平成２７年度修士論文

配列類似性検索のFPGAを用いた並列演算

大学院情報システム学研究科
情報ネットワークシステム学専攻

学 籍 番 号 :　1452014
氏 名:　中井　康介
主任指導教員: 吉永　努 教授
指 導 教 員 :　長岡　浩司 教授
指 導 教 員 :　笠井　裕之 准教授
提出年月日 :　平成28年1月28日

(表紙裏)

目 次

図 目 次

表 目 次

第1章　序論

２種の生物学的配列から類似箇所を算出する配列類似性検索は，バイオインフォマティクスの分野で広く利用されている．また BLAST（Basic Local Alignment Search Tool）[1, 2, 3] は，既知な配列データベース [4] から，クエリ配列と類似した配列を検索し，類似度を示す高速なシーケンスアラインメントアルゴリズムとして利用されている．

この配列データベースは，学者により新しい配列が日々発見され，年々増加している [5]．例えば，UniProtKB/SwissProt データベース（2015_12 リリース）[6] は，配列データベースのひとつであり，550,116 の配列エントリを保持している．日々増加する配列データベースのようなビッグデータ解析において，データの増加速度の方がプロセッサの速度向上よりも速いため，計算性能の向上が求められる．

この向上方法として，専用ハードウェアによる BLAST の処理の一部または全体を，FPGA にオフロードする研究は多く行われてきた [7, 8, 9, 10, 11]．FPGA によるデータの入力速度で計算する手法に取り組むことは解析時間の短縮に大きく貢献する．別の向上方法として，複数の計算機による BLAST の並列化の取り組みがある [12]．mpiBLAST は PC クラスタ上で BLAST を並列演算するオープンソースであり，MPI（Message Passing Interface）を利用することで，処理の高速化を図っている．

計算処理を高速化する点と計算処理を並列化する点，ふたつの観点からの BLAST の高速化取り組み方法があるが，どちらの研究も個別の観点からの実装である．そこで両方の観点から BLAST の専用ハードウェアによる並列演算の取り組みが必要となる．

本研究では，先行研究で行われた，並列処理するためのデータパーティショニングのハードウェア実装 [13] と BLAST における処理負荷が高い部分のハードウェア実装 [11] を統合し，mpiBLAST で行われる CPU 処理の一部を専用ハードウェアにオフロードする．また BLAST の一部処理をデータ I/O の速度で処理を行える専用ハードウェアを開発し，BLAST の並列演算の高速化を図る．

本論文は以下のように構成される．2 章では，関連研究および関連技術について述べ，3 章で，先行研究の概要，提案回路および実装の概要について述べ，4 章で，提案手法を組み込んだハードウェアを構築し評価を行う．最後に，5 章で結論を述べる．

第2章　背景

2.1　バイオインフォマティクス

バイオインフォマティクスは情報学と生物学が融合した学問である [14]．生物学には，ゲノムのデータから得られた配列の生物学的情報の決定，2 種の生物のゲノム配列から遺伝子構造の共通点の探索がある．これらの生物学的情報は配列データベースとして集合され，生物学的情報の発見への比較として用いられる．配列データベースは日々増加しており，コンピュータの利用が必要不可欠である．

2.2　配列類似性検索

配列類似性検索とは，ある塩基配列が遺伝子またはタンパク質配列中から類似した文字列パターンを見つける技術である．2 つの配列間の類似性を比較することをペアワイズアラインメント [1, 15] と呼ぶ．これにより配列の立体構造予測または生物学的な機能の予測が可能になる．

2.3　FPGA : FPGA: Field-Programmable Gate Array

先行研究の工藤 [13] より引用する．FPGA は，CPLD (Complex Programmable Logic Device) と並んで多く出荷されているデバイスであり，ハードウェア記述言語（HDL：Hardware Description Language）を用いて，ユーザが構成を設定できる集積回路という特徴を持つ．Look-Up Table（LUT）を用いて書き換え可能な回路を実現することで，大規模な回路を実装が可能である [16, 17, 18].

2.3.1　FPGA の構造

FPGA は，LUT を基本ロジックブロックとする書き換え可能なデバイスである．LUT が n 入力 m 出力であれば，入力 n ビット，出力 m ビットの任意の論理関数を実現するためのメモリである．このメモリを，SRAM や Flash ROM，Anti-Fuse ROM などで構成し，値を書き込むことで任意の論理関数を実現できる．現在，商用 FPGA のほとんどは 6 ビット 1 出力の LUT で構成されており，この場合，1 つの LUT はアドレス幅 6 ビット，ワード幅 1 ビットのメモリとみなす．

SRAM 型 FPGA は一般的な CMOS 半導体プロセスで製造されるため，最新のプロセスを利用することができる．また，プロセスの微細化によるスケーラビリティがある．FPGA を構成する基本の構成要素を図 2.3.1(a) に示す．これを多数繋げていくことによって図 2.3.1(b) のように大きな回路を構成する．このため Island-style FPGA の構造は均一であり，ブロック数を変化させるだけでロジックサイズの異なる製品を容易に作成可能である．

Logic Block は LUT を含む，書き換え可能な回路を実現するためのブロックである．図 2.3.1(a) において 1 本線で示したように，隣接する Connection Block への配線を持つ．また，Connection

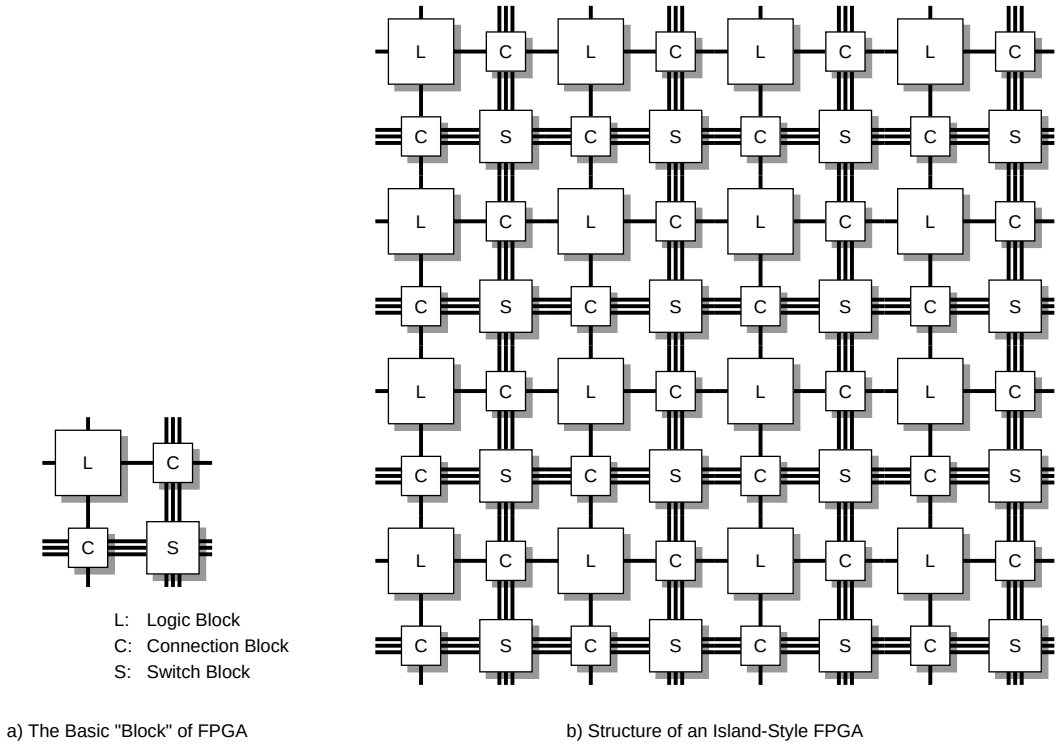

図 2.3.1: Island-Style FPGA の基本ブロックと全体の構造

Block は，隣接する Logic Block への配線を持ち，図 2.3.1(a) において 3 本線で示したように，FPGA 中の大域的な配線との間を接続するパストランジスタを用いた書き換え可能なスイッチである．縦横に並ぶ平行な三本の配線の交点には，同様に書き換え可能なスイッチである Switch Block が設置されている．このようにして FPGA は回路と配線の双方で書き換えが可能になる．

2.3.2 FPGA の例: Avaldata APX-880A

本研究で使用した FPGA ボード (APX-880A) を図 2.3.2 に示す．また，そのストレージの拡張に用いる拡張ボードを図 2.3.3 に示す．APX-880A[19] はアバールデータ社が開発した大容量の高速書き込みおよび高速読み出しが可能なメモリを搭載する FPGA ボードである．本ボードは，光通信モジュールと SD カードコネクタを搭載しており，ネットワークとストレージへのデータ転送を高速に行うことができる．全ての機能はボード上の FPGA から制御可能である．また，ホストマシンからのアクセス経路として，PCI Express Gen2x4 を採用しており，ホストマシンからもデータの転送や各種制御が可能である．

本ボード上の FPGA には，SD カードコントローラ，光通信コントローラ，PCIe コントローラのモジュールが実装されている．これらは 1 つのバススイッチに接続されており，自由にアクセスすることができる．またストレージでは DMA コントローラが実装されており，ホストマシンへの負荷をかけずに，データの転送を行うことができる．さらに，データ転送や計算のバッファとして使用可能な SDRAM とそのコントローラも搭載している．ホストマシンからの制御に必要なすべてのレジスタは，レジスタマップモジュールにより，PCIe のアドレス空間にマッピングされている．

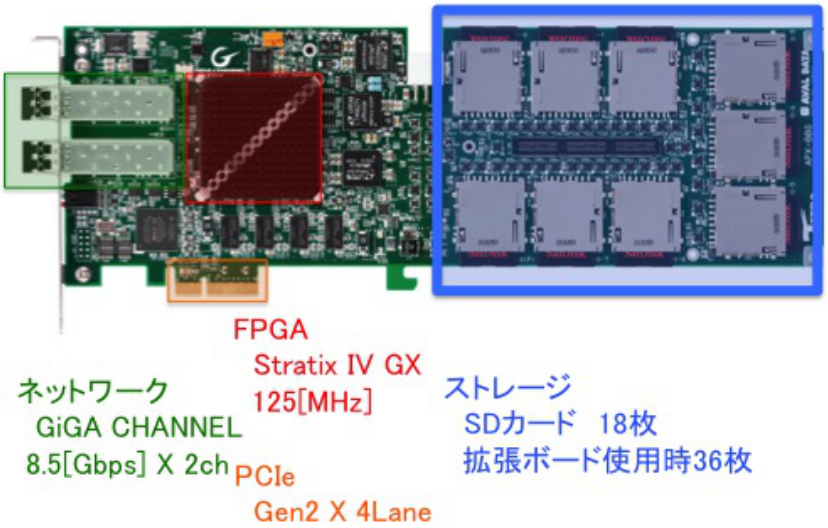

図 2.3.2: APX880A ボード

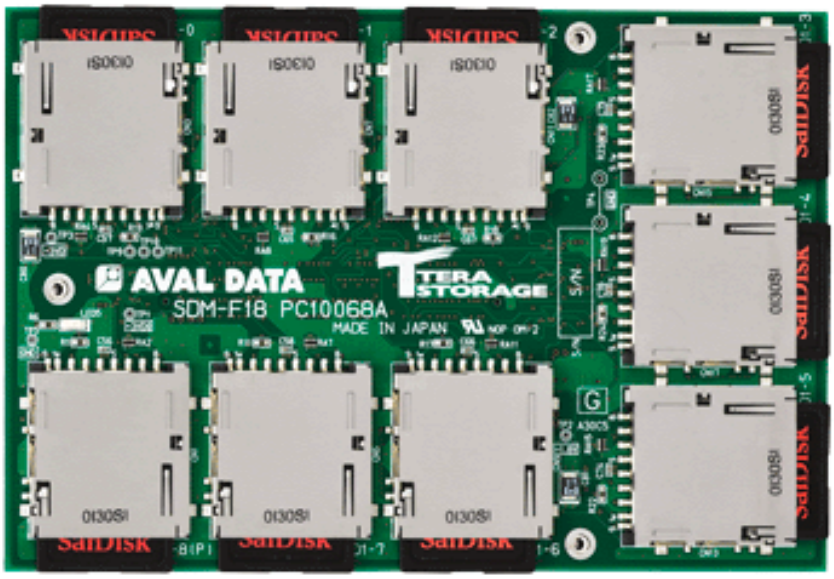

図 2.3.3: APX880A 拡張ボード

2.4 BLAST: Basic Local Alignment Search Tool

2.4.1 BLAST の概要

Basic Local Alignment Search Tool（BLAST）[1, 2, 3] はペアワイズアラインメントのツールであり，バイオインフォマティクスの中で最も広く利用されている．問い合わせ配列（クエリ配列）と検索にかけられる塩基配列データベースを指定すると，クエリ配列に類似した塩基配列データベース中のエントリがランキング付けされて表示され，類似部分文字列についてアラインメント結果が表示される．

本研究では，ローカルの環境で実行可能な NCBI Local BLAST および mpiBLAST[12] について取り組む．

以降の文章では NCBI Local BLAST を BLAST として表記する．

2.4.2 BLAST の各種プログラムについて

BLAST では検索クエリと検索対象のデータベースにあわせて，複数のプログラムが実装されている（表 2.1）．BLASTN は 4 種，BLASTP は 24 種の文字からなる塩基配列同士のペアワイズアラインメントを行う．本研究では BLASTP についてハードウェアを用いた高速化を行う．

2.4.3 FASTA フォーマット

BLAST において，データベースおよびクエリなどのシーケンスデータは，FASTA フォーマットで記述される．

FASTA フォーマットはプレーンテキストであり，ヘッダ行とシーケンス文字列で構成されている．ヘッダ行には先頭に “>” で示され，以降にそのシーケンス文字列の識別情報が一行で記載される．ヘッダ行の下には実際のシーケンス文字列が記述される．データベース配列，または検索したいクエリ配列が複数ある場合，シーケンス文字列の記述が終わると，図 2.4.1 のように次のシーケンスデータのヘッダ行およびシーケンス文字列というように続く．

本研究では，FASTA フォーマットである実際の配列データベース [4] を実データベースと呼ぶ．また BLAST では，事前に FASTA フォーマットの配列データベースに対し，ヘッダ行の識別情報，シーケンス文字列，インデックスの 3 種のバイナリファイルへと再フォーマットする過程がある．これにより後述する BLAST のアルゴリズムの高速化を図っている．

表 2.1: BLAST の各種プログラム

プログラム	クエリ配列	データベース
BLASTN	塩基配列	塩基配列
BLASTP	タンパク質	タンパク質
BLASTX	塩基配列	タンパク質
TBLASTN	タンパク質	塩基配列
TBLASTX	翻訳された塩基配列	翻訳された塩基配列

>gi|137335551|gb|EBT52160.1| hypothetical protein GOS_7260019 [marine metagenome]gi|142036080|gb|ECV04742.1| hypothetical protein GOS_2971017 [marine metagenome]
MTGQRIGYIRVSTFDQNPERQLEGVKVDRAFSDKASGKDVKRPQLEALISFARTGDTVVVHSMDRLARNLDDLRRIVQTL
TQRGVHIEFVKEHLSFTGEDSPMANLMLSVMGAFAEFERALIRERQREGIALAKQRGAYRGRKKSLSSERIAELRQRVEA
GEQKTKLAREFGISRETLYQYLRTDQ
>gi|142020295|gb|ECU90029.1| hypothetical protein GOS_2999736 [marine metagenome]gi|142022179|gb|ECU91854.1| hypothetical protein GOS_2996212 [marine metagenome]gi|142032408|gb|ECV01282.1| hypothetical protein GOS_2977717 [marine metagenome]gi|142032413|gb|ECV01287.1| hypothetical protein GOS_2977722 [marine metagenome]gi|142032467|gb|ECV01341.1| hypothetical protein GOS_2977818 [marine metagenome]gi|142032512|gb|ECV01385.1| hypothetical protein GOS_2977496 [marine metagenome]gi|142039359|gb|ECV07954.1| hypothetical protein GOS_2964639 [marine metagenome]gi|142039367|gb|ECV07962.1| hypothetical protein GOS_2964651 [marine metagenome]gi|142047930|gb|ECV15622.1| hypothetical protein GOS_2953343 [marine metagenome]
MIKWFRQKLHLDNKMREANILQHSLHQYCPELHLKRLNSLMLASKALIECKTLTLTELGRNLPTTARTKHNIKRIDRLLG
NTHLHQERLAVYQWHASLICSGNPMPIVLVDWSDIREHKRIMALRASIAFNGRSITLYEKSYPLSEQCSKASHNGFLADL
AKILPLHVTPLIVTDAGFKVPWYKEVEAHGWFWLSRIRGTVQFADIGAENWRAVRSTHDLANGQAKSLGCKTLTKTNPIN
CHLTLYRSKPKGRTNQRSTRTNCHHPSAKTYSTSAKEPWVLASNLPPESRSPKQLVNLYAKRMQIEETFRDLKSPAYGFG
LRQSRTNSPERFDIILLIALMVQCLLWLVGLHAQQQGWDKHFQANTIGHRTVLSTIRLGLEVLRRPDYQITEKELLAAWV

図 2.4.1: 実 FASTA フォーマットの配列一部

2.4.4 BLASTP の計算手順

BLASTP では全ての文字列についてアラインメントを行うと時間がかかるため，ハッシュテーブルを利用して細かい単位でクエリとデータベースのサブアラインメントを行う．アラインメントスコアが高くなりそうな部分について事前に検討し，計算時間の短縮を図ることを目的としている．

BLASTP は大きく分けて以下の 4 つのフェーズによって計算が行われる．

1. Lookup Table Phase（LTP）
2. Collecting Seeds Phase（CSP）
3. Ungapped Alignment Phase（UAP）
4. Gapped Alignment Phase（GAP）

Lookup Table Phase

最初のフェーズでは，まずクエリ配列を 3 文字ずつの細かな配列に区切ったクエリワードを作成する（図 2.4.2 参照）．続いて，24 種の文字からできる 3 文字の配列を全て想定し，各クエリワードと 1 文字ずつ比較して，ハッシュテーブルにより類似度を計算する．次に想定される 3 文字の BLOSUM62 の合計値がクエリワードに対し，T 以上になったものを neighboring words と呼ぶ（図 2.4.2 参照）．T は neighboring threshold といい，どの程度の類似度を許容するかの設定を行う値であり，初期値は 12 とされる．12 以上となった neighboring words が次のフェーズで用いられる．この過程で使われるハッシュテーブルを置換行列と呼ぶ．置換行列とは，塩基配列やタンパク質の解析において，あるアミノ酸 a_i が他のアミノ酸 b_j，あるいは同じアミノ酸 a_i にどれだけ似ているかの指標を示すのに利用される．BLAST では BLOSUM（ブロック置換行列）が用いられており，BLOSUM62 が初期置換行列として使用されている．

Collecting Seeds Phase

LTP 処理で作成された neighboring words について，データベースのシーケンスと完全一致する部分を検索する．完全一致した部分は seed と呼び，次のフェーズで用いられる．図 2.4.3 において

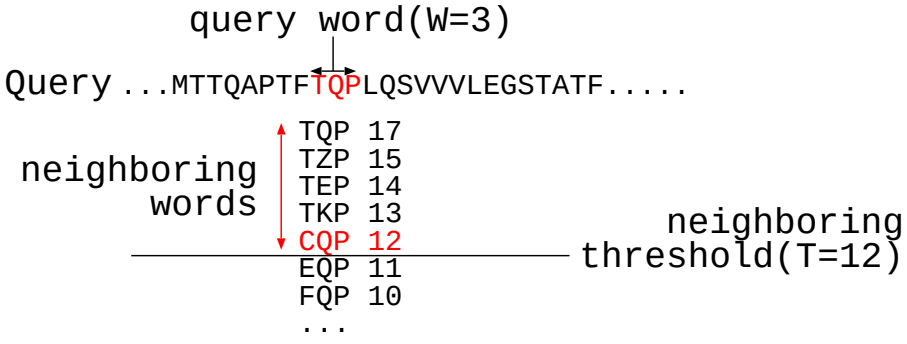

図 2.4.2: Lookup Table Phase

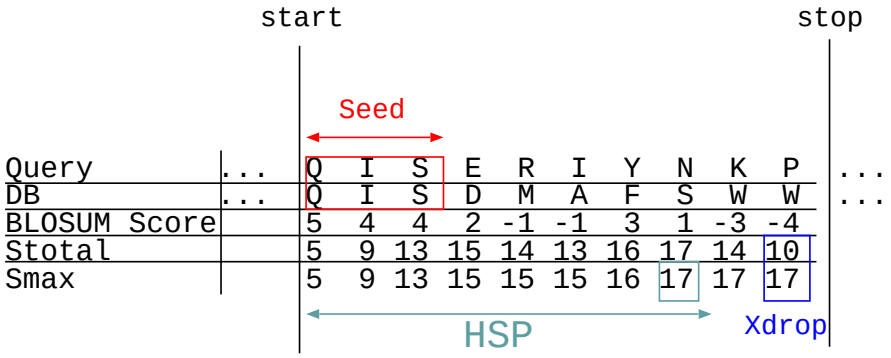

図 2.4.3: Seeding Phase および Ungapped Alignment Phase

seed は Query および DB の “QIS” となる．

Ungapped Alignment Phase

Seed を開始位置としてアラインメントの拡張を行う．本来は一方向だけでなく両方向にアラインメントの拡張を行うが，ここではハードウェアに，より適した片側方向に拡張する手法 [20] を説明する．アラインメントスコアの計算は以下の手順で 1 文字ずつ行われる．

1. クエリと DB の各文字について BLOSUM スコアを参照
2. BLOSUM スコアの合計値 Stotal（Total Substitution Score）を計算
3. それまでの Stotal の最大値 Smax を保持
4. Smax から Stotal へしきい値 X-drop（一般的に 7）下がるまで計算の拡張

拡張されたアラインメントのうちスコアがしきい値 S（11 を用いた）以上のものを，high-scoring segment pairs（HSP）と呼ぶ．この HSP が Gapped Alignments Phase の入力となる．図 2.4.3 において HSP は “QISERIYN” および “QISDMAFS” である．本研究では，UAP 処理について片側方向に拡張する手法を用いる．また 2 種の配列から生成される HSP の内，最もしきい値が高い HSP を算出することを目的とする．

Gapped Alignment Phase

UAP 処理にて算出された HSP について，Smith-Waterman アルゴリズム [21] を用いて，ギャップを含んだアラインメントを行う．

SW アルゴリズムは 2 つの文字列の局所的な類似部分列を抽出するための手法である．2 つの文字列が，同じ位置に同一の文字が並ぶように操作を行い，類似部分列を抽出する．

これにより 2 種の配列の正確な類似部が判明する．

全ての配列において GAP 処理を行う場合，非常に計算時間がかかってしまうため，LTP 処理，CSP 処理，UAP 処理を事前に行うことで，GAP 処理の負荷を大幅に軽減する．

2.5 mpiBLAST の概要

mpiBLAST は，BLAST の並列実装である．mpiBLAST では，主に 2 つのフェーズに分けて，文字列のアラインメントを行う．始めに mpiformatdb コマンドを用いて，データベース配列を任意の数に分割し，それぞれについて内部で NCBI BLAST の formatdb を呼び出し，配列データベースの再フォーマットを行う Data Partitioning フェーズがある．分割後のデータサイズを一定となるように分割を行い，各ノードの処理時間を揃えることで，mpiBLAST 全体の処理の高速化を行っている．次に Open MPI の mpirun コマンドを介して mpiblast コマンドを各ノードで実行する BLAST フェーズがある．mpiblast を実行すると，文字列のアラインメントを行うプロセスは分割されたデータベースのうち，自分の担当の分割データベースから文字列のアラインメントを行う．mpiBLAST では少なくとも 2 プロセスがタスクのスケジューリングおよびファイル出力のコーディネートに使用されるため，実際に mpirun コマンドで実行するプロセス数は，BLAST 検索を行う場合に比べ 2 プロセス増加する．pc クラスタを利用した mpiBLAST のイメージを図 2.5.1 に示す．

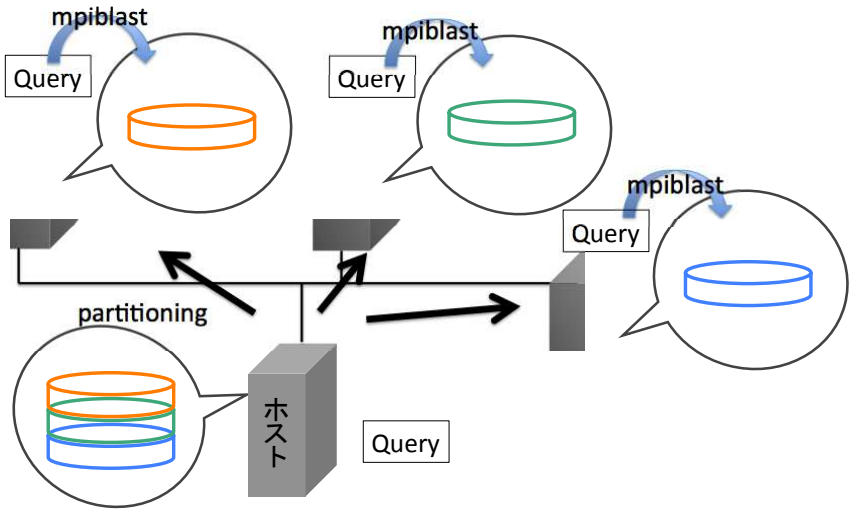

図 2.5.1: mpiBLAST のイメージ図

2.6 関連研究：FPGA を用いた BLAST の高速化

BLAST の処理の一部や全体を，専用ハードウェアを実装した FPGA にオフロードする研究は多く行われてきた [7, 8, 9, 10]．しかし，一方で BLAST の処理全体をオフロードした場合 [9] は比較できる配列長がクエリ，DB 共に 100 であり，実データベースでの検索ができないため十分に実用性があるとは言えない状況にあった．

FPGA を用いた SW アルゴリズムの高速化は多くなされてきたが，BLAST では，GAP 処理が占める割合が少なくなるように，それまでのフェーズで GAP 処理への入力（HSP）を減らす仕組みが導入されている [2]．須戸 [11] から BLASTP の処理の割合から，GAP 処理は UAP 処理に対し非常に少なく，またアルゴリズムが複雑であるため，ソフトウェアでの実行がハードウェアよりも高効率である．

本研究では，並列処理の観点からの取り組み, および実データベースと工藤 [13] のストリーム処理エンジンを用いることで BLAST の高速化を図る．

第3章　提案回路および実装

3.1　専用ハードウェアによる配列類似性検索の並列処理

本研究では前述の mpiBLAST を参考に，専用ハードウェアによる実データベースとの配列類似性検索の並列処理を行う．

mpiBLAST の 2 つのフェーズから，Data Partitioning のフェーズに工藤のハードウェア [13] を用い，各 FPGA ボードへ送信された分割データベースへの配列類似性検索に須戸のハードウェア [11] を用いる．また BLASTP の LTP から UAP までのフェーズにて，データベースの入力速度での計算が可能な構造のハードウェア設計を行う．FPGA ボードは，アバールデータ社製 APX-880A を用いる．

3.1.1　先行研究のハードウェアの統合

工藤のハードウェアにて，ホストにある配列データベースは，各々の FPGA の SD に分割された配列データベースが格納される．この際，APX880A の仕様上データバスは 128bit である．

BLAST 処理を行うには，須戸のハードウェアは UAP 処理に対してのみであり，事前に行われる LTP 処理と CSP 処理が必要となる．LTP 処理に必要となる配列がクエリのみであることからソフトウェアでの実行とし，CSP 処理はクエリおよびデータベースが必要なことから新たにハードウェア設計を行った．また須戸のハードウェアの入力は，検出された箇所からのクエリ配列，データベース配列，各々のインデックスの 4 つである．この 4 つをまずブロック RAM に格納し，その後適宜ブロック RAM から取得し処理が行われる．ストリーム処理エンジン [13] と須戸のハードウェアを用いた場合，流れてくるデータベースから須戸のハードウェアで UAP 処理対象をブロック RAM へと格納する際 UAP 処理が終わる前に次の処理対象が格納され続け，ブロック RAM の許容量を超えてしまう．このことからブロック RAM を排除し，CSP モジュールで UAP モジュールへの入力とリセットのタイミングを制御するように設計する．先行研究におけるハードウェア実装と各 BLAST 処理の流れを図 3.1.1 に示す．

CSP 処理のハードウェア

単一のクエリ配列において LTP 処理から生成される neighboring words を，SD にある配列データベースから検出する回路を作成する．

クエリ配列と neighboring words を register に配置し，配列データベースを回路に転送する．neighboring word を “SWW”，データベース配列のヘッダ行 “>NAME1”，シーケンスを “QISDMAFSWWTA” とした時の一連の流れを図 3.1.3，3.1.2，3.1.4 に示す．データベース配列をクエリ同様に 3 文字ずつの細かな配列に区切り，neigboring words とマッチしたインデックスを返す（図 3.1.3 の index）．一度のデータバスにデータベースの配列が収まらないため，配列を 3 文字に区切

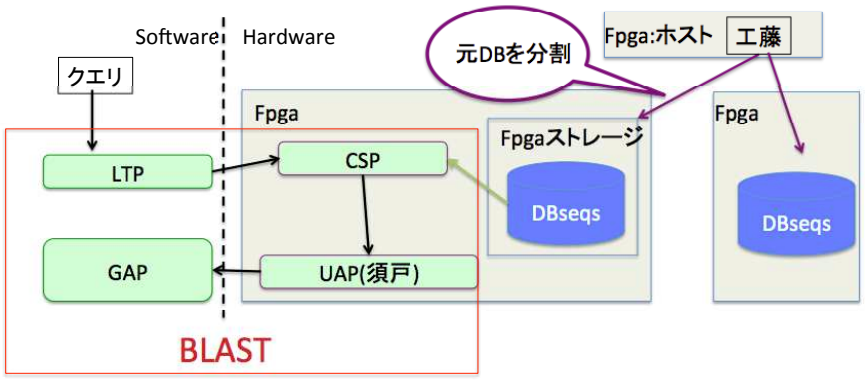

図 3.1.1: 先行研究を統合したハードウェア実装

る際，一つ前のデータバスで転送される文字列の末尾 2 文字を先頭にマージする（図 3.1.2 のマージされた DB 配列）．

次に須戸のハードウェアには検出位置からのクエリ配列，データベース配列を送信する必要がある．入力速度で流れるデータベースを，検出した index によってデータバスの先頭にくるようにデータバスを変更すると，index 以降の配列文字の続きがいくつあるかわからず，データバスの管理が複雑になる．そこで DB_index（図 3.1.3 の”S”）と同じ箇所に並ぶようにクエリを register から取得し，Query_index より前の文字を null に変更する．対応する 2 つの文字に対し，片方の文字を null にすることで，index 前の HSP 計算スコアを 0 にする（図 3.1.4）．

実データベースの FASTA の構造から各配列のヘッダにある”>” を用い，CSP モジュール，HSP モジュールの計算結果を初期化する．これによりデータベース内にある各配列毎に HSP 計算を可能にする．上記を実現したハードウェアの回路を図 3.2.1 に示す．

上記のハードウェア実装により，データの入力速度での計算を可能にする．しかしデータベースの転送におけるバス幅 128bit に対し，HSP 処理（須戸）を行うモジュールのバス幅が 32bit なことから，計算速度は転送速度の 1/4 になってしまう．また最も長い HSP を探すには，須戸の提案回路ひとつでは足りず，クエリとデータベース配列の検出回数分並べる必要があることがわかった．また CSP 処理で検出する度に検出位置からのデータベース配列とクエリ配列を UAP へ入力するため，データの転送が増えるほど計算速度が下がる．

FPGA 全体のリソース量，データの入力速度での転送と制御に使われるリソース量（ベースライン）と，および 1 つのデータベース配列につき 1 つの HSP 計算が行われる時のリソース量（提案回路）を追加した値を表 3.1 に示す．表 3.1 より，ベースラインに対し，提案回路を追加することで，Logic Utilization を 5%ほど消費してしまう．このことからデータの入力速度で HSP 計算が複数行われる時に，十分に拡張性があるとは言えない．

図 3.1.2: DB の末尾 2 文字のマージと 3 文字リストの生成の様子

3.1.2　データの転送速度で計算可能な構造のハードウェア

前節の計算速度，および拡張性の少ない問題を解決するため，新たに LTP，CSP，UAP 処理を行うハードウェア設計を提案する．ソフトウェア上で BLASTP の処理を速めるため，LTP 処理がまず行われるが，図 3.1.5，図 3.1.6，図 3.1.7 に示すように，データベース配列に対し，クエリを 1 文字ずらし，subquery として配置する．対応するデータベース配列の文字と subquery の文字同士のスコアを BLOSUM テーブルから算出する（図 3.1.6）．次に連続する 3 つのスコアを合計し triplet として算出する（図 3.1.7）．triplet を計算することで，ソフトウェア上で行われる neighboring threshold と同様にしきい値（T=12）以上になるかを判定し，CSP 処理と同様に UAP 処理の開始位置を判定できる．これにより neighboring words を作成する工程をなくすことができる．

次に UAP 処理について，あるクエリとデータベースにおいて，最も長い HSP を探すために必要な HSP 計算モジュールは 2 つである．これは図 3.1.8 に示すように，HSP 計算中に新たに HSP 計算を行う場合，後の HSP 計算が前の HSP 計算より長くなる可能性があるためである．図 3.1.8 の（A）の計算中に，新たに（B）,（C）の計算が行われた時，triplet の末尾の BLOSUM スコアが正の値（B）か負の値（C）かで場合分けすることができる．（B）の場合，HSP 終了判定に扱われる smax，stotal，xdrop の推移は（A）と必ず同じになる．これは xdrop の条件が-7 に対し，triplet の値が 12 以上なため，smax が必ず更新されるからである．（C）の場合，開始地点での smax と stotal が同じなため，（A）とは別に値が推移する．そのため（C）の HSP が（A）より長くなる可能性が

表 3.1: ハードウェア実装のリソース使用量

	リソース	ベースライン	提案回路
Logic Utilization[%]	100	75	80
Combinational ALUTs	182,400	81,129	85,049
Memory ALUTs	91,200	432	432
Dedicated logic registers	182,400	106,400	114,532

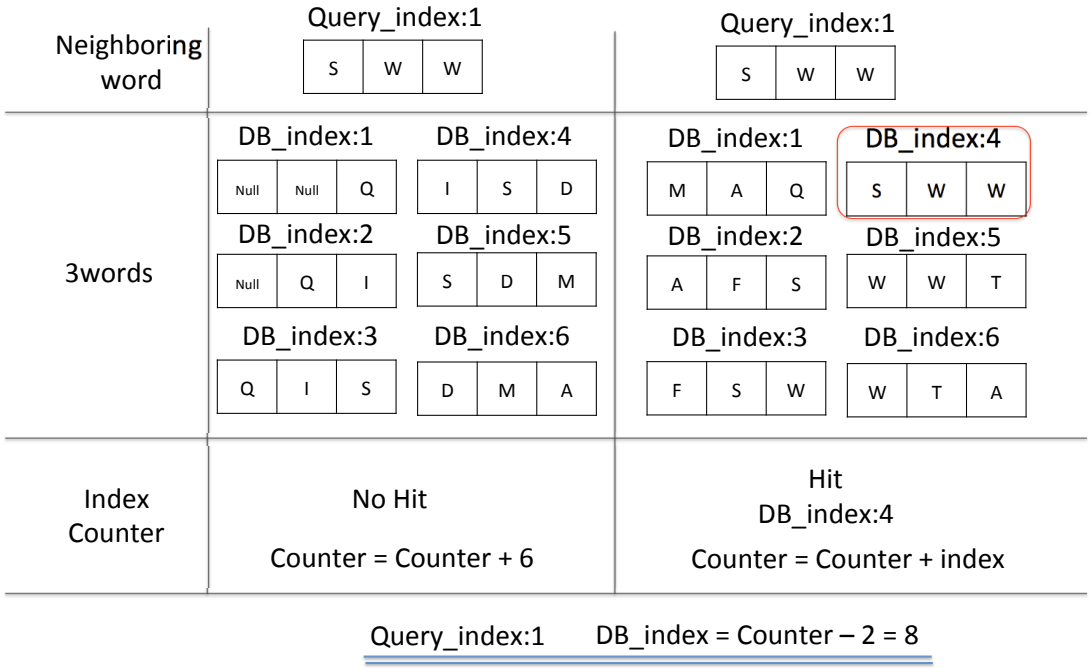

図 3.1.3: Neighboring Word と 3word の matching の様子

ある．また（C）の計算中に上記と同じことが発生した場合,（A）を計算が終了していれば,（A）を計算していたモジュールで計算を行う.（A）と（C）両方が計算中の時，HSP 計算終了位置が（A）と（C）で同じになるため,（C）の HSP 計算を破棄し，新たに計算を行う．このように HSP 計算が行われている時，新たに計算開始位置が見つかる場合でもモジュールは 2 つで十分である.

HSP 計算が終わった時，その Stotal 値，開始位置，終端位置を保存し，より長いものが現れたら更新する．実データベースで計算を行う際は，この 2 つのモジュールを SubQuery（図 3.1.5）の個数分用意する.

SubQuery が 1 つの場合のハードウェア回路図を図 3.2.2 に示す．上記のように実装することで，実データベースの転送速度でクエリに対し最も長い HSP を検出でき，拡張するための要素は，一度に扱うクエリの文字数のみである.

query		-	S	W	W	L	T
database		F	S	W	W	T	A
smax		0	4	15	26	26	26
stotal		0	4	15	26	25	25

図 3.1.4: クエリをずらした時の HSP 計算の様子

query		P	F	E	R	H	N	C	W
database		P	F	H	A	H	N	C	W
DB		P	F	H	A	H	N	C	W
SubQuery0		P	F	E	R	H	N	C	W
SubQuery1		F	E	R	H	N	C	W	P
SubQuery2		E	R	H	N	C	W	P	F
SubQuery3		R	H	N	C	W	P	F	E
SubQuery4		H	N	C	W	P	F	E	R
SubQuery5		N	C	W	P	F	E	R	H
SubQuery6		C	W	P	F	E	R	H	N
SubQuery7		W	P	F	E	R	H	N	C

図 3.1.5: データベース配列に対しクエリを一文字ずつずらした配置の様子

Bscore0		7	6	0	-1	8	6	9	11
Bscore1		-4	-3	0	-2	1	-3	-2	-4
Bscore2		-1	-3	8	-2	-3	-4	-3	1
Bscore3		-2	-1	1	0	-2	-2	-2	-3
Bscore4		-2	-3	-3	-3	-2	-3	-4	-3
Bscore5		-2	-2	-2	-1	-1	0	-3	-2
Bscore6		-3	1	-2	-2	0	0	-3	-4
Bscore7		-4	-4	-1	-1	0	1	-3	-2

図 3.1.6: 文字に対応した BLOSUM62 のスコアの様子

Triplet0			13	5	7	13	23	26
Triplet1			-7	-5	-1	-4	-4	-9
Triplet2			4	3	3	-9	-10	-6
Triplet3			-2	0	-1	-4	-6	-7
Triplet4			-8	-9	-8	-8	-9	-10
Triplet5			-6	-5	-4	-2	-4	-5
Triplet6			-4	-3	-4	-2	-3	-7
Triplet7			-9	-6	-2	0	-2	-4

図 3.1.7: neighboring threshold の様子

	query	C	D	I	W	H	W	K	I	F	Q	M	L	L	A	Q	R	P	K
	database	C	D	I	L	H	W	C	M	C	R	P	A	Q	G	D	R	F	L
	blosum62	9	6	4	-2	8	11	-3	1	-2	1	-2	-1	-2	0	0	5	-4	-2
	TripletSum	-	-	19	8	10	17	16	9	-4	0	-3	-2	-5	-3	-2	5	1	0
HSP																			
A	smax	-	-	19	19	19	25	36	36	36	36	36	36	36	36	36	36	36	36
	stotal	-	-	19	17	25	36	33	34	32	33	31	30	28	28	28	33	29	27
	Xdrop	-	-	-	2	0	0	3	2	4	3	5	6	8	8	8	3	7	9
	1.更新された時、二種類(i,ii)のパターンがある																		
B	smax	-	-	-	-	-	17	17	17	17	17	17	17	17	17	17	17	17	17
	stotal	-	-	-	-2	6	17	14	15	13	14	12	11	9	9	9	14	10	8
	xdrop	-	-	-	-	-	0	3	2	4	3	5	6	8	8	8	3	7	9
	1(i).triplet末尾のblosumスコア値が0以上の時、Aと計算終了条件は同じ																		
C	smax	-	-	-	-	-	-	16	17	17	17	17	17	17	17	17	17	17	17
	stotal	-	-	-	-	8	19	16	17	15	16	14	13	11	11	11	16	12	10
	xdrop	-	-	-	-	-	-	0	0	2	1	3	4	6	6	6	1	5	7
	1(ii).負の時、HSPがAより長い場合がある																		

図 3.1.8: HSP 計算の様子

3.2 実装

統合ハードウェアでは以下のモジュールを実装した.

- データベース配列の末尾 2 文字を次の先頭 2 文字と合わせるモジュール
- 同じ配列であるかを 3 文字ずつ判定するモジュール
- 開始位置を計算するモジュール
- 開始位置からのクエリとデータベースの配列を転送するモジュール

改善ハードウェアでは以下のモジュールを実装した.

- BLOSUM62 の値を参照するモジュール
- クエリとデータベースの配列の triple 計算を判定するモジュール
- 1 文字について Ltotal，Lmax，終了判定を計算するモジュール
- 2 つ HSP 計算を管理するモジュール

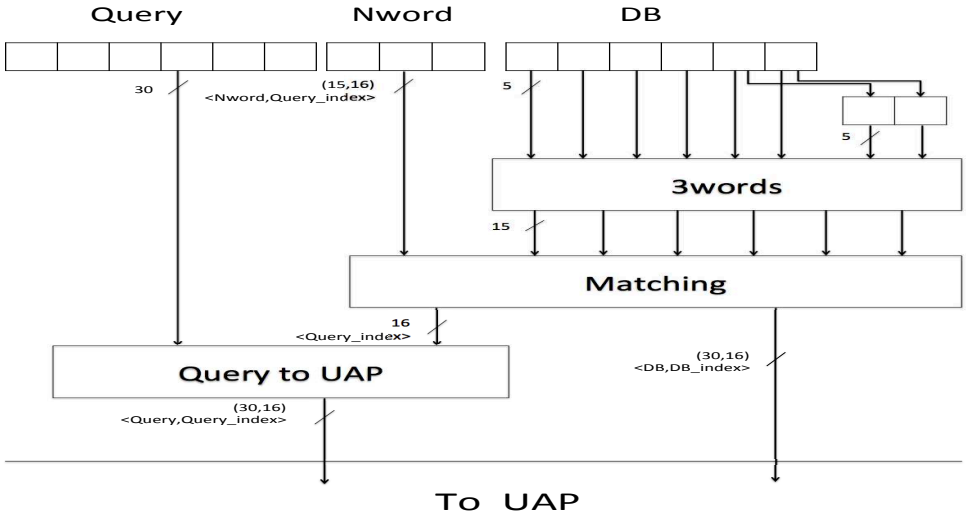

図 3.2.1: CSP 処理のハードウェア

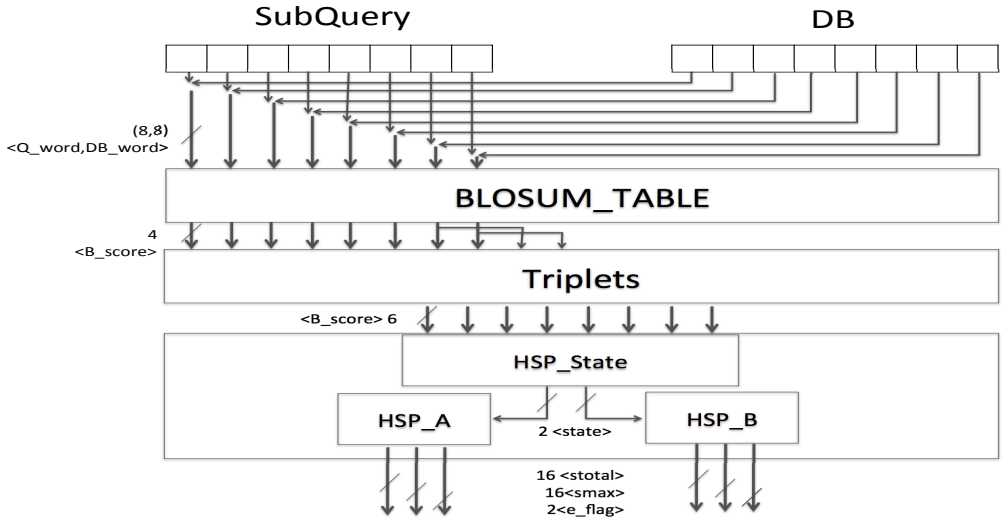

図 3.2.2: 改善のハードウェア

第4章　評価

4.1　評価環境

提案回路について，FPGA ボード（APX-880A）を使用し，APX-880A を搭載したホストを 4 台用いて，4 ノードで構成する PC クラスタを使用する．PC クラスタに用いたホストの構成を表 4.1 に示す．またホストのソフトウェア環境を表 4.2 に示す.

表 4.1: 実験用ホストの仕様

Node	4	
使用機種	Dell Precision T5610	
CPU	Intel Xeon E5-2630 v2	2.60[GHz]
メモリ	4GBx4 DDR3 RDIMM 1866MHz	16 [GB]
ネットワーク	Broadcom BCM957810A1008G	10[Gbps]
SSD	Crucial CT480M500SSD1	480[GB]
HDD	Seagate ST2000DM001	2[TB]

表 4.2: 実験用ホストのソフトウェア環境

OS	CentOS 6.7 kernel-2.6.32-504.1.3.el6.x86_64
SW Compiler	gcc-4.4.7
Compiler Option	-O2
HW CAD	Altera Quartus 10.1sp1
mpiBLAST	mpiBLAST-1.6.0

4.2 評価方法

始めに改善ハードウェアのリソース使用量について評価する．

次に Data Partitioning フェーズでの実行時間，および BLAST フェーズの CSP，LTP，UAP の実行時間について mpiBLAST と比較評価する．Data Partitioning フェーズにおいて，工藤のハードウェア実装では，分割対象となるデータベースを送信ノードのストレージである SD に格納し，分割処理を行う．処理によって得られた分割されたデータは受信用の 3 ノードの SD に格納する．これに対して，ソフトウェア実装である mpiBLAST の実験では，データベースをホストの SSD に格納し，2.5 節で述べた mpiformatdb 処理を行う．処理によって得られた分割データをホストの 10 gigabit Ethernet を用いて他のノードに分散する．

BLAST フェーズにおいて，2.6 節で述べたように，GAP 処理の負荷を軽減するため，事前に処理が行われる．3.2 節で実装した改善ハードウェアと同じ処理箇所のソフトウェアを比較評価する．

評価に使用した実データベースはデータサイズが 1.73GB の env_nr である．最後に GAP 処理をホストで行った際の，配列類似検索の並列処理全体の見積もりについて評価する．

4.2.1 改善ハードウェアのリソース使用量

3.1.2 節において，一度に計算できるクエリの文字数を 16，SubQuery を 1 つの時に実装した時のリソース使用量を表 4.3 に示す．本来，文字数 16 のクエリに対し，必要な SubQuery は 16 種類である．改善した HW のスループットは，984[MB/sec] であり，実データベースでの入力速度での計算が確認できた．しかし表 4.3 から 1 つの SubQuery を行う際に必要な combinational ALUTs は 5,613 であり，一度に計算できるクエリ配列が 16 文字程度しかなく非常に短いという問題が生じた．

表 4.3: 改善 HW 実装のリソース使用量

	リソース	ベースライン	提案回路
Logic Utilization[%]	100	75	80
Combinational ALUTs	182,400	81,129	86,742
Memory ALUTs	91,200	432	432
Dedicated logic registers	182,400	106,400	106,400

4.2.2 Data Partitioning フェーズの実行時間の評価

env_nr を各ノードへデータを分割する実行時間の評価を表 4.4 に示す．また FPGA での分割は，予めホストにて Padding を行う必要がある．Padding 前後のデータサイズを表 4.5 に示す．

mpiBLAST では，mpiformatdb で実行される時間は formatdb に 231.9 秒，format 後のデータを分割する時間は 24.9 秒である．工藤のモジュールでは，ホストで Padding を行った時間は 222.0 秒，Padding 後のデータを分割する時間は 1.7 秒である．formatdb 処理および Padding 処理は，どちらも env_nr のテキスト処理をホストで行っており，差が 4.5%ほどで実行時間の差はほぼない．デー

タベースの分割時間については，工藤 [13] の実験と同様のスループットが確認でき，14.6 倍の高速化が確認された．Data Partiotioning フェーズ全体の実行時間としては，1.1 倍ほどの高速化が確認された．

表 4.4: Data Partitioning フェーズの実行時間比較

	テキスト処理時間 [s]	データ分割時間 [s]	全体実行時間 [s]
mpiBLAST	231.9	24.9	256.8
HW	222.0	1.7	223.7

表 4.5: env_nr データベースの Padding 後のデータサイズ比較

	Padding 前 [GB]	Padding 後 [GB]
env_nr	1.73	1.78

4.2.3 BLAST フェーズの実行時間の評価

mpiBLAST では，各ノードへのデータサイズが均等になるよう分割されるため，各ノードでのデータサイズは 600MB ほどである．mpiBLAST での BLAST フェーズはクエリの配列により実行時間が異なる．そのため，本評価では，タンパク質において最も短い配列”1LY2” であると最も長い配列”Q8WZ42” をクエリとし，実行時間を測る．配列長はそれぞれ 20 と 34,350 である．一方ハードウェアの実装において，BLAST フェーズの実行時間は，データベースのデータサイズに依存する．これはデータの入力速度での計算を可能に設計を行ったからである．

mpiBLAST の BLAST フェーズにおける実行時間は GAP 処理も含まれている．そのため BLASTP のプロファイル（表 4.7，表 4.8）から GAP の実行時間の割合を調べ，LTP から GAP 前までの処理時間を逆算する．

各ノードでの提案した改善ハードウェア実装における実行時間の見積もり，および mpiBLAST での 2 種のクエリによる実行時間を表 4.6 に示す．

実データベースを 3 つに分割した際，各ノードの BLAST の実行時間は同じだった．これは単一のクエリに対し，非常に大きいデータベースと配列類似性検索行うため，処理時間よりもデータの読み書きに起因するものと考えられる．

GAP 処理時間は表 4.8 から，Blast_SemiGappedAlign，s_RestrictedGappedAlign，ALIGN_EX の 3 つの実行時間である．配列 “1LY2” において，GAP 処理時間の時間割合は 0%であり，UAP 処理である BLASTAaWordFinder は全体の 10%ほどある．クエリの neighboring words を配列データベースから検出しているが，HSP が統計学的に類似と言えるほど拡張しないと s_BlastAaExtendTwoHit が判断しているため，実行結果では類似した配列が配列データベースに存在しないとあった．

改善 HW により，配列”1LY2” での BLAST フェーズは 6.5 倍，配列”Q8WZ42” での BLAST フェーズは 840 倍の高速化が見込める．

表 4.6: BLAST フェーズの実行時間比較

	nodeA	nodeB	nodeC
	LTP to UAP[s]	LTP to GAP[s]	LTP to GAP[s]
mpiBLAST(1LY2)	3.9	3.9	3.9
mpiBLAST(Q8WZ42)	504.2	504.1	504.2
改善 HW	0.6	0.6	0.6

4.2.4 配列類似性検索の並列処理の評価

4.3.1 節で用いたたデータベース”env_nr” と 4.3.2 節で用いたクエリ”1LY2”，”Q8WZ42” との配列類似性検索を行う．GAP 処理は 2.4.4 節で述べたように，処理が複雑かつ実行時間割合が少ない．そのためホストで GAP 処理を行う．その際の mpiBLAST の実行時間と FPGA による並列処理の全体実行時間の見積もりを表 4.9 に示す．

配列”1LY2” では，GAP 処理がなく，BLAST の実行時間も短い．4.3.1 節で述べたテキスト処理がどちらも実行時間全体の 9 割を占めことから，1.16 倍，配列”Q8WZ42” では，BLAST の実行時間がテキスト処理よりも多く，2.64 倍の高速化が見込まれる．

表 4.7: 1LY2 のプロファイル結果

nodeA		nodeB		nodeC	
% time	name	% time	name	% time	name
69.58	s_BlastSmallAaScanSubject	75.97	s_BlastSmallAaScanSubject	70.29	s_BlastSmallAaScanSubject
8.33	BlastAaWordFinder	10.30	BlastAaWordFinder	9.21	BlastAaWordFinder
7.92	s_BlastSearchEngineCore	4.29	s_BlastSearchEngineCore	5.86	s_BlastSearchEngineCore
2.92	s_BlastAaExtendTwoHit	1.29	s_BlastAaExtendTwoHit	1.67	s_BlastAaExtendTwoHit
0.00	Blast_SemiGappedAlign	0.00	Blast_SemiGappedAlign	0.00	Blast_SemiGappedAlign
0.00	s_RestrictedGappedAlign	0.00	s_RestrictedGappedAlign	0.00	s_RestrictedGappedAlign
0.00	ALIGN_EX	0.00	ALIGN_EX	0.00	ALIGN_EX

表 4.8: Q8WZ42 のプロファイル結果

nodeA		nodeB		nodeC	
% time	name	% time	name	% time	name
44.19	BlastAaWordFider	44.41	BlastAaWordFider	43.86	BlastAaWordFider
27.21	s_BlastAaExtendTwoHit	27.18	s_BlastAaExtendTwoHit	27.50	s_BlastAaExtendTwoHit
10.60	s_BlastSmallAaScanSubject	10.49	s_BlastSmallAaScanSubject	10.44	s_BlastSmallAaScanSubject
8.07	Blast_SemiGappedAlign	8.12	Blast_SemiGappedAlign	8.23	Blast_SemiGappedAlign
7.55	s_RestrictedGappedAlign	7.55	s_RestrictedGappedAlign	7.68	s_RestrictedGappedAlign
1.26	ALIGN_EX	1.09	ALIGN_EX	1.11	ALIGN_EX

表 4.9: 並列処理の実行時間比較

	mpiBLAST[s]	FPGA による並列処理 [s]	高速化率
Query(1LY2)	260.7	224.3	1.16
Query(Q8WZ42)	864.4	327.7	2.64

第5章　結論

配列データベースの増加速度がプロセッサの向上速度よりも速いことから配列類似性検索の高速化が期待されている．本研究では，BLAST の高速化について，複数の計算機を用いた並列演算とアルゴリズムの高速化に着目し，両方を実現できるハードウェアを提案し，FPGA ボードを用いて高速化を示した．さらに FPGA の利点である自由なロジック構成が可能なことから，データの転送速度でのデータに対し計算処理ができるハードウェアを提案した．先行研究と本論文で実装したハードウェアについて，mpiBLAST との実行時間の比較評価を行った．Data Partitioning フェーズでは 1.1 倍，BLAST フェーズでは 6.5〜840 倍の高速化を確認した．またホストで行う BLAST の箇所を想定した時の全体実行時間は 1.16〜2.64 倍の高速化が見込まれる．

今後は，改善ハードウェアにおける実装で，一度に行えるクエリ配列の文字数の拡張を提案し，FPGA のリソースについて再度評価を行う．またハードウェア実装から得られた出力をホスト上で入力となるように，アルゴリズムのさらなる調査と実実行時間比較を評価する．

謝辞

本研究を進めるにあたり，ご指導を頂きました吉永努教授に感謝の意を表します．ゼミにおける議論を通し，多くのご示唆を頂きました．

また，同講座吉見真聡助教にも研究を進めるにあたり多くのご助言を頂き，感謝致します．

最後に日常の議論を通じて多くのご指摘，ご協力を下さいましたネットワークコンピューティング学講座の皆様に感謝致します．

参考文献

[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, Vol. 215, No. 3, pp. 403 – 410, 1990.

[2] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, Vol. 25, No. 17, pp. 3389–3402, 1997.

[3] The Developer's Guide to BLAST. https://github.com/elucify/blast-docs/wiki.

[4] NCBIBLAST Database. http://ftp.ncbi.nlm.nih.gov/blast/db/FASTA/.

[5] Monthly Data Submitter and Session to DDBJ. http://www.ddbj.nig.ac.jp/breakdown_stats/ddbj_submissions-e.html.

[6] UniProtKB/Swiss-Prot Release. http://web.expasy.org/docs/relnotes/relstat.html.

[7] A. Jacob, J. Lancaster, J. Buhler, and R.D. Chamberlain. FPGA-accelerated seed generation in Mercury BLASTP. In *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pp. 95–106, April 2007.

[8] P. Laczkó, B. Fehér, and B. Benyo. FPGA-based BLAST prefiltering. In *Intelligent Engineering Systems (INES), 2010 14th International Conference on*, pp. 303–306, May 2010.

[9] 石川淑, 田中飛鳥, 宮崎敏明. FPGA を用いた BLAST アルゴリズムの高速化. 情報処理学会論文誌, Vol. 55, No. 3, pp. 1167–1176, mar 2014.

[10] Yoshiki Yamaguchi, HungKuen Tsoi, and Wayne Luk. FPGA-Based Smith-Waterman Algorithm: Analysis and Novel Design. In *Reconfigurable Computing: Architectures, Tools and Applications*, Vol. 6578 of *Lecture Notes in Computer Science*, pp. 181–192. Springer Berlin Heidelberg, 2011.

[11] 須戸里織. 配列類似性検索の FPGA による高速化. 電気通信大学大学院情報システム学研究科修士論文, Feb 2015.

[12] Darling, A. E., L. Carey, and W. - C. Feng. "the design, implementation, and evaluation of mpiblast". Jun 2003.

[13] 工藤龍, 須戸里織, オゲヤースィン, 寺田裕太, 吉見真聡, 入江英嗣, 吉永努. 複数 FPGA ボードを用いたビッグデータ分割処理の高速化. 信学情報, Vol. 114, No. 427, pp. 193–198, Jan 2015.

[14] David W. Mount. *Bioinformatics Sequence and Genome Analysis, Second Edition*. Cold Spring Harbor Laboratory, 2005.

[15] W R Pearson and D J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences of the United States of America*, Vol. 85, No. 8, pp. 2444–2448, Apr. 1988.

[16] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *File-Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[17] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for Deep-submicron FPGAs*.

[18] 三好健文. インターフェース ZERO No.4. CQ 出版株式会社, 2013.

[19] AVAL DATA. 高速ストレージボード：APX880A. http://www.avaldata.co.jp/products/z1_embedded_zz/avalother_products/apx880/apx880.html.

[20] Joseph Lancaster, Jeremy Buhler, and Roger D. Chamberlain. Acceleration of Ungapped Extension in Mercury BLAST. *Microprocess. Microsyst.*, Vol. 33, No. 4, pp. 281–289, June 2009.

[21] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, Vol. 147, No. 1, pp. 195–197, March 1981.