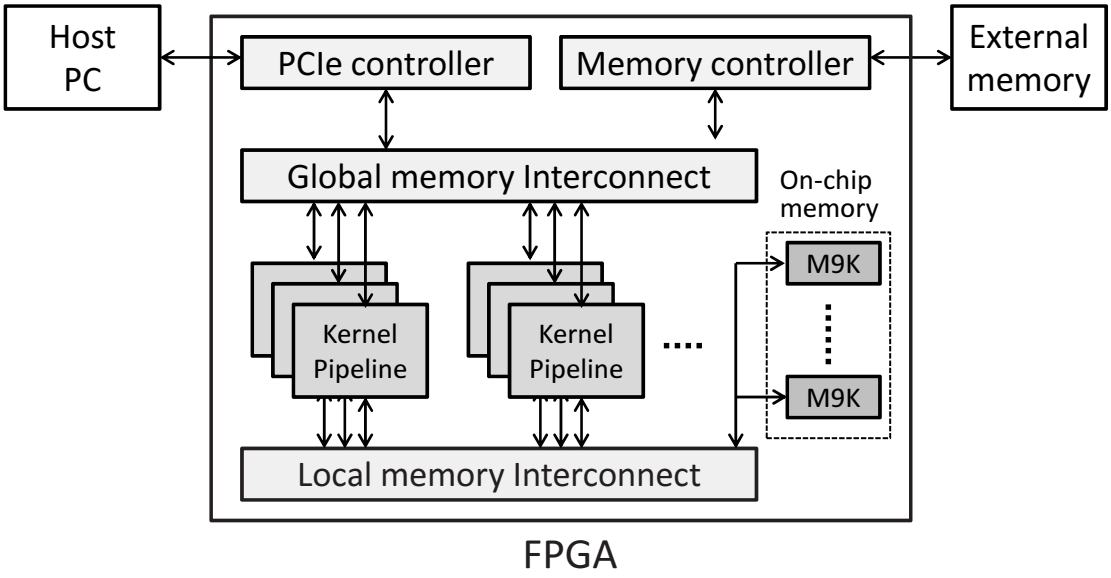


図 3.6: OpenCL により生成されるアーキテクチャモデル

とにより，コンパイラによってシフトレジスタが FPGA 上に自動的に実装される．

図 3.8 にタイムステップ方向のステンシル計算処理の OpenCL 記述例を示す，3 行目の記述により，シフトレジスタ内のステンシル領域内のグリッドデータを入力してステンシル演算が実行される．また，4 行目から 13 行目の記述により，ステンシル演算結果が次のタイムステップにおけるシフトレジスタ，または外部メモリに入力される．タイムステップ方向のステンシル計算処理の記述について，1 行目の Loop unroling の pragma 記述により,2 行目以降の for 文のループが展開される．そのため，実装されるアーキテクチャはステンシル計算を処理するモジュールがタイムステップの順番に接続される．外部メモリのグリッドデータをステンシル計算アクセラレータに転送してから，depth 回のステンシル計算が実行された結果を外部メモリに書き込む処理までの全ての処理については，コンパイラによって自動的にパイプライン化されている．

ステンシル計算の並列度の変更について，本研究では，OpenCL で定義されたベク

タ型と呼ばれるデータ型を使用して並列度を変更している．ベクタ型は int 型，float 型などによる複数のデータを 1 つにまとめたデータ型であり，GPU などで SIMD 演算を実行する場合に使用されている．外部メモリ間とのグリッドデータ転送，ステンシル演算処理をベクタ型を用いて SIMD 化することにより，個々のグリッドを逐次的に処理する場合と比較してリソースの増加量や制御オーバーヘッドを削減することができる

```
1.    #pragma unroll
2.        for (int j = 0; j > DEPTH; j++) {
3.    #pragma unroll
4.            for (int i = N * 2 + 1; i > 0; --i) {
5.              shift[ j ][ i ] = shift[ j ][i - 1];    //シフトレジスタ内のデータシフト
6.            }
7.        }
8.       shift[0][0] = in[count]; //   シフトレジスタにデータ入力
```

図 3.7: シフトレジスタ構造の OpenCL 記述

```
1.#pragma unroll
2.  for (int j = 0; j > DEPTH; j++) {
3.       sten[ j ]=0.25*(shift[j][0*N+1]+shift[j][1*N+0]+shift[j][1*N+2]+shift[j][2*N+1]);
4.       if ( j<DEPTH -1){
5.           if (count!=boundary) {
6.              shift[j+1][0] = sten[ j ];    //次ステップのシフトレジスタに計算結果転送
7.           }
8.       }
9.       else{
10.           if (count!=boundary){
11.             out [count - latency] = sten[ j ];    //外部メモリに計算結果転送
12.           }
13.      }
14.  }
```

図 3.8: タイムステップ方向のステンシル計算処理の OpenCL 記述

3.4 ステンシル計算アクセラレータの最適設計法

本節では，ステンシル計算アクセラレータを FPGA に実装するための制約条件，およびステンシル計算アクセラレータのアーキテクチャを最適化するための計算時間見積もりの導出手法について説明する．

3.4.1 制約条件

ステンシル計算アクセラレータを FPGA 実装するための制約条件として，LUT・メモリモジュール等の FPGA のリソース量の上限による制約と，外部メモリの最大帯域の制約の 2 つがあげられる．

FPGA の LUT，メモリモジュール等のリソース使用量の上限による制約条件について，外部メモリ制御部・PCI 通信部については設計自由度によらず一定のリソース使用量のアーキテクチャが作成されると仮定できる．一方で，ステンシル計算アクセラレータのリソース使用量は depth，ステンシル演算の並列度の 2 つの設計自由度に依存して変化する．シフトレジスタと PE で構成されるタイムステップごとのステンシル計算モジュールについて，シーケンサなどの制御部は一定のリソース使用量のアーキテクチャが作成されると仮定できる．一方で，PE，シフトレジスタなどのリソース使用量は並列度に対して 1 次関数的に増加するとと仮定できる．このようなステンシル計算モジュールが depth 個実装されるため，リソース使用量の上限を R_{MAX} とすると，リソース使用量の制約は式 (3.2) で与えられる．

$$R_{MAX} \geq R_{base} + depth \times (R_{ctrl} + P_{sten} \times R_{PE}) \tag{3.2}$$

ここで，R_{base} は外部メモリ制御部・PCI 通信部などのリソース使用量，R_{ctrl} はステンシル計算モジュールにおいて設計自由度に依存しない部分のリソース使用量，P_{sten} はタイムステップあたりのステンシル演算の並列度, R_{PE} は P_{sten}あたりのリソ

ース使用量の平均増加量（主に PE のリソース増加量）である．式 (3.2) より,P_{sten} ごとの depth の上限を導出することができる．

外部メモリの帯域の制約条件について，FPGA のボードごとに外部メモリとの転送帯域の性能は，搭載されているメモリの性能および FPGA ボードの構造に依存している．そのため，FPGA 上のデータパスの要求するデータ転送速度が外部メモリとの最大帯域を超えた場合は，外部メモリの読み出しにストールが発生して処理性能が低下する．データパスの要求するデータ転送速度について，1 回のデータ転送におけるデータ量と動作周波数（1 秒あたりのデータ転送回数）の積で導出される．1 回のデータ転送におけるデータ量について，SIMD によるデータ転送により演算語長とステンシル演算の並列度の積で導出される．そのため，外部メモリとの最大帯域を B_{MAX} とすると，外部メモリとの最大帯域による制約は式 (3.3) で与えられる．

$$B_{MAX} \geq W_{width} \times P_{sten} \times f \tag{3.3}$$

ここで，W_{width} は演算語長，f は動作周波数である．式 (3.3) より，FPGA 上に実装できる P_{sten} の上限が導出される．

3.4.2　計算時間見積もり式

本節ではステンシル計算アクセラレータを最適化するための計算時間の見積もり式の導出について説明する．本研究は 3.3.2 章で説明したように，OpenCL を用いてアーキテクチャを設計しているため，FPGA 上でステンシル計算を処理するためのカーネルと，カーネル起動，FPGA-HostPC 間のデータ転送を制御するためのホストコードが実行される．対象とする計算時間については，HostPC から FPGA ボード上の外部メモリに計算領域分のグリッドデータを転送してから，ステンシル計算を規定のタイムステップ数実行した計算結果のデータを HostPC に転送するまでの時間とする．図 3.9 にステンシル計算処理のタイムチャートを示す．ステンシル計

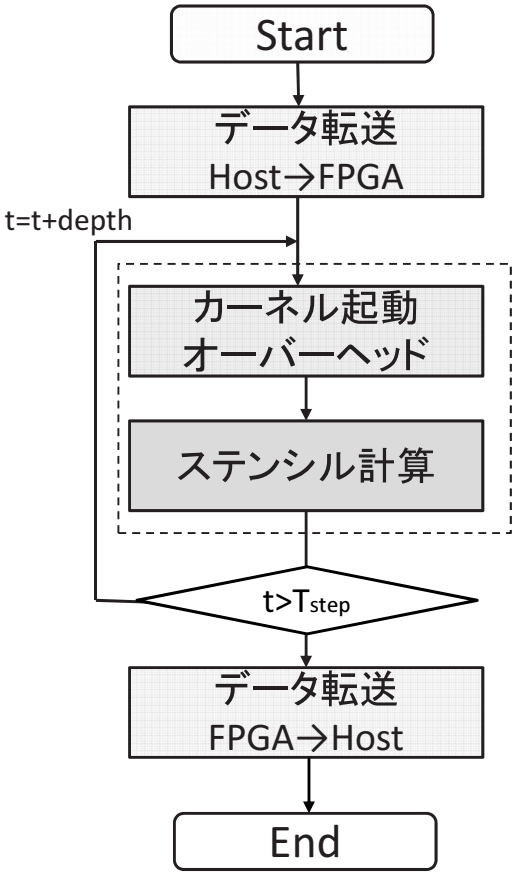


図 3.9: ステンシル計算のタイムチャート

算処理全体に必要なタイムステップ数を T_{step} とすると，depth 回のステンシル計算を処理するカーネルを起動する回数は，$\lceil T_{step}/depth \rceil$ 回となる．

そのため，ステンシル計算処理の全体の計算時間は式 (3.4) で与えられる．

$$t_{total} = t_{HF} + \left\lceil \frac{T_{step}}{depth} \right\rceil \times (t_{kern} + t_{sten}) + t_{HF} \tag{3.4}$$

ここで，t_{HF} は HostPC から FPGA ボードへのデータ転送時間，t_{kern} はカーネル起動オーバーヘッド，t_{sten} は depth あたりのステンシル計算時間，t_{HF} は FPGA ボードから HostPC へのデータ転送時間である．

t_{sten} の見積もりについて，ステンシルアクセラレータのデータパスはストールしないパイプラインであるため，計算にかかるサイクル数はパイプラインへのデータ投入数とデータパスのレイテンシの合計で求められる．外部メモリからのデータ入力は 3.3.2 章で説明したように，ベクタ型のデータ型を用いて複数グリッドのデータを同時に転送している．そのため，データ投入数 N_{data} については，計算領域内のグリッド数をステンシル計算の並列度で割った数値に等しくなる．

データパスのレイテンシについては，外部メモリ読み出しレイテンシ l_{ddr} とタイムステップあたりの計算モジュールのレイテンシの合計で見積もることができる．タイムステップあたりの計算モジュールのレイテンシは，シフトレジスタにグリッドデータが入力されてから，該当するグリッドデータのステンシル演算が実行されるまでのレイテンシ l_{shift} と，ステンシル演算処理する PE のレイテンシ l_{PE} の合計で見積もることができる．計算時間については，処理にかかるサイクル数を 1 秒あたりのサイクル数の値である動作周波数で割ることで求められる．そのため，t_{sten} は式 (3.5) で与えられる．

$$t_{sten} = \left(l_{ddr} + depth \times (l_{shift} + l_{PE}) + \frac{N_{grid}}{P_{sten}} \right) \times \frac{1}{f} \tag{3.5}$$

ここで，N_{grid} は計算領域内のグリッドデータ数である．

式 (3.4) および式 (3.5) より，ステンシル計算の処理時間は設計自由度 $depth$, P_{sten} の数値から計算されることが確認できる．そのため，ステンシル計算時間の最小化は $depth$, P_{sten} の全探索を実行することにより実現することができる．設計自由度を最適化するための探索空間については，2 つの制約式 (3.2)，(3.3) によって上限が設定されるため，非常に短い探索時間で最適化ができると考えられる．設計自由度の最適化の評価結果については， 3.5 章で詳しく考察する．

3.5 評価

3.5.1 見積もり式による処理時間最小化の評価

本研究では，Terasic DE5-NET board [34] にステンシル計算アクセラレータの実装評価を行った．この FPGA ボードは，FPGA に Altera Stratix V を搭載している．また，外部メモリとして，2GB の DDR3 SDRAM を 2 枚搭載している．外部メモリの最大通信帯域は 12.8GB/s であり，制約式 (3.3) の B_{MAX} の値に使用される．OpenCL による FPGA 実装のため OpenCL コンパイラは，AOCL14.1[27] を使用している．

ステンシル計算のアプリケーションについて，今回の評価では，熱解析に使用される 2 次元 laplace 方程式を対象アプリケーションとしている．式 (3.1) と同じ形式で laplace 方程式を表現すると，式 (3.6) のようになる．

$$v_{j,i}^{t+1} = 0.25 \times (v_{j-1,i}^{t} + v_{j+1,i}^{t} + v_{j,i-1}^{t} + v_{j,i+1}^{t}) \tag{3.6}$$

計算領域内のグリッド数は 1024×1024，全体のタイムステップ数は 10000，演算精度は単精度浮動小数点計算に設定している．境界条件は，タイムステップによって値が変動しないディレクレ条件 $(v_{j,i}^{t+1} = v_{j,i}^{t})$ を設定している．

FPGA に実装されたステンシル計算アクセラレータの動作周波数については，OpenCL コンパイラにより FPGA マッピングが実行された後に決定されるため，FPGA 実装前に正確な動作周波数を見積もることは困難である．そのため，本研究ではステンシル計算アクセラレータを実装したときの動作周波数の数値が変化する範囲を調査して，上限値と下限値の 2 つの計算時間の見積もり時間を導出する．図 3.10 にステンシル計算アクセラレータの動作周波数の変化を表すグラフを示す．演算並列度ごとのデータ系列について，depth が大きくなるに連れて動作周波数が下がる傾向が見られる．この理由について，実行に必要なリソース量が増え，実装されるアーキテクチャが複雑化するためにクロックの制約が厳しくなると考えられる．設計自由

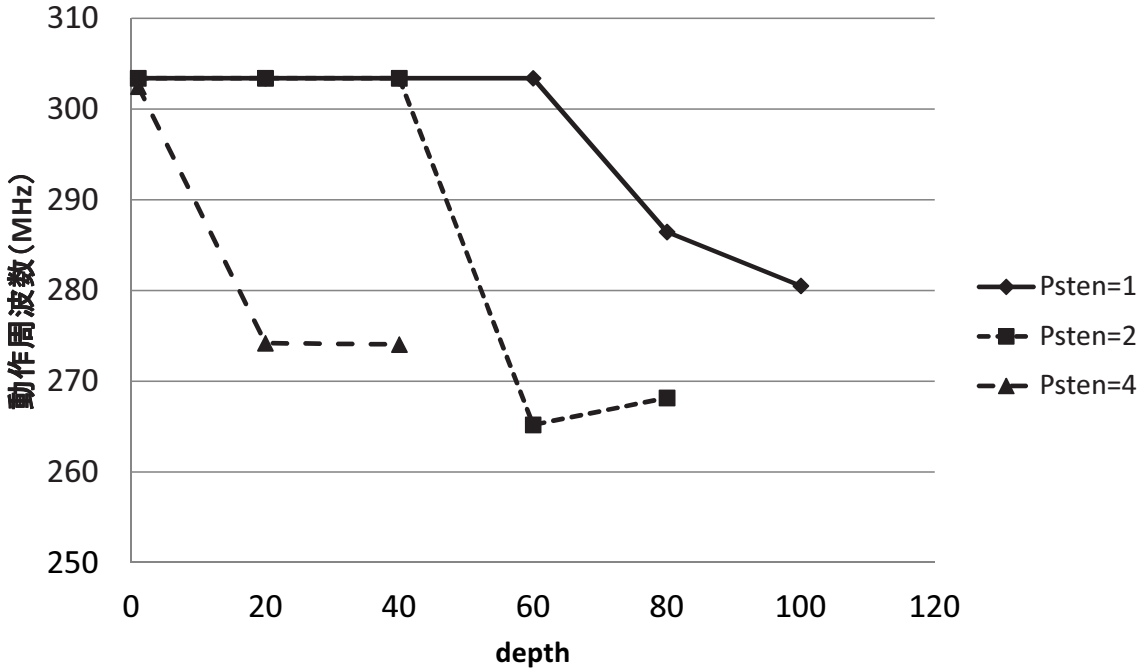


図 3.10: 設計自由度とステンシル計算アクセラレータの動作周波数のグラフ

度を変化させた場合の動作周波数の変動範囲については，250MHz から 300MHz までの範囲で動作周波数が変動することを確認している．そのため，今回の評価では動作周波数の範囲の上限を 300MHz，下限 250MHz として，計算時間見積もりの入力に使用している．計算時間の実測値については，式 (3.4),(3.5) で動作周波数の入力値を 300MHz とした見積もりの下限値と 250MHz とした見積もりの上限値の範囲内に存在することが予想される．

外部メモリ帯域による制約より，P_{sten} の最大値を式 (3.3) を用いて算出する．今回の評価では $B_{MAX} = 12.8(GB/s), \quad W_{width} = 4(byte)$ となるため，P_{sten} の最大値は動作周波数が 300MHz の場合は 10, 250MHz の場合は 12 となる

ステンシル計算アクセラレータのリソース使用量については，LUT とレジスタの使用量を表す Logic utilization，メモリモジュールの使用量を表す RAM blocks，乗算器などの DSP ユニットの使用量を表す DSP Units の使用率が算出される．リソー

表 3.1: リソース使用量 ($P_{sten} = 1$)

depth	1	5	10	20
Logic utilization	46203(20%)	51226(22%)	57242(24%)	69575(30%)
RAM blocks	362(14%)	380(15%)	406(16%)	466(18%)
DSP Units	0(0%)	0(0%)	0(0%)	0(0%)

表 3.2: リソース使用量 ($P_{sten} = 4$)

depth	1	5	10	20
Logic utilization	49006(21%)	65795(28%)	86563(37%)	129131(55%)
RAM blocks	366(14%)	402(16%)	451(18%)	551(22%)
DSP Units	0(0%)	0(0%)	0(0%)	0(0%)

ス制約を定義するためには，ステンシル計算アクセラレータの実装時において使用率が最も大きいリソースの種類を調べる必要がある．表 3.1，3.2 に FPGA にステンシル計算アクセラレータを実装した場合のリソース使用量を示す．これらの表から，Logic utilization の使用率が最も大きいことが確認されている．そのため，リソース使用量の制約式 (3.2) について，Logic utilization の使用量を基準にして depth の最大値を算出することができる．

式 (3.2) の入力パラメータの値について，実装評価の結果から $R_{base} = 45013$, $R_{ctrl} = 225$，$R_{PE} = 964$ となった．リソース使用量の上限値について，本研究では見積もりの誤差，および OpenCL コンパイラの性能を考慮して, FPGA 全体の搭載量の 90%($R_{MAX} = 212148$) を上限に設定している．図 3.11 にタイムステップあたりのステンシル演算の並列度と depth の最大値の関係を表すグラフを示す．このグラフより，ステンシル演算の並列度が増加すると depth の上限が減少するトレードオフの関係が確認できる．

式 (3.4),(3.5) より導出されたステンシル計算時間の見積もり値と実測値の比較評

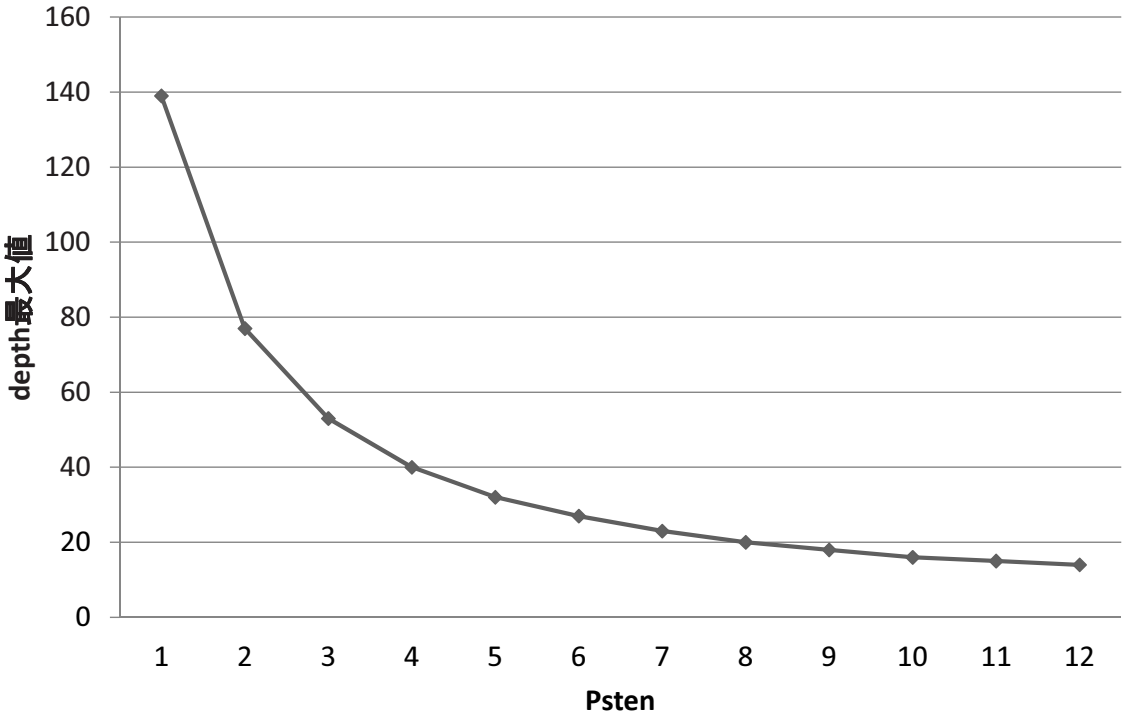


図 3.11: ステンシル演算の並列度ごとの depth の最大値のグラフ

価について考察する．式 (3.4) の入力パラメータの値について，実装評価の結果から，$t_{HF} = 7.47(ms)$, $t_{kern} = 0.0415(ms)$, $t_{FH} = 2.87(ms)$ となった．式 (3.5) の入力パラメータの値について，実装評価の結果から，$l_{ddr} = 180$, $l_{PE} = 18$, $l_{shift} = 1024/P_{sten} + 1$ となった．

図 3.12，3.13，3.14 にそれぞれ $P_{sten} = 1, 2, 4$ の場合の depth ごとのステンシル計算時間の見積もり値と実測値を比較したグラフを示す．図 3.12,3.14 について，見積もり値の上限 (f=250MHz の計算値) と下限 (f=250MHz の計算値) の間に実測値が存在することが確認できる．また，実装結果より，depth を大きくするとステンシル計算の処理時間の見積もり値，実測値がともに小さくなることが確認できる．このことから，FPGA 内部で処理するタイムステップ数を増やしたことにより，外部メモリのアクセス回数およびカーネル起動回数を削減した効果が現れていることが

確認できる．一方で，図 3.13 のように見積もり値の上限よりも実測値が大きくなるケースも見られた．理由としては，OpenCL コンパイラによって合成されたアクセラレータの動作周波数が小さくなってしまうことが考えられる．

図 3.15 にステンシル演算の並列度に対する計算時間の最小値を比較したグラフを示す．計算値の上限 (f=300MHz) の場合は $P_{sten} = 4$, 下限 (f=250MHz) の場合は $P_{sten} = 6$ の場合に計算時間の最小値が得られることが確認された．しかしながら，計算時間の最小値について，ステンシル演算並列度が 2 から 10 までの範囲では大きな違いは確認されなかった．また，計算時間の実測値は見積もり値の上限近くの値が計測されていることが確認できる．理由については，リソース量のほぼ上限の回路を実装しているため，動作周波数が下限値の近くまで低下しているためであると考えられる．計算時間の実測値についても，設計自由度ごとの大きな違いは確認されなかった．そのため，ステンシル計算アクセラレータの最適な設計自由度の選択については，計算時間の他に消費電力の違いなど，別の性能指標について評価する必要がある．

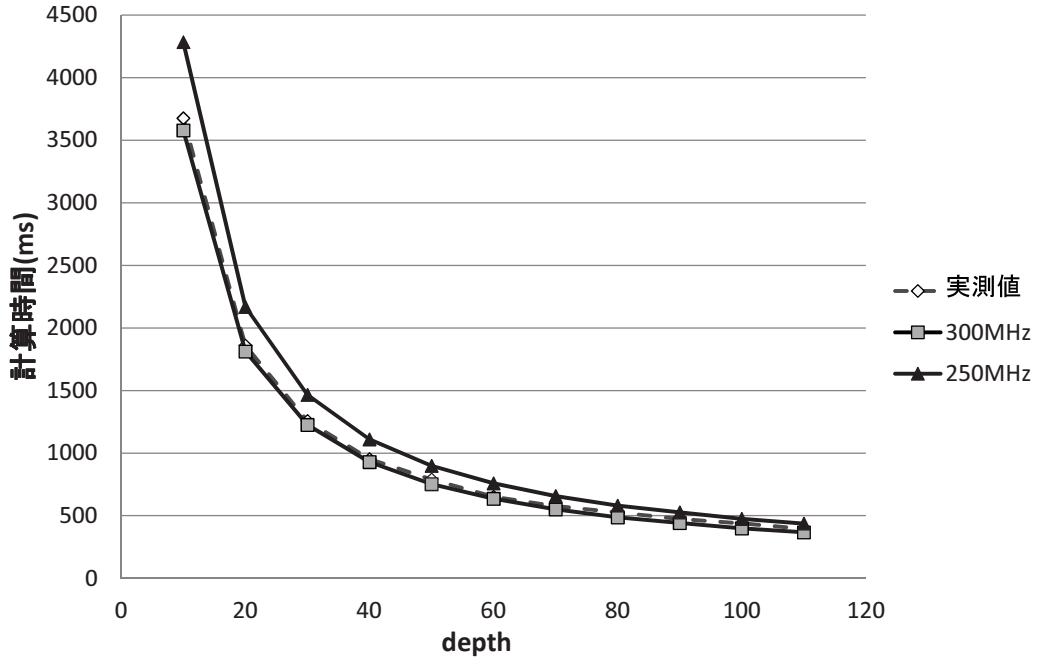


図 3.12: depth ごとのステンシル計算時間の計算値と実測値 ($P_{sten} = 1$)

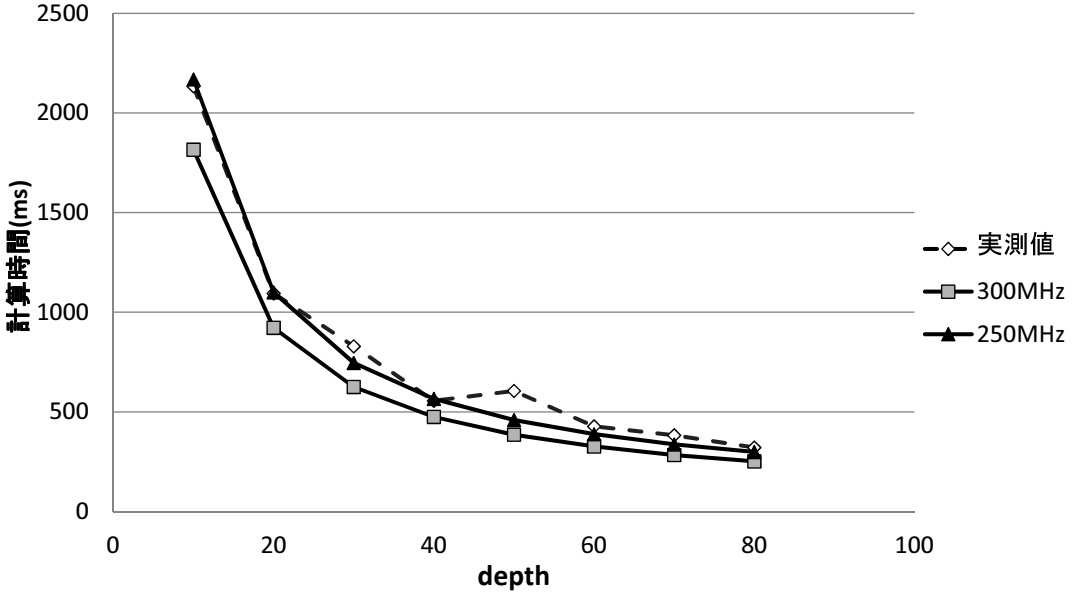


図 3.13: depth ごとのステンシル計算時間の計算値と実測値 ($P_{sten} = 2$)

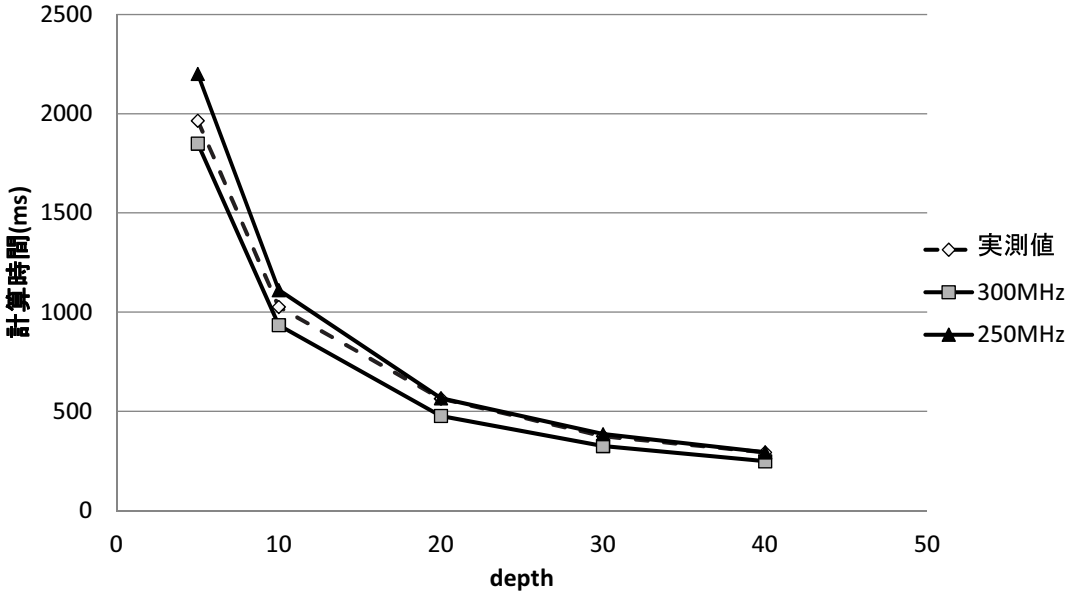


図 3.14: depth ごとのステンシル計算時間の計算値と実測値 ($P_{sten} = 4$)

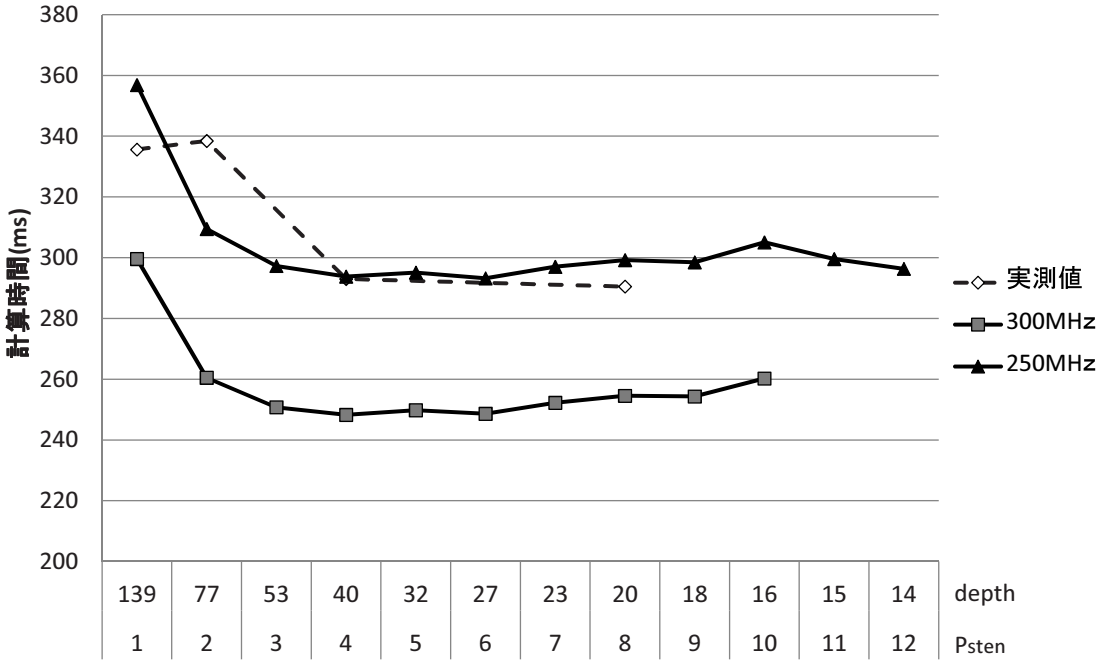


図 3.15: ステンシル演算の並列度に対する計算時間の最小値と実測値の比較

図 3.16: 外部メモリ-FPGA 間の I/O パッド，配線

3.5.2　設計自由度ごとの消費電力の評価

FPGA ベースのステンシル計算アクセラレータの消費電力については，FPGA の消費電力の他に，図 3.16 のような外部メモリとのデータ転送のための I/O パッド，配線の消費電力を考慮する必要がある．外部メモリ-FPGA 間の配線，I/O パッドの消費電力量はデータ転送量に依存して大きくなる．そのため，depth が大きくステンシル演算並列度が小さいアクセラレータについては，外部メモリとのデータ転送量の削減によって消費電力が小さくなっていることが予想される．

図 3.17 に電力測定器の接続法など，消費電力を測定した条件の概要を示す．FPGA ボードが搭載されている PC の消費電力について，電力測定器 (HIOKI AC/DC POWER HiTESTER 3334) を PC の電源ラインに接続して PC 全体の消費電力を計測している．ステンシル計算の処理における消費電力の計測法について，FPGA におけるラプラス方程式の処理実行中の PC の消費電力と，ラプラス方程式の処理を実行していない定常状態の PC の消費電力の差分を求めている．消費電力量は，処理時間と消費電力の積であり，ラプラス方程式の処理で消費される電力エネルギーの量を示す．

表 3.3 にステンシル計算アクセラレータの設計自由度ごとの処理時間，消費電力量の比較結果を示す．表 3.3 より，ステンシル演算の並列度が 1 の場合の消費電力量について，演算並列度が 8 の場合の消費電力量と比べて約 21 %削減されているのが確認できる．この結果は，ステンシル計算アクセラレータの消費電力について，外部メモリとのデータ転送における I/O パッド，配線の消費電力の影響が無視できな

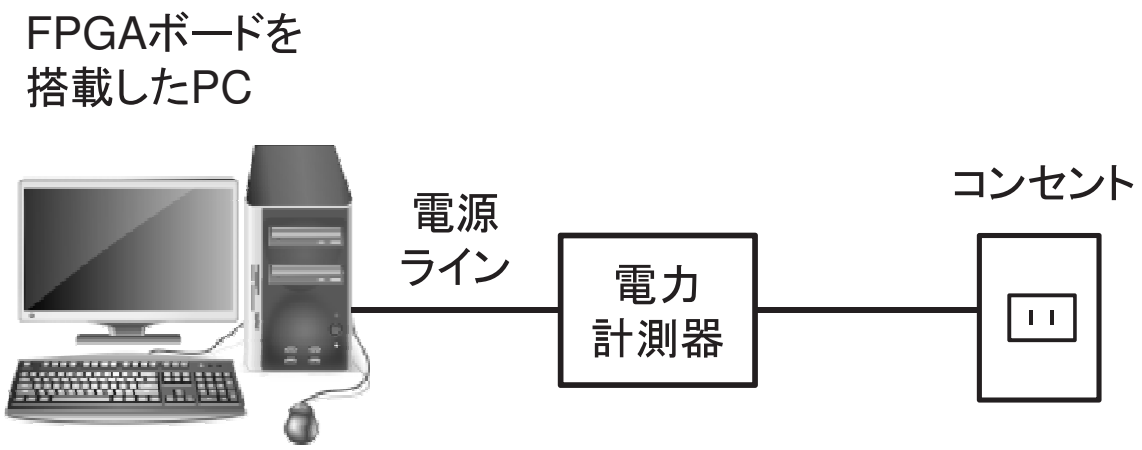


図 3.17: 電力測定器の接続法

表 3.3: FPGA によるステンシル計算アクセラレータの設計自由度ごとの処理時間，消費電力量の比較

P_{sten}	1	2	4	8
depth	139	77	40	20
計算時間 (s)	0.336	0.338	0.293	0.290
消費電力 (W)	21.5	23.1	26.6	31.6
消費電力量 (W・s)	7.2240	7.8078	7.7938	9.1640

いことを示している．結論として，ステンシル計算アクセラレータの設計自由度を選択について，ステンシル計算の処理時間が同程度の場合は，消費電力を小さくするために depth が大きくなるような設計自由度を選択することが最適である考えられる．

本研究で提案する FPGA ベースのステンシル計算アクセラレータと CPU，GPU との性能比較について，ラプラス方程式の処理時間，消費電力，消費電力量を比較評価した．CPU のラプラス方程式の実装では，Intel Xeon E5-1650 v3 に OpenMP を用いて実装している．GPU のラプラス方程式の実装では，nVIDIA Geforce GTX

表 3.4: FPGA アクセラレータと GPU,CPU との処理時間，消費電力量比較

	Stratix V	GTX 960	Xeon E5-1650 v3 (1 thread)	Xeon E5-1650 v3 (6 threads)
計算時間 (s)	0.29	1.02	12.56	2.7
消費電力 (W)	31.6	120.9	28.7	99.1
消費電力量 (W・s)	9.614	123.18	360.472	267.57

960 に CUDA7.5 を用いて実装している．消費電力，消費電力量については，図 3.17 と同様の計測法を用いている．

表 3.4 にラプラス方程式の計算時間と消費電力の比較を示す．FPGA アクセラレータと CPU(6 thread) の比較評価では，FPGA 実装により CPU の約 9 倍の処理の高速化を実現している．さらに，FPGA の消費電力量は CPU の約 96%削減されている．FPGA アクセラレータと GPU の比較評価では，FPGA 実装により GPU の約 3 倍の処理の高速化を実現している．さらに，FPGA の消費電力量は GPU の約 92%削減されている．これらの評価結果から，提案するステンシル計算の FPGA アクセラレータは，CPU，GPU と比較して大幅な処理の高速化と消費電力量の削減を実現していることが確認できる．

3.6　結言

本章では，ステンシル計算の FPGA アクセラレータについて，FPGA 内部で処理するタイムステップ回数 (depth) とタイムステップあたりのステンシル演算の並列度を考慮した最適設計法を提案した．最初に，内部メモリに中間計算結果を格納して外部メモリアクセス回数を削減したステンシル計算アクセラレータについて，OpenCL を用いて設計した．また，設計自由度を最適化するために，処理時間の見積もり式を用いた計算時間の最小値の評価，および消費電力の計測を行った．評価

の結果について，ステンシル演算の並列度が一定であればリソース制約下で depth を大きくすること，計算時間が同程度の場合は，消費電力を小さくするために depth が大きくなるような設計自由度を選択することが最適であることが結論付けられた．CPU，GPU との性能比較では，FPGA に実装したステンシル計算アクセラレータの提案アーキテクチャは CPU，GPU と比較して大幅な処理の高速化と消費電力量の削減を実現していることを確認した．

第4章　簡潔グラフ表現に基づく最短経路探索アクセラレータ

4.1　まえがき

近年のビッグデータ処理の中で，道路ネットワーク，コンピュータネットワーク，ソーシャルネットワークなど，大規模グラフ構造を処理するアプリケーションは様々な場面で用いられている．このような大規模グラフ処理の中で，2点間の最短経路・距離を求める最短経路問題は，ナビゲーションシステム，コンピュータネットワークなど様々な分野に応用されている．

最短経路問題の処理の高速化に関して，これまでのアルゴリズムの計算量削減やデータ構造の改良など，ソフトウェア的なアプローチによる研究が主に行われている．一方で，GPU, FPGAなどのアクセラレータによる最短経路問題の高速化に関する研究は，単純な計算で並列度が大きいワーシャル-フロイド法の高速化が行われている [35, 36]．しかしながら，ダイクストラ法などの最短経路検索アルゴリズムは逐次的な処理を必要とする部分が大きく，複雑な制御フローが要求されるため，GPUのような単純な並列アーキテクチャによる高速化は困難である．現在，大規模な最短経路問題の処理には，マルチコアCPUによるPCクラスタが主に用いられている [37]．しかしながら，大規模なPCクラスタでは，演算および冷却に必要な電力費などの運用コストの増大が大きな問題となっている．大規模グラフにおける最短経路問題の高速かつ低消費電力な処理を実現するために，本章では大規模グラフにおける最短経路問題の処理のためのFPGAアクセラレータのアーキテクチャを提案する．

大規模グラフにおける最短経路問題の処理では，入力グラフのデータ量および最

短経路検索の計算に必要な中間データの記憶量の削減が課題となっている．入力グラフのデータ量について，外部メモリの容量以上のデータ量を持つグラフを処理する場合は，通信帯域が小さい外部ストレージにデータを記憶する必要がある．そのため，外部ストレージとのデータ転送時間が最短経路問題の処理の高速化におけるボトルネックとなる．最短経路検索の中間データの記憶量について，最短経路検索アルゴリズムでは，各ノードにおける暫定的な最短経路・最短距離のデータを更新するために，これらのデータをメモリに記憶する必要がある．中間データのデータ量が内部メモリの容量以上の場合は，外部メモリとのデータ転送がアルゴリズムの処理中に発生してしまうため，処理時間が大きくなる．

これらの問題を解決するために，本章は簡潔グラフ表現を用いたグラフデータ圧縮，および中間データの記憶量削減を実現できる最短経路検索アクセラレータのアーキテクチャを提案する．入力グラフのデータ構造について，高速なデータ処理とコンパクトな記憶容量を両立する簡潔グラフ表現を使用している．これにより，大規模な入力グラフを外部メモリに記憶することが可能になり，外部ストレージを使用する場合と比べてデータ転送時間を削減することができる．また，中間データの記憶量削減について，代表的な最短経路探索アルゴリズムであるダイクストラ法のノードデータのアクセスパターンに注目している．最短経路および最短距離のデータについて，最短経路が確定したノードデータを新規のノードデータに書き換えることにより，最短経路探索に必要なノードデータの記憶量の削減を実現している．

4.2 簡潔グラフ表現に基づくデータ圧縮とFPGAによる簡潔グラフ表現の処理

本節では，簡潔データ構造に基づくグラフデータのデータ圧縮法について説明する．さらに，簡潔グラフ表現を処理するための FPGA によるハードウェアのアーキテクチャについて提案する．

4.2.1 簡潔データ構造

大規模グラフ処理，テキストデータ処理に代表されるようなビッグデータ応用アプリケーションの処理時間について，プロセッサとストレージ間のデータ通信量が大きくなる．そのため，ストレージとのデータ転送時間がビッグデータ応用アプリケーションの処理時間の大部分を占めているケースが多く存在している．ビッグデータ処理を高速化する方法の 1 つとして，データ圧縮によりストレージ間のデータ転送量を削減する手法があげられる．データを圧縮する手法について，zip,tar などのファイル圧縮法がコンピュータ上のファイルシステムに広く用いられている．しかしながら，圧縮したデータを解凍するための計算時間が大きいこと，圧縮したデータのランダムアクセスが出来ないことなどを考慮にいれると，ビッグデータ処理の高速化のためのデータ圧縮法には適していないと考えられる．

本研究では，効率的な大規模グラフ処理を実現するために，簡潔データ構造によるグラフデータの圧縮を利用している．簡潔データ構造は，高速なデータ処理とコンパクトな記憶容量を両立できるデータ構造として提案されている [38, 39]．通常のデータ圧縮とは異なり，簡潔データ構造は圧縮したデータを展開する操作を行わずにランダムアクセスなどのデータ処理を実行することができる．ビッグデータ処理の需要が大きい近年では，簡潔データ構造を利用したビッグデータ処理がグラフ構造・木構造の解析，テキストデータ検索，遺伝子データ解析など幅広い分野で応用されている．

簡潔データ構造における操作について，rank，select と呼ばれる操作が主に用いられている．rank 操作はデータ内の特定の範囲における文字数をカウントするために行われる操作であり，文字列 B において位置 x までの文字 q の数をカウントする操作は $rank_q(B, x)$ と表現される．select 操作は特定のデータを検索するため行われる操作であり，文字列 B において x 番目の文字 q が出現する位置を返す操作は $select_q(B, x)$ と表現される．簡潔データ構造では，これらの 2 つの操作を定数時間 $O(1)$ で処理することを保証するために，辞書データなどの補助データを小さい記憶容量で実装して効率的な処理を実現している．

4.2.2　簡潔グラフ表現

点（ノード）と辺（エッジ）で構成されるグラフデータをメモリ上で表現する方法として，隣接行列と隣接リストが主に使用されている．

図 4.1 のグラフ構造を隣接行列で表現すると，図 4.2(a) のようになる．隣接行列の要素はエッジの重みであり，隣接していないノード間のエッジの重みは無限大（INF）で定義している．隣接行列はデータアクセスが容易なこと，グラフのエッジの数が大きい場合に効率的なデータ記憶ができることが利点としてあげられる．しかしながら，エッジの数が小さいグラフに対しては，接続されていないノード間のエッジの重みを示すためのメモリ領域が必要になるため，記憶量のオーバヘッドが大きくなってしまう．

図 4.1 のグラフ構造を隣接リストで表現すると，図 4.2(b) のようになる．隣接リストは，ノードごとに隣接ノード番号とエッジの長さをリスト化してそれぞれ格納している．隣接リストはエッジの分のみデータを記憶するため，ノードの数が小さいグラフに対しては小さい記憶量に抑えられることが利点としてあげられる．しかしながら，特定のエッジデータにアクセスする場合については，ノードに接続しているエッジの長さがそれぞれ異なるため，隣接行列よりも複雑な処理が必要になる．隣接行列と隣接リストの選択について，グラフの密度が大きい場合は隣接行列が，グ

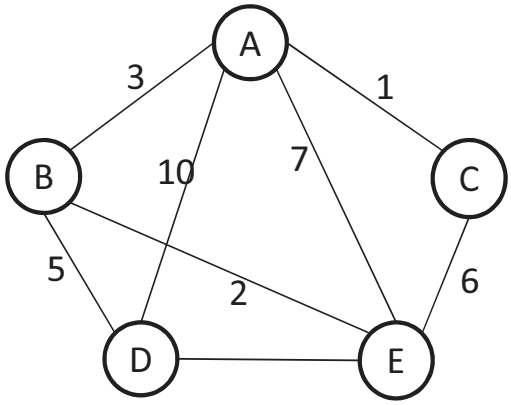


図 4.1: 入力グラフ

	A	B	C	D	E
A	0	3	1	10	7
B	3	0	INF	5	2
C	1	INF	0	INF	6
D	10	5	INF	0	8
E	7	2	6	8	0

(a) 隣接行列

ノード	隣接ノード(エッジの長さ)
A	B(3),C(1),D(10),E(7)
B	A(3), D(5),E(2)
C	A(1),E(6)
D	A(10),B(5), E(8)
E	A(7),B(2),C(6),D(8)

(b) 隣接リスト

図 4.2: グラフの表現法

ラフの密度が小さい場合は隣接リストが主に用いられる．道路ネットワーク，ソーシャルネットワークなどの実世界に存在する大規模グラフの密度は，完全グラフと比べると非常に小さいため，本研究は隣接リストに基づく簡潔グラフ表現を利用している．

図 4.2(b) の隣接リストをメモリに記憶する場合について， 図 4.3 に示されるような配列によって記憶される．隣接ノードとエッジの重みのデータがエッジ数の要素を含む配列になっている．また，特定のノードに接続されているエッジのデータに

アクセスために，隣接ノードの集合の区切りの位置を示す Index 情報がノード数の要素を含む配列として別に記憶されている．グラフの密度が小さいい場合について，エッジ数に対するノード数の割合が大きいため，Index 情報の情報量が大きくなる．

図 4.2(b) の隣接リストを，簡潔グラフ表現を用いてメモリに記憶した場合は，図 4.4 のようになる．図 4.3 との違いについて，隣接ノードの集合の区切りの位置を示す Index 情報を，0 と 1 のビット情報の集合であるビットベクトルで表現している．このビットベクトルでは，隣接ノードの区切りになる位置には 1 が，それ以外の位置では 0 が格納されている．特定のノード x に接続されているエッジのデータにアクセスする場合は，頂点データの集合の区切りの位置を示すビットベクトル B に対して，セレクト操作 $select_1(B, x-1)$ と $select_1(B, x)$ を計算することにより，図 4.3 の Index 情報を用いた場合と同様の処理を行うことができる．また，図 4.3 の Index 情報は，図 4.4 のビットベクトルの select の計算結果をテーブル化したものに等しくなっている．

ノード数 n, エッジ数 m のグラフについて，図 4.3 の配列による隣接リストの表現と図 4.4 の簡潔グラフ表現のデータ量について比較する．隣接ノードとエッジの重みのデータを含む配列については共通のため，Index を示す配列とビットベクトルのデータ量を比較する必要がある．Index 情報はエッジのアドレスを示すデータがノードの分だけ必要であるため，Index の配列にデータ量は $n \times log_2 m$ bit になる．一方で，簡潔グラフ表現におけるビットベクトルの長さはエッジ数に等しいため，ビットベクトルのデータ量は m bit になる．簡潔データ表現によるグラフデータ圧縮が有効になる条件については，ノードあたりのエッジの本数の平均値 $(\frac{m}{n})$ が式 (4.1) を満たす必要がある．

$$\frac{m}{n} \geq log_2 m \tag{4.1}$$

図 4.5 に Index を示す配列と簡潔グラフ表現におけるビットベクトルのデータ量

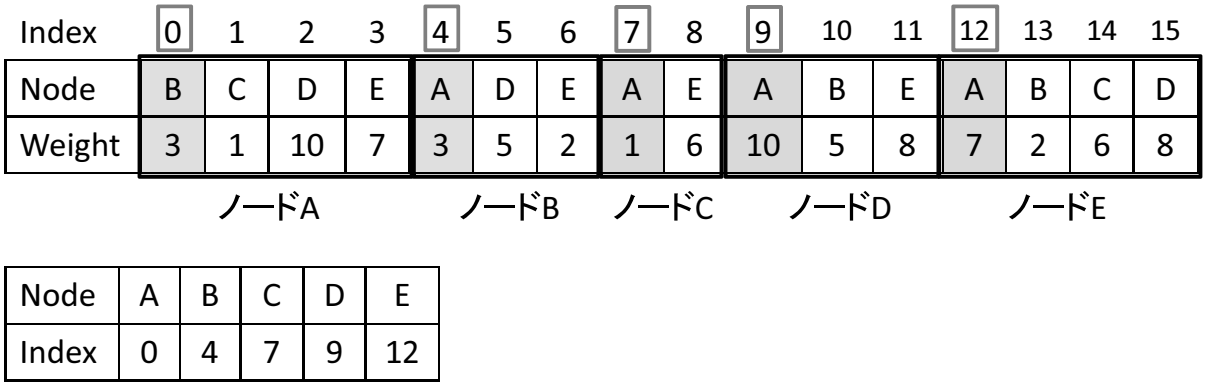


図 4.3: 配列による隣接リストの表現

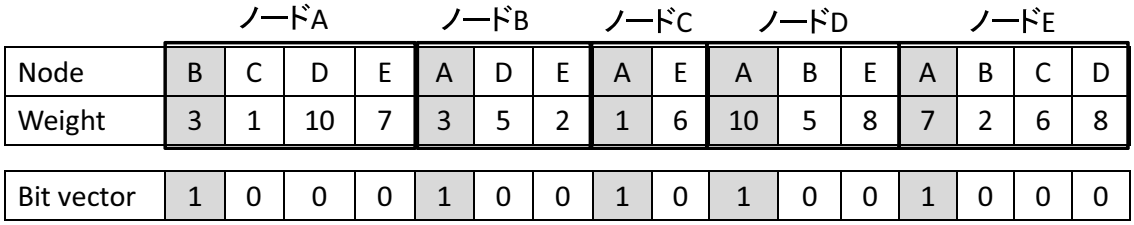


図 4.4: 簡潔グラフ表現

を比較したグラフの概形を示す．実世界に存在する大規模グラフ構造のエッジ数を 100 万から 10 億程度と仮定すると，$log_2 m$ の値は 20 から 30 程度になる．簡潔グラフ表現におけるデータ圧縮が有効なグラフ構造について，道路ネットワークのグラフ構造はノードあたりのエッジの本数が非常に小さいグラフ．そのため，図 4.4 の簡潔グラフ表現のビットベクトルのデータ量のほうが少なくなり，簡潔グラフ表現によるデータ圧縮が有効になる．一方で，ソーシャルネットワークのグラフ構造の中には，ノードあたりのエッジの本数の平均値が 100 以上になるものが存在する．このようなエッジの本数が大きいグラフの場合は，図 4.3 のように Index 情報を直接記憶したほうが簡潔グラフ表現よりもデータ量が少なくなる．

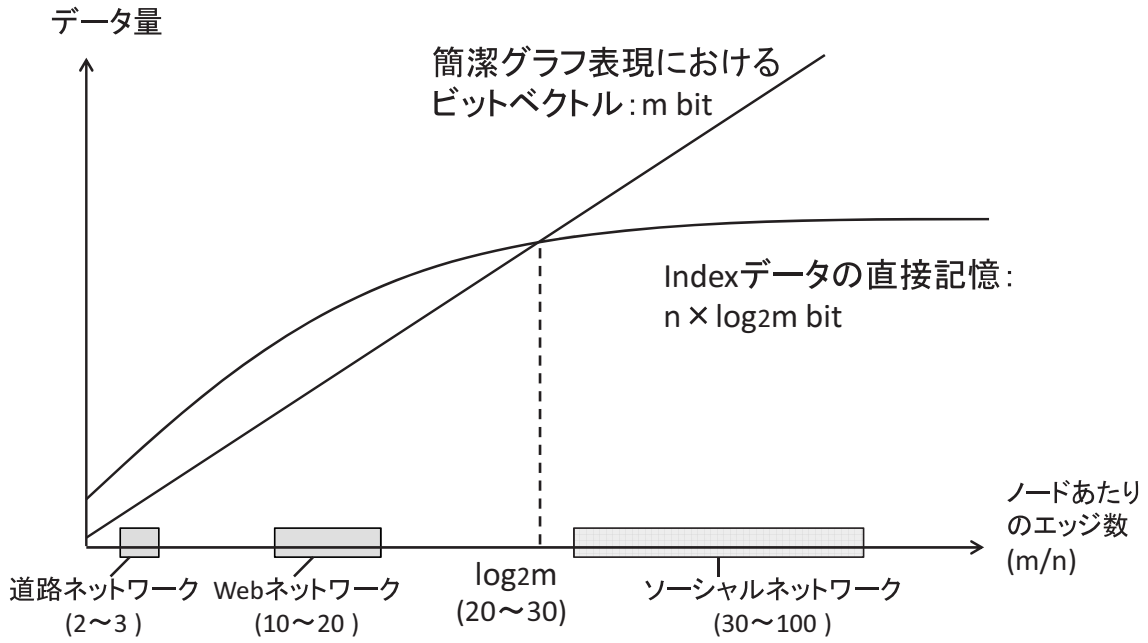


図 4.5: Index を示す配列とビットベクトルのデータ量

4.2.3　FPGA による簡潔グラフ表現の処理

簡潔グラフ表現で記憶されたグラフデータに高速にアクセスするためには，図 4.4 における頂点データの集合の区切りの位置を示すビットベクトルの select 計算を高速処理する必要がある．ビットベクトルの select 操作を補助データ無しで処理する場合は，ビットベクトルの先頭から 1 の数を逐次的に数える必要があるため，効率的な処理が難しくなる．そのため，select 操作をハードウェアで効率よく処理するための補助データを小さい記憶容量で実装する必要がある．

select 操作の補助データの実装における先行研究では，ビットベクトルを大ブロックと小ブロックに階層化している select の索引付きデータ構造が提案されている [40]．しかしながら，計算に必要な補助データのデータ量が大きいこと，ビットベクトルの疎密にあわせてブロックの分割サイズを変更するなど複雑な制御が必要なことなどから，ハードウェア実装においては効率的な処理を行うことは難しい．

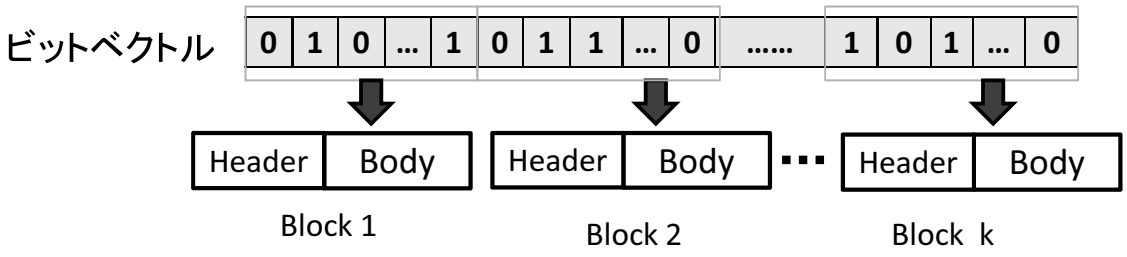


図 4.6: ビットベクトルのブロック化

本研究は，ハードウェア処理を前提とした簡潔データ構造の補助データ実装に関する先行研究 [41] における rank 操作向け補助データの実装法を select 操作に応用している．先行研究 [41] では，圧縮されたテキストデータにおける rank 操作の計算のための補助データとして，図 4.6 のようなデータ構造を実装している．図 4.6 のデータ構造では，ノード数に対応するビットベクトルを一定数のノードに対するビットベクトルの情報が含まれている block に分割している．block データは header と body で構成される．header は先頭のデータまでの select の数値を格納している．body は分割したビットベクトルを格納している．select 操作の計算は，対応するエッジ情報の集合の位置を示すビットベクトルが含まれている block を抽出して，header と body の値からハードウェアによる並列計算で実行される．

block データから select 操作を処理するハードウェアについて，入力される block のデータ長に合わせて設計する必要がある．先行研究 [41] の rank 操作の計算では，block の大きさは一定で，ハードウェアによる効率的な rank 操作の計算を実現することが容易である．一方で，簡潔グラフ表現処理における select 操作において，効率的なデータアクセスのために block ごとのノードの集合の数を等しくする必要がある．この場合，body の長さは一定数のノードごとに分割したグラフのエッジ数に等しくなるため，図 4.7 のように body の長さが block によって異なる．そのため，select 操作を処理するハードウェアを設計する場合については，block の最大の長さに合わせて設計する必要が有るため，ハードウェアのリソース増大，稼働率の低下

などの問題が発生してしまう．

本研究では block ごとの長さを揃えるために，図 4.8 のように可変ワード長へ拡大した簡潔データ構造を提案する．可変ワード長へ拡大した簡潔データ構造では，header と body の他に，body の分割の有無を判定するフラグ，および分割した body の格納場所を示すポインタを追加している．さらに，分割した body を記憶するためのマルチワード専用の領域を確保している．body が一定長よりも長い block は block データを分割して，マルチワード専用の領域に分割した body と分割した部分までの select の値を格納する．

block の大きさについては，本研究では外部メモリとの通信ビット幅によって決定している．header に対して block の長さが十分大きい場合は，補助データである header によるメモリ記憶量の増加量を抑えることができる．

図 4.9 に簡潔グラフ表現を処理するユニットのハードウェアの概要を示す．簡潔グラフ処理ユニットは，select 計算ユニットおよび外部メモリコントローラによって構成されている．block のデータが外部メモリからメモリコントローラによって，header と body の値から select の値を計算するユニットに入力される．最短経路検索ユニットが特定のノードに接続したエッジ情報ににアクセスする場合について，select 計算ユニットによって計算された select の値からエッジ情報の格納アドレスが算出される．その後，最短経路検索の計算に必要な隣接ノード番号，辺の長さの情報が外部メモリから最短経路検索ユニット転送される．

図 4.10 に select を計算するユニットのハードウェアの概要を示す．最初に，外部メモリから転送された body を分割して，1 の数の合計を求める popcount[42] の計算をハードウェアによって並列処理する．次に，popcount の値から body の中で検索するノード番号に対応する情報が含まれている部分を抽出する．最終的に，抽出された部分のビットベクトルを select の値を計算したテーブルに入力し，header の値と合計することで select の値が導出される．

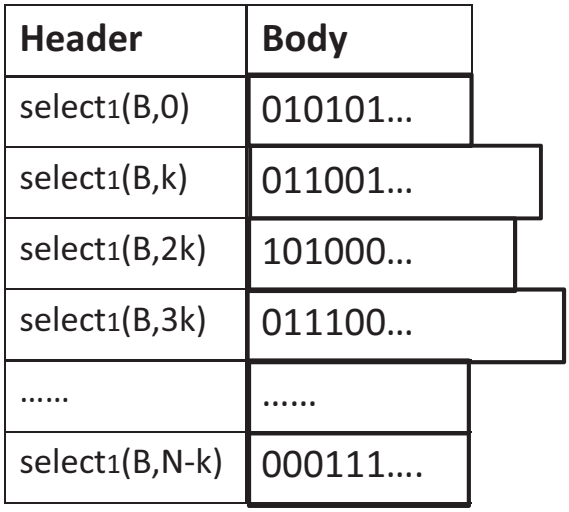


図 4.7: Select のデータ構造の問題点

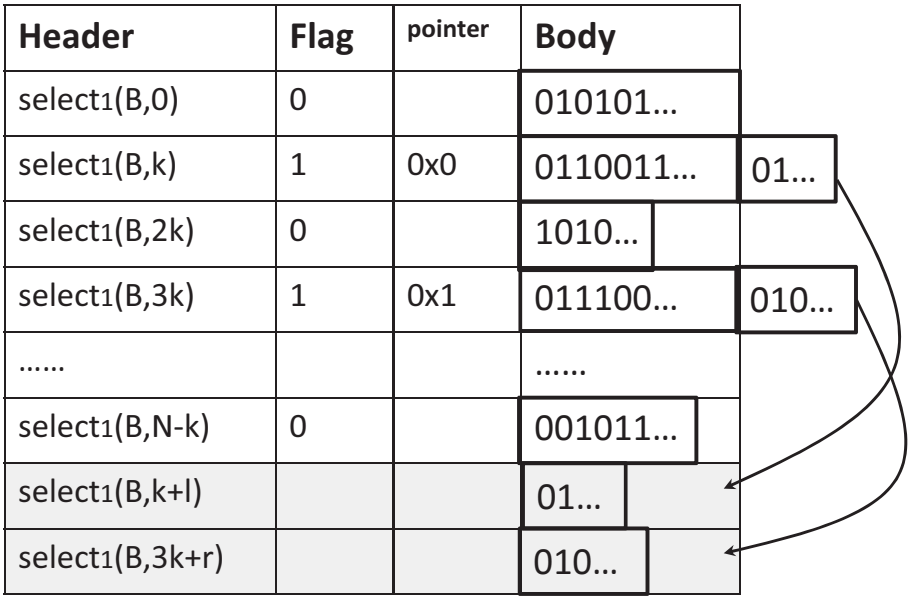


図 4.8: 可変ワード長へ拡大した簡潔データ構造

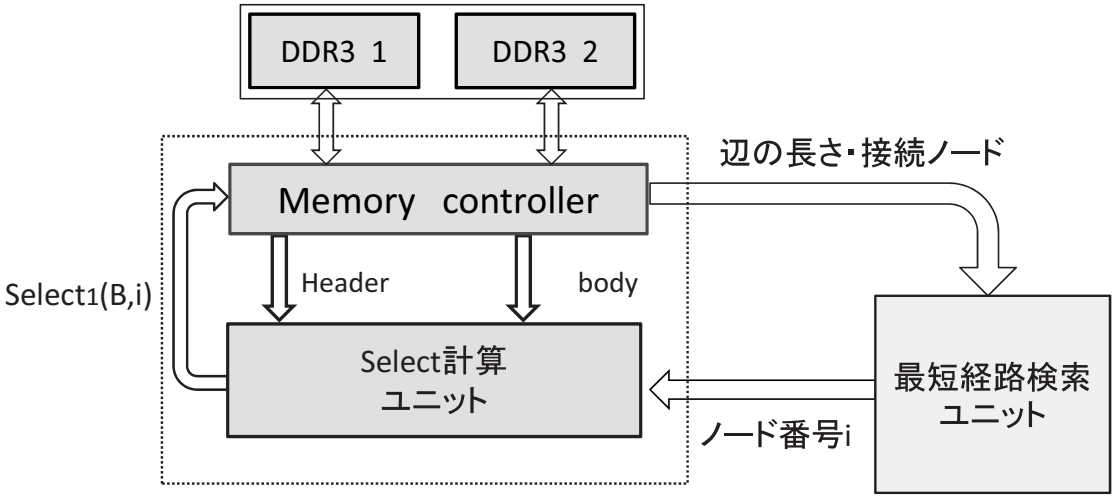


図 4.9: 簡潔グラフ処理ユニット

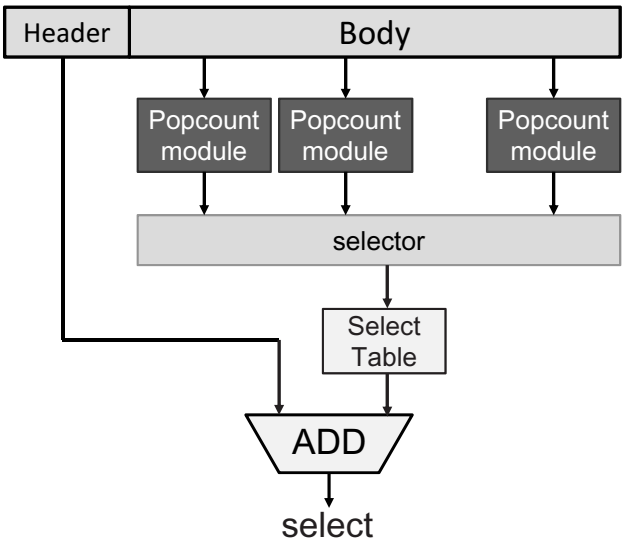


図 4.10: select 計算ユニット

4.2.4 簡潔グラフ表現によるデータ圧縮の評価

グラフデータの圧縮効果を見るために，アメリカの道路ネットワーク [43] について，図 4.3 の隣接リストの配列による表現と，図 4.4 の簡潔グラフ表現のデータ量を比較した．簡潔グラフ表現のデータ量については，図 4.8 に示されるような可変ワード長へ拡大したビットベクトルの簡潔データ構造のデータ量を示している．図 4.8 における block 長は 512bit，マルチワードに分割されたブロックの割合は 20%と仮定している．

図 4.11 にアメリカの道路ネットワークについて，Index データの配列による表現と簡潔グラフ表現のデータ量の比較を示す．アメリカ全土の道路ネットワーク（約 2395 万ノード，5833 万エッジ）に対して，簡潔グラフ表現におけるビットベクトルの簡潔データ構造のデータ量について，隣接リストの Index データの配列のデータ量に比べて約 88%削減されていることが確認された．

図 4.12 にエッジの重みと接続ノードを含むグラフデータ全体のデータ量の比較を示す．この場合について，圧縮された Index データのデータ量に対して，圧縮されていないエッジの重みと接続ノードを示すデータ量の割合が大きいため，簡潔グラフ表現のデータ量の削減率が約 18%にとどまっていることが確認された．そのため，今後の課題として，エッジの重みと接続ノードの配列データを含むグラフデータの圧縮率を向上させる方法を検討する必要がある．エッジの重みと接続ノードを示すデータについては，図 4.3 のようにそれぞれ配列として格納されているため，ノード番号またはエッジの重みを並べたテキストデータとして扱うことが可能である．そのため，先行研究 [44] で提案されているようなハードウェア処理向けのテキストデータ圧縮法が応用できると考えられる．

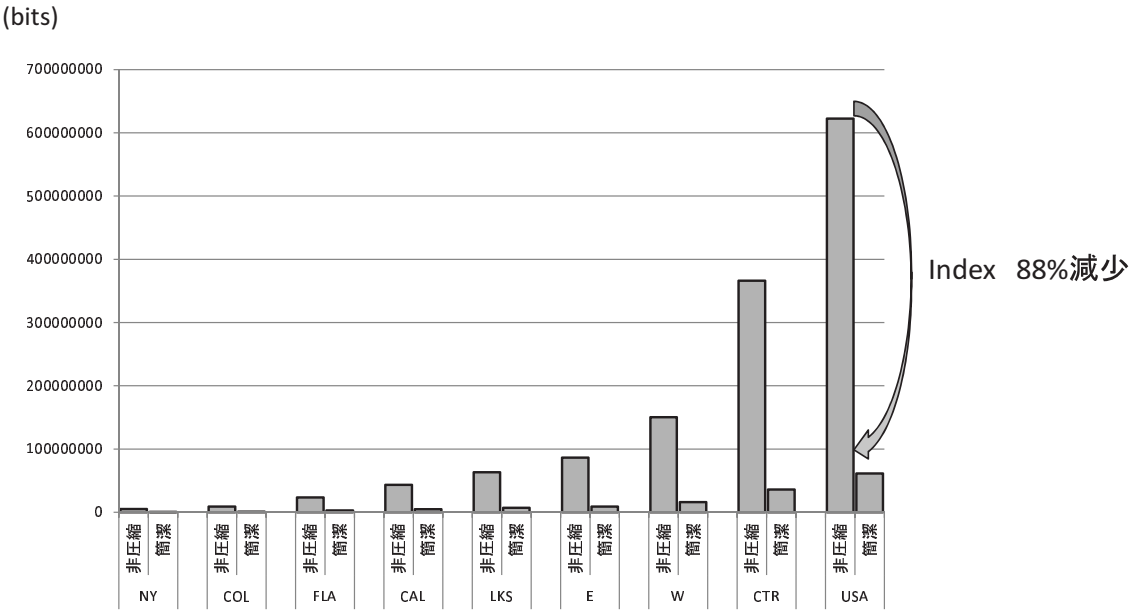


図 4.11: Index のデータ量

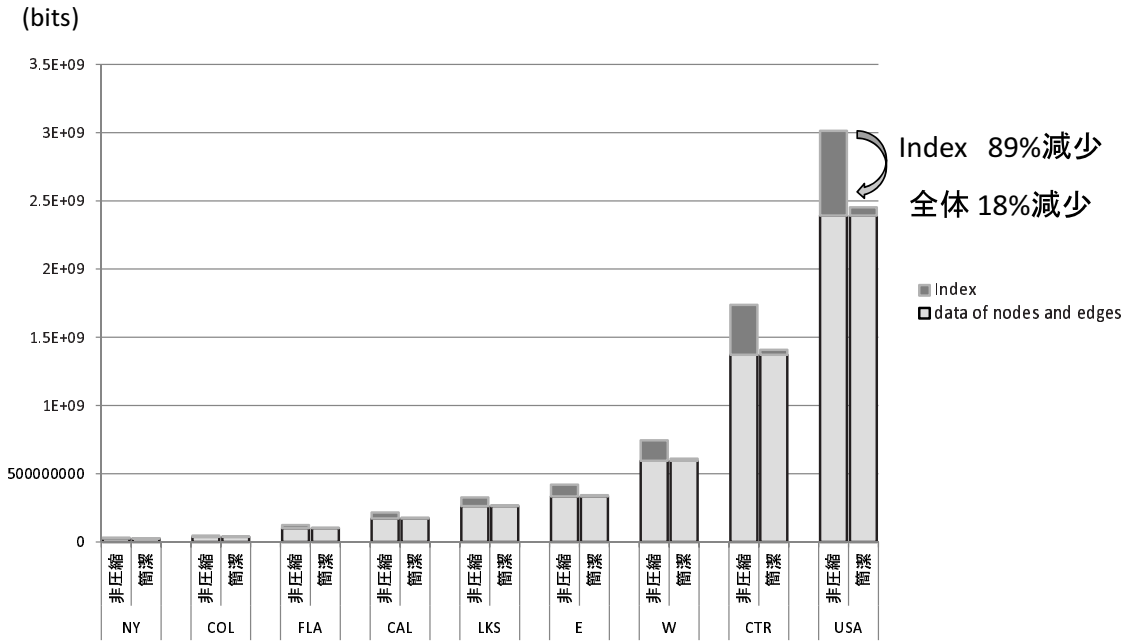


図 4.12: グラフ全体のデータ量

4.3 中間データ記憶量を削減した最短経路問題処理ユニット

本節では，ダイクストラ法を用いた最短経路問題処理を高速化するFPGAアクセラレータの設計について，暫定的な距離データ等のダイクストラ法の処理における中間結果の記憶量の削減に注目する．ダイクストラ法のノードデータアクセスの特性について説明し，メモリ内部のノードデータの書き換えによるデータ記憶量を削減する手法について説明する．また，メモリ内部のノードデータの書き換えを実現可能な最短経路問題処理ユニットのアーキテクチャを提案する．

4.3.1 ダイクストラ法

グラフ構造の処理の中で，2点間の最短経路・距離を求める最短経路問題は，ナビゲーションシステム，コンピュータネットワークなど様々な分野に応用されている．大規模グラフにおける最短経路問題を高速かつ低消費電力で処理するためには，最短経路検索に特化したアクセラレータをFPGAに実装することが有用である．

最短経路問題には，一つの始点ノードから全ノードにおける最短経路を求める単一始点最短経路問題（SSSP），全ノード間における最短経路を求める全点対最短経路問題(APSP)などがあげられる．SSSPを解くアルゴリズムとして，ダイクストラ法[45]，ベルマン-フォード法[46]などが提案されている．APSPを解くアルゴリズムとして，ワーシャル-フロイド法[47]などが提案されている．これらの最短経路検索アルゴリズムの中で，ダイクストラ法は計算量が小さくシンプルな演算のみで実装できるため，多くのアプリケーションで利用されている．

ダイクストラ法による最短経路検索アクセラレータのFPGA実装に関連する先行研究について，[48, 49, 50]など，複数の先行研究が報告されている．これらの先行研究では，暫定距離更新，最小距離検索などのダイクストラ法の演算タスクに対応した並列回路の設計に注目したFPGA実装が行われている．しかしながら，先行研究で提案されたアーキテクチャは，大規模なグラフの処理におけるメモリアクセス

のオーバーヘッドの増加，使用リソース量などが考慮されていない．そのため，本研究では，大規模なグラフ処理に対応できるようにするために，ダイクストラ法における中間データ記憶量の削減を実装した最短経路問題処理のためのアーキテクチャを提案する．

ダイクストラ法による一点対全点問題の処理は次のようなステップで実行される．

Step1: 入力グラフに対して，始点ノードの暫定的な距離を 0 に，それ以外のノードの暫定的な距離を無限大に設定する

Step2: 最短経路, 距離が未確定のノードに対して，暫定的な最短距離を有するノードを検索する．最短距離を有するノードを現在ノードに設定し，始点からの最短距離・経路を確定する

Step3: Step2 で選択された現在ノードに接続されている最短経路, 距離が未確定のノードまでのエッジの重みと現在ノードまでの最小距離を足した数値について，現在ノードに接続している未確定ノードにおける暫定的な距離の数値を比較する．エッジの重みと現在ノードまでの最短距離の合計のほうが暫定的な距離よりも小さい場合は，このノードにおける暫定的な距離，経路を更新する

Step4: 全てのノードまでの距離が確定するまで Step2, Step3 を繰り返す

Step2 と Step3 の処理では，最短距離が未確定のノードにおける暫定的な距離データを参照するため，効率的な処理のためにはそれらのデータを内部メモリに記憶する必要がある．しかしながら，FPGA 上の内部メモリの容量は限られているため，暫定的な距離データの記憶量を削減することが重要になる．そのため，ダイクストラ法のノードデータのアクセス範囲に注目したデータ書き換えによる記憶量削減に

ついて提案する．

図 4.13 に示されるような場合のダイクストラ法の処理に必要なノードデータの領域について，現在ノード (C) および現在ノードに隣接または最短距離が確定しているノードに隣接した最短距離が未確定のノード (D,E) のデータのみアクセスされる．図 4.14 のようにノード D までの最短経路が確定して現在ノードが移動した場合について，現在ノード (D) および現在ノードに隣接または最短距離が確定しているノードに隣接した最短距離が未確定のノード (E,F) のデータのみアクセスされる．このように，現在ノードが移動した場合のダイクストラ法の処理について，前回の現在ノードデータ (C) は今後のダイクストラ法の処理で使用されないことが確認できる．また，新しく現在ノードに隣接したノードのデータ (F) へのアクセスが必要になるため，メモリ領域を新たに確保する必要がある．

そのため，現在ノードにおける距離のデータを専用レジスタ領域に記憶して，新たな現在ノードの隣接したノードのデータを前回のノードデータに上書きすることが可能になる．最短経路検索アクセラレータにおける具体的なノードデータの上書き方法については，4.3.2 章でダイクストラ法の処理の流れと合わせて説明する．

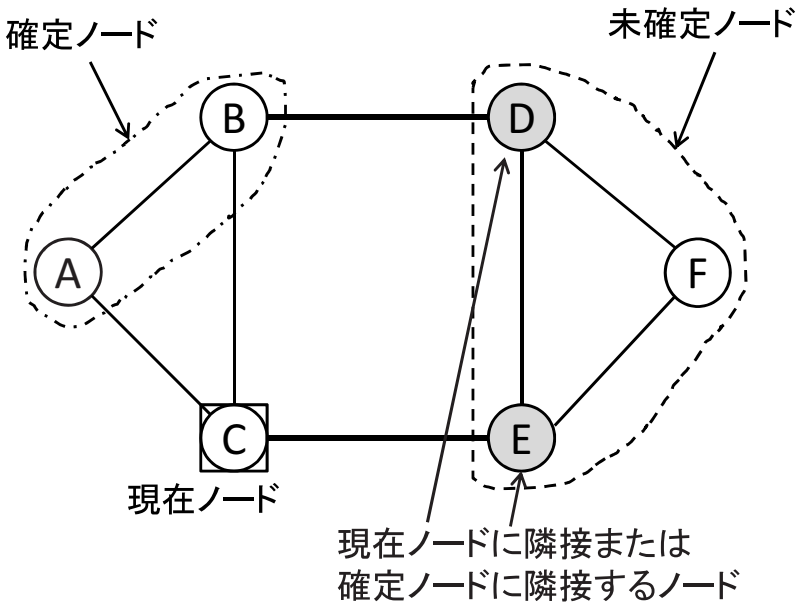


図 4.13: 現在ノード (C) と現在ノードまたは最短距離が確定したノードに隣接する未確定ノード (D,E)

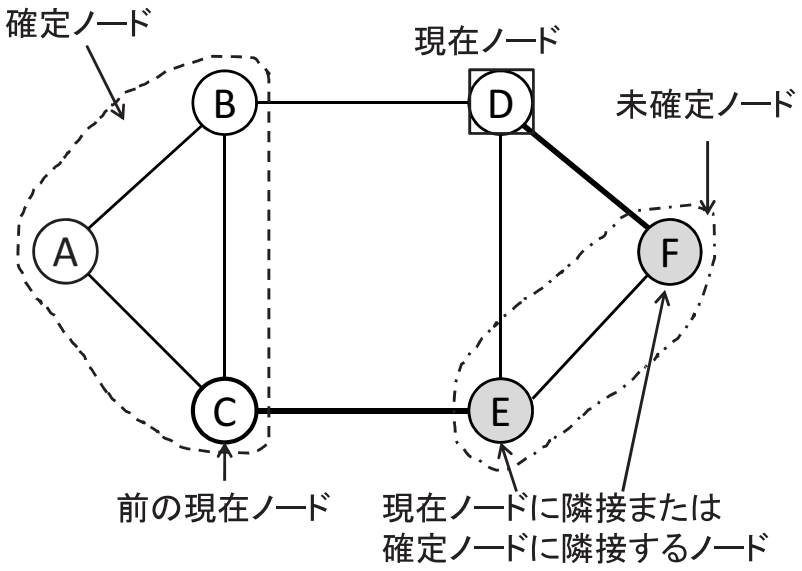


図 4.14: 現在ノード (D) と現在ノードまたは最短距離が確定したノードに隣接する未確定ノード (E,F)

4.3.2 最短経路検索ユニットのアーキテクチャ

ダイクストラ法に基づく最短経路検索ユニットのFPGA実装について，4.3.1章で説明したようなメモリ使用量削減手法の実現の他に，ダイクストラ法の演算処理の並列化について考慮する必要がある．4.3.1章で説明したダイクストラ法のステップごとの処理の中で，Step2における最短距離の検索処理，およびStep3の現在ノードに接続している未確定ノード番号の検索処理の演算は並列化することができる．一方で，2つ以上のノードにおける最短経路を同時に確定させることは不可能であるため，現在地ノードの更新は逐次的に処理する必要がある．

図4.15最短経路検索ユニットのアーキテクチャを示す．このアーキテクチャはダイクストラ法における距離の更新，最小距離の検索を並列処理するためのPE，PEの計算結果から最短経路を持つノード番号と距離データを選択するためのセレクタ，現在ノードの最短距離，ノード番号を格納する現在ノードレジスタなどで構成されている．現在ノード番号を簡潔グラフ構造処理ユニットに入力することにより，外部メモリ内の簡潔グラフ構造から最短経路検索に必要な隣接ノード番号，エッジ重みのデータを取得することができる．

図4.16にPEとノードメモリのアーキテクチャを示す．ノードメモリには，暫定的な距離が設定されているノード番号と暫定距離，最短経路における1つ前のノードの情報が記憶されている．PEには，メモリ内のノード番号のアドレスを検索するモジュール，ノードメモリ内の最短経路を検索するモジュール，現在ノードまでの最短距離とエッジの重みの合計および暫定距離を比較して，合計の方が小さい場合は暫定距離を更新するモジュールが搭載されている．PE，ノードメモリはFPGAに並列的に実装されているため，最短距離の検索処理，および現在ノードに接続している未確定ノード番号の検索処理については並列計算による高速な処理が可能となっている．

暫定距離の更新処理では，最初に図4.17のようにノード番号検索モジュールによってPE間で並列処理が行われる．この場合はノード番号8の暫定距離データが記憶

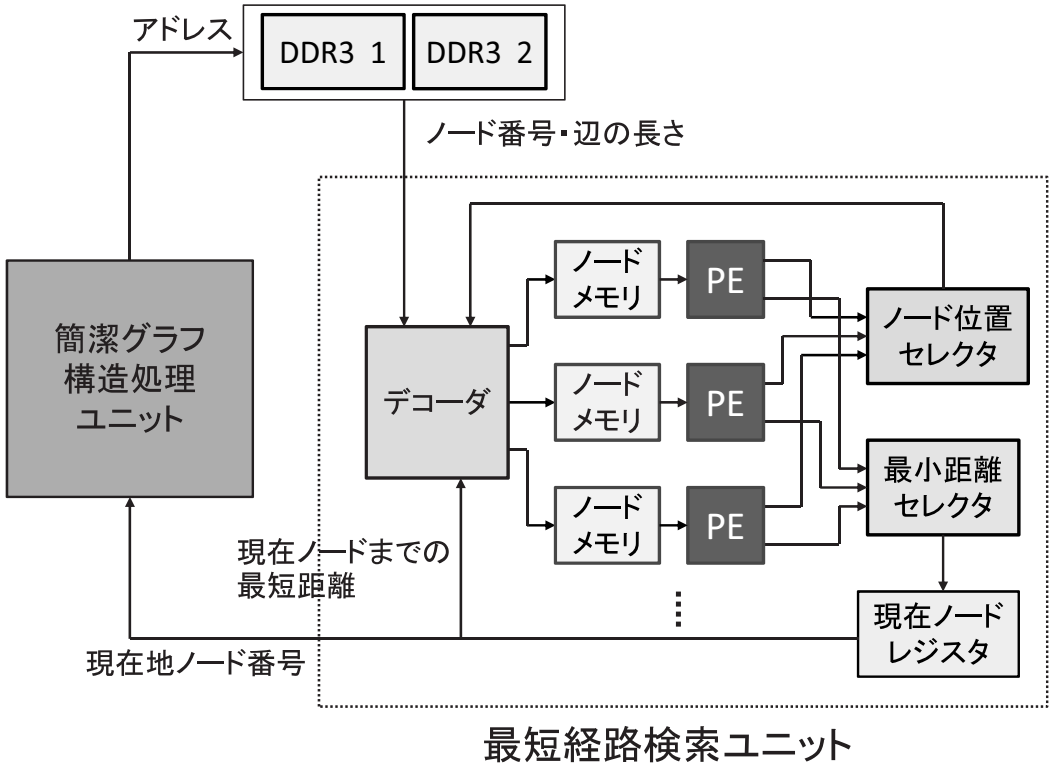


図 4.15: アーキテクチャの概要

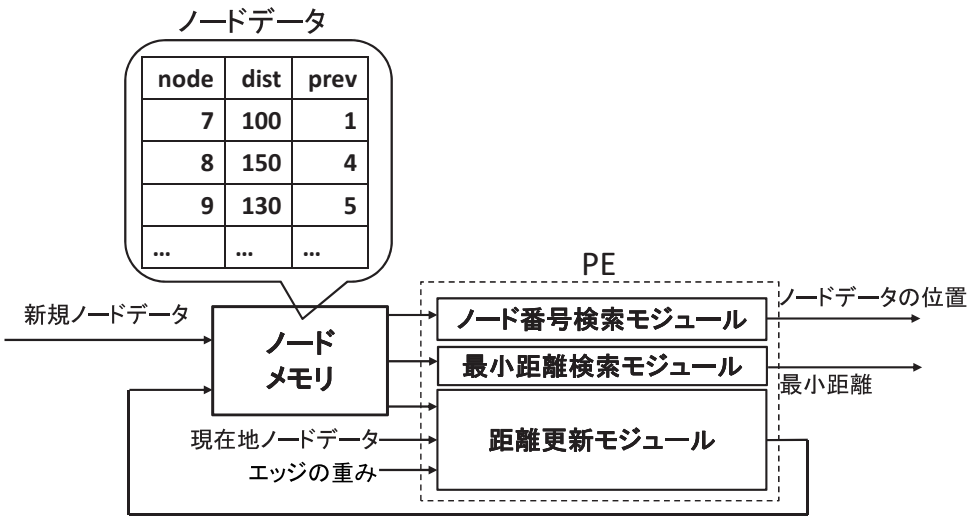


図 4.16: PE・ノードメモリのアーキテクチャ

されているノードメモリ，アドレスが検索される．

次に，ノード 8 における暫定的な距離データの更新について，現在ノード 6 までの最短距離とノード 6-8 間のエッジの重みの合計との比較が，図 4.18 のように実行される．図 4.18 の場合は，現在ノード 6 までの最短距離とノード 6 とノード 8 を接続しているエッジの重みの合計の値のほうが小さいため，ノード 8 における暫定距離，最短経路において前に接続しているノード番号のデータが更新される．

現在ノードにおける最短距離の確定について，ノードメモリ内の全暫定距離の最小値の検索が最短経路検索モジュールによって行われる．図 4.19 では，このノードメモリ内ではノード 7 が最小距離を有している．全ノードメモリ内でノード 7 が最小距離を有する場合はノード 7 が現在ノードに選択されるため，ノード番号 7 とノード 7 における暫定距離データが現在ノードレジスタに記憶される．

これらのデータが現在ノードレジスタに記憶された後は，図 4.20 のように新たに暫定的な距離と経路が設定された最初の未確定ノードデータを現在ノードの領域に上書きすることができる．2 番目以降の未確定ノードデータについては，ノードメモリ内の空いている領域に記憶される．

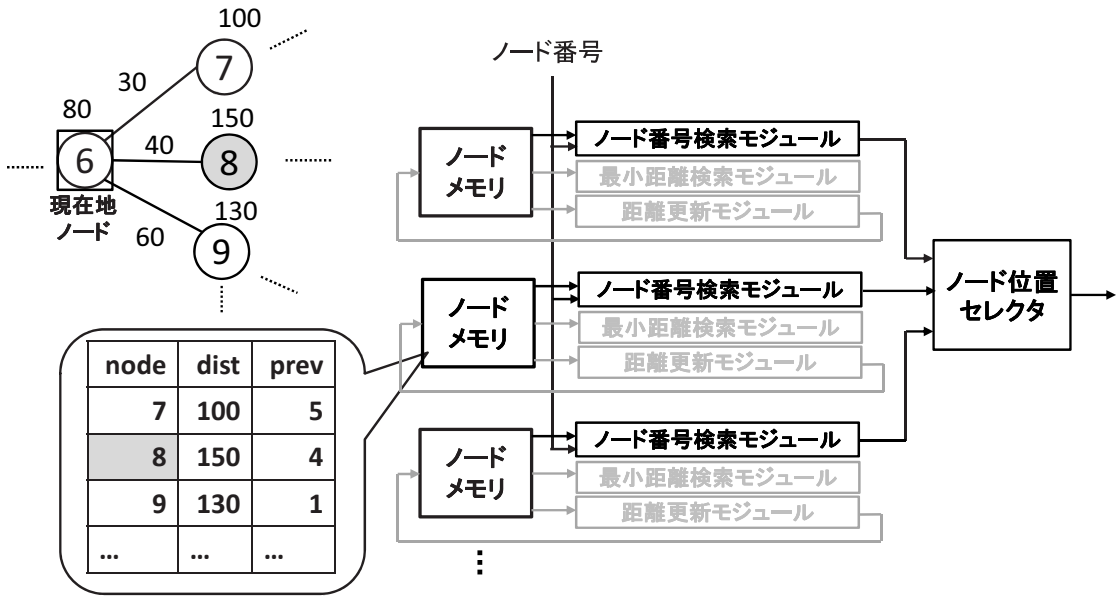


図 4.17: ノード番号の検索

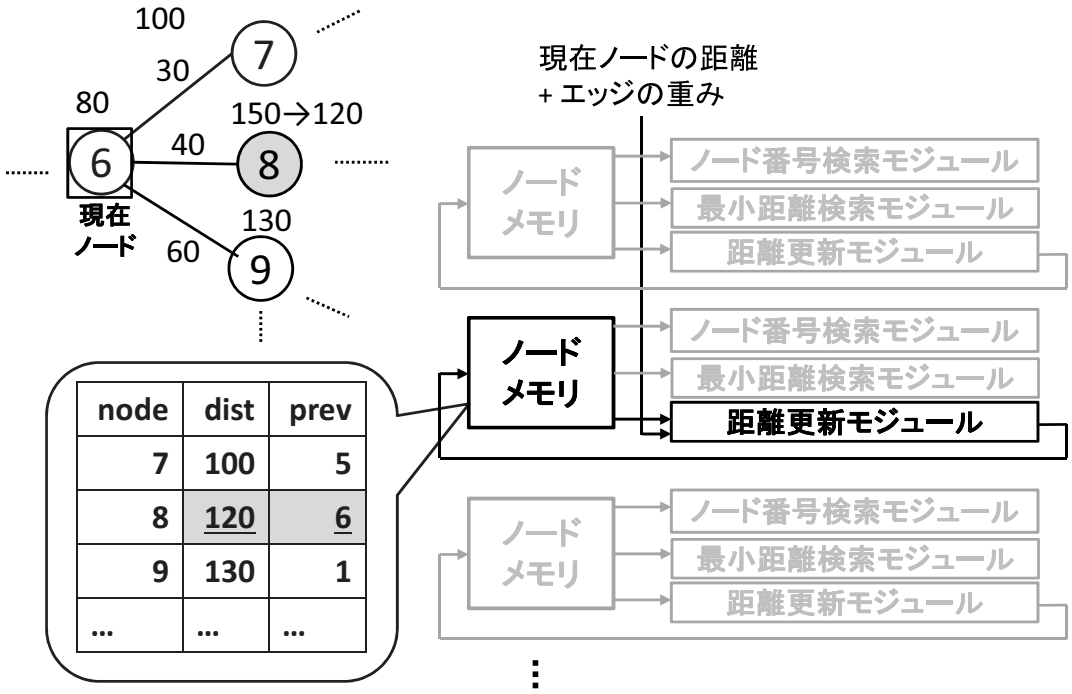


図 4.18: 暫定距離の更新

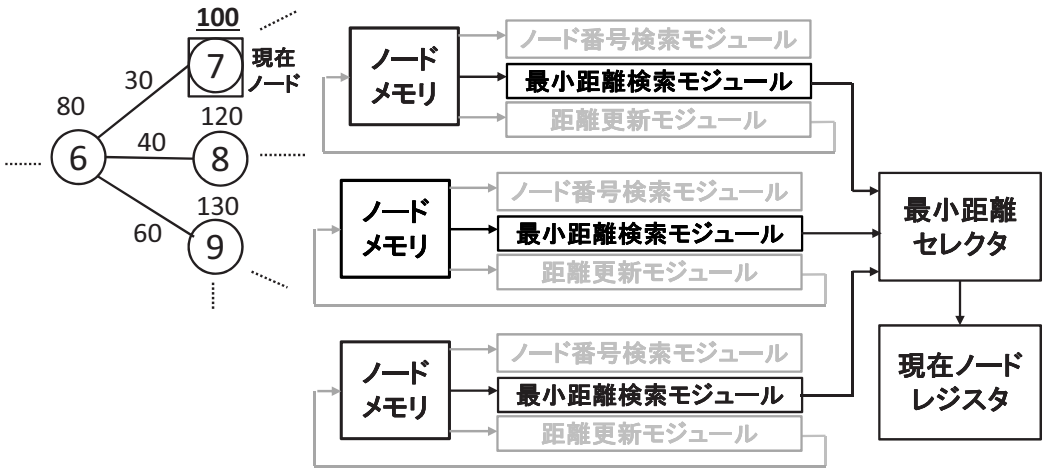


図 4.19: 最短距離の検索

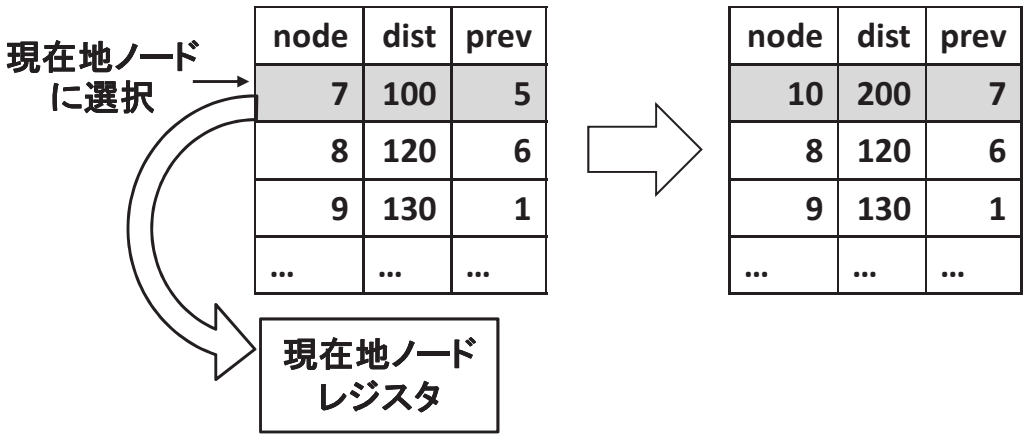


図 4.20: ノードデータの上書き

表 4.1: 最短経路検索ユニットのリソース量

LE	Register	Memory bit	DSP
1509(0.2%)	607(0.1%)	196608(0.4%)	0

4.3.3　評価

今回の評価では，簡潔グラフ表現の処理ユニットの動作確認が出来なかったため，最短経路検索ユニットの実装評価からのリソース量の評価および最短経路検索の処理時間の見積もりについて評価を行う．

最短経路検索ユニットについて，Altera Stratix V を搭載した Terasic DE5 board [34] に Altera Quartus 13.1 を用いて実装評価を行った．今回の試作では，PE およびノードメモリ数を 8，ノードメモリに格納できるノードデータ数が 4096 ノードのアーキテクチャを実装した．最短経路検索ユニットの動作確認について，ソフトコア CPU である NiosII に最短経路検索ユニットを接続して動作確認した．その結果，一点対全点の最短経路検索について，ノードごとの最短距離を正しく計算していることを確認した．

表 4.1 に最短経路検索ユニットのリソース量の評価結果を示す．今回の評価では動作確認小規模な最短経路検索ユニットを実装された最短経路検索ユニットのリソース量と FPGA 上で利用できる全体のリソース量の割合から，最大で PE を約 320 実装し，約 100 万ノードをノードメモリに格納できるような最短経路検索ユニットのアーキテクチャを FPGA ボードに実装できる見積もりが得られた．

FPGA で処理できるノード数の上限は，内部メモリの容量によって制約される．現在の FPGA で利用可能な内部メモリの容量は数十 MB 程度が上限となっているため，ノード数が数千万スケール以上に達するような大規模なグラフを単数の FPGA で処理することは極めて困難である．このようなメモリ容量の問題点の解決法として，複数の FPGA ボードを連結して大規模な最短経路検索ユニットのアーキテクチャを実装する方法を提案することができる．

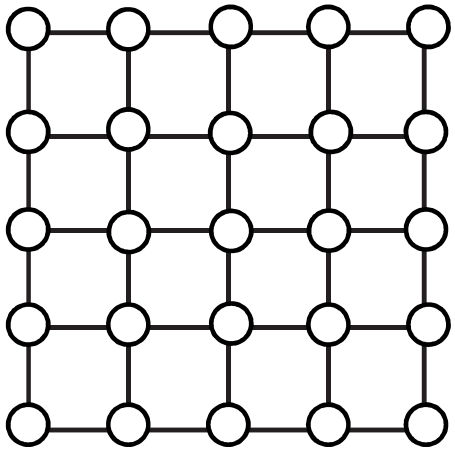


図 4.21: 格子グラフ

表 4.2 に図 4.21 に示されるような格子グラフに対して，提案する最短経路検索ユニットにおける処理中にノードメモリに記憶されるノードデータ数を示す． ノードメモリに格納されるノードデータ数は全体のノード数の約 1/3 程度であることが確認されたため，格子グラフのような形状，密度を持つグラフについては，ノードメモリに記憶できるノードの 3 倍のグラフを処理できると考えられる．ノードメモリに格納されるノードデータ数の割合について，グラフの密度が小さい場合はノードメモリに格納されるノードデータ数を小さくなると考えられる．一方で，完全グラフのように始点と他の全ノードが隣接しているグラフの処理に関しては，始点以外の全ノードにおける暫定距離を格納する必要がある．そのため，本研究で提案したアクセラレータは，道路ネットワークなどの密度が小さいグラフ処理に向いていると結論付けられる．

ダイクストラ法による一点対全点の最短経路問題の計算時間の見積もりについて，外部メモリのアクセス時間，ダイクストラ法の並列処理にかかるサイクル数から見積もりを求めることができる．表 4.3 に CPU と FPGA アクセラレータにおける最短経路問題の計算時間を示す．CPU におけるダイクストラ法の処理速度と比較する

表 4.2: ノードメモリ内の記憶ノードデータ数

全ノード	256	1024	4096
ノードメモリ内の記憶データ	87	316	1329
割合	0.340	0.309	0.324

表 4.3: 計算時間の評価

	1024 ノード，3968 エッジ	4096 ノード，16130 エッジ
FPGA	0.34ms	1.51ms
Core2 Quad	9.38ms	94.0ms

と，ノード数が 4096 のグラフの最短経路検索に対して，CPU の約 60 倍の処理速度が出ていることが確認できる．また，グラフの規模が大きくなるにつれて並列処理の効果が大きくなることが確認できる．今後の課題として，道路ネットワーク，ソーシャルネットワークなどの実世界に存在する大規模なグラフ構造を処理した場合の処理時間，メモリ使用量を評価する必要がある．

4.4　結言

本章では，簡潔グラフ表現に基づくデータ圧縮されたグラフデータを処理するための簡潔グラフ処理ユニット，およびダイクストラ法における暫定距離のデータの記憶量を減らす最短経路ユニットを有する最短経路検索アクセラレータを提案した．最初に，簡潔グラフ表現のビットベクトルにおける select 操作をハードウェア処理するために，可変ワード長へ拡大した簡潔データ構造およびおよび簡潔グラフ表現を処理するための select 操作を計算するユニットのアーキテクチャを提案した.．グラフデータの圧縮量について，アメリカ全土の道路ネットワークに対してエッジデータのアドレスを示す Index のデータ量が約 88%，グラフデータ全体のデータ量の約 18%がが削減されていることが確認された．次に，最短経路ユニットにおける記憶量

の削減について，ダイクストラ法のノードデータのアクセス範囲に注目して，ノードメモリ内のデータ書き換えが可能なアーキテクチャを提案した．最短経路導出中にノードメモリに格納されるノードデータ数について，格子グラフにおける最短経路問題を処理した場合は全体のノード数の 1/3 程度であることを確認した．FPGA アクセラレータにおける一点対全点の最短経路問題の処理速度の見積もりについて，CPU における処理速度と比較すると，ノード数が 4096 のグラフの最短経路検索に対して CPU の約 60 倍の処理速度を実現できる見積もりが得られた．

第5章　結言

以上，第2章から第4章に渡って，外部メモリ・ストレージとのデータ転送のオーバーヘッド削減に着目したFPGAアクセラレータの設計手法を提案した．

2章では，FPGAベースMIMDアクセラレータのアーキテクチャモデルについて，Window画像処理におけるデータ転送と演算処理をオーバーラップを考慮したMIMDアクセラレータの最適設計法を提案した．コア数，演算並列度などのMIMDアクセラレータの設計自由度について，処理時間を最小化する設計自由度を導出するために，データ転送と演算処理をオーバーラップを考慮した処理時間の見積もり式を導出した．MIMDアクセラレータのFPGAボードへの実装評価の結果より，処理時間の見積もり式は実測値と比較しても十分な精度が得られていること，全探索によりオーバーラップを考慮した処理時間を最小化する設計自由度が導出されていることを確認した．また，従来のカスタム化されたヘテロジニアスマルチコアプロセッサとの比較評価により，本研究で提案するFPGAベースのMIMDアクセラレータは，処理速度，消費電力においてカスタムプロセッサと遜色ない性能を実現していることを確認した．

3章では，内部メモリの効率的利用によって外部メモリアクセス回数を削減したステンシル計算アクセラレータのアーキテクチャについて，内部メモリで処理するタイムステップ数とタイムステップごとのステンシル演算の並列度の設計自由度を最適化するアクセラレータの最適設計手法を提案した．最初に，内部メモリに中間計算結果を格納して外部メモリアクセス回数を削減したステンシル計算アクセラレータのアーキテクチャについて，OpenCLを用いたFPGAへの実装法を提案した．ま

た，設計自由度を最適化するために，処理時間の見積もり式を導出して実測値との比較評価を行った．計算時間，消費電力の評価結果について，ステンシル演算の並列度が一定であればリソース制約下で depth をできるだけ大きくすること，計算時間が同程度の場合は，消費電力を小さくするために depth が大きくなるような設計自由度を選択することが最適であることを結論付けた．さらに，CPU，GPU との性能比較について，FPGA に実装したステンシル計算アクセラレータの提案アーキテクチャは，CPU，GPU との比較して大幅な処理の高速化と消費電力量の削減を実現していることを確認した．

4 章では，最短経路問題を処理する FPGA アクセラレータについて，高速処理とコンパクトな記憶量を両立する簡潔グラフ表現に基づくグラフデータ圧縮，および最短経路問題の処理における暫定的な距離データの記憶量を削減することにより，データ転送量を削減したアーキテクチャを提案した．簡潔グラフ表現をハードウェア処理するために，可変ワード長へ拡大した簡潔データ構造，および簡潔グラフ表現処理ユニットのアーキテクチャを提案した．グラフデータの圧縮量について，アメリカ全土の道路ネットワークに対して，エッジデータのアドレスを示す Index のデータ量が約 88%，グラフデータ全体のデータ量の約 18%が削減されていることが確認された．また，最短経路検索における記憶量の削減について，ダイクストラ法のノードデータのアクセス範囲に注目して，ノードメモリ内のデータ書き換えが可能なアーキテクチャを提案した．最短経路導出中にノードメモリに格納されるノードデータ数について，格子グラフを処理した場合は全体のノード数の 1/3 程度であることを確認した．FPGA アクセラレータにおける一点対全点の最短経路問題の処理速度の見積もりについて，CPU における処理速度と比較すると，ノード数が 4096 のグラフの最短経路検索に対して CPU の約 60 倍の処理速度を実現できる見積もりが得られた．

本研究の今後の展望として，複数の FPGA ボードを接続した FPGA クラスタによる超大規模計算向けの FPGA カスタムアクセラレータへの応用が期待できる．FPGA

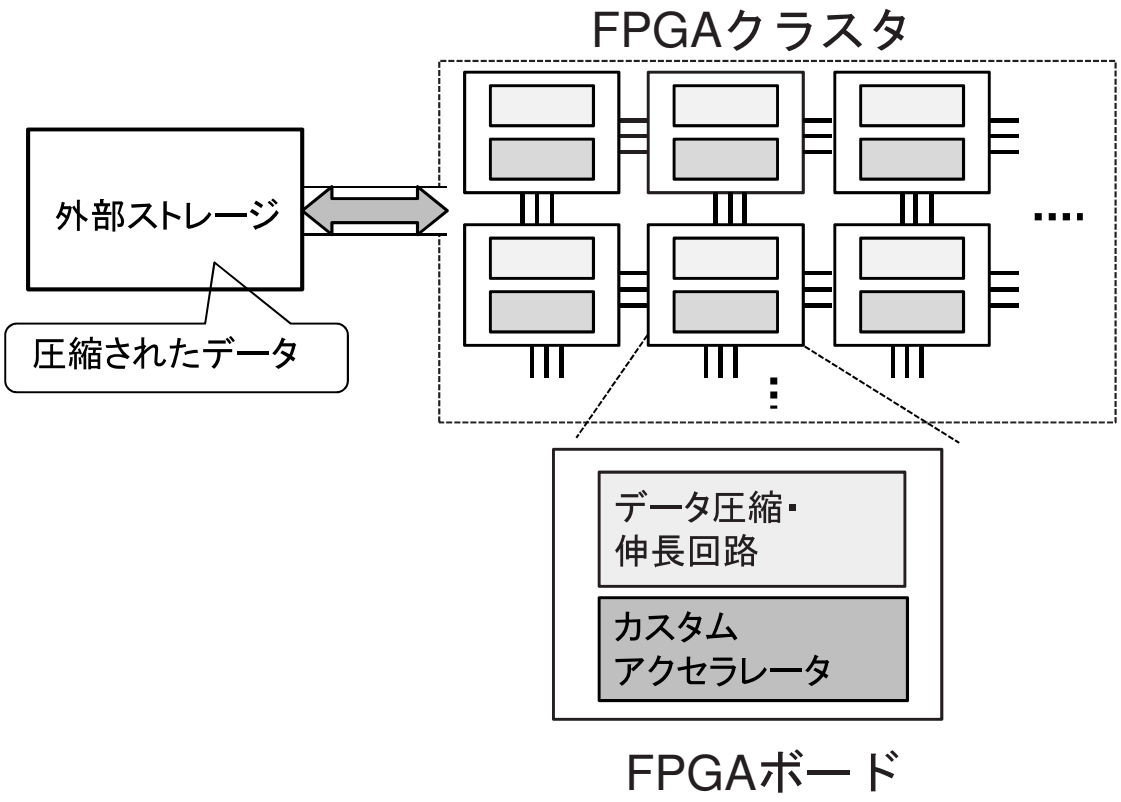


図 5.1: FPGA クラスタにおける本研究の応用

ボードを複数使用したアーキテクチャにおける処理時間について，FPGA ボード間のデータ転送および外部ストレージと FPGA 間のデータ転送時間を考慮する必要がある．そのため，図 5.1 のように，本論文の 4 章で提案した簡潔表現を用いたデータ圧縮により，ボード間・ストレージ間のデータ転送量削減が実現可能であると考えられる．さらに，FPGA ボード上のカスタムアクセラレータの設計について，本論文の 2 章および 3 章で提案したような，データ転送を考慮したアクセラレータの最適設計法が非常に有用であると考えられる．

参考文献

[1] Top500 Supercomputing Sites, http://www.top500.org/system/177232

[2] NVIDIA, “CUDA C Programming Guide v7.5” ,2015.

[3] A. Putnam et al. “Large-Scale Reconfigurable Computing in a Microsoft Datacenter” , A Symp. High Performance Chips (HOT Chips 26), 2014.

[4] H. Noda , M. Nakajima , K. Dosaka , K. Nakata , M. Higashida , O. Yamamoto , K. Mizumoto , T. Tanizaki , T. Gyohten , Y. Okuno , H. Kondo , Y. Shimazu , K. Arimoto , K. Saito and T. Shimizu “The design and implementation of the massively parallel processor based on the matrix architecture”, IEEE Journal of Solid-State Circuits Vol. 42, No. 1, pp.183-192, 2007.

[5] H. Shikano, M. Ito, M. Onouchi, T. Todaka, T. Tsunoda, T. Kodama, K. Uchiyama, T. Odaka, T. Kamei, E. Nagahama, M. Kusaoke, Y. Nitta, Y. Wada, K. Kimura and H. Kasahara”, “Heterogeneous Multi-Core Architecture That Enables 54x AAC-LC Stereo Encoding”, IEEE Journal of Solid-State Circuits Vol.43, No.4, pp.902-910, 2008.

[6] M.Motomura, “A Dynamically Reconfigurable Processor Architecture”, Microprocessor Forum, 2002.

[7] Y. Takei, H. M. Waidyasooriya, M. Hariyama and M. Kameyama, “Evaluation of an FPGA-Based Heterogeneous Multicore Platform with SIMD/MIMD Cus-

tom Accelerators", IEICE Transactions on Fundamentals, Vol.E96-A, No.12, pp.2576-2586, 2013.

[8] Y. Kobayashi, M. Hariyama and M. Kameyama, "Memory Allocation for Multi-Resolution Image Processing" IEICE Transactions on Information and Systems, Vol.E91-D, No.10, 2008.

[9] H. M. Waidyasooriya, M. Hariyama and M. Kameyama, "Memory Allocation for Window-Based Image Processing on Multiple Memory Modules with Simple Addressing Functions", IEICE Transactions on Fundamentals, Vol.E94-A, NO.1, pp.342-351, 2011.

[10] H. M. Waidyasooriya, Y. Ohbayashi, M. Hariyama and M. Kameyama, "Memory Allocation Exploiting Temporal Locality for Reducing Data-Transfer Bottlenecks in Heterogeneous Multicore Processors", IEEE Transactions on Circuits and Systems for Video Technology, Vol.21, No.10, pp.1453-1466, 2011.

[11] H. M. Waidyasooriya, Y. Ohbayashi, M. Hariyama and M. Kameyama, "Memory-Access-Driven Context Partitioning for Window-Based Image Processing on Heterogeneous Multicore Processors", IEICE Transactions on Information and Systems, Vol.E95-D, No.2, pp.354-363, 2012

[12] H. Ta and S. Lee, "High-performance computing model for 3D camera system", Proc. Int. Conf. Robotics and Biomimetics (ROBIO), pp.354-359, 2011.

[13] K. Iwai, T. Kurokawa and N. Nishikawa, "AES encryption implementation on CUDA GPU and its analysis", Proc. 2010 First Int. Conf. Networking and Computing, pp.209-214, 2010.

[14] T. Chen , Z. Sura , K. O'Brien and J.K. O'Brien, "Optimizing the use of static buffers for DMA on a CELL chip", Proc. the 19th Int. Conf. Languages and compilers for parallel computing, pp.314-329, 2006.

[15] N. Dalal and B. Triggs "Histograms of Oriented Gradients for Human Detection", Proc. IEEE Computer Society Conf. Computer Vision and Pattern Recognition, pp.886 - 893, Vol.1, 2005.

[16] D.G. Lowe, "Object Recognition from Local Scale-Invariant Features", Proc. IEEE Int. Conf. Computer Vision, pp.1150 - 1157, Vol.2, 1999.

[17] Xilinx, "Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit", `http://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html`

[18] S. Dutta, V. Rajagopolan, B. Taylor and R. Wittig, "Xilinx Zynq Embedded Processing Platform", *A Symp. High Performance Chips (HOT Chips 23)*, 2011.

[19] W.Augustin, V.Heuveline and J. P. Weiss, "Optimized StencilComputation Using in Place Calculation on Modern Multicore Systems", Proceedings of Intemational European Conference on Parallel and Distributed Computing, pp 772-784 ,2009.

[20] M. Wittmann, G. Hager and G. Wellein, "Multicore-aware Parallel Temporal Blocking of Stencil Codes for Shared and Distributed Memory", in Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium , pp.1-7, 19-23 2010

[21] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-Performance Code Generation for Stencil Computations on GPU Architectures", In Proceedings

of the 26th ACM international conference on Supercomputing (ICS '12), pp.311-320, 2012.

[22] J.Meng, and K. Skadron, "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations", International Journal of Parallel Programming 39.1, pp115-142, 2011.

[23] M.Shafiq, M. Pericas, R. de la Cruz, M. Araya-Polo, N. Navarro and E. Ayguade, "Exploiting Memory Customization in FPGA for 3D Stencil Computations" , International Conference on Field-Programmable Technology(FPT '09), pp.38-45, 2009.

[24] R.Kobayashi, S. Takamaeda-Yamazaki and K. Kise, "Towards a Low-Power Accelerator of Many FPGAs for Stencil Computations" , Third International Conference on Networking and Computing (ICNC), pp.343-349, 2012.

[25] K. Sano, L. Wang and S. Yamamoto "Prototype Implementation of Array-processor Extensible over Multiple FPGAs for Scalable Stencil Computation", SIGARCH Comput. Archit. News, Vol.38, No.4, pp.80-86, 2011.

[26] K. Sano, H.Hatsuda and S. Yamamoto "Multi-FPGA Accelerator for Scalable Stencil Computation with Constant Memory Bandwidth" IEEE Transactions on Parallel and Distributed Systems , vol.25, no.3, pp.695-705, 2014.

[27] Altera corporation, "Altera SDK for OpenCL Programming Guide", https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[28] Khronos group, http://www.khronos.org/opencl/

[29] D. Chen and D. Singh, “Fractal Video Compression in OpenCL:An Evaluation of CPUs, GPUs, and FPGAs as Acceleration Platforms”, Design Automation Conference (ASP-DAC) 18th Asia and South Pacific, pp.297-304, 2013.

[30] Nallatec, “40Gbit AES Encryption Using OpenCL and FPGAs”, http://www.nallatech.com/40gbit-aes-encryption-using-opencl-and-fpgas/

[31] K. Hill, S. Craciun, A. George, H. Lam, “Comparative analysis of OpenCL vs. HDL with image-processing kernels on Stratix-V FPGA”, IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP), pp.189-193, 2015

[32] D. Neto, “Optimizing OpenCL for Altera FPGAs ”, International Workshop on OpenCL, 2014.

[33] Altera corporation, “Altera SDK for OpenCL Best Practices Guide”, https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_optimization_guide.pdf

[34] Terasic, “DE5-Net FPGA Development Kit”, http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=158&No=526

[35] G. J. Katz and J. T. Kider Jr, “All-Pairs Shortest-Paths for Large Graphs on the GPU”, In Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, 2008.

[36] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, “Parallel FPGA-Based All-Pairs Shortest-Paths in a Directed Graph”, In Proceedings of Parallel and Distributed Processing Symposium, 2006.

[37] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. “Pregel: a System for Large-Scale Graph Processing”, In Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, pp. 135-146, 2010.

[38] G. Jacobson, “Space-efficient Static Trees and Graphs,” 30th Annual Symposium on Foundations of Computer Science, 1989.

[39] J. I. Munro and V. Raman, “Succinct representation of balanced parentheses,static trees and planar graphs,” in Proceedings of the 38th Annual Symposium on Foundations of Computer Science, pp. 118-126, 1997.

[40] A. Golynski, “Optimal Lower Bounds for Rank and Select Indexes” Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, pp.370-381, 2005.

[41] H. M. Waidyasooriya, D. Ono, M. Hariyama and M. Kameyama, “An FPGA Architecture for Text Search Using a Wavelet-Tree-Based Succinct-Data-Structure”, International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), pp.354-359, 2015.

[42] H. S. Warren, “Hacker ’s Delight (2nd edition)”, - Chapter 5.

[43] “9th DIMACS Implementation Challenge - Shortest Paths”, `http://www.dis.uniroma1.it/challenge9/download.shtml`

[44] H. M. Waidyasooriya, D. Ono, M. Hariyama and M. Kameyama, “Hardware-Oriented Succinct-Data-Structure based on Block-Size-Constrained Compression”, Seventh International Conference of Soft Computing and Pattern Recognition (SoCPaR 2015), pp.136-140, 2015.

[45] E. W. Dijkstra, “A Note on Two Problems in Connexion with Graphs”, Numerische Mathematik, 1(1): pp.269-271, 1959.

[46] R. Bellman, “On a Routing Problem”, Technical report, DTIC Document, 1956.

[47] R. W. Floyd, “Algorithm 97: Shortest Path”, Commun. ACM, 5(6) pp.345-346, June 1962.

[48] M. Tommiska and J.Skytta. “Dijkstra’s Shortest Paths Algorithm in Reconfigurable Hardware”, In Proc. Field Programmable Logic and Applications, pp.653-657, 2001.

[49] I. Fernandez, J. Castillo, C. Pedraza, C. Sanchez, and J. I. Martinez, “Parallel Implementation of the Shortest Path Algorithm on FPGA” In Proc. 4th Southern Conf. on Programmable Logic., pp.245-248, 2008.

[50] K. S. T.K.Priya and P. Kumar, ”Hardware Architecture for Finding Shortest Paths”, In Proc. IEEE Region 10 Conf., pp. 1-5, 2009.

謝辞

本論文は，著者が東北大学大学院情報科学研究科 情報基礎科学専攻 知能集積システム学分野 (亀山・張山研究室) において行った研究を取りまとめたものであります．本研究を推し進めるにあたり，多くの方々からご協力とご助言を頂きました．

恩師 亀山充隆教授には，終始熱心なご指導とご鞭撻を頂きました．先生から学ばせて頂いた原理を深く追求してゆく姿勢は，研究を行う上で最も重要なものだと実感しております．加えて，先生の研究，教育に対する熱意溢れるご姿勢と，ご討論などでの有益なるご助言から多くを学ばせて頂いたことを銘記し，ここに深く感謝の意を表します．

本研究をまとめるにあたり，東北大学大学院情報科学研究科 青木孝文教授ならびに東北大学大学院工学研究科 大町真一郎教授にはご専門の立場から有意義なご意見を賜りました．先生方のご意見により本論文がさらに充実したことを銘記し，ここに深く感謝申し上げます．

東北大学大学院情報科学研究科 張山昌論准教授には，常日頃から有益なるご助言と激励のお言葉を頂きました．先生の研究に対する常に前向きなご姿勢と，実行力から多くを学ばせて頂いたことを銘記し，ここに深く感謝いたします．

東北大学大学院情報科学研究科 Hasitha Muthumala Waidyasooriya 助教には，常日頃から有益なるご意見とご助言を頂きました．ここに深く感謝いたします．

日頃の研究室生活において様々な面でご支援，ご協力いただいた東北大学工学部佐々木明夫技術職員 (再雇用) を始めとする研究室諸氏には心より御礼申し上げます．

最後に，長きにわたる研究生活を応援し支えて下さった両親に感謝の意を表し，本論文を結びます.

武井　康浩

2016 年 1 月 20 日