ショートリードマッピング
高速計算システムの研究

２０１８年3月

曽我部　陽光

ショートリードマッピング高速計算システムの研究

曽我部　陽光

システム情報工学研究科
筑波大学

２０１８年３月

目 次

図 目 次

表 目 次

第1章
序論

本研究では，書き換え可能なLSIであるField Programmable Gate Array（以下，FPGAと呼ぶ）と画像処理用のプロセッサであるGraphics Processing Unit（以下，GPUと呼ぶ）を用いて，遺伝子情報処理分野におけるショートリードマッピングと呼ばれる問題を高速に処理する計算機システムの構築を行う．

ショートリードマッピングは，DNAシーケンシングに必要な情報処理である．DNAシーケンシグとは，対象DNAの塩基配列を決定することであり，幅広い応用範囲を持つ．特にヒトのDNAシーケンシングは，オーダメイド医療と呼ばれる遺伝的特徴に基づいた新しい医療の実現に向けて，需要が高まっている．

ショートリードマッピングには，技術的な課題が主に2つある．第一に，処理するショートリード数が膨大なため，処理速度が重要である．例えば，ヒトのDNA解析の場合は100塩基長のショートリードを約10億個処理することが必要となる．そのため，ショートリードマッピングが，DNAシーケンシングのボトルネックになっている．第二に，ショートリードと参照配列が100%一致するとは限らず，ある程度変異を含んでいても正しくマッピングできることが求められる．これらを踏まえ，本研究では，処理速度とマッピング成功率の2つの観点からショートリードマッピングの手法を評価する．

DNAシーケンシングは，高速かつ安価な処理が求められ，普及のためには，個人や大学の研究室レベルで導入可能な安価なコンピュータシステムで高性能（高効率）なショートリードマッピング処置を実現することが重要であると考えた．本研究では，パーソナルコンピュータとそれに付加したFPGAあるいはGPUという構成の比較的小規模な計算機システムで高速化を図る．

本論文では，ソートと並列比較に基づくショートリードマッピング並列処理アルゴリズムを提案し，メモリアクセスがボトルネックとならずFPGA及びGPUの回路資源量に応じて高速化可能な手法を提案する．本手法では，初めにショートリードのソーティングを行い同種のデータを集め，集まったデータを1000個以上並列に比較する．

本手法を FPGA 及び GPU システム上に実装し，その有効性を明らかにした．有力なソフトウェアマッピングツールである BWA-MEM [1] と比較して，FPGA システムで約 18 倍，GPU システムで約 8 倍の高速化を達成した．また，マッピング成功率に関しては，ショートリードの変異が少ない場合で BWA-MEM と同程度，変異が多い場合で BWA-MEM を上回る結果が得られた．

本研究で提案する手法は，従来のハードウェアシステムとは異なり，メモリ転送速度の制約をほとんど受けない．したがって，今後の LSI 技術の発展によって，より大規模な FPGA，GPU が開発されるに伴い，より高速な処理が実現可能となる．このことは，提案手法が，本論文で示したハードウェア構成において他の手法に対して優れているだけでなく，将来開発されるであろう，より大規模なハードウェアを用いれば，より優れた性能が実現可能であることを意味する．DNA シーケンサーの性能（ショートリードの生成速度）向上及び DNA シーケンシングの需要拡大が予想されているが，CPU の性能向上が鈍化しているため，CPU の性能に依存するソフトウェアアプローチでは，将来求められる性能に対応できない．提案手法を用いれば，大規模なハードウェアを用いることで将来においても十分な性能が達成できると考えられる．

ショートリードマッピングは，処理の潜在的な並列性が非常に高い一方で，大規模なデータに対して局所性が低い参照を頻繁に行う問題である．このような問題は，メモリ転送速度がボトルネックとなるため，FPGA や GPU を用いた並列処理で大幅な高速化を実現することはできないと考えられて来た．本研究は，その定説に挑戦したものであり，本研究の成果を通して，他の同様の問題に対しても，突破口のヒントを与えうるものである．

本論文の構成を以下に述べる．まず，第 2 章で，ショートリードマッピングについてその背景や問題の特徴を述べ，主要な既存のアルゴリズムについても述べる．第 3 章では，ショートリードマッピング高速化のためにこれまでに提案された既存の FPGA システム，GPU システムについて述べる．第 4 章では，本論文で提案するソートと並列比較に基づくショートリードマッピング並列処理アルゴリズムについて述べる．第 5 章では，提案手法の FPGA を用いたシステム構築について，第 6 章では，GPU を用いたシステム構築について述べる．第 7 章では，本 FPGA システムおよび GPU システムを既存のソフトウェアマッピングツール，GPU システム，FPGA システムと性能比較を行い，提案手法の有効性を明らかに

する．最後に，第 8 章で，まとめと今後の課題について述べる．

第2章
ショートリードマッピング

本章では，ショートリードマッピングについて説明し，既存のソフトウェアとそのアルゴリズムについて概説する．

2.1　ショートリードマッピングの概要

DNA シーケンシングとは，対象 DNA の塩基配列をアデニン（A），シトシン（C），グアニン（G）, チミン（T）の 4 種類の塩基の並びとして決定することであり，遺伝子解析における最も基本的で重要な問題のひとつである．DNA シーケンシング技術は，生物学以外にも，医学，創薬研究，農業分野，法医学など幅広い応用範囲を持つ．特に，オーダメイド医療と呼ばれる個人の DNA 配列に基づいた医療の実現が注目されており，DNA シーケンシングの需要が高まっている．オーダメイド医療は，DNA 配列によって，将来の疾患の予測や適切な治療法の推定を行うのが目的である．

2000 年代後半に，Next-Generation Sequencing（NGS）[2] と呼ばれる技術が登場し，従来に比べて非常に高速で低価格な DNA シーケンシングが可能となった．図 1 に，NGS 技術を用いた DNA シーケンシングの概要を示す．DNA の塩基配列を高速に自動で読み取る装置を DNA シーケンサーと呼び，この装置から得られるデータは，ランダムに断片化した固定長の塩基配列（この固定長の塩基配列をリードと呼ぶ）である．NGS では，次世代シーケンサーと呼ばれる DNA シーケンサーが用いられ（この次世代 DNA シーケンサー自体を NGS（Next-Generation Sequencer）と呼ぶ場合もある），従来型に比べ著しく性能が向上しており，ヒトゲノムを一日以内に読み出すことも可能である．また，次世代シーケンサーで読み出されるリードは，数百塩基長以下の短いリードであるため，このリードをショートリードと呼ぶ．

次世代シーケンサーが読みだしたショートリードは，DNA 配列全体でどの部分に位置していたかは不明であり，このままではショートリードから元の DNA 配列を決定することはできない．モデル生物の場合では，ゲ

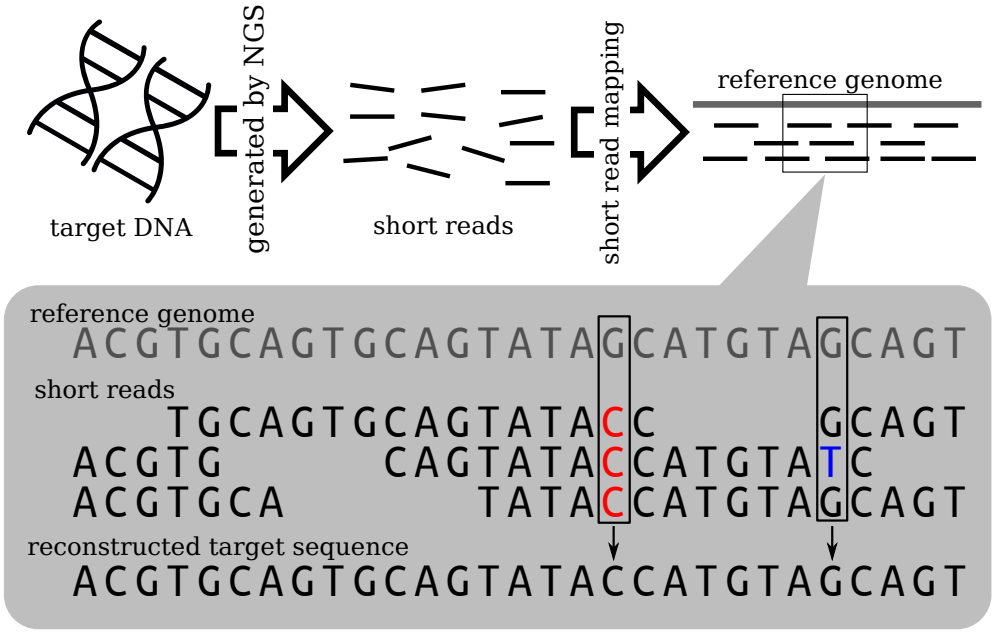


図 1: NGS 技術を用いた DNA シーケンシングの概要

ノムが解読されているため，ショートリードをその生物種の既知のゲノム（参照配列と呼ぶ）と照らし合わせる（マッピング）ことで，各ショートリードが全体の DNA 配列のどこに位置しているかを決定することができる．この処理を，ショートリードマッピング（あるいは，ショートリードアライメント）と呼ぶ．ショートリードマッピングとは，ショートリード毎に参照配列中でもっとも類似している位置を見つける処理である．最後に，参照配列とマッピングされたショートリードの比較から読み出された DNA の遺伝的特徴を得る．図 1 にあるように，読み出された塩基配列の信頼性を確保するために，DNA シーケンサーで複数重複したショートリードを生成している．図 1 の青文字の ‘T’ は，DNA シーケンサーの読み出し時のエラーであり，赤文字の ‘C’ は，対象 DNA の遺伝的な特徴である．DNA シーケンサーの読み出し時のエラーは，単にランダムに発生することを利用し，重複してマッピングされたショートリードを使って，読み出し時のエラーを打ち消す．

次世代シーケンサーは，非常に高速であるため，ショートリードマッピングが DNA シーケンシングのボトルネックとなっており，その高速化が課題となっている．また，次世代シーケンサーは，登場以後，その性

能および低価格化がムーアの法則を超える速度で向上を続けており [3]，ショートリードマッピングの高速化の需要は，ますます増大すると言われている．

ショートリードマッピングには，大きく分けて 2 つの課題がある．一つ目は，処理速度である．非常に大量のショートリードを処理する必要があるために，優れた処理速度が求められる．例えば，ヒトの全ゲノムシーケンシングでは，参照配列が 3G 塩基長であり，100 塩基長のショートリードを 1G 個程度処理する必要がある（約 30 重複）．2 つ目は，変異を含んだ検索能力である．参照配列とショートリードは，非常に類似しているものの，僅かに変異を含んでいる．この変異を発見し，対象 DNA の遺伝的特徴を検出することが，DNA シーケンシングの目的でもあるが，変異を含んだ検索は，この問題を非常に難しくしている．

変異には，SNV（Single Nucleotide Variant），indel（insert/deletion），SV（Structual Variation）等がある．SNV は，ある 1 塩基が他の塩基に置換する変異である．indel は，短い塩基配列が挿入/欠損する変異である．SV は，より大規模な変異であり，いくつか種類がある．例えば，特定配列の繰り返し回数が変化したり，一定の長さの塩基配列が反転するなどが含まれる．複雑な変異ほど検出は難しくなる．例えば，長さ L の塩基配列間の SNV を発見する計算量は単に $O(L)$ であるが，indel の場合，シーケンスアラインメントと呼ばれる処理が必要になり，計算量は $O(L^2)$ である．indel の場合は，挿入や欠損によって対応する塩基の関係がずれるため，ギャップを挿入し対応する塩基が同じ箇所に並ぶようにする（アラインメント）．例えば，“CAGT” と “CAAGT” の場合，“CA-GT” と “CAAGT”（‘-’ がギャップを意味する文字）というようギャップ入れアラインメントを行う．このように，変異を考慮する場合，その計算量が非常に増えることが，ショートリードマッピングをより難しくしている．実際に任意の変異を考慮するのは，計算量の問題から困難なため，ヒューリスティックスが用いられる．SNV は，計算しやすく実際の DNA 上での出現量も多いために，ほぼすべてのリードマッピングツールで対応されている．一方で，indel に関しては，初期のショートリードマッピングツールの中にはギャップに対応しておらず，検出できないものもある．

以上に述べたように，ショートリードマッピングは，処理速度，変異を含んだ時に正しくマッピングできるか（マッピング率）という 2 点で評価すべきである．

ここで，ショートリードマッピングで使われるファイル形式について

述べる．次世代シーケンサーが読み出したショートリードは，FASTQ 形式 [4] で与えられるのが一般的である．FASTQ 形式は，テキストベースのファイル形式であり，塩基配列と塩基毎にその信頼度を示すクオリティスコアを ASCII 文字に変換したものが付加されている．マッピング結果は，SAM 形式 [5] で出力される．SAM 形式は，マッピング結果を示すためのテキストベースのファイルであり，FASTQ 形式にマッピング位置やそのマッピング位置の信頼度，アラインメントによるギャップの位置等を付加した形式である．参照配列は，FASTA 形式 [6] で与えられる．FASTA 形式は，単に塩基コード（‘A’，‘C’，‘G’，‘T’ や不明を意味する ‘N’ 等）が並んだテキストファイルである．以上をまとめると，ショートリードマッピングは，FASTQ 形式で与えられたショートリードを FASTA 形式で与えられた参照配列にマッピングし，その結果を SAM 形式で出力するという処理になる．

2.2 ショートリードマッピングアルゴリズム

高速で高精度なショートリードマッピングを実現するために，様々なアルゴリズムが提案されてきた．それらのアルゴリズムでは，[7] で述べられているように，seed-and-extend 法と呼ばれるヒューリスティックスが用いられることが多い．

seed-and-extend 法では，ショートリードマッピングを 2 段階に分けて行う．1 段階目は，seeding と呼ばれ，ショートリード全体ではなく，一部を使ってマッピングの候補位置を検出する．このマッピングの候補位置を Candidate Alignment Location(CAL) と呼ぶ．ショートリードと参照配列の間には変異が存在するため，ショートリード全体を使うと多くの変異を考慮しなければならず，計算量が増える．そこで，ショートリードの一部だけであれば，含まれる変異は，少なくなる（あるいは，全くなくなる）ため，簡単な計算で済む．seeding 段階では，ショートリードの一部だけ（seed と呼ぶ）を使って，参照配列と一致する場所を見つけ，その場所を CAL とする．つまり，CAL は，ショートリードの少なくとも一部が一致する参照配列中の位置である．2 段階目は，extension と呼ばれ，seeding 段階で得られた CAL について，ショートリード全体をつかって，置換・欠損・挿入の変異を考慮しながら評価する．一般には，Smith-Waterman 法 [8] が使われることが多い．seeding 段階で，複数の CAL が得られて，

extension 段階で，それらの CAL を評価し最良のものを選ぶというのが，seed-and-extend 法の基本的な方針である．

seeding 段階では，参照配列中で seed との一致箇所を高速に検出するために，事前に参照配列から作成した検索用 index を用いることが一般的である．[9]，[10]，[11] では，数々のショートリードマッピングアルゴリズムを，index のデータ構造によって FM-index 法 [12] と hash-index 法という 2 種類に分類している．

2.2.1 FM-index 法

FM-index [12] とは，Paolo Ferragina と Giovanni Manzini によって開発された部分文字列用の検索用 index である．FM-index は，長さ l の塩基配列の参照配列中で出現位置を，参照配列の長さに依存せず $O(l)$ で計算できるという特徴を持つ．またメモリ消費量も比較的少なく，ヒトゲノムで 4 GB 程度の容量となる．

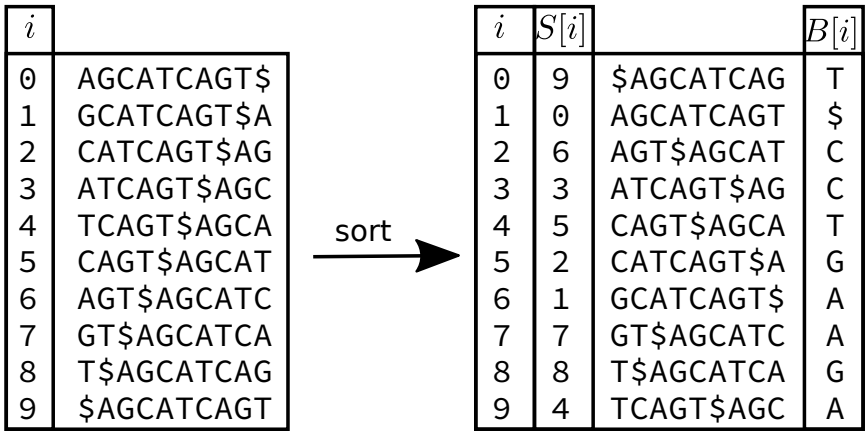


図 2: Burrows-Wheeler Transform(BWT)

FM-index では，Burrows-Wheeler Transform(BWT) [13]（ブロックソートとも呼ぶ）を利用する．図 2 に，参照配列を “AGCATCAGT” とした時の BWT の例を示す．図 2 の左の表では，“AGCATCAGT” の末尾に区切り文字 ‘$’ を加えた文字列を一文字づつ左に循環シフトした文字列を生成している．i は，単に通し番号であり，左に循環シフトした回数でもある．図 2 の右の表は，循環シフトによって生成された文字列を辞書順にソート

したものである．$S[i]$ は，suffix array と呼ばれ，循環シフトして生成された文字列のソート前の位置を示す．$B[i]$ は，“AGCATCAGT” の BWT であり，循環シフトして生成された文字列をソートしたものの末尾の文字を集めて文字列にしたものである．この例の場合，“AGCATCAGT$” を BWT したものが $B[i]$ =“T$CCTGAAGA” である

i	$S[i]$		$B[i]$
0	9	$AGCATCAG	T
1	0	AGCATCAGT	$
2	6	AGT$AGCAT	C
3	3	ATCAGT$AG	C
4	5	CAGT$AGCA	T
5	2	CATCAGT$A	G
6	1	GCATCAGT$	A
7	7	GT$AGCATC	A
8	8	T$AGCATCA	G
9	4	TCAGT$AGC	A

$Occ[c, i]$

	A	C	G	T
0	0	0	0	1
1	0	0	0	1
2	0	1	0	1
3	0	2	0	1
4	0	2	0	2
5	0	2	1	2
6	1	2	1	2
7	2	2	1	2
8	2	2	2	2
9	3	2	2	2
$C[c]$	0	3	5	7

図 3: FM-index の概要

FM-index では，BWT で得られた $S[i]$,$B[i]$ の他に $Occ[c, i]$,$C[c]$ という配列を利用する．図 3 に，参照配列を “AGCATCAGT” とした時の FM-index の例を示す．$Occ[c, i]$ は，$B[i]$ 上で i 地点までの文字 c の出現回数を示す配列である．$C[c]$ は，テキスト中での文字 c より辞書順が小さい文字の総出現回数である．

FM-indexでは，$S[i]$,$B[i]$,$Occ[c, i]$,$C[c]$ を用いて，検索を行う．FM-indexでは，塩基配列 $\boldsymbol{X}$ の参照配列中の出現位置は，suffix array 上の区間 $[s(\boldsymbol{X}),e(\boldsymbol{X})]$ として与えられる．例えば，文字列 $\boldsymbol{X}$=“CA” とした時の検索結果は，$[s(\boldsymbol{X}),e(\boldsymbol{X})]$=[4,5] となり，“CA” の “AGCATCAGT” 中での出現回数は 2(=5-4+1) 回とわかる．suffix array 上の区間 $[s(\boldsymbol{X}),e(\boldsymbol{X})]$ から，出現場所を得るには，$S[i]$ を参照すればよく，$S[4] = 5$,$S[5] = 2$，つまり，‘CA’ は，‘AGCATCAGT’ の位置 5 と 2 に出現していることがわかる．

実際に，ある文字列の出現位置を検索するためには，次に示す定義にしたがって再帰的に計算を行う．文字列 $\boldsymbol{X}$ の先頭に文字 a を追加した文

字列の $s(a\boldsymbol{X})$，$e(a\boldsymbol{X})$ は，$\boldsymbol{X}$ を使って，次のように再帰的に定義される．

$$\begin{aligned} s(a\boldsymbol{X}) &= C(a) + Occ[a, s(\boldsymbol{X}) - 1] - 1 \\ e(a\boldsymbol{X}) &= C(a) + Occ[a, e(\boldsymbol{X})] \end{aligned}$$

表 1: 塩基配列 'CA' を検索する場合の FM-index の計算例

Iteration	a	$\boldsymbol{X}$	s	e
Init			0	9
1	'A'	$\varnothing$	$C['A'] + Occ['A', 0-1] + 1 = 1$	$C['A'] + Occ['A', 9] = 3$
2	'C'	'A'	$C['C'] + Occ['C', 1-1] = 4$	$C['C'] + Occ['C', 3] = 5$
Return			4	5

表 1 に，塩基配列 'CA' を検索する場合の FM-index の計算例を示す．表 1 では，再帰的な定義にしたがって，まず，a='A'，$\boldsymbol{X}=\varnothing$ について計算し，次に，a='C'，$\boldsymbol{X}$='A' について計算するという 2 回の計算で結果が得られている．表 1 にあるように，まず初めに $s(\varnothing) = 0$，$e(\varnothing) = 9$ と初期化する．そして，a='A'，$\boldsymbol{X}=\varnothing$ として，定義に基づいて $s(`A') = 1$ と $e(`A') = 3$ を計算する．次に，a='C'，$\boldsymbol{X}$='A' として，前回の結果である $s(`A') = 1$ と $e(`A') = 3$ を利用して，$s(`CA') = 4$ と $e(`CA') = 5$ を計算する．これで，'CA' の検索が完了する．このように，FM-index では，$a\boldsymbol{X}$ というように先頭方向 1 塩基づつ拡張していくことで，長さ l の塩基配列の参照配列中で出現回数を $O(l)$ で計算できる．

ここまでの説明は，FM-index を用いた exact match の検索について述べたが，ショートリードと参照配列は，僅かな変異を含むことが想定される．FM-index では，基本的に exact match の検索しかできないが，$a \in ['A', 'C', 'G', 'T']$ とすることで，$\boldsymbol{X}$ の先頭に任意の 1 塩基を追加した塩基配列を検索できる．ただし，この方法は，許容する変異量に対して計算量が指数関数的に増加する．

実際の参照配列は非常に長く，ヒトゲノムの場合は約 3G 塩基長となるため，$S[i]$,$Occ[c, i]$ は，各要素を 4 B としてもそれぞれ，12 GB，48 GB と膨大な容量になってしまうが，[14] において，圧縮手法が提案されており，実際のソフトウェア [15]，[16] 等で用いられている．[14] では，$Occ[c, i]$ を一定間隔（典型的には，128）に飛び飛びで記憶する（$Occ[c, 128n]$（n は任

意の整数）のみ記憶する）ことで，$Occ[c,i]$ の容量は 1 GB 以内になる．記憶されていない間の領域 ($i = 128n + k$（n は任意の整数,k は $1 \leq k \leq 127$ の整数）) は，$Occ[c,i]$ の定義にしたがって，$B[128n]$ から $B[128n + k]$ まで読み出し，c の出現回数をカウントし，$Occ[c, 128n]$ に加算することで復元する．この手法では，メモリ使用料を大幅に削減することができる代わりに計算量を増加させるが，$Occ[c,i]$ の復元はビット演算で比較的高速に計算することができる．

FM-index 法は，利用する index の容量が少なく（ヒトゲノムで 4GB 程度）PC の限られたメモリ空間でも実行できるため，ソフトウェアでは最も有力なアルゴリズムである．FM-index を用いるソフトウェアマッピングツールは BOWTIE [15],BOWTIE2 [17],BWA [16],BWA-SW [18],BWA-MEM [1],SOAP2 [19] 等，数多くある．これらのソフトウェアは，FM-index を用いる点では共通しているが，変異を含んだ検索への対応が大きく異なる．

BOWTIE [15] では，ショートリード全体を FM-index で検索するが（そのため，seed-and-extend 法には属さない），変異については，$a \in [{}'A', {}'C', {}'G', {}'T']$ のように 1 塩基の置換変異を深さ優先探索で検索する．したがって，許容する変位量に対して計算量が指数関数的に増加する．そのため，ギャップには対応しない，2 塩基程度の置換のみ対応する等，許容する変異量を限定することで，高速な処理を可能にしている．

BWA [16] では，ショートリード全体を FM-index で検索するが（そのため，seed-and-extend 法には属さない），変異を含んだ検索は幅優先探索を利用し BOWTIE よりも多くの変異に対応し，ギャップにも対応している．しかしながら，処理速度では BOWTIE に劣る．

BOWTIE2 [17] では，BOWTIE を seed-and-extend 法に改良し，リード長が 100 塩基程度でも十分な性能が得られるようになっている．ギャップにも対応している．

BWA-MEM [16] では，ショートリード全体ではなく部分文字列の exact match の検索にのみ FM-index を使い，seed-and-extend 法に属するアルゴリズムを提案している．BWA のように，FM-index で変異を含んだ検索は行わない．ショートリード中で，なるべく長い exact match(Maximum Exact Match(MEM)) を見つけてマッピング候補とし，ショートリード全体は，SW 法で評価する．MEM を見つけるために，[18] で提案された FM-index を（$\boldsymbol{aX}$，$\boldsymbol{Xa}$ というように）双方向に延長できるように拡張した FMD-index を用いている．BWA のように FM-index で変異を含ん

だ検索を行わず，FM-index の利用を exact match の検索にのみ限定しているため，効率がよく，ショートリード長が 50 塩基を越えるような場合は，BWA よりもマッピング精度，処理速度ともに優れる．

このように，FM-index を用いるソフトウェアマッピングツールは，BOWTIE に代表されるように FM-index で変異の含んだ検索を行い FM-index だけでリード全体の比較を行うものと，BWA-MEM のように FM-index を部分配列（seed）の exact match のみに用い seed-and-extend 法に属するものがあり，リード長が 50 塩基を超える場合は，seed-and-extend 法に属する BWA-MEM が効率的とされる．

2.2.2 hash-index 法

hash-index 法に基づくアルゴリズムは，基本的に seed-and-extend 法に属する．hash-index 法に属する具体的なアプリケーションとしては，BFAST [20]，SOAP [21]，SeqMap [22]，PASS [23]，GASSST [24]，PerM [25]，MOSAIK [26],novo align [27] 等がある．

ここからは，BFAST で用いられる hash-index 法を例に説明する．BFAST は，seed-and-extend 法に基づいた手法である．seeding 段階では，ショートリードから抜き出した seed について，hash table を参照することで，参照配列上での出現位置を調べ，CAL を得る．extension 段階での，ショートリード全体の比較には挿入・欠失・置換を考慮した近似塩基配列比較法である Smith-Waterman アルゴリズム [8] を用いてそれぞれの CAL をスコア付けし，CAL の中から最良のものを選択する．Smith-Waterman アルゴリズムは，塩基の置換，挿入，欠損を考えた編集距離を算出するアルゴリズムであり，詳細は次項（第 2.2.3 項）で説明する．

ここからは，Hash-index 法の一例である BFAST [20] に基づいたアルゴリズムを図 4 を使用して説明する．図 4 では，参照配列を “ACGTAACGTAGC”，seed の長さを 4 塩基としている．hash-index 法では，hash table を用いて seed の参照配列上の位置を得る．ヌクレオチドは，‘A’，‘C’，‘G’，‘T’ の 4 種類のため，それぞれ 00，01，10，11 と 2bit で符号化される．hash 関数は，seed の先頭 2 塩基（4bit）を hash 値とする関数とする．また，seed の下位 2 塩基（4bit）を key と呼ぶ．hash table は，index table と CAL table という 2 つのテーブルで構成される．まず，図 4 の左に示すように，最初に参照配列中のすべての seed が列挙される．この場合，

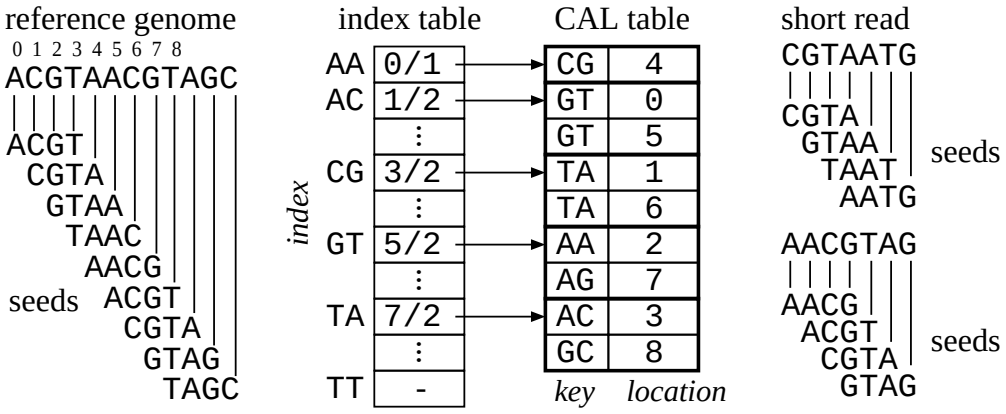


図 4: Hash-index 法

参照配列の長さは，12 塩基長のため，$12 - 4 + 1 = 9$ 個の seed が参照配列中に存在する．参照配列中の全ての seed は，CAL table にその参照配列中の出現位置と共に記録される．ただし，CAL table には，seed 全体ではなく，key のみを保存する．CAL table は index table を通して参照される．index table は，seed の hash 値をアドレスとして，参照される．index table は，同じ hash 値を持つ seed の CAL table 中の領域を指し示している．hash table は，seed を入力とし，その参照配列中の出現位置を出力する．例えば，“GTAA” という seed の出現位置を調べるとき，まず先頭 2 塩基 “GT” $(1011_2 = 11_{10})$ が hash 値として得られ，この値をアドレスとして index table を参照し，‘5/2’ というデータが読みされる．この，‘5/2’ とは，CAL table 中の 5 番目から 2 個（{“AA,2”}，{“AG,7”}）という領域を示す．この領域は，同じ hash 値を持つ参照配列の seed，すなわち，“GT” から始まる seed を示す．{“AA,2”}，{“AG,7”} は，それぞれ，“GTAA” が 2，“GTAG” が 7 という位置で，参照配列中に出現していることを意味する．このように，CAL table は，index table を通してアクセスされ，先頭 2 塩基（hash 値）については，暗黙的に区別されるため，seed の下位 2 塩基の key 領域だけを CAL table に保存している．このように，seed の全領域でなく，key 領域のみを保存するのは，CAL table の容量を削減でき，検索時のデータ転送量を減らすことができるため，処理速度の観点から重要である．index table と CAL table は，参照配列にのみ依存するため，マッピング処理の前に作成しておくことができる．

ここで，ショートリードとして，“CGTAATG’が，与えられたとする．このショートリードには，“CGTA”，“GTAA”，“TAAT”，“AATG” という 4 個の seed が存在する．全ての seed に対して，hash table を参照して，参照配列中での出現場所を得る．例えば，“CGTA” という seed の参照配列中の位置を得るため，“CG” を使って index table を参照する．CAL table から {“TA,1”}，{“TA,6”} を読み取り，key が一致する 1 と 6 が CAL に加えられる．同様にして，“GTAA”，“TAAT”，“AATG” の出現場所を調べ，最終的には 1 と 6 が CAL として得られる．得られた CAL を用いて，SW 法により，ショートリード全体と参照配列とが比較され，最も類似している 1 が最終的な場所として選ばれる．

seed の長さには，処理速度とマッピング率のトレードオフの関係がある．短すぎる seed は，非常に多くの CAL を発見してしまい，SW 法を計算する回数が増えてしまう．seed-and-extend 法では，seeding 段階では，正解を含むなるべく少ない数の CAL を見つけることが求められる．短すぎる seed は，参照配列中で何度も出現し，一意な場所を示しにくいため，無駄に CAL の数を増やしてしまう．一方で，長すぎる seed は，マッピング率を下げてしまう．長い seed では，seed の中に変異を含む可能性が高くなる．hash table の参照では，seed に変異が発生していないことが前提となるため，seed 中に変異が発生していると正しい CAL を見つけることができない．そのため，長すぎる seed は，マッピング率に悪影響を与える．

BFAST [20] では，このトレードオフの関係を，実験的に評価し解決している．まず，seed の長さと参照配列中での出現回数の関係について，参照配列をヒトゲノムとし，seed の出現回数の分布を，seed の長さを 10 塩基長から 50 塩基長まで調査することで明らかにしている．seed 長を 18 塩基とすると，50%の seed が 1 回のみ出現し，seed 長を 22 塩基とすると，80%の seed が 1 回のみ出現する．この結果を元に，seed 長は，18 から 22 塩基程度が，よいとされ，ショートリードの長さが，50 塩基以上ある場合は，22 塩基が実行速度も考慮すると優れているとされる．

また，BFAST では，出現回数が 8 回より多い seed を使用しない．例えば，seed 長が 22 塩基のときに，80%が 1 回のみ出現するのに対し，約 20%は，8 回より多く出現し，さらに約 8%は，1024 回以上出現するという特徴がある．ゲノム配列では，特定の並びが反復して出現することが多いため，このように，反復して出現する seed が存在する．反復して出現する seed は，その出現位置が CAL となるため，CAL の数を増やして

しまう．このために，出現回数が 8 回以内の seed のみ（約 80%）を使用する．

hash 値の定義域を l_{idx}bit とすると，index table の長さは，$2^{l_{idx}}$ となる（図 4 では，$l_{idx} = 4$ である）．l_{idx} が大きいほど，hash の衝突が減り，CAL table 上のデータ読み出し量や比較回数が減るため高速だが，index table の容量が指数関数的に増加する．BFAST では，index table の容量が 1GB 程度になる $l_{idx} = 28$ が一般的である．key は，メモリのバウンダリの関係から，seed の下位 32bit（4 B）である．

hash-index 法の処理をまとめると，次のようになる．

各ショートリードに対して
1. ショーリードの各 seed に対して
   a. seed から hash 値を計算し，index table を参照して
   b. CAL table から対象領域を読みだして
   c. key を比較して，一致した時，CAL へ加える
2. 全ての CAL をについて，ショートリード全体で Smith-Waterman 法を使って評価し，最良のものを選ぶ．

2.2.3 Smith-Waterman 法

参照配列とショートリードを塩基配列の欠損や挿入等を考慮して比較する場合，シーケンスアラインメントと呼ばれる処理が必要になる．典型的には，動的計画法に基づいたアルゴリズムである Smith-Waterman 法 [8]（以下，SW 法と呼ぶ）を用いる．

長さ L のショートリードと長さ M の参照配列を SW 法で比較する場合，$L \times M$ のマトリックス $H(i,j)$ を式 1 に基づいて計算する．

$$H(i,j) = \max \begin{cases} 0 & \\ H(i-1,j-1) + s(a_i, b_j) & (Match/Mismatch) \\ H(i-1,j) + g & (Deletion) \\ H(i,j-1) + g & (Insertion) \end{cases} \quad (1)$$

ただし，a_i は，ショートリードの i 番目の文字，b_j は，参照配列の j 番目の文字，$s(a_i, b_j)$ は，スコア・マトリックスとも呼ばれ，ショートリードマッピングでは，単に，a_i と b_j が一致すれば 1 で，一致しなければ -1，g はギャップペナルティと呼ばれ，-2 とする．

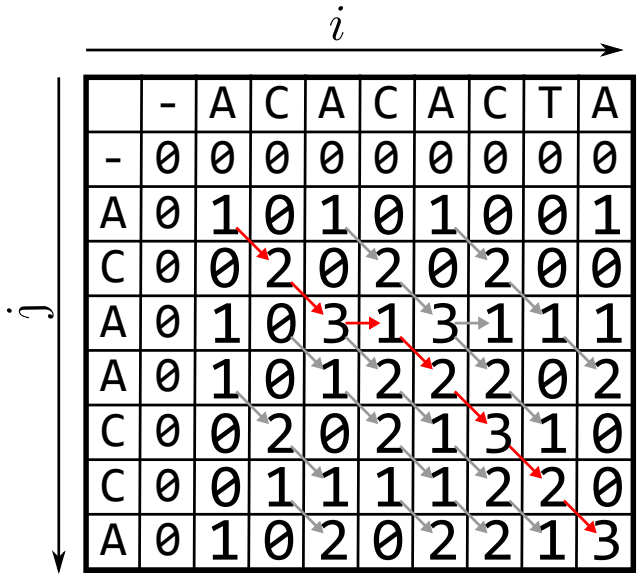


図 5: SW 法の計算例

図 5 に，“ACACACTA” と “ACAACCA” に対する SW 法の計算例を示す．i は 0 から L，j は 0 から M までの全ての $H(i, j)$ を式 1 に基づいて計算している．また，各要素を式 1 に基づいて計算するとき，最大値としてどの値を選んだかを矢印で示しており，この矢印を逆向きにたどることで，ギャップの挿入位置が特定できる．この場合，参照配列である “ACACACTA” に対し，ショートリードを “ACA-ACCA” とすれば，このショートリードが参照配列に対して 1 塩基欠損・1 塩基置換であることがわかり，そのスコアが 3 であることがわかる．ショートリードマッピングの出力形式である SAM 形式でアラインメント結果を示すために使われる CIGAR 形式では，“3M1D4M” と表し，ギャップの位置を示す．SW 法の計算量は $O(LM)$ である．

また，この例では，ギャップペナルティは，単に g で，ギャップの長さに対して線形だが，affine gap [28] と呼ばれる，ギャップの開始時と延長時のスコアを分ける手法も用いられる．

第3章
既存の高速化システム

本章では，ショートリードマッピングを高速化するために提案された既存の FPGA システム及び GPU システムについて述べる．ショートリードマッピングの高速化の需要に応えるために，FPGA や GPU を用いた高速化システムの研究が数多くあり，その一部を本章で紹介する．

3.1 既存の FPGA システム

Field-Programmable-Gate-Array(FPGA) とは，任意の論理回路を構成可能な集積回路である [29].

FPGA をショートリードマッピングのアクセラレータとして用いる場合，アルゴリズムの専用回路を設計し，並列，パイプライン化して高速化を図る．また，DNA を構成する塩基は，4 種類であるため，2bit で符号化でき，長さ N 塩基の DNA 配列は，長さ $2 \times N$ の bit 列として扱うことができるが，これは処理毎に演算幅を自由に設定できる FPGA との相性がいい．そのため，FPGA を用いたショートリードマッピングの高速化システムがいくつも提案されてきた．

まず，FPGA の概要を述べる．図 6 に，Xilinx 社製 FPGA の構成の概要を示す．FPGA を構成する基本的な要素は，Configurable Logic Block(CLB)，それらを接続するプログラマブルなスイッチブロック，外部とのインターフェイスにあたる IO Block である．他に，ハードウェアブロックとしては，Block RAM と呼ばれるオンチップ・メモリ，乗算器として DSP などが搭載されている．CLB は，任意の論理演算を実現するプログラマブルな要素であり，Xilinx 社製 Vitrex-5 以降の FPGA では，6 入力 LUT とフリップフロップから構成されている．また，FPGA で大規模な問題を解く需要があるため，外部メモリとして DRAM が搭載されている FPGA ボードが多くあり，ショートリードマッピングを行う場合はこのようなデバイスを使う．

FPGA で設計を行う場合，回路をハードウェア記述言語（Verilog HDL [30] と VHDL [31] が一般的）で記述する．ハードウェア記述言語で記述

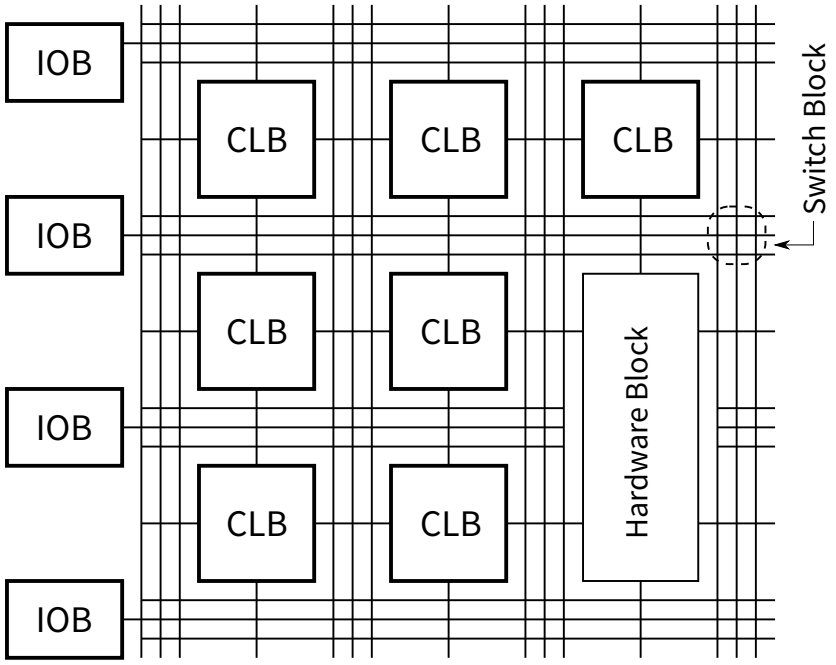


図 6: Xilinx 社製 FPGA の構成の概要

された回路は，各 FPGA ベンダーが提供する合成ツールを用いて，回路データに変換され，FPGA に回路データを転送することで設計した回路が実装される．

次に，既存の FPGA ショートリードマッピングシステムについて概説する．

[32] では，最初に複数をショートリードを FPGA 上に読み込み，参照配列全体を先頭から読みながら，読み込まれた参照配列と複数のショートリードを並列に比較していくアプローチが提案されている．FPGA 上で保持可能なショートリードの数は限られているために，多くのショートリードを処理するためには，何度も参照配列全体を読み込む必要があるため，性能は限定的である．

[33] では，ショートリードと参照配列間で任意の塩基数の置換が発生してもマッピングできることが保証されている FPGA システムが提案されているが，ギャップには対応しておらず，また性能はソフトウェアと比べ僅かに劣る．

[34] は，FM-index 法を使った最初の FPGA 実装であるが，FM-index がオンチップメモリに記憶されることが前提であり，小さな参照配列に

しか対応できないため，ヒトゲノムは扱えず，応用範囲は限定的である．また，FM-index では，exact match しか行っておらず，その成果は基礎的なものにとどまっている．

[35] では，CPU と FPGA を使ったヘテロジニアスなショートリードマッピング計算システムが提案されている．ショートリードと参照配列を毎回 FPGA へ転送しているために PCIe インターフェイスがボトルネックになり，高速化は限定的である．また，ギャップにも対応していない

[36] では，シンプルなシストリック方式の比較方式を提案し，数千に及ぶ大規模並列なショートリードと参照配列の比較を実現している．Xilinx 社製 Virtex-6 XC6VLX550T を用いて，性能は，BOWTIE と同程度である．

[37] では，BFAST に基づいた hash-index 方式の FPGA システムが提案されている．4GB の DDR3 SODIMM が 2 個搭載されいてる Xilinx 社製 Virtex-6 FPGA が搭載された FPGA ボードを 8 個使った構成で，BFAST, BOWTIE に対してそれぞれ，250 倍，31 倍の高速化を実現している．最も高速な FPGA システムの一つである．

[38] では，hash-index 方式の FPGA システムが提案されている．extend ステージでは，Needleman-Wunsch [39]（NW）法が用いられる．NW 法は，SW 法と計算方法はほとんど同じだが，SW 法は $H(i, j)$ が負の数を取らず，0 でリセットされることが異なる．ショートリードと参照配列はある程度似ていることが想定されるため，ギャップの数を制限することで計算量を削減している．ボトルネックは，seeding で性能評価は，ヒトゲノムではなく大腸菌で行われている．Xilinx 社製 Virtex5 LX330 を使用し，BWA と比較して約 5 倍の高速化を達成している．

[40] では，FM-index 方式の FPGA システムが提案されている．Xilinx Virtex-6 SX475T で，BWA と比較して 10 倍の性能を達成し，[37] と同等のハードウェア資源を利用すると同程度の性能が達成できると見積もっている．

[41] では，BWA の FPGA 実装が提案されている．$Occ[c, i]$ の容量を削減するために新たなエンコード方式が提案されている．

[42] では，BOWTIE の FPGA 実装が提案されている．BOWTIE と完全互換な結果が保障されていおり，8 スレッドの BOWTIE と比べ 12 倍の高速化を実現している．BOWTIE と完全互換のために，ギャップには対応していない．

3.2 既存のGPUシステム

まず初めに，Graphics-Processing-Unit(GPU) について概要を述べる．GPU とは，画像処理に特化したプロセッサであり，[43] では，

> GPU は今日ではあらゆる PC，ラップトップ・コンピュータ，デスクトップ・コンピュータ，ワークステーションに搭載されるに至っている．最も基本的な形の GPU は，ウィンドウ・ベースのオペレーション・システム，グラフィカル・ユーザー・インターフェイス，ビデオ・ゲーム，ビジュアル・イメージング・アプリケーション，ビデオに活用されている．2D や 3D のグラフィックス，画像，ビデオを生成する．

と述べられている．優れた並列プロセッサとして GPU を利用し，上記以外の汎用目的の演算処理をすることを，GPU コンピューティングと呼ぶ．[43] では，同じ GPU を汎用目的に利用する場合でも，従来の画像処理用の API を用いる場合を GPGPU（general purpose computing on GPU），汎用目的のための並列プログラミング用の言語と API を利用する場合を GPU コンピューティングと呼び区別している．

GPU コンピューティングの環境として，NVIDIA 社が提供する開発プラットフォームである Compute Unified Device Architecture（CUDA）[44] があり，C，C++ベースでのプログラミングで NVIDIA 社の GPU を汎用目的の演算装置として利用できる．CUDA では，GPU 上でマルチスレッド実行されるカーネル関数を記述する．数千スレッドを同時に実行可能だが，SIMT(Single-Instruction Multiple-Thread) と呼ばれるアーキテクチャで動作し，各スレッドが異なる命令を実行する場合は実行効率が落ちる．

CUDA では，CUDA コアと呼ばれる並列に動作する演算ユニットを使用する．以下は，NVIDA 社の GTX Titan Xp を対象として説明する．GTX Titan Xp では，CUDA コアを 3840 個搭載しており，それらは SM（Streaming Multiprocessor）と呼ばれる単位に分割され，SM あたり 128 コア，全体で 30 個の SM が搭載されている．

CUDA では，GPU 内で実行する関数をカーネル関数と呼び，カーネル関数は実行時に指定したスレッド数で動作する．

カーネル関数のスレッドは，図 7 に示すような階層性を持つ．スレッドの集合をブロックと呼び，ブロックの集合をグリッドと呼ぶ．ブロック内

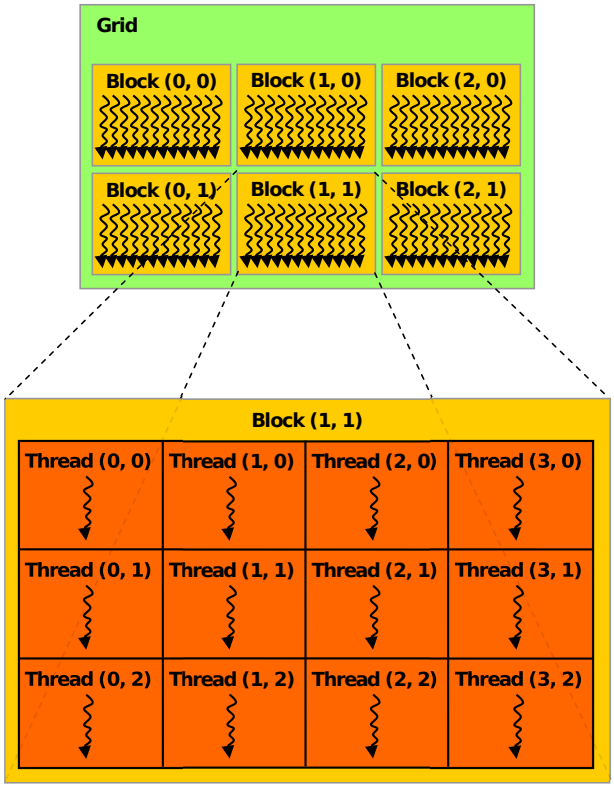


図 7: GPU におけるスレッドの階層性（ [44] より引用）

のスレッド，及び，グリッド内のブロックは 3 次元的に定義可能である．カーネル関数は，SIMT というモデルで実行されるが，それは 32 スレッドを 1warp とし，warp 内で，同じ命令を共有することに由来する．条件分岐などで，warp 内のスレッドで別の命令をする場合には実行効率が落ちる（warp divergence と呼ばれる）．

次に，CUDA でも用いるメモリ資源について述べる．表 2 に，CUDA でも用いる各メモリ資源と特徴を示す（各容量は，GTX TITAN Xp の例）．Register file は，スレッド・ローカルな変数を格納し，一つ 4B で SM あたり 64K 個搭載されている．shared memory は，オンチップメモリであり，同一ブロックのスレッド間でデータを共有する場合に使われ，SM あたり 96KB の容量である．global memory は，オフチップメモリであり，大容量で，ホスト PC と GPU 間でデータを共有する場合にも利用される．Register file と shared memory は高速に読み書きできる一方で少容量であるのに対し，global memory は低速・大遅延で大容量である．

表 2: CUDA における主要なメモリ資源

名称	保存場所	アクセス範囲	容量
Register file	on-chip	スレッド	64K/SM
shared memory	on-chip	ブロック	96KB/SM
global memory	off-chip	任意（ホスト PC も可）	12 GB

CUDA で高性能なシステムを実現するには，メモリボトルネックを回避する必要があり最適なメモリ資源の利用が重要な課題となる．

ここからは，既存の GPU を用いたショートリードマッピング高速化手法について概説する．

CUSHAW [45] では，CUDA を用いて，FM-index 法に基づいたアルゴリズムで Bowtie に対して約 6 倍の高速化を達成している．BWA では，変異を含んだ検索を幅優先探索で行っていたが，深さ優先探索で行うことで，メモリ消費量の削減をし，メモリ資源量が限れた GPU に最適化している．ギャップを含む検索には対応していない．

CUSHAW2-GPU [46] では，CUSHAW2 の GPU 実装が提案されている．CUSHAW2 では，seed-and-extend 法に基づき，seed は，FM-index 法を用いて，参照配列と一致するなるべく長い seed（Maximum-Exact-Match）を見つける．ギャップを含む検索に対応している．

barraCUDA [47] では，FM-index 法に基づいたソフトウェアである BWA [16] を改良し，GPU を用いた高速化手法を実現している．１つの GPU で，CPU の 1 コアに対して 6 倍程度の高速化を達成している．また，複数の GPU を使った高速化にも対応している．

SOAP3 [48] では，FM-index 法に基づいたソフトウェアマッピングツール SOAP2 [19] の GPU 実装が提案されている．ギャップには対応しておらず，4 塩基の変異までに対応している．1GPU で BOWTIE に対して 7 倍程度の高速化を実現している．

Arioc [49] では，hash-index 法に基づいた GPU を用いた高速化手法が提案されている．1GPU で，BOWTIE に対して 10 倍程度の高速化を実現している．また，複数の GPU を使った高速化に対応している．

第4章
ソートと並列比較に基づくショートリードマッピング並列処理アルゴリズム

本研究では，書き換え可能なLSIであるFPGAと画像処理用の演算ユニットであるGPUを用い，FPGAまたはGPUを付加したパーソナルコンピュータ（以下，ホストPC と呼ぶ）という比較的小規模な計算機システムで，ショートリードマッピングの高速化を図る．

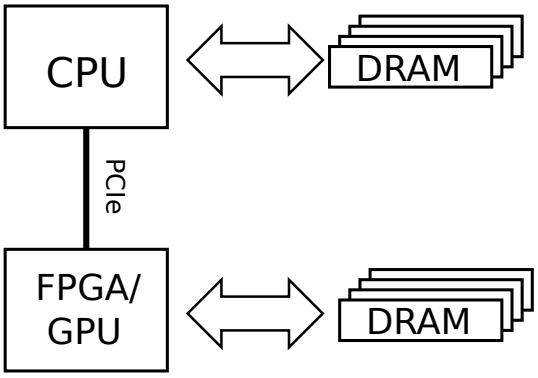


図 8: ショートリードマッピング高速計算システムの構成図

図8に，ショートリードマッピング高速計算システムの構成図を示す．本システムは，ホストPC とPCI Expressによって接続されたFPGA またはGPUを用いて高速なショートリードマッピングの実現を目指す．

hash-index法は，FM-index方に比べ，メモリ参照の回数が少なく単純であるため，大規模な高速化が期待できると考え，BFASTで使われているhash-index法に基づいたアルゴリズムで高速化を図る．seedの長さは，BFASTと同様に22塩基とする．

FPGA，GPU，いずれの場合に置いても，高速化を実現する上で需要なのは，並列処理である．ショートリードマッピングでは，ショートリード毎，あるいはseed毎に独立して処理することが可能であり，容易に並列処

理することができる．ヒトの全ゲノムシーケンシングを行う場合，100 塩基長のショートリードを 10 億本程度処理する必要があるが，このショートリードは全て並列に処理可能であり，潜在的な並列性は非常に高い．一方で，ショートリードリードから抜き出した seed の参照配列中での出現位置を高速に検索するために，事前に hash table を作成しておき，その hash table を読み込んで検索するが，そのサイズは数 GB 以上になるためオフチップ・メモリに保存されており，処理ごとにプロセッサに読み込んでくる必要がある．プロセッサの並列演算能力に比べ，メモリアクセスは遅く，処理のボトルネックになる．ショートリードのデータ次第でアクセス先も変わるため予めオンチップ・メモリに読み込んでおくことはできず，メモリアクセス局所性が低いため，キャッシュするという方策は有効に働かない．このような理由から，ソフトウェアで用いられるアルゴリズムをそのまま適応しても，メモリアクセスがボトルネックになり，大規模な高速化は難しい．

本研究では，メモリアクセスがボトルネックとならず FPGA 及び GPU の回路資源量に応じて高速化可能な並列処理手法を提案する． 本手法は，第 4.1 節のソートと並列比較については [50]，第 4.2 節の segment indexing 及び第 4.3 節の flexible match については [51]，第 4.4 節の可変 mask を用いた hash 関数については [52] で発表を行っている．

本章では，FPGA および GPU に共通するアルゴリズムについて説明し，具体的な実装については，FPGA システムについては第 5 章，GPU システムについては第 6 章で述べる．また，長さ L，M 塩基の塩基配列に対して，SW 法の計算時間は，逐次処理の場合 $O(LM)$ であるが，並列計算では $O(L+M)$ であることが知られており，FPGA や GPU では高速に計算することができる．そのため，CAL を計算する seeding がボトルネックになるため，本章では，seeding 処理の高速化手法を中心に議論する．

参照配列は，NCBI [53] よりダウンロードしたヒトゲノム GRChv38 を使用する．ショートリード長は，NGS 分野で最大シェアを誇る Illumina 社 [54] の Illumina HiSeq 2000 等で使われる 100 塩基を対象として議論する．

ショートリードと参照配列のヌクレオチドは，2bit で符号化する．ショートリードの FASTQ 形式および参照配列の FASTA 形式のファイルには，‘A’，‘C’，‘G’，‘T’ 以外の曖昧コードと呼ばれる文字が僅かに含まれる（典型的なものは，不明や任意を示す ‘N’）．[16] では，このような曖昧コー

ドをランダムに ‘A’，‘C’，‘G’，‘T’ に置き換えている．この方法では，1塩基を単純に 2bit で符号化できるようになるが，一方で，ランダムに変換されたために間違った一致を生じさせる可能性がある．その確率は非常に低いと [16] で述べられており，マッピング率が低下する恐れもあるが，可能性は非常に小さいと考えられる．

また，基本的な前提として，index の作成には，膨大な時間をかけても問題はない．例えば，ヒトゲノムについて作成した index は，全人類の DNA シーケンシングに使い回せるように，ある生物種の参照配列の index は，その生物種のあらゆる個体のマッピングに利用することができるため，index の作成時間は，マッピングの処理時間に加える必要はなく，十分な時間をかけることができる．実際に本手法では，ヒトゲノム用の index は，作成した index 構築用プログラムによって十数時間程度で構築されるが，マルチスレッド化等の高速化は特に行ってない．本手法では，マッピング処理速度を上げるために，膨大な時間をかけて最適な index を作りこむアプローチをとる．

4.1 アルゴリズムの概要

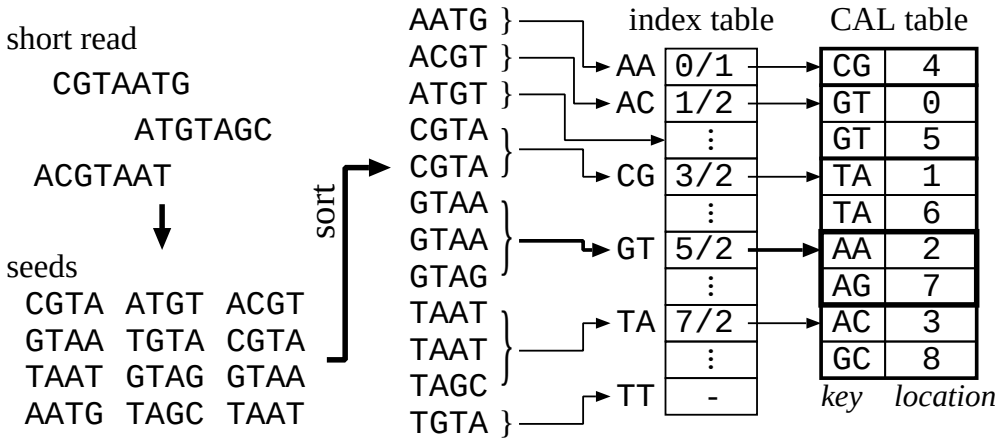


図 9: ソートと並列比較に基づくショートリードマッピング並列処理アルゴリズムの概要

図 9 に，ソートと並列比較に基づくショートリードマッピング並列処理アルゴリズムの概要を示す．図 9 では，3 個のショートリードが与えられ，それらの seed が全て抜き出されている．抜き出された seed は hash 値，つまり，先頭 2 塩基でソートされている．ここでは，同じ “GT” を先頭に持つ 2 個の “GTAA” と 1 個の “GTAG” という 3 個の seed がある．“GT” をアドレスとして index table にアクセスして，CAL table から {“AA”,2} と {“AG”,7} が読み出される．

ショートリードから抜き出された “GT” を先頭に持つ 3 個の seed の key（“AA”，“AA”，“AG”）は，CAL table から読み出された（{“AA”,2} と {“AG”,7}）と並列に比較される．この例では，3 個の key の比較を並列に行い，さらに，index table と CAL へ 1 回ずつの合計 2 回のアクセスを行う．1 個の seed ずつで処理した場合は，6 回のアクセスが必要であり，つまり，この例では，3 個の seed を同時に処理することで CAL table へのアクセス回数を 1/3 に減らしている．この手法では，seed を並列に比較することと CAL table を記憶した DRAM バンクへのアクセス回数を削減することにより性能向上を図ることができる．これ以降では，この同時に比較する個数を P とする．

ソートと並列比較に基づくショートリードマッピング並列処理アルゴリズムは，次に示すような処理手順となる．

(I) seeding

(1) ソート: 与えられたショートリード中の全ての seed を抜き出し，バケットソートを用いてその hash 値でソートする．

(2) key の並列比較: あるバケットが P 個の seed を持った時（同じハッシュ値を持つ seed を，P 個集めた時），DRAM 中の index table 及び CAL table を参照して候補を読み出し，P 個の seed の key を並列に比較する．

(II) extension

(1) スコアリング: 得られた CAL について，ショートリード全体を使って Smith-Waterman 法を用いて評価し，最良のものを見つける．

より高い性能を達成するためには，なるべく多くの同じ hash 値を持つ seed を集める必要がある（より大きい P を実現する）．しかしながら，hash 値のビット幅である l_{idx} が大きくなると，同じ hash 値を持つ seed の数は少なくなる．l_{idx} を小さくすると，同じ hash 値を持つ seed を多く集めやすくなるが，hash の衝突が増え，key の比較回数が増える．

メモリアクセス回数と P と key の比較回数を明らかにするために，予備実験を行った．この予備実験では，参照配列のランダムな位置から 100 塩基を抜き出し，1 塩基あたり 1%の確率で変異を加えたものを 10M 個用意し，ショートリードとした．変異は，置換，欠損，挿入の割合を 0.8, 0.1，0.1 とした．この割合は，[55]，[56] で報告されている実際のヒトの DNA シーケンシグの分析結果である SNV と indel の出現比率（SNV が indel の約 4 倍）を元にしたものである．

表 3: hash table へのアクセス回数と key 比較回数

P	l_{idx} [bit]	#accesses	#cmp	#cmp/P
1	28	1.00000	1.0	1.00
500	14	0.00202	830.2	1.68
1000	14	0.00101	830.2	0.84
1500	14	0.00064	830.2	0.57
2000	14	0.00052	830.2	0.43

表3に，各Pについて，index table，CAL table へのアクセス回数（#accesses）と key の比較回数（#cmp）を示す．#accesses，#cmp，#cmp/P は，$P = 1$，$l_{idx} = 28$ の時が 1 となるように正規化している．$P = 1$，$l_{idx} = 28$ は，ソートと並列比較を行わない場合にあたり，BFAST と同じである．l_{idx} は，バケットソートを行う関係上，メモリ容量により制限されるが，ここでは，$l_{idx} = 14$ とした．実際の l_{idx} と P の値は，第 5 章（FPGA を用いたシステム構築），第 6 章（GPU を用いたシステム構築）で述べる．ソートと並列比較を行わない $P = 1$ の場合に比べ，$P \neq 1$ の場合は，index table，CAL table へのアクセス回数（#accesses）が大幅に減る．基本的に P 個の seed で同時にアクセスするので，$1/P$ となるが，同じ hash 値の seed を P 個集められない場合もあるので，僅かに $1/P$ より大きな値を持つ．ソートと並列比較を行わない場合に比べ，$P \neq 1$ の場合は，key 比較回数（#cmp）が非常に大きくなってしまう．これは，$l_{idx} = 28$ が $l_{idx} = 14$ となったため，hash の衝突が多くなり，CAL table 上での key の比較回数が増えるためである．P 個の key を同時に比較した場合の key 比較回数（#cmp/P）では，$P \geq 1000$ の時に．ソートと並列比較を行わない場合よりも小さくなる．これは，$P \geq 1000$ の場合，ソートと並列比較によって高速化可能なことを意味する．ソートと並列比較を行わない場合より高速になる P のしきい値は，この場合 1000 であるが，l_{idx} 等にも依存するため一定ではない．ここで示したは，ソートと並列比較によって，hash table へのアクセス回数が大幅に減り，l_{idx} が大きくなり比較回数が増えても，十分大きな P を実現できれば，高速化可能なことである．

また，従来の方法では，あるショートリードの seed を順番に処理し，CAL を発見するのに対し，ソートと並列比較では，非同期的に処理される．つまり，あるショートリードの CAL が，任意のタイミングで得られることになり，ショートリード全体を使って SW 法で比較するためには，全てのショートリード（とその最良の CAL）をメモリ上に保持しておく必要がある．

4.2 segment indexing

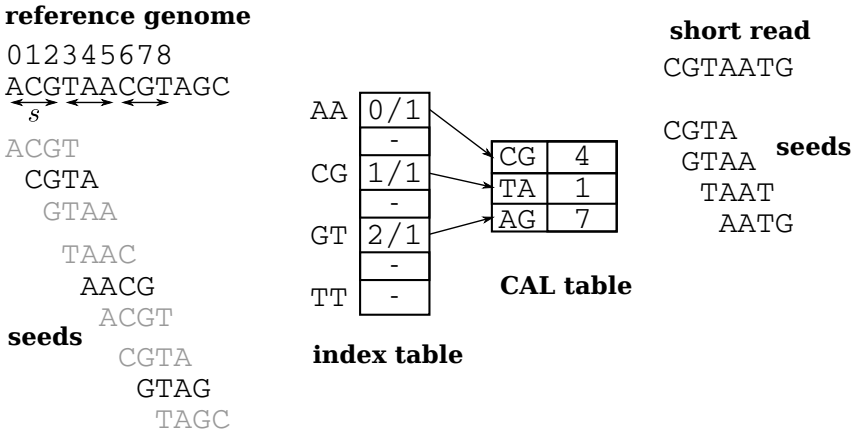


図 10: segment indexing の概要

本節では，segment indexing について説明する．図 10 に，segment indexing の概要を示す．segment indexing では，参照配列を長さ s 塩基間隔のセグメントに分け，このセグメント内で最大で一つの seed だけを登録するようにする．図 10 では，$s = 3$ としている．参照配列上の 0 から 2 番目の位置から始まる “ACGT”，“CGTA”，“GTAA” の 3 個の seed が最初のセグメントとなり，この中で {“CGTA”, 1} のみを hash table に登録する．次のセグメント以降も同様に処理する．これを segment indexing と呼び，hash table に登録する seed の数を $1/s$ の減らすことで，hash table 上で hash 衝突による key の比較回数を $1/s$ に減らし，処理速度の向上を図る．ここで，ショートリードとして，“CGTAATG” が与えられたとする．このショートリードは，4 個の seed を持つが，“CGTA” のみが，hash table と一致し，CAL として ‘1’ を得て，参照配列中の位置 ‘1’ にマッピングされる．これは，第 2.2.2 項で図 4 を用いて議論した最終的なマッピング位置と同じであり，segment indexing により hash table に登録する seed の数を $1/s$ に減らしたにも関わらずに同じ結果が得られる．

このように，ショートリードには，複数の seed が存在し，その中で一つでも正しい CAL を示せば良いため，segment indexing により hash table に登録する seed の数を $1/s$ に減らしたとしてもそのままマッピング率が下がるわけではない．例えば，100 塩基長のショートリードには，$79(= 100 - 22 + 1)$ 個の seed が存在し，この中で一つでも正しい CAL を示せば

良い．全ての seed を利用しないという意味では，BFAST でも同様で，参照配列中での出現回数が 8 回以内の seed（参照配列中の全 seed の約 80%にあたる）のみを利用している．BFAST の場合，参照配列に対して全く変異が発生してないショートリードの場合，ある seed が hash table に登録されている確率を 0.8 とすると，$79 \times 0.8 = 63.2$ という高い冗長性がある．実際には，ショートリードと参照配列の間にある程度変異が発生していることが想定されるため，このような冗長性を確保している．この冗長性のため，参照配列中での出現回数が 8 回以内の seed のみ，つまり，80%の seed を利用することによるマッピング率の低下は僅かである．

segment indexing では,「冗長性があるため，参照配列中の全ての seed を hash table に登録しなくてもよい」という考え方を応用し，hash table に登録する seed の約 $1/s$ に減らす（s は自然数）．key の比較回数は $1/s$ になるため，大きな s は，実行速度は優れているが，マッピング率は低下する．このように s によって，実行速度とマッピング率を調節できるため，マッピング率が BFAST より少し良い程度の fast mode($s = 12$) とより高いマッピング率の accurate mode($s = 4$) の 2 種類の実装を検討する．

本方式は基本的には BFAST に従っており，ここまでは，出現回数が 8 回より多い seed を hash table に登録しないとしてきたが，segment indexing においてこのしきい値を条件的に緩和することによって，BFAST 以上のマッピング率を達成することについて述べる．マッピングが失敗する場合の典型的なケースは，ショートリードから抜き出した全ての seed が hash table に登録されておらず CAL を一つも見つけられない場合である．BFAST では，参照配列中での出現回数が 8 回以内の seed のみを利用するため，参照配列に対して全く変異が発生してないショートリードでも，ショートリード中の全ての seed が参照配列中の出現回数が 8 回より多いの seed の場合は，hash table を参照しても，CAL が見つけられずマッピングが失敗する．

表 4 に，各条件におけるマッピング率と CAL 数の比較を示す．この実験では，参照配列のランダムな位置から 100 塩基を抜き出し，参照配列に対して変異なしショートリードを 1M 個用意した．ショートリード全体（100 塩基）が一致する箇所が参照配列中で複数箇所存在する場合あるため，変異がないショートリードであるにも関わらず，全てのケースでマッピング率が 100%にはならない．BFAST では，参照配列中での出現回数が 8 回以内の seed のみしか利用しないという特徴があるため，FM-index 法に基づくソフトウェアと比べ，変異なしのショートリードでのマッピング率が低

いという特徴がある．FM-index では，再帰的に検索文字列を延長できるために可変長の seed になるが，hash-index 法では，固定長の seed となるためである．表 4 に示すように，BFAST では，94.07%，BWA-MEM では，96.06%と FM-index 法に基づく BWA-MEM のマッピング率が高い．

hash table に登録する seed の出現回数のしきい値を t と呼ぶ．BFAST では，$t = 8$ である．本方式では，CAL 数の増加を極力抑えながら，hash table に登録する出現回数のしきい値 t を条件付きで緩め，マッピング率を向上させる方法を提案する．まず，基本的な処理は，セグメント内で最も出現回数が少ない seed を見つけ，その出現回数がしきい値 t 以内の場合，hash table に登録する．ただ，t は固定値ではない．自分自身を含め左右のセグメントを 1-近傍，左右に二つ目まで計 5 のセグメントを 2-近傍というように，自分自身を含め前後 r 個のセグメントを r-近傍と呼ぶ．r-近傍全てのセグメントで seed が未登録の場合については，r ごとに個別に t を設定できるようにした．これは，近傍のセグメントで登録できていない場合の方がマッピング率に与える影響が大きいため，その場合についてしきい値 t を緩めるという考えた方に基づくものである．

fast mode（$s = 12$）では，1 から 3 近傍が未登録の場合についての t をそれぞれ $16, 128, 512$，accurate mode（$s = 4$）では，1 から 7 近傍が未登録の場合についての t をそれぞれ $16, 32, 64, 128, 256, 512, 1024$ としている．

fast mode の場合の segment indexing による hash table の構築は，次のような処理手順となる．

1. 全セグメントについて，$t = 8$ で探索する
2. 1-近傍のセグメントが全て未登録の場合は，$t = 16$ として再探索する
3. 2-近傍のセグメントが全て未登録の場合は，$t = 128$ として再探索する
4. 2-近傍のセグメントが全て未登録の場合は，$t = 512$ として再探索する

ここで「探索」と呼んでいるのは，セグメント内で最も出現回数が少ない seed を見つけ，その出現回数がしきい値 t 以内の場合，hash table に登録するという処理である．accurate mode の場合も，処理段階は多くなるが同様である．

ただし，このように t を可変化することにより 8 回より多く出現する seed を登録することができるが，登録できるのは同じ seed について 8 回までにしている．これは，hash table を参照する際に，最大一致回数が 8 回に制限されていると，メモリ確保の問題等，利点が大きいためである．

実行速度の観点から，CAL の数，すなわち SW 法の計算回数は少ないほどよい．そのため，なるべく CAL 数を増やさずに高いマッピング率を達成するのが重要である．表 4 を使って，本手法が，CAL 数の増加を抑えながら高いマッピング率を達成したことを示す．また，ここでは，議論を簡単にするために，fast mode のみ比較している．segment indexing で，BFAST と同様に $t = 8$ で固定した場合は，表 4 にあるように BFAST と同程度のマッピング率（93.85%）になる．また，しきい値 t を設けずに，全てのセグメントで一つの seed を登録した場合では，BFAST より高く BWA-MEM と同程度のマッピング率（96.06%）を達成するが，ショートリードあたりの平均 CAL 数 (2637.1) が大幅に増え，実行時間が大幅に増える．$t = 512$ で固定した場合は，BFAST より高く，BWA-MEM に接近する高いマッピング率（95.88%）になるが，ショートリードあたりの平均 CAL 数が 39.8 とやや多い．BFAST で，$t = 8$ というようにしきい値を設けたのは，CAL 数を増加を抑えるためであり，しきい値を単純に緩めれば，マッピング率は向上するが，CAL 数が大幅に増加する．

r-近傍によって t を可変化した場合は，高いマッピング率（95.55%）を保持しつつ，ショートリードあたりの平均 CAL 数は，5.6 と少ない．つまり，しきち値 t を可変化することで，512 に固定した場合と近いマッピング率を持ちながら，少ない CAL 数となっている．以上により，t を近傍セグメントの状況によって可変化する本方式では，ショートリードあたりの平均 CAL 数を抑えながら，マッピング率向上が達成されていることがわかる．

fast mode と accurate mode のマッピング率の差は，主にショートリードと参照配列の間に変異が存在する場合に顕著に見れるため，表 4 では差が小さい．表 4 で示す，ショートリードに変異が全く無い場合のマッピング率は，s ではなく，可変的に緩めるしきい値 t の最大値に大きく依存する．

CAL table の容量は，fast mode で 3.3 GB，accurate mode で 9.5 GB になる．

表 4: マッピング率と CAL 数の比較

t	ショートリードあたりの平均 CAL 数	マッピング率 [%]
segment indexing (fast mode $s = 12$) t 可変	5.6(11.0)	95.55
segment indexing (accurate mode $s = 4$) t 可変	14.9(33.8)	95.79
segment indexing (fast mode $s = 12$) $t = 8$ 固定	1.7(6.9)	93.85
segment indexing (fast mode $s = 12$) $t = 512$ 固定	39.8(54.2)	95.88
segment indexing (fast mode $s = 12$) 閾値なし	2637.1(3200.7)	96.06
BFAST	-	94.07
BWA-MEM	-	96.06

（平均 CAL 数の括弧内の値は，重複した CAL を数えた場合である）

4.3 flexible match

本節では，変異が発生したショートリードに対するマッピング率の向上を目指す flexible match について述べる．hash table での key の比較の際に，1 塩基の置換・欠損・挿入を許容した一致処理にする．この 1 塩基の変異を許容する比較を，flexible match と呼ぶ．

flexible match により，seed の key 領域に変異が発生した場合でもマッピングできるようになる．exact match から，flexible match に置き換えると，key の比較処理の複雑性が増し，実行速度が低下する，一方で，変異が発生したショートリードに対しても高いマッピング率が期待できる．ただし，単純に exact match から，flexible match に置き換えただけでは，ショートリードあたりの平均 CAL 数を大幅に増やしてしまう．

表 5: 変異率 1%の場合のショートリードあたりの平均 CAL 数

	平均 CAL 数
exact match	5.4
flexible match(raw)	164.2
flexible match	13.9

表 5 に，fast mode における変異率 1%のショートリードにおける平均 CAL 数を示す．flexible match(raw) は，単純に flexible match に置き換えた場合であるが，平均 CAL 数が，164.2 と，exact match の 5.4 と比べ大幅に増える．これでは，SW 法の計算回数が増え，処理時間が大幅に増えてしまう．

key の比較が exact match である BFAST では，実行速度の観点から平均 CAL 数を減らすために，出現回数が 8 回以上の seed を hash table に登録しないようにしていた．これは，次に示す二つに性質から，CAL 数の抑制に働いていると考える（以下，それぞれ，性質 1，性質 2 と呼ぶ）．

1. 参照配列中で出現回数が 8 回より多い seed は，hash table を参照した際に，1 回も一致しない．

2. 任意の seed で，hash table を参照した際に，一致するのは 8 回以内である（一つの seed で得られる CAL 数は 8 個以内である）．

exact match の場合では，この二つの性質が hash table に参照配列中に 8 回より多く出現する seed を登録しないことで同時に満たされている．

key の比較が flexible match に変わると，この二つの性質は，単純に hash table に参照配列中に 8 回より多く出現する seed を登録しないだけでは満たされない．flexible match では，参照配列中で出現回数が 8 回より多い seed が，出現回数が 8 回以内の seed と一致することが起こりうる．そのため，hash table に参照配列中に 8 回より多く出現する seed を登録しないだけでは，性質 1 は満たされない．また，key，つまり，16 塩基長の塩基配列に対し，1 塩基変異した塩基配列は約 150 通り（置換：$16 \times 3 = 48$ 通り，欠損・挿入：$15 \times 4 = 60$ 通りで，元の塩基配列によっては重複があるため一定ではない）あるため，flexible match で比較した場合，それぞれ最大 8 回まで登録できるので，最悪のケースで，$150 \times 8 = 1200$ 回程度一致することがあり，性質 2 は満たされない．

flexible match でも同様の性質を保証することで，平均 CAL 数を抑えることができる．flexible match で同様の性質を保証ために，次の抑制処理を行う（それぞれ性質 1，性質 2 に対応する）．

1. 出現回数が 8 回以上の seed で，hash table を flexible match で参照した際，一致した CAL table 上の key を flexible match で使用しないようにする．

2. 任意の seed で，hash table を flexible match で参照した際に，8 回より多く一致する場合，一致した全ての CAL table の key を flexible match で使用しないようにする．

この抑制処理の対象となるのは，hash table の全エントリの 10%以内である．表 5 に示す flexible match は，この抑制処理を行った場合であり，平均 CAL 数が 13.9 個まで減少する．

この抑制処理は，flexible match で比較した際に，頻繁に一致し CAL の数を増やすことにつながる hash table 上の一部のエントリを flexible match で比較しないようにするものであるが，それら抑制処理の対象になったエントリは exact match による比較には用いられる．そのため，実際に hash table から除外するのではなく，単に flexible match では比較しないようにする（exact match では利用する）．flexible match で比較するものと比較しないものの管理は，CAL table で領域分けをし，index table で管理する．

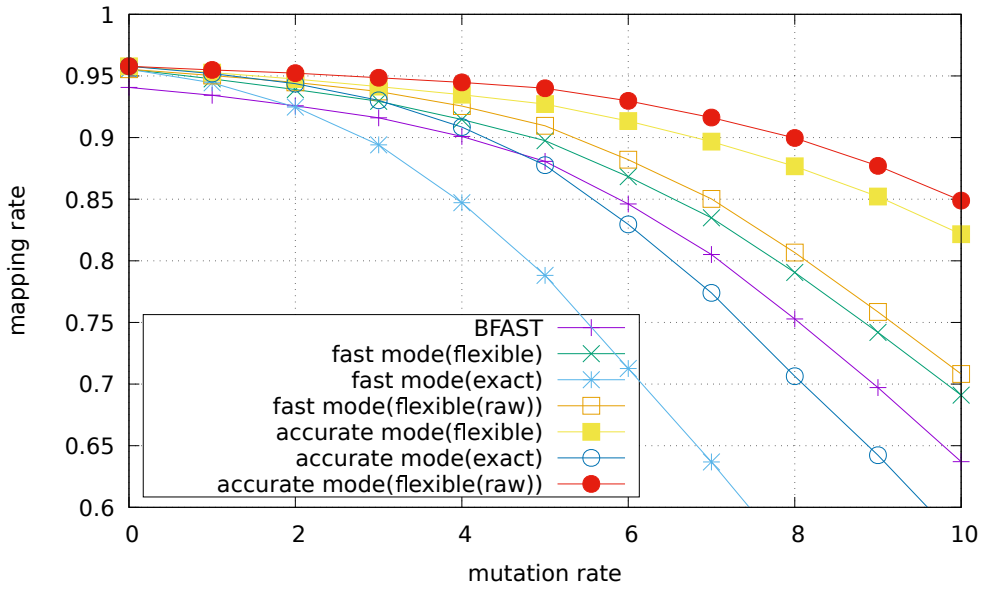


図 11: マッピング率の比較

予備実験を行い，flexible match を利用した場合におけるマッピング率の向上を確認した．図 11 に，flexible match の場合，exact match の場合，flexible match で抑制処理を行わなかった場合（flexible (raw)）について fast mode と accurate mode のマッピング率を示す．この予備実験では，参照配列のランダムな位置から 100 塩基を抜き出し，1 塩基あたり $m\%$の確率で変異を加えたものを 1M 個用意し，ショートリードとした．このグラフの縦軸は，マッピング成功率，横軸はショートリードの変異率 m である．exact match の場合は，変異率が大きくなると，BFAST よりもマッピング率が低くなる．flexible match では，exact match とくらべ，マッピング率が高く，特に m が大きい場合はその差が顕著である．flexible(raw) と flexible のマッピング率の差は小さいが，表 5 に示すように CAL の数を大幅に減らすことができるため，抑制処理を行った flexible が優れている．

fast mode では，全ての m で BFAST と比較して，数%程度マッピング率が高い．accurate mode では，特に m が大きい場合，fast mode よりもマッピング率が大幅に高い．以上により，flexible match と抑制処理の有効性が示された．

flexible match は，単純な exact match と比べ複雑な演算となるが，そのことによる性能の悪化については，FPGA および GPU で性質が異なる

ため，第5章（FPGAを用いたシステム構築），第6章（GPUを用いたシステム構築）で述べる．

4.4 可変 mask を用いた hash 関数

従来手法である BFAST [20] とその FPGA 実装である Olson 等の FPGA システム [37] では，$l_{idx} = 28$ であり，hash 関数としては，seed の先頭 14 塩基，28bit をそのまま hash 値とする関数を利用している．これは，シンプルで高速に計算でき，また，seed の先頭 14 塩基について hash 値で暗黙的に識別できるため CAL table 上に seed の全領域（22 塩基（44bit））を保存せずに seed の key 領域（末尾の 16 塩基 (32bit)）のみを保存すれば良いためである．高速化の観点から，外部メモリからのデータ読み出しがボトルネックになりやすいため，22 塩基長 (44bit) 全体を保存せずに，key 領域（32bit）のみを保存すれば良いという特徴は優れている．これから説明する可変 mask を用いた hash 関数でも，同一の hash 値を持つ seed については，先頭 6 塩基（12bit）が同一であることを保証することで，先頭 6 塩基（12bit）については暗黙的に識別し，従来手法と同様に，実際に CAL table に保存する key 領域は 16 塩基（32bit）とする．

4.4.1 理論的背景

ここでは，22 塩基長（44bit）の seed を，16bit に写像する関数 $hash()$ について議論し，処理速度にどのように影響を与えるかを明らかにする．また，この議論では，hash table としては，議論を単純化するために，segment indexing 等を行わずに参照配列中の全ての seed を登録するものとする．まず初めに，いくつかの変数を定義する．

R ： 参照配列から抜き出される全ての seed の集合
SR ： マッピングする全てのショートリードから抜き出される seed の集合
X_i ： $\{a \in X_i, hash(a) = i\}$

X_i（X は，R または SR）は，X のうち，i 番目の hash bucket に写像される部分集合である．また，$|X|$ は，集合 X の要素数を示す．参照配列としては，ヒトゲノム (NCBIv38) を利用するため，$|R|$，つまり，参照配列から抜き出しうる全ての seed は，逆相補鎖配列を含め $2.8G \times 2$ 個となる．

ここで，hash table 上での hash 衝突による key の比較回数は，次のように定義される（ここでは，ソートと並列比較も行わず，純粋に key の比

較回数を示す）.

$$T_c = C_c \times |SR|$$

ここで，C_c は，各 seed の hash 衝突による平均比較回数とする．ここでは，重要であるは，C_c であるが，C_c は，次のように記述できる．

$$C_c = \sum_{i \in index} p_i |R_i|$$

ただし，$index = \mathcal{N}_{2^{16}}$ （$hash()$ の出力の定義域であり，$[0, 2^{16} - 1]$) である．p_i は，ある seed が，i に写像される確率である．この式は，単に，ある seed が，hash 値として i 持つ場合，R_i の中から一致するものを見つけるために，$|R_i|$ 回の比較をすることを意味する．また，ある seed が，i に写像される確率である p_i は，一様分布ではないため，単純な $|R_i|$ の平均で C_c を定義することができず，i に写像される確率 p_i を考慮する必要がある．p_i は，次のように記述できる．

$$p_i = \frac{|SR_i|}{|SR|} \simeq \frac{|Ri|}{|R|}$$

この近似については，ショートリードと参照配列の差は非常に小さいという前提を利用している．ここまでを踏まえて，C_c を再び記述すると次のような式になる．

$$C_c \simeq \sum_{i \in index} \frac{|R_i|}{|R|} |R_i| = \frac{\sum_{i \in index} |Ri|^2}{|R|}$$

この式は，T_c を最小化するためには，C_c を最小化する必要があり，C_c は，$|R_i|$ が均一になった時に最小化されるということを示している．つまり，p が一様分布になるような $hash()$ が，得られれば，高速化が達成できる．これは，hash table について一般的に言われていることであり，hash-index 法に基づいたショートリードマッピングについても同様のことが言える．

また，C_c の最小値は，$C_{min} = \frac{|R|}{|index|}$ であり，これは，全ての $|R_i|$ が，均一化されて $\frac{|R|}{|index|}$ となった時である．

表 6 に，C_c，C_{min}，C_c/C_{min} （高速化率）を示す．ただし，$hash()$ としては，従来手法で用いられているような seed の先頭 l_{idx}bit をそのまま用いる関数を考えている．表 6 に示したように，l_{idx} を大きくすれば，C_c が

表 6: C_{min} と単純に先頭 l_{idx}bit を hash 値とした場合の C_c

l_{idx}	14	16	18	20
C_c	701728.7	228917.5	88532.7	42867.3
C_{min}	349282.6	87320.6	21830.2	5457.5
C_c/C_{min}	2.0	2.6	4.1	7.9

小さくなり処理速度が向上する．一方で，l_{idx} が大きいほど $|R_i|$ がより不均一になり，hash 関数の改善による高速化達成率が大きくなる．例えば，$l_{idx} = 16$ の時は，hash 関数の改善により最大で，2.6 倍の T_c の高速化が達成できる．つまり，均一な hash 関数を実現することで大きな高速化が期待できることがわかる．

4.4.2 可変 mask を用いた hash 関数

表 6 が示すように，既存の hash-index 方で典型的に用いられている seed の先頭 l_{idx} を hash 値とする方法では，不均一性があり，性能を悪化させていた．ここでは，seed ごとに違った mask を利用することで均一な分布を実現する hash 関数を提案する．ここで使う mask 処理は，与えられたビット列から，mask によって指定された領域のビットを抜き出す処理である．例えば，$x = x_4x_3x_2x_1x_0$ というビット列と $m = 10110$ という mask が与えられた時に，$mask(x, m) = x_4x_2x_1$ とする処理である．

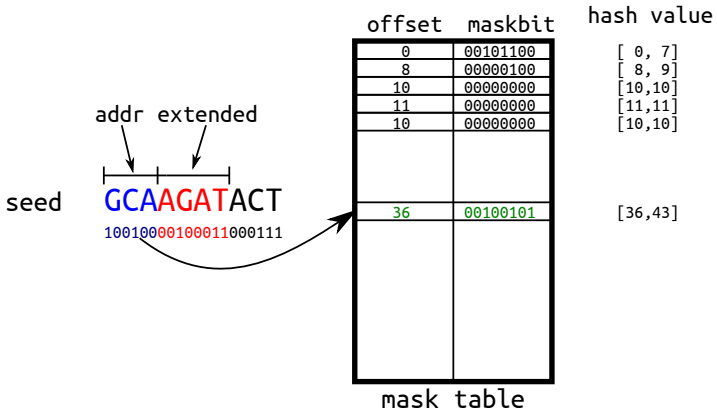
以降の議論では，seed 先頭の l_{addr} bit の領域を addr 領域，addr 領域に続く l_{ex} bit の領域を extended 領域と呼ぶ．可変 mask を用いた hash 関数では，事前に作成した mask table と呼ばれるテーブルと，seed の addr 領域，extended 領域を使用して計算され，出力の定義域は $[0, 2^{l_{idx}} - 1]$ である．l_{idx}，l_{addr}，l_{ex} は，独立して設定でき，l_{addr} は，l_{idx} より大きくても良い．

この処理において重要なのは，事前に計算しておく mask table であり，これは offset と mask という要素からなる長さ $2^{l_{addr}}$ のテーブルである．

hash 関数は次のように定義される．

$$
\begin{aligned}
&hash(addr\_bits, extended\_bits) = \\
&\quad mask\_table[addr\_bits].offset + \\
&\quad mask(extended\_bits, mask\_table[addr\_bits].mask)
\end{aligned}
$$

この式が示すように，まず初めに，与えられた seed の addr bit をアドレスとして mask table から offset と mask が読み出される．次に，読み出された mask を使って seed の extended bit に対して mask 演算を行い，その結果が offset と加算される．mask table から読み出した mask を使って，extended bit に対してマスク演算を行うが，この時の mask は，111000 や 111110 のように先頭から 1 が連続するビット列である必要はなく，任意のビット列で良い．むしろ，連続するビット列を用いると，塩基配列中のパターンの出現頻度の不均一性の影響を受けるため，‘1’ が連続するビット列より，一様に分布するようなビット列の方が優れている．




```
hash(GCAAGATACT) = offset[100100] + mask(00100011 , maskbit[100100])
                 =      36₁₀       + mask(00100011 , 00100101)
                 =      36₁₀       +        101
                 =      41₁₀
```

図 12: 可変 mask を用いた hash 関数の計算例

図 12 に，可変 mask を用いた hash 関数の計算例を示す．図 12 では，$l_{addr} = 6$ bit，$l_{ex} = 8$bit，seed が 20bit（10 塩基長）としている．“GCAAGATACT” という seed が与えられた時に，まず初めに，addr bit 領域であ

る “GCA”（“100100”）と extended bit 領域である “AGAT”（“00100010”）が抜き出される．“GCA”（“100100”）をアドレスとして，mask table にアクセスし，mask として “00100101”，offset として “36_{10}” が読み出される．次に，extended bit 領域である “AGAT”（“00100010”）に対して mask“00100101” を使って mask 演算し，“101” を得る．offset の 36 と $101 = 5_{10}$ を加算し，41 がこの seed の hash 値として得られる．この場合，addr bit 領域として “GCA” を持つ seed に，extended bit 領域に mask 演算することにより，extended bit 領域の差を反映して 8 通り（$2^{pop\_count(mask:\text{“}00100101\text{”})}$）の hash 値（その値は，36 から 43）を算出できる（$pop\_count(x)$ は，ビット列 x のうち ‘1’ になっているビット数を数える関数とする）．これは，$|R_i|$ を均一化するために，$|R_i| > C_{min}$ である（平均より大きい hash bucket）$|R_i|$ を複数に分割することを意味する．

また，図 12 では，mask table の 3 行目と 5 行目の offset が同じ “10” であり mask が “00000000” あるが，これは，異なる addr bit を持つ seed に対して同じ hash 値を持たせるためである．これは，$|R_i|$ を均一化するために，$|R_i| < C_{min}$ である（平均より小さい hash bucket）二つの $|R_i|$ を一つに統合することを意味する．このように，extended 領域の mask 処理によって，同じ addr bit を持つ seed を複数の hash 値に分割，また，offset に同じ値を持たせることによる別の addr bit の seed を同じ hash 値に統合する，この 2 つを組み合わせることで，参照配列に特化した均一な分布を持つ hash 関数を実現することができる．

4.4.3 mask table の計算方法

可変 mask を用いた hash 関数では，事前に計算しておいた mask table を利用するが，その性能は，mask table に大きく依存する．mask table は，ショートリードとは関係なく参照配列によってのみ計算され，その参照配列中での塩基の並びの出現分布に基づいて，最適な mask table を計算し，均一な分布を持つ hash 関数を実現する．本項では，mask table の計算方法について述べる．

$C_c = \frac{\sum_{i \in index} |R_i|^2}{|R|}$ を最小化する mask table を計算する必要があるが，C_c は，全ての $|R_i|$ が $\frac{|R|}{2^{l_{idx}}}$ となった時に最小になる．最適な mask table を計算する問題は，組合せ最適化問題であり，offset と mask の組み合わせは膨大であるため，最適解を計算するのは困難である．そこで，シンプ

ルなヒューリスティックスアプローチを提案し，mask table の計算を行った．この方法では，addr bit ごとに計算を行うが，図 12 のように一つの addr bit が複数の hash 値を持つことがある．そのことを考慮すると，ある addr bit=k が C_c に与える影響 score[k] は，次のように定式化できる．

$$\mathrm{score}[k] = \frac{\sum_{i=0}^{p(k)} |R_{offset[k]+i}|^2}{p(k)}$$

ただし，$p(k) = pop\_count(mask[k])$ である．例えば，図 12 では，score[100100（“GCA'')] $= \frac{|R_{36}|^2+\cdots+|R_{43}|^2}{8}$ となる．

1. mask table が，先頭 $2^{l_{addr}}$ を hash 値にするように初期化される．すなわち， $offset[i] = i$， $mask[i] = 0$ とし，addr bit がそのまま hash 値となるように初期化する．

2. 最大の score[k] となる k を見つけ，ランダムに $mask[k]$ の 0 を 1 に変える．これは，addr bit $= k$ の時，$p(k)$ 通りの hash 値を持っていたが，それが $2p(k)$ 通りに増えること意味する．そして，$offset[i]$ $(k < i < 2^{l_{index}} - 1)$ に $p[k]$ を加算する．これは，hash 全体の定義域が，$p[k]$ 増えたことを意味する．

3. 最小の score[k] となる k（mask[k] $= 0$ のもの）を見つけ，その k と同じ先頭 12bit を持つ最小の k' を見つけ，$offset[k]$ と $offset[k']$ を同じ値に変更する．addr bit $= k$，k' は，同じ hash 値を持つようになり，これにより，hash 関数の全体定義域が 1 減少する．$offset[i]$ $(k < i < 2^{l_{index}} - 1)$ を 1 減算する．これを，hash 関数の全体定義域が $2^{l_{idx}}$ に収まるまで繰り返す．

4. 2 と 3 を収束するまで繰り返す．

5. 全ての mask[k] について，$p(k)$ を変更しない範囲で，最適なものに変更する．ビット列である mask[k] は，$p(k)$ 個の 1 を持つが，同じ $p(k)$ 個の 1 を持つビット列（${}_{l_{ex}}\mathrm{C}_{p(k)}$ 通り）について全て score[k] を計算し，score[k] を最小化するビット列に mask[k] を変更する．

このヒューリスティックスアプローチは，計算完了に数時間程度が必要であるが，ある参照配列について一度計算すればよいので，問題はない．この mask table の計算時間は，index 作成時の問題であるので，マッピングの計算時間とは関係がない．

表 7: $l_{idx} = 16$ の場合の C_c の高速化率と mask table の容量 (Mb)

		l_{addr}				
		14	15	16	17	18
l_{ex}	8	2.02 (0.38)	2.11 (0.75)	2.15 (1.50)	2.21 (3.00)	2.26 (6.00)
	10	2.11 (0.41)	2.16 (0.81)	2.20 (1.63)	2.28 (3.25)	2.29 (6.50)
	12	2.17 (0.44)	2.21 (0.88)	2.21 (1.75)	2.31 (3.50)	2.32 (7.00)
	14	2.22 (0.47)	2.25 (0.94)	2.26 (1.88)	-	-
	16	2.25 (0.50)	-	-	-	-

† 完全に均一化された場合の高速化率は，2.62 である．

表7に，従来の先頭l_{idx}bit を抜き出す場合の C_c と，ヒューリスティックスアプローチを使って計算した mask table を用いた C_c' の高速化率（C_c/C_c'）と mask table の容量を示す．$l_{idx} = 16$ に固定し，l_{addr} と l_{ex} については，いくつかの値を評価している．表 7 では，‘-’ となっている箇所もあるが，これは，計算時のメモリ使用量の問題から，$l_{addr} + l_{ex} \leq 30$ 場合のみ計算しているためである．mask table の容量は，$2^{l_{addr}} \times \{l_{idx} + l_{ex}\}$ bit である．l_{addr} と l_{ex} は，大きな程，高い高速化率を達成するが，mask table の容量とのトレードオフの関係がある．例えば，FPGA 実装の場合，mask table を FPGA 内に格納する必要があるため，l_{addr} と l_{ex} に制限を与えなければならない．表 7 では，$l_{addr} = 18$ と $l_{ex} = 12$ の時，最大高速化率 2.32 が得られるが，完全に均一化した場合の $C_{min} = 2.62$ に近い．そのため，このヒューリスティックスアプローチは，十分な探索能力を有していると思われる．今回の実装では，mask table の容量を考慮し，$l_{addr} = 15$, $l_{ex} = 14$ とした．

表 8: 各 l_{idx} における可変 mask を用いた hash 関数の C_c の高速化率

l_{idx}	14	15	16	17	18
expected speedup	1.86	2.07	2.25	2.46	2.67

表 8 に，$l_{addr} = 15$，$l_{ex} = 14$ とし，l_{idx} を 14 から 18 にした時の C_c の高速化率を示す．表 8 は，どの l_{idx} についても，本手法が高速化に寄与することを示している．また，l_{idx} が大きくなるほど，高い高速化率を達成しているが，これは，表 6 の結果とも一致する．

4.5 解候補の確率的削除

seeding 段階で得られた CAL（解候補）は，extension 段階で SW 法を使って評価されて，得られた CAL の中から最良のものを選択する．本節では，SW 法を計算する回数を減らすために，最良になる確率が低い CAL を確率的に削除する手法について述べる．

提案手法では，BFAST のように，ショートリードから抜き出した seed を順番に処理するのではなく，いくつものショートリードから抜き出した seed を hash 値でソートし，同じ hash 値の seed を同時に処理する．そのため，ショートリードから抜き出した各 seed は，非同期的に処理されるため，すべてのショートリードのマッピング位置や SW 法で計算したスコアをメモリ上に保存しておく必要がある．それぞれ，best.location, best.score と呼ぶ．

基本的な処理は，あるショートリードについて，新しい CAL が得られた時に，SW 法で評価することで，best.score を更新するか確認する．しかし，解候補（CAL）の確率的削除では，新しく得られた CAL を SW 法で評価しない場合もある．次に述べる 2 つの条件は，最終的なマッピング率に全く影響を与えない CAL の削除である．

1. 新しい CAL が best.location と一致する場合，新しい CAL を評価しない．

2. best.score が上限値の場合，新しい CAL を評価しない．

best.score の上限値とは，ショートリード全体が参照配列と完全に一致する場合であり，ショートリード長が，100 塩基，SW 法における一致のスコアが 1 とすると，best.score の上限値は 100 である．

CAL の確率的削除では，最終的なマッピング率に悪影響を与える可能性がある CAL の削除を行う．CAL の確率的削除は，best.score の上限値と best.score の差に基づいて，新しく得られた CAL を SW 法で評価するかを確率的に決定する．

best.score の上限値と best.score の差（これを d とする）から評価する確率を与える関数 $p(d)$ を削除確率関数と呼び，$p(d)$ によって評価するかを決定する．$p(d)$ が，どのような関数が最適であるかは難しい．$p(d)$ は，確率を表す関数のため，上限が 1 である．また，$d = 0$ の場合は，上記の条件 2 と同様で SW 法で評価する必要がないため，$p(d)$ は，原点を通る

($p(0) = 0$)．さらに，d が大きいほど，すなわち現状の best.score が低いほど，新しく得られた CAL が現在の best.score を上回って更新する確率が高いため，$p(d)$ は単調増加関数である．

CAL の確率的削除は次に述べる 2 つの前提に基づいている．

1. d が大きいほど，新しく得られた CAL が best.score を更新する可能性が高い．

2. 変異が少ない（最終的な best.score が大きい）ほど，ショートリード中の別の seed が何度も同じ CAL を見つけてくることが多い．

2 について説明を加える．図 4 において，ショートリード “CGTAATG” で，seed “CGTA” から 1 と 6，“GTAA” から 1（CAL table では，2 だが，この seed のショートリード中の位置である 1 で補正をすると（すなわち，ショートリードの先頭からみた位置に変換すると）$2 - 1 = 1$）を CAL として得られ，1 については重複している．このように，変異が少ないショートリードほど，別の seed が同じ CAL を何度も見つけてくることが多い．CAL の確率的削除によって，マッピングが失敗する典型的なケースは，d が少ないにもかかわらず，さらに良いスコアをもつ CAL があり，それを SW 法で評価せずに削除してしまう場合である．このようなケースの場合，そのさらに良いスコアをもつ CAL は，別の seed が何度も見つけてくることが想定されるため，その複数回の試行により確率的な削除を回避することが期待できる．

原点を通り，単調増加という条件で最も単純な関数である線形関数 $p(d) = kd$（k は正の実数）として，SW 法を計算する回数と，マッピング率について予備実験を行った．この実験では，変異率 1.5%のショートリードを 64M 個を使用する．SW 法における各スコアは，一致が 1，不一致が -2，affine gap を用いてギャップの開始が -5，ギャップの延長が -2 とする．

表 9 に，削除確率関数の k とマッピング率 と SW 法の計算回数（#SWs）の関係を示す．k が大きいほど，高確率で削除せずに評価することになり，マッピング率は高くなるが，#SWs は多くなり，計算量が増える．今回は，マッピング率とのバランスから，$k = 160$ とした．このとき，マッピング率の低下が僅かである一方で，SW 法の計算回数は確率的削除を行わない場合の 60%程度になり，計算量を大幅に減らすことができていることがわかる．

表 9: 変異率 1.5%のショートリードにおける削除確率関数とマッピング率と SW 法の計算回数の関係

k	マッピング率 [%]	#SWs
確率的削除をしない	94.568	21.7
400	94.568	17.6
240	94.557	14.9
200	94.552	14.3
160	94.536	12.8
80	94.477	10.2

第5章
FPGAを用いたシステム構築

本章では，ソートと並列比較に基づくショートリードマッピング並列処理アルゴリズムのFPGAへの実装について述べる．本章の内容は，[57]で発表を行っている．

5.1 FPGAシステムの全体像

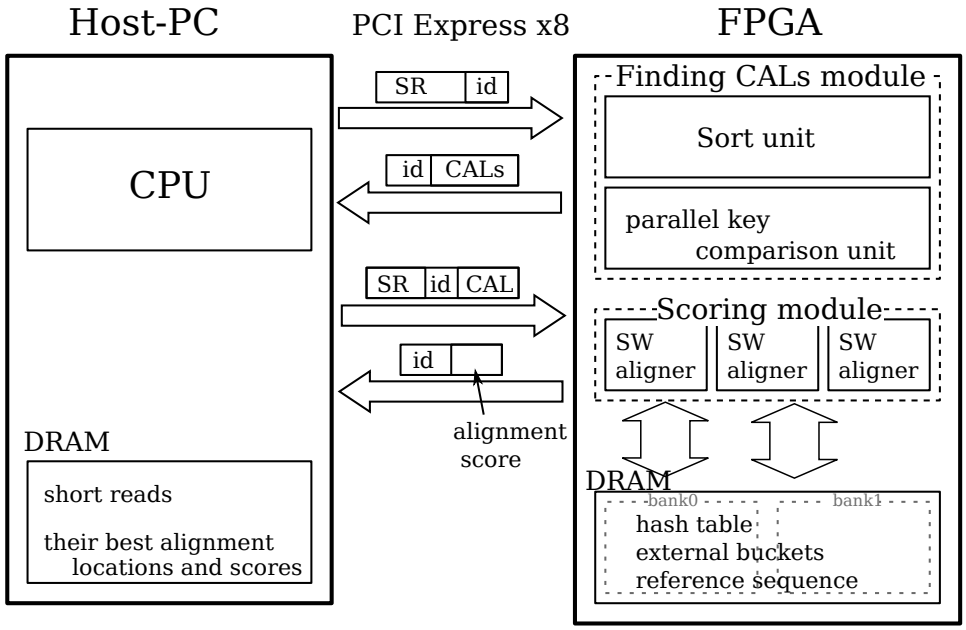


図 13: FPGAシステムの全体像と，PCとFPGA間のデータの流れ

図13に，本FPGAシステムの全体像と，PCとFPGA間のデータの流れを示す．

‘SR’は，ショートリードを意味する．hash tableと参照配列は，FPGAのオフチップメモリに事前に格納されている．

まず，ホストPCから，ショートリードとそのID（ショートリードを特定するための通し番号）をFPGAに送る．“Finding CALs module” は，与えられたショートリードのCALを算出する．このモジュールは，ソート・ユニット（“sort unit”）と並列比較ユニット（“parallel comparison unit”）から構成される．ホストPCから与えられたショートリードは，ソート・ユニットに送られる．このユニットでは，ショートリードから全てのseedを抜き出し，hash値によりソートする．あるhash値を持つseedがP個集められたときに，ソート・ユニットは停止し，そのP個のseedは，並列比較ユニットで並列に比較される．並列比較ユニットは，比較結果として一致したIDとCALのペアをホストPCに送る．ホストPCでは，各ショートリードのbest locationとスコアを管理しており，FPGAから送られてきたIDとCALのペアを元に必要であれば，SW法によりスコア計算するためにショートリードとCALをFPGAに送る．Scoring moduleは，並列に動作するいくつかのSW alignerで構成されている．

5.2 ソートの実装

本節では，どのようにして効率的に同じhash値を持つP個のseedを集めるかを説明する．その機能は，ソート・ユニットで実現されており，ソート・ユニットでは，バケット・ソートが用いられる．ここで問題になるのは，各バケットへのデータ書き込みである．本システムでは，メモリ転送効率を上げるために，FPGAのオンチップメモリに格納される内部バケットと，オフチップメモリに格納される外部バケットを用いる．

図14に，ソート・ユニットのブロック図を示す． まずはじめに，与えられたショートリードをシフトレジスタに格納する．ショートリードは，2bitずつ（つまり，1塩基ずつ）左にシフトされ（“counter” がシフトした回数を記憶している），最上位44bitがseedとして抜き出される．本システムの対象であるショートリードが100塩基長，seedが22塩基長の場合，一つのショートリードから79個のseedが生成される．“counter” の値は，seedのショートリード中での位置を表す（そのseedがショートリード中のどの位置から抜き出されたか）．“hash generator” では，44bitのseedからl_{idx}bitのhash値を計算する．“hash generator” では，オンチップメモリで構成されたmask tableを使い，可変maskを用いたhash関数の計算をしている．seedのkey領域（下位32bit）とショートリードのID

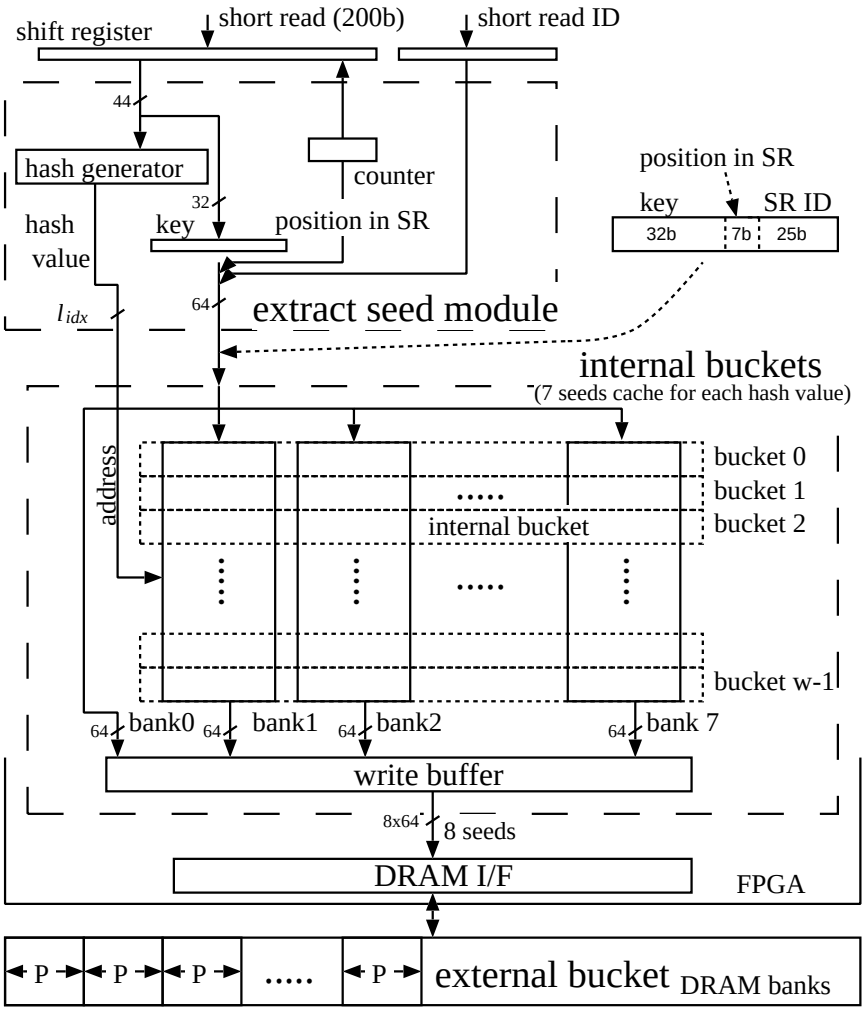


図 14: ソート・ユニットのブロック図

(25bit) と seed のショートリード中での位置 (7bit) の計 64bit（8B）が，バケットへの入力となる．オフチップメモリに格納されている外部バケットは，各 hash 値につき，最大 P 個の seed を格納することができる（8B $\times P$）．外部バケットの総容量は，$2^{l_{idx}} \times P \times 8$B となる．FPGA のオフチップメモリに格納されている内部バケットは，各 hash 値につき 7 個の seed を保存し，外部バケットのキャッシュメモリとして機能する．内部バケットの総容量は，$2^{l_{idx}} \times 7 \times 8$B となる．内部バケットに 8 個目の seed が与えれられたときに，8 個の seed（7 個は内部バケットに記憶されているもの）を外部バケットにバースト転送で書き込む．内部バケットは，オフチップメモリの外部バケットにアクセスする回数を減らし，バースト転送によりデータ転送の効率化を実現する．

ソート・ユニットは，1 クロックサイクルあたり 1 個の seed を処理する．外部バケットが一杯になったときに，ソート・ユニットは停止し，P 個の seed を外部バケットから読み出して，並列比較ユニットが動作する．

大きな l_{idx} ほど，hash の衝突が減り効率的な処理が期待できるが，内部バケットの容量の制限により，l_{idx} は制限される．BFAST [20] では，$l_{idx} = 28$ となるが，本システムでは，オフチップメモリの内部バケットの容量の制限により 14 から 18 が使われる（FPGA オフチップメモリのリソース量に依存する）．hash の衝突の量は増えるものの，内部バケットを使ったバースト転送によるメモリアクセスボトルネック回避し，1 クロックサイクルあたり 1 個の seed の処理を実現している．また，hash の衝突については BFAST より増えるものの，key 比較の並列化や，可変 mask を用いた hash，segment indexing により，十分な高速化が実現されている．

5.3 key の並列比較の実装

key の並列比較は，並列比較ユニット（parallel comparison unit）で実現されいている．図 15 に，並列比較ユニットのブロック図を示す．このユニットでは，ショートリードから抜き出した最大 P 個の seed を効率的に CAL table と比較する．ある外部バケットが一杯になったときに，P 個の seed が外部バケットから読み出されて，並列比較ユニットに与えれられる．外部バケットに格納されている seed は，seed の key 領域（seed の下位 32bit）とショートリードの ID（25bit）と seed のショートリード中での位置 (7bit) で構成される．同時に，index table を通して，CAL table

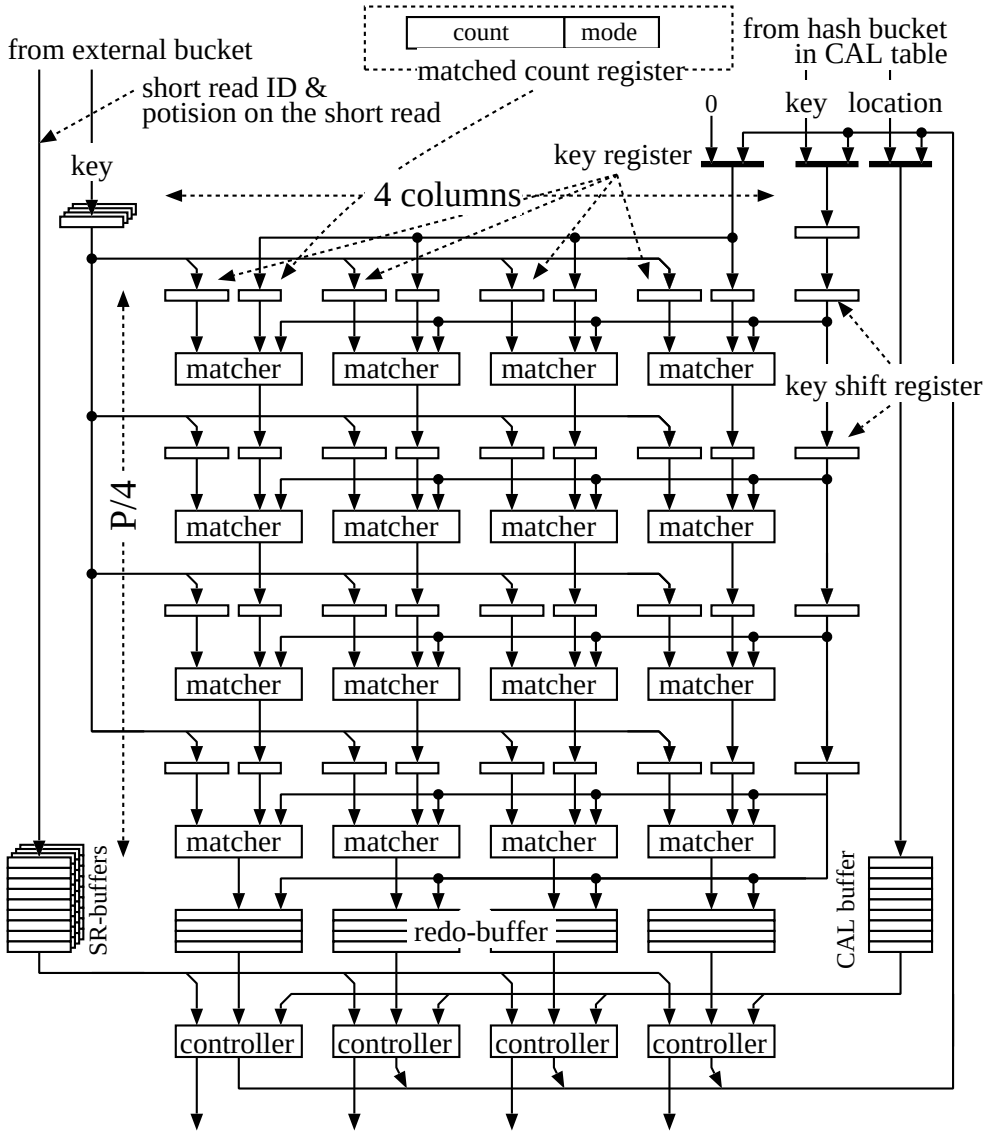


図 15: 並列比較ユニットのブロック図

から対応する key と location が読み出され，並列比較ユニットに与えられる．並列比較ユニットでは，ショートリードから抜き出した P 個の seed を CAL table と並列に比較する．並列比較ユニットでは，2 段階に分けて比較し，最終的には，一致したショートリード ID と CAL のペアを出力する．この 2 段階の処理は，“matched count register” の mode 領域が管理することで実現する．

表 10: 各モードにおける key 一致時の動作

mode	key 一致時の動作
first stage	“count” の値を 1 増加させる
second stage (not reach)	もし “count” の値が 0 ならば，“count” の値を 現在の行番号に変え，mode を ‘lock’ に変更する． それ以外は，“count” の値を 1 減少させる．
lock(second stage)	何も行わない

表 10 に，各モードにおける key が一致した時の動作を示す．モードは 3 通りで，2bit で表される．1 段階目は，一致した回数を数えるのみで．2 段階目で，実際に一致したペアを得る．

まず，1 段階目の動作について説明する．1 段階目の役割は，CAL table から読み出された各 key がそれぞれの列で何回一致するかを数え上げることである．外部バケットから 1 クロックサイクル毎に 4 個の seed（seed の key 領域（seed の下位 32bit）とショートリードの ID（25bit）と seed のショートリード中での位置 (7bit)）がバースト転送で読み出される．外部バケットから読み出された seed の key は，対応する “key register” に保持される．1 クロックサイクルあたり 4 個の seed を読みだし，$4x + k$ 番目に読み出された seed の key は，x 行目の k 列目の “key register” が記憶し，$P/4$ クロックサイクルで，P 個の “key register” 全てがセットされる．CAL table から読み出された key は，“key shift register” の最上部に一つずつ与えられる．“key shift register” は，1 クロックサイクル毎にデータをシフトダウンしていき，各行の 4 つの “key register” と並列に比較する．“matched count register” の count は，最初に 0 に初期化され，“key shift register” と同期してシフトダウンされる．“matched count register” の “count” は，同じ行の “key register” と “key shift register” が一致した時に 1 を加算される．CAL table から読み出された key は，最初に “key shift register” の最上部に与えられ，最下部に来た時に ($P/4$ クロックサ

イクルかかる)，その key は，全ての P 個の seed の key と比較されたことになる．“key shift register” と “matched count register” は，“matched count register” の “count” が 0 でなければ，“redo-buffer” に書き込まれる．そのとき，各列の 4 個の “matched count register” の “count” は，その key が各列で何回一致したかを意味する．外部バケットから読み出されたショートリードの ID（25bit）と seed のショートリード中での位置 (7bit) は，“SR-buffer” に書き込まれる．これらのデータは，最下部の出力部分でのみ必要で，key の比較には必要ないためである．同様に，CAL table から読み出された location も “CAL buffer” に書き込まれる．“SR-buffer” は，P 個のショートリードの ID（25bit）と seed のショートリード中での位置 (7bit) を記憶し，2 段階目で “key register” の位置（何行目か）をアドレスとして，対応するデータを出力する．“CAL buffer” は，単に長さ $P/4$ のバッファであり，入力されたデータが $P/4$ クロックサイクル後に出力される．

次に，2 段階目の動作について説明する．CAL table から対応するデータを全て読み出した後，2 段階目の動作が開始する．2 段階目の動作の目的は，1 段階目の結果を利用し，実際に一致したショートリード ID と location のペアを見つけることである．それぞれの “redo-buffer” は，“matched count” の “count” を m の時，m 個のコピーを作り，“key shift register” の先頭に再投入する．この時，“matched count” の初期値は，j $(0 \leq j < m)$ とする．この初期値によって，j 番目に一致した場所を記録するように動作し，m 個の一致した結果全てを見つけられるようにする．

“key register” と “key shift register” が一致した時には，表 10 に基づいて動作する．一致した時に，“count” が 0 の場合は，現在の行数を “count” に書き込み，“count” を変更しないように “mode” を変更する．“count” が 0 以外の場合は，“count” の値を 1 減算する．2 段階目で，key が末尾に来た時，“count” は,「何行目の “key register” と一致したか」を示す．この “count” を値を使って，“SR-buffer” からショートリードの ID と seed のショートリード中での位置を読み出し，“CAL　buffer” の出力と合わせることで，ショートリード ID と CAL のペアを得る．このショートリード ID と CAL をホスト PC に送り返す.

図 16 に，key 比較ユニットの動作例を示す．図 16 では，本来 4 列であるものを議論しやすいように 1 列にしている．実際の key 比較ユニットでは，“key shift register” のみ共用し，“key register” と “matched count register” は 4 列毎に独立して存在する．図 16 の左側の列は，1 段階目

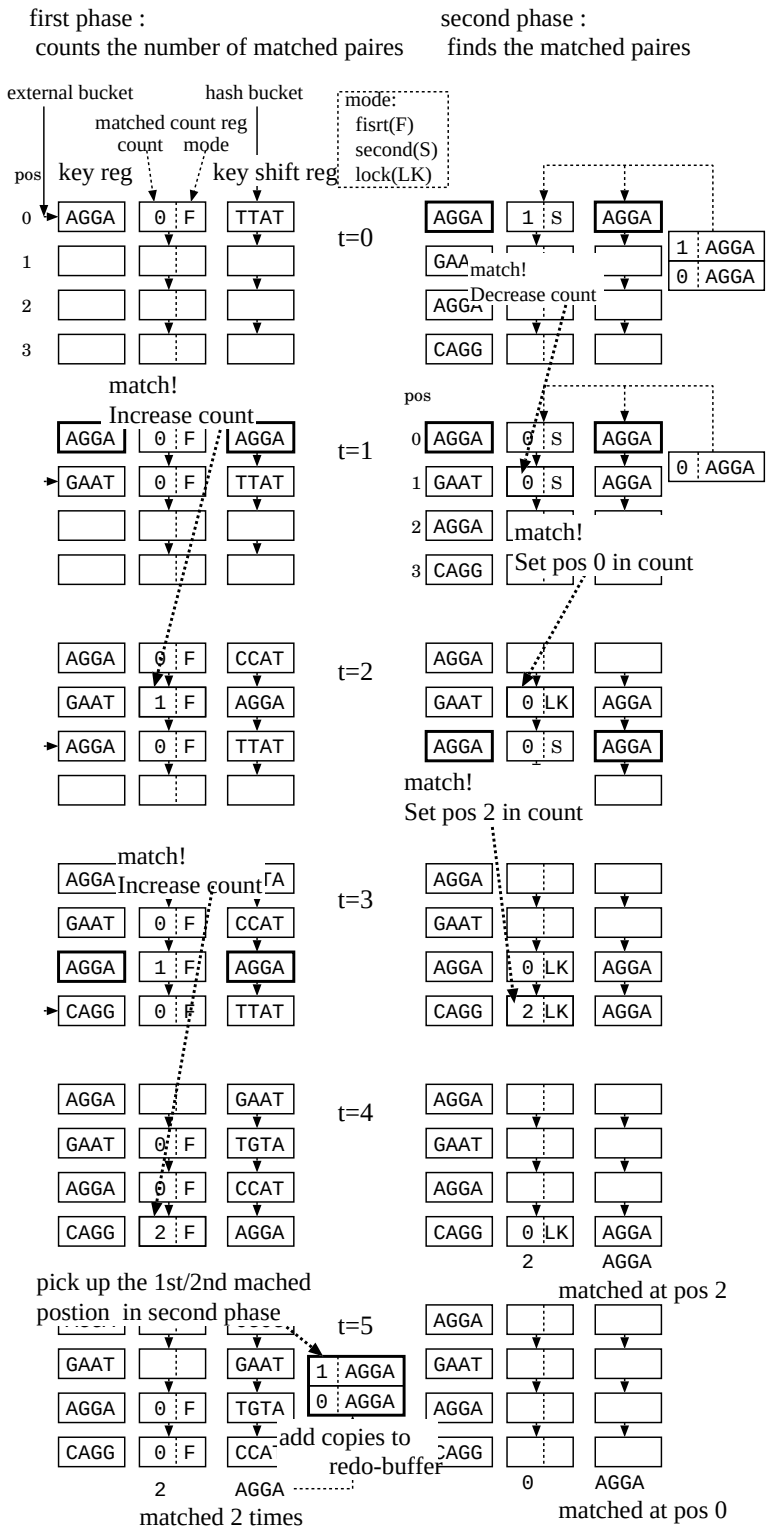


図 16: 並列比較ユニットにおける処理の一例

(first phase) の動作を示し，右側は 2 段階目 (second phase) の動作を示す．ここでは，まず，1 段階目の動作を図 16 を用いて説明する．ここでは，$t = 0$ クロックサイクル目に，外部バケットから “AGGA” という key が読み出され，0 番目の “key register” に格納される．同時に CAL table から，“TTAT” とう key が読み出され，“key shift register” の最上部に与えられる．“matched count register” は，“count” を 0，“mode” を ‘F’ に初期化する．$t = 1$ クロックサイクル目では，外部バケットから “GAAT” が読み出され，1 番目の “key register” に格納される．CAL table から，“AGGA” が読み出され，“key shift register” の最上部に与えられる．“matched count register” と “key shift register” の値はシフトダウンされる．このとき，0 行目の “key register” と “key shift register” が一致する．表 10 に基づいて，“matched count register” の “count” を 1 増やす（$t = 2$ クロックサイクル目の 1 行目の “matched count register” を参照）．$t = 4$ クロックサイクル目では，$t = 0$ クロックサイクル目で “key shift register” の最上部に与えられた “AGGA” が最下部に到達している．この時の，“matched count register” の “count” は ‘2’ となっており，これは，“AGGA” が “key register” と 2 回一致したことを意味する．“AGGA” を複製し，“AGGA” を 2 個 “redo-buffer” に書き込む．このとき，それぞれに，‘0’,‘1’ という数値をもたせる．これは，2 段階目で，それぞれ，‘0’,‘1’ 番目に一致したものを記録することを意味する．

次に，2 段階目の動作例を説明する．1 段回目の結果で，“AGGA” が 2 回一致したことがわかっており，{“AGGA”,‘1’}，{“AGGA”,‘0’} を “key shift register” と “matched count register” の初期値とする．$t = 0$ クロックサイクル目では，“redo-buffer” から読み出された {“AGGA”,‘1’} が “key shift register” と “matched count register” の先頭に格納される．このとき，0 行目の “key register” と “key shift register” が一致するため，表 10 にしたがって，“count” の値を 1 減算する（$t = 1$ の 1 行目を参照）．$t = 1$ クロックサイクル目では，“redo-buffer” から読み出された {“AGGA”,‘0’} が，“key shift register” と “matched count register” の先頭に格納される．このとき，0 行目の “key register” と “key shift register” が一致するため，表 10 にしたがって動作する．この場合，“count” の値が 0 のため，“count” に現在の行番号である 0 を格納する（$t = 2$ の 1 行目を参照）．このとき，“mode” も “LK” 変更することで，今後この “count” の値を変更しないようにする．このデータは，$t = 5$ で末尾から出力され，0 番目の位置で一致したことがわかる．$t = 2$ クロックサイクル目では，“redo-buffer” から，

“AGGA”,‘1’ として与えられたデータが，2 行目で “AGGA”,‘0’ となっており，“AGGA” が一致し，“count” が 0 となっているため，現在の行番号 2 を “count” に書き込む．

“AGGA” は，“key register” と 2 回一致するが，“redo-buffer” から，{“AGGA”,‘1’} として再投入された key は 2 回目に一致した場所を，{“AGGA”,‘0’} として再投入された key は 1 回目に場所を記憶する．これによって，1 段階目の結果を利用し，2 段階目の処理で，全ての一致箇所を発見することができる．

1 段階目では，4 列が完全に並列に動作するの対し，2 段階目では，1 列ずつ動作する．これは，“key shift register” が 1 列しかなく，これを 4 列で共有しているためである．

この key 比較ユニットでは，最大で，1 クロックサイクルあたり外部バケットから 4 個の seed(4×8B) を，CAL table から 1 行分のデータ（8B）を読みだす必要がある．本 FPGA のシステムのメモリ帯域は，この要求に対して十分な性能が求められる．

5.3.1 Matcher

本節では，並列比較ユニットで使われる Matcher について説明する．Matcher は，key を一致比較するユニットであり，並列比較ユニットに P 個存在する．図 17 に，Matcher のブロック図を示す．図 17 では，2bit で符号化された 16 塩基（32bit）の key0 と key1 が比較されている．’Exact matching’ は，塩基（2bit）毎に比較し，各塩基が一致したかを求める（出力は 16bit）．’Exact matching’ の出力が全て 1（つまり，0xFFFF）の場合は，key0 と key1 が完全に一致している．’#zero==1’ は，入力のビット列中の 0 の数を数え，1 だった場合は，1 を出力する．’leading gen’ は，最上位ビットから，連続する 1 を取り出す処理を行う (例えば，`1110XXXX` ⇒ `11100000`，`0XXXXXXX` ⇒ `00000000` となる)．2bit（1 塩基分）左または，右にシフトした key1 と key0 を ‘exact matching’ に与えることで，任意の位置の 1 塩基の挿入・欠損を検出することができる．‘valid bit’ は，flexible match よる過剰な一致を防ぐために付加された 1bit である．‘valid bit’ が 0 の場合は，key0，key1 の 16 塩基全てが一致する場合のみを検出し，‘valid bit’ が 1 の場合は，変異が 1 塩基以内の場合を検出する．

本回路は，exact match の場合に比べ，より多くの回路規模を必要とするが，並列比較ユニットの Matcher 以外の要素も含めると，全体の回路

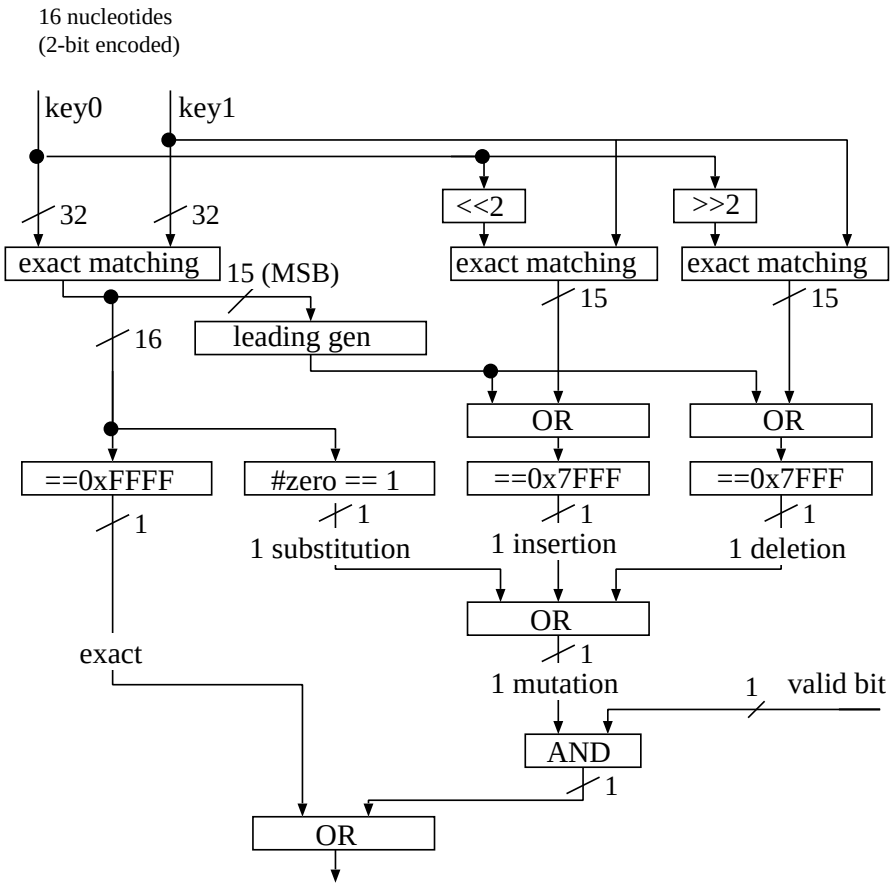


図 17: Matcher のブロック図

規模が 2 倍程度なる程度である．このため，key の比較が exact match から flexible match になると，同一の回路規模で実現できる P を半分程度にする必要がある．

5.4 Smith-Waterman 法の実装

FPGA を用いた SW 法については，ショートリードマッピングとは独立した遺伝子情報処理分野の問題として，様々な研究がなされている [58] [59] [60] [61].

ショートリードマッピングにおける SW 法の計算は，それらと比べて，

次の 2 点を考慮すれば，非常に簡単な問題であることがわかる．第一に，非常に短いシーケンス間の計算である．一般的な SW 法の FPGA 実装では，シーケンスが FPGA のレジスタで記憶できない程の大容量なものを想定していることが多いが，本問題ではショートリードのために非常に小さい．第二に，要素は‘A’，‘C’，‘G’，‘T’ の 4 種類のヌクレオチドのみである．一般的な SW 法の FPGA 実装では，アミノ酸を対象としていることが多く，$s(a_i, b_j)$ も複雑なスコアリング・マトリックスを用いることが多い（本問題では，単に，一致の場合 +1，不一致の場合 −2 であるためにスコアリング・マトリックスを持つ必要がない）．したがって，SW 法の実装において新規性は全く必要なく，既存研究に習えば十分に必要な性能が得られる．

本モジュールは，ショートリード（200bit）と CAL（32bit）が与えられ，CAL を元に，オフチップメモリから参照配列上の対応する領域を 256bit 分（128 塩基）読み出す．次に，ショートリード 100 塩基重と参照配列 128 塩基長の SW 法の計算を行う．SW 法の計算部分の回路は，SW Aligner と呼び，100 個の PE からなるシストリックアレイで構成され，100+128 クロックサイクルで計算が完了する．結果のスコアをホスト PC に送り返す．SW Aligner は，軽量な回路であり，複数個実装することで高速化を図る．

SW 法における各スコアは，affine gap を用いて，一致が 1，不一致が −2，ギャップの開始が −5，ギャップの延長が −2 とする．

5.5 実行時間の定式化と性能予想

本節では，総実行時間を定式化し，任意のハードウェアリソース量における性能予想を行う本システムの処理は，ソート処理，key の比較，SW 法の計算にわかれ，ソート処理，key の比較はどちらか一方のみが動作し，SW 法の計算は，並列に動作する．以上を踏まえると，ソートの実行時間を T_{sort}，key 比較の実行時間を T_{cmp}，SW 法の実行時間を T_{SW} とした時に，本システムの総実行時間 T_{total} は，次の式で表される．

$$T_{total} = \max(T_{sort} + T_{cmp}, T_{SW})$$

このとき，SW 法の計算回数（CAL の総数）は少なく，並列に動作する

SW Aligner を複数実装することで，十分高速に処理できるため，$T_{total} = T_{sort} + T_{cmp}$ となる．T_{sort} は，内部バケットを用いて，効率的にメモリアクセスすることで，1 クロックサイクルあたり 1 個の seed を処理できる．そのため，T_{sort} は次のように記述できる．

$$T_{sort} = N_s \times t$$

ただし，N_s は，処理する seed 数（ショートリード数 $\times$ 79），t は，動作周波数の逆数である．

次に，T_{cmp} は，次の式で表される．

$$T_{cmp} = \{N_s/2 + C_c \times N_s/P + N_{matched} + N_s/P \times L_d\} \times t$$

ただし，$N_{matched}$ は，key 比較の総一致回数であり，L_d は，DRAM アクセス時のレイテンシである．key の比較を行う並列比較ユニットは，2 段階で動作し，2 段階目の動作が，$N_{matched}$ に依存する．実験によると，$N_{matched}$ は，各ショートリードあたり 14 回程度であり，seed あたりに換算すれば，0.18 回程度になるので，$0.18 \times N_s$ とも書け，他の項と比べて影響は十分に小さい．

T_{total} は，次の式で表される．

$$\begin{aligned} T_{total} &= T_{sort} + T_{cmp} \\ &= \{N_s \times 3/2 + C_c \times N_s/P + N_{matched} + N_s/P \times L_d\} \times t \end{aligned}$$

T_{total} の定式化を行ったが，実際に性能予想を行うためには，平均比較回数である C_c を求める必要がある．表 11 に，fast mode の場合の各 l_{id} における C_c の値と内部バケット容量を示す．

次に，定式化した T_{total} を用いて，シミュレーションにより，性能予想を行った．本シミュレーションでは，データセットを，100 塩基長のショー

表 11: 内部バケットの容量と C_c

l_{idx}	14	15	16	17	18
内部バケット容量 [MB]	0.875	1.75	3.5	7	14
C_c	29155	14917	7701	4032	2245

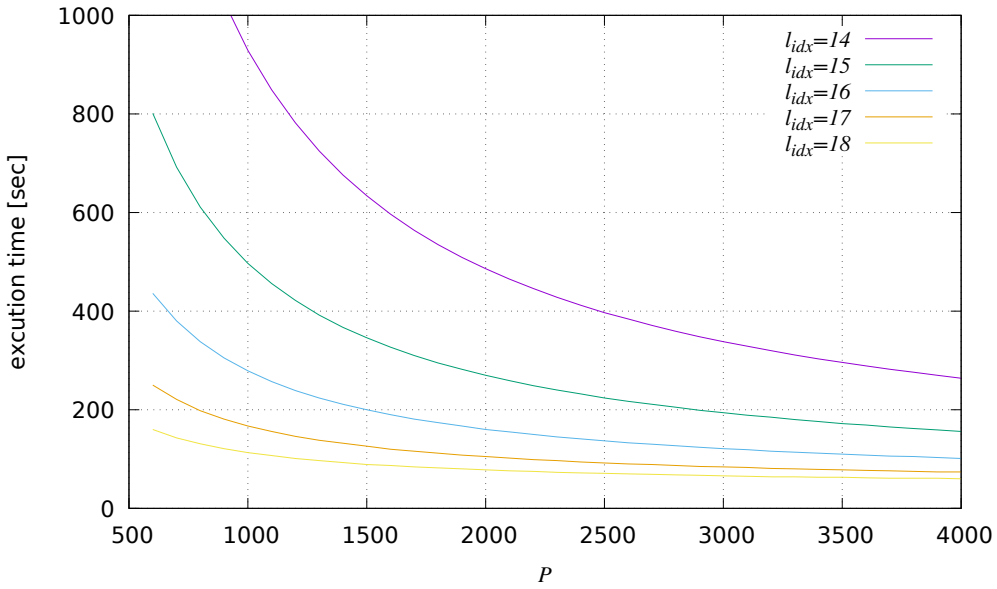


図 18: シミュレーションによる性能予想

トリード $64M$ 個とした．また，fast mode の index を使用するとし，回路の動作周波数を 200MHz，4 GB DDR3-800 ×2 以上のオフチップメモリを持つことを前提としている．DRAM アクセスレイテンシは，$L_d = 100$ クロックサイクルとした．$L_d = 100$ は，実際の DRAM アクセスレイテンシと比べ十分大きいが，L_d が，T_{total} に与える影響は非常に小さい．

図 18 に，シミュレーションによる性能予想の結果を示す．図 18 では，縦軸が実行時間（秒），横軸が P である．P は，並列比較ユニットで同時に比較する key の数であり，使用する FPGA の CLB 数に依存する．図 18 に示すように，l_{idx} が大きいほど性能が向上するが，表 11 に示すように，大容量の内部バケットが必要である．すなわち，使用する FPGA の BRAM 容量によって，l_{idx} が制限される．

表11と図18を用いれば，任意のFPGAについて，そのCLB数とBRAM容量に基づいて，性能を予想することができる．ただし，最後にP個のseedが集まらなかったバケットを処理するため（P個以下の並列比較になり効率が落ちる），実際の性能は，図18よりは僅かに劣る．

図18に示すように，Pが大きくなれば，より高い処理性能が実現できる．このシミュレーションは，オフチップ・メモリの性能を固定して行っており，図18の結果は，使用するFPGAのCLB数が増加すれば，より大きなPが実現でき，メモリ性能に関係なく高速化できることを意味する．

5.6 実装結果

本論文で提案を行うソートと並列比較に基づくショートリードマッピング並列処理アルゴリズムを，実際にFPGAに実装するためには，使用するFPGAのハードウェアリソース量（CLB数，BRAM容量）に応じて，P，l_{idx}を決定しなければならない．key比較の並列度であるPは，key比較ユニットで並べられる比較器の数によって決まり，使用するFPGAのCLBの量によって決定される．一方，l_{idx}は，ソートでの内部バケットの容量によって，使用するFPGAのBRAM容量によって決定される．大きなl_{idx}は，key比較回数を減らす一方で，多くのBRAM容量が必要になる．表11に示すように，大きいl_{idx}は，C_cを減らし性能を向上させるが，より多くの内部バケット容量を必要とする．本論文では，実装デバイスをXilinx社製のVirtex-7 XC7VX690Tとした．このFPGAが搭載されたFPGAボードは，オフチップメモリとして，4 GB DDR3-800を2バンク持つ．また，BRAMの総容量は，52.92Mbit(6.615MB)である．表11に示す内部バケット容量を考慮し，なるべく大きい$l_{idx} = 16$とした．また，$P = 2048$とした．

accurate modeではのCAL tableが9.5 GBになり，オフチップメモリの容量から，実装できない．そこでFPGA実装については，fast modeのみ実装を行った．

Scoring Module内では，SW Alignerを8個実装した．Scoring Moduleは，ソーティングとkeyの並列比較とは独立して動作し，十分な数のSW Alignerが実装されていれば，処理時間は，ソーティングとkeyの並列比較によって決まる．8個のSW Alignerは，十分な数であることを実験的に確認している．

表 12: FPGA 実装における各リソースの使用率（$P = 2048$）

#LUTs(K)	#Registers(K)	#BRAMs	freq(MHz)
336(78%)	120(14%)	898(61%)	200

本システムを，Verilog HDL で設計し，Virtex-7 XC7VX690T に実装した．合成ツールは，Xilix 社の Vivado 2014.2 を使用した．動作周波数は，DRAM インターフェース (DDR3-800) に同期し，200MHz とした．表 12 に，実装結果を示す．括弧内は，各リソースの使用率を示す．

表 13: FPGA 実装における主なメモリ消費

on-chip(BRAM)	mask table	0.12 MB
	内部バケット	3.50 MB
off-chip DRAM banks	index table	1.25 MB
	reference genome	0.78 GB
	外部バケット	1.0 GB
	CAL table	3.3G GB

表 13 に，オンチップ（BRAM) およびオフチップメモリに格納される各テーブルの容量を示す．

第6章
GPUを用いたシステム構築

本章では，提案手法のGPU実装について述べる．対象デバイスは，GTX Titan Xp（1.58 GHz）とする．CUDAは，バージョン 8.0を使用する．

6.1 GPUシステムの全体像

本手法の処理は, ソート処理, keyの並列比較, SW法の計算に分かれる．ソート処理は，ホストPC上で行い，GPUで実行するカーネル関数はkeyの並列比較を行う比較カーネルとSW法の計算するスコア・カーネルの2種類である．

FPGA実装では，内部バケットの容量によって l_{idx} の大きさに制限があった．GPU実装では，ホストPCでソートするため，その制限はない．そのため，より大きな l_{idx} を実現できるが，大きすぎる l_{idx} では，P が小さくなり，keyの並列比較の効率が悪くなり，性能が悪化する．このトレードオフについては，実験的に求め，より性能が優れている $l_{idx} = 20$, $P = 2048$ とした．

表 14: GPUシステムにおける主要なメモリ消費

GPU（global mem）	CAL table	3.3(9.5)GB
	参照配列	0.78 GB
ホストPC	ソート用のバケット	18.4GB
	ショートリードとベストスコア（64M個分の場合）	6.1GB

表14にGPUシステムにおける主要なメモリ消費を示す．CAL tableの容量は，fast modeで，カッコ内の値はaccurate modeである．

GPUシステムを設計する上では，次に示す3つの項目について注目して工夫を行った．

(i) GPUの大規模な並列演算資源の利用率を高める

(ii) GPU のアイドルタイムを極力減らす

(iii) ホスト PC で行うソーティング処理が，ボトルネックならないようにする

(i) については，GTX Titan Xp は，3840 個の CUDA コアを持ち，CUDA コアはパイプライン化されているため，実際に搭載されている総コア数の複数倍のスレッド数を同時に実行した方が演算効率が良い．そのため，より大規模なスレッド数を持つカーネル関数を定義する必要がある．

まず，(i) に対応する実装上の工夫として，Hyper-Q の利用がある．CUDA による GPU の基本的な利用方法は，(1)CPU のメモリから GPU のメモリ（global memory）に入力データを転送，(2) カーネル関数を実行，(3) 結果を CPU のメモリに転送，という段階に分かれる．これらの処理は，GPU 上のストリームと呼ばれるキューに登録され，(1) から (3) が順番に実行される．NVIDIA 社の kepler アーキテクチャ以降の GPU では，Hyper-Q と呼ばれる複数のストリームを持つ機能がある．ストリーム内で実行の順序関係は守られるが，複数のストリームを使えば，ストリーム間の実行関係については順序関係に制約がなく，GPU が自動的に負荷の割当を考えて処理を分配する．この機能を利用すれば，実行資源の利用率が上がりやすくなるため，高速化できる．本システムは，Hyper-Q を利用し，ホスト PC の各スレッドが，独立したストリームを持ち，独立してカーネル関数を呼び出す．

比較カーネルでは，一つのバケットが $P = 2048$ 個の seed を持ち，このバケットを 32 個同時に処理するようにした．また，スコア・カーネルでも，同時に複数の CAL を処理するようにして，GPU の大規模な並列演算資源の利用率を高められるように設計した（これについては，第 6.4 節で，定量的な評価を行っている）．

次に，(ii)，(iii) に対応する実装上の工夫を述べる．処理の基本的な流れは，次のようになる．

1. ホスト PC でソート処理を行いバケットを集める
2. 集まったバケット 32 個を GPU に転送し，比較カーネルで，key の比較を行い CAL を求める
3. 得られた CAL を元に，スコア・カーネルを呼び出し，各 CAL を SW 法で評価し，評価結果に応じてベストスコアを更新する

基本的には，上記の順番で処理すれば良いのだが，実際には，(ii)，(iii) を考慮して，より複雑な順序で計算を行う．例えば，ホスト PC は，比較カーネルの終了を待たずにソーティングを再開して，次の比較カーネル用のバケットを集めたほうが効率が良い．

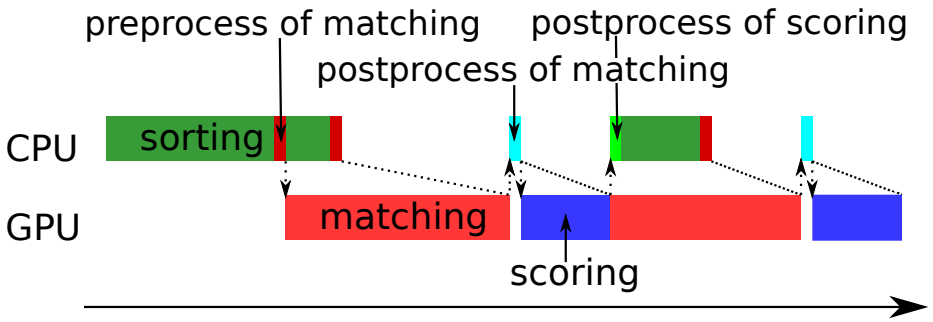


図 19: GPU 実装における処理のタイムライン

図 19 に，GPU システムにおけるホスト PC と GPU の処理のシーケンスを示す．ホスト PC では，FASTQ 形式のファイルからショートリードを読み出し，2bit に符号化した後，各 seed を抜き出して hash 値を計算し，各バケットに追加する．図 19 では，この処理をまとめて，“sorting” と表現している．各バケットは，P 個の seed で一杯となり，32 個の一杯になったバケットが集まった時に，これらのバケット中のデータを GPU に転送する（“preprocess of matching”）．次に，比較カーネル（“matching”）を呼び出し，GPU で先ほど転送した 32 バケット分の key の並列比較が実行される．ホスト PC は，比較カーネルの終了を待たずにソーティングを再開する．再開したソーティング処理により再度 32 個のバケットが集まると，そのバケットを次の比較カーネルの実行のために転送し，その後，ホスト PC は，前回呼び出した比較カーネルの終了と同期する．その結果をホスト PC のメモリに転送し，その結果を元に，SW 法で評価するショートリードと CAL のペアを GPU に転送する（“postprocess of matching”）．次に，スコア・カーネル（scoring）を呼び出し，ホスト PC は，スコア・カーネルの終了を持ち，スコア・カーネルで計算した SW 法のスコアをホスト PC に転送し，ベストスコアを更新したか確認する（“postprocess of scoring”）．その後，先ほど転送したバケットを入力とする比較カーネルを呼び出す．

このように，(ii)，(iii) を考慮して，GPU とホスト PC でなるべくアイ

ドルタイムを作らないように設計している．図 19 は，ホスト PC のある 1 スレッドに注目した場合であり，実際はホスト PC の各スレッドが独立して同様の処理を行う．そのため，GPU のアイドルタイムはより少なくなり，GPU の大規模な並列演算資源の利用率を高めることができる．

表 15: GPU システムにおける各処理の実行スレッド数

処理	実行スレッド数
ソート	8 スレッド（ホスト PC）
key の並列比較	65536 スレッド（$32 \times P(= 2048)$）
SW 法	1638400 スレッド † （16384×100）

† 最小でこの数字であり，key の比較結果に応じてより大きなスレッド数で実行される．

表 15 に，GPU システムにおける各処理の実行スレッド数を示す．ソートについては，ホスト PC で行うため，今回使用する CPU である「Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz」に依存し，8 スレッドで実行する．key の並列比較は，比較カーネルと呼ばれるカーネル関数で計算される．比較カーネルでは，一つのバケットが $P(= 2048)$ 個の seed を持ち，このバケットを 32 個同時に処理する．各スレッドが一つの seed に対応するため，$32 \times P(= 2048)$ スレッドで実行される．SW 法は，スコア・カーネルと呼ばれるカーネル関数で計算される．スコア・カーネルは，複数の CAL を同時に評価する．同時に評価する CAL 数が最小で 16384 で，それぞれショートリードの長さの並列度を持つので，16384×100 スレッドで実行される．

6.2 ソートの実装

ソートの目的は，バケットソートを用いて同じ hash 値の seed を P 個集めることであり，key の並列比較で，集められた P 個の seed を CAL table と並列に比較する．FPGA システムでは，FPGA の BRAM を利用して内部バケットを構成し，キャッシュメモリとして利用することで，バースト転送によりデータ転送の効率化を図った．GPU 実装で，同様の手法を行

うには，内部バケットを shared memory で構成する必要があるが，その容量は 96KB と小さいため，FPGA と同様の手法は困難である．

したがって，ソート処理は，ホスト PC 上で行い，同じ hash 値の seed が P 個集まったバケット中のデータを GPU へ転送する方式を採用する．ここで問題となるのは，ホスト PC 上で十分高速にソート処理が行えるかということであるが，ホスト PC と GPU 非同期的に動作するため，ホスト PC でソート処理を行っている間に GPU は別の処理を実行できる．そのため，ソート処理が，GPU の総処理時間と比べて十分に高速で，ボトルネックにならない限りで，ホスト PC におけるソート処理の実行時間は総処理時間にほとんど影響を与えない．

6.3 key の並列比較の実装

key の並列比較は，比較カーネルで行う．比較カーネルでは，32 個のバケット（同じ hash 値をもつ P の seed）が入力として与えられ，global memory から CAL table を読みだして key の比較を行う．カーネル関数の 1 スレッドが，1 個の seed に対応し，総スレッド数は，$32 \times P(= 2048) = 65536$ となる．CAL table の読み出しはコアレッシング [44] を利用し，効率的な転送を図る．コアレッシングとは，カーネル関数の各スレッドが，global memory の連続したアドレスにアクセスするときに，一つのメモリ・トランザクションでまとめアクセスすることで，効率的なメモリアクセスを行うことである．

flexible match は，ビット演算により実装される．ここで，4 バイト整数型（16 塩基）の key0 と key1 を比較する flexible match のソースコードを示す．このコードでは，ビット演算により flexible match（key0 と key1 が 1 塩基以内の変異であるか）を検出している．

```
__device__ bool flexible_match(const uint32_t
   key0, const uint32_t key1) {
 uint32_t bit0 = ~(key0 ^ key1);
 uint32_t bit1 = ~((key0 << 2) ^ key1);
 uint32_t bit2 = ~((key0) ^ (key1 << 2));

 bit0 = ((bit0 & (bit0 << 1)) | 0x55555555);
 bit1 |= 0x00000003;
 bit1 = (bit1 & (~(bit1 + 1)));

 bit2 |= 0x00000003;
```

```
  bit2 = (bit2 & (~(bit2 + 1)));

  return ((bit0 | (bit0 + 1)) == 0xFFFFFFFF) ||
     ((bit0 | bit1) == 0xFFFFFFFF) ||
         (((bit0 | bit2) == 0xFFFFFFFF));
}
```

6.4 Smith-Waterman 法の実装

GPU を用いた SW 法の高速化はいくつか存在するものの，[62] で指摘されているように，CUDASW++ [63]，GPU-BLAST [64] は，ヌクレオチドではなくタンパク質限定のものであり，対象もショートリードではなく長い塩基配列である．第 5.4 節で FPGA における SW 法の実装について述べたように，ショートリードマッピングにおける SW 法は，シーケンス長が短く，ヌクレオチド限定であるために，一般的な SW 法より簡単な問題である．

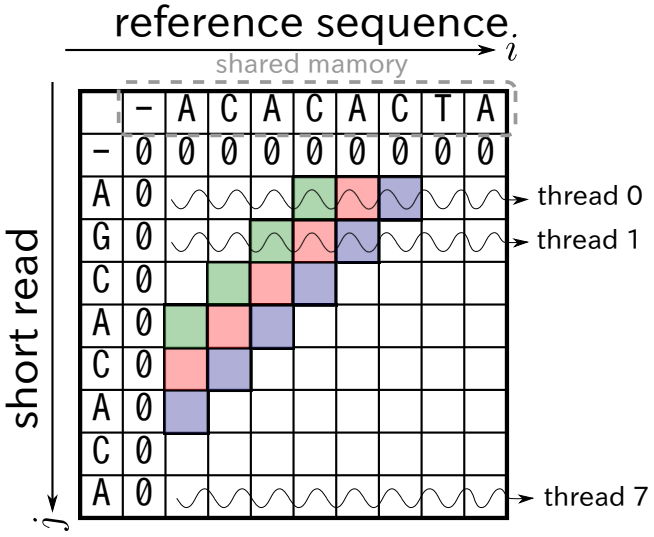


図 20: GPU における SW 法の計算

そこで，あらたに GPU を用いた SW 法の実装を提案する．SW 法を計算するカーネル関数を，スコア・カーネルと呼び，ショートリードと CAL が与えられ，スコアを返す．与えられた CAL から，対応する参照配列配

列中の領域を 128 塩基分読みだす（ギャップがあるため，ショートリードより少し長めに用意している）．

図 20 に，スコア・カーネルの構成を示す．図 20 では，ショートリードは，“AGCACACA” であり，マトリクスの縦軸に対応する．参照配列は，“ACACACTA” であり，マトリクスの横軸に対応する．SW 法は，定義にしたがって $H(i,j)$ を計算するが，$H(i,j)$ の計算には，$H(i-1,j)$，$H(i,j-1)$，$H(i-1,j-1)$，a_i(参照配列の i 番目の文字)，b_j(ショートリードの j 番目の文字) が必要である．この依存関係にしたがって，図 20 中の青で示される場所の $H(i,j)$ の計算には，緑と赤で示される場所の $H(i,j)$ が必要である．逆に言うと，緑と赤で示される場所の $H(i,j)$ があれば，青で示される場所の全ての $H(i,j)$ が同時に計算可能である．

ショートリードの長さを L，参照配列の長さを M とする．スコア・カーネルでは，1 ブロックが，一つのショートリードと参照配列中のペアの SW 法を計算し，全 L スレッドで実行される．k 番目のスレッドは，$H(i,k)$ を計算する（図 20 中の波線）．各スレッドは，スレッドローカルな値（レジスタ）としてショートリードの一文字（k 番目のスレッドは b_k）を持つ（つまり，スレッド 0 なら ‘A’，スレッド 1 なら ‘G’）．参照配列は，shared memory に保存され，各スレットから読み出される．また，$H(i,j)$ は，shared memory に保存するが，全ての場所を保存する必要はなく，図 20 中の緑と赤で示される部分の長さ L の配列 2 個でよい．このようにショートリードを shared memory に保存しなくてよいこと，$H(i,j)$ を全領域保存しなくて良いことがメモリの使用効率を大幅に上げ実行効率を改善している．

表 16: ブロック長と SW カーネルの性能

ブロック長	処理時間 [秒]
32	21.1
256	3.5
1024	2.9
4096	2.6
16384	2.5
65536	2.5

スコア・カーネルの性能評価を行った．本評価では，10M 個のショー

トリードと CAL のペアの SW 法の計算をする時間を測定している．1 つの CAL の評価に，1 つのカーネル関数を呼び出したのでは，カーネル関数呼び出し時のオーバーヘッドの影響が大きく性能が悪い．そこで，スコア・カーネルは，1 ブロックで 1 個のショートリードと CAL のペアを処理し，1 つのカーネル関数で同時に複数ブロックを処理する．その同時に処理する数をブロック長と呼ぶ．表 16 に，ブロック長と SW カーネルの性能を示す．ブロック長が短すぎると，カーネル関数を読みだす回数が多く，カーネル関数呼び出し時のオーバーヘッドが性能を悪化させている．さらに，実行するカーネル関数の総スレッド数が，GPU の並列演算能力に比べて少ないため，演算資源の利用率が低くなり，十分な性能が発揮できていないことがわかる．ブロック長が一定以上になると，カーネル関数呼び出し時のオーバーヘッドは無視できる程度になり，GPU の並列演算能力も十分効果的に働くため，性能が頭打ちになる．

この結果を元に，スコア・カーネルは，比較カーネルの出力である CAL が 16384 個以上溜まった時に，SW カーネルで処理するように設計した．つまり，図 19 に示すように，比較カーネルの後で必ず，スコア・カーネルが呼び出されるのではなくて，一定個数まで溜めておいて，一定個数以上集まった時にまとめてスコア・カーネルで処理するようにした．

第7章
性能評価

本章では，マッピング精度と処理性能について他の手法と比較し，本論文で提案するソートと並列比較に基づくショートリードマッピング並列処理アルゴリズムの有効性を明らかにする．提案手法では，FPGA を用いたシステム構築と GPU を用いたシステム構築を行っており，それらの性能を他のソフトウェアマッピングツール，他の FPGA システム，GPU を用いたマッピングツールと比較する．

マッピング精度については，変異を含んだショートリードを模擬データとして作成し，ショートリードの変異量と正しくマッピングできる割合の関係を明らかにすることで，提案手法の有効性を示す．実際のショートリードマッピングにおいて，参照配列とショートリードは，非常に類似しているものの僅かに変異を含んでおり，変異を含んだショートリードに対するマッピング能力が求められる．

処理性能については，実データを用いて評価を行い，提案手法の有効性を示す．ショートリードマッピングは，非常に大量のショートリードを処理する必要があるために，優れた処理速度が求められる．

7.1 マッピング精度の比較

本節では，提案手法と他の手法のマッピング率を比較する．参照配列は，NCBI [53] よりダウンロードしたヒトゲノム GRChv38 を使用する．実データは，正解のマッピング位置が不明なため，マッピング率を正確に測定することは困難である（単に，マッピングできたかどうかしかわからない）．各手法のマッピング率を正確に比較するために，参照配列のランダムな位置から 100 塩基を抜き出し，1 塩基あたり $m\%$の確率で変異を加えたものを 1M 個用意した．このデータセットでは，参照配列からショートリードを抜き出した地点を正解のマッピング位置として記憶している．変異は，置換，欠損，挿入が，0.8，0.1，0.1 の比率で分布し，この割合は，[55]，[56] で報告されている実際のヒトの DNA シーケンシグの分析結果である SNV と indel の出現比率（SNV が indel の約 4 倍）を

元にしている．

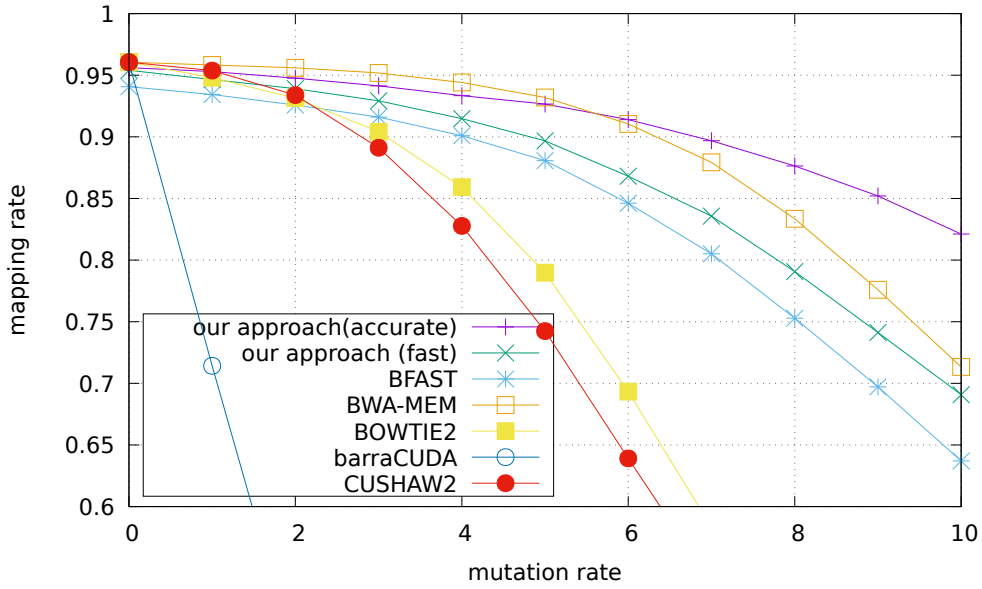


図 21: 各手法のマッピング率の比較

図 21 に，ショートリードの変異率 m が 0%から 10%の場合における，各手法のマッピング率を示す．このグラフでは，縦軸がマッピング率で，単にマッピングできたかどうかではなく，正解のマッピング位置が得られた割合を示す．横軸は，ショートリードの 1 塩基あたりの変異率 m[%] である．この実験では，提案手法のマッピング率と最も高精度なソフトウェアマッピングツールの 1 つである BWA-MEM [16]，BOWTIE2 [17]，提案手法の元になった hash-index 法に基づくソフトウェアである BFAST [20]，さらに GPU を用いたマッピングツールである barraCUDA [65] と CUSHAW2-GPU [46] を比較している．

図 21 に示すように，どの手法も m が大きくなるほど，マッピング成功率が下がる．提案手法の accurate mode は，m が小さいときに BWA-MEM に近接し僅かに劣る程度のマッピング率を示し，m が大きい時には BWA-MEM を大きく上回るマッピング率を達成した．提案手法の fast mode では，BWA-MEM には劣るものの BFAST と比べて全ての m で優れたマッピング率を示した．また，accurate mode，fast mode ともに，BWA 以外のマッピングツールと比べて，全ての m で高いマッピング率を示した．以上により，本手法のマッピング率は優れていると言える．

7.2 処理性能の比較

本節では，実データを用いて，各手法の処理速度を比較する．本実験では，ヒトの全ゲノムシーケンシングのためにIllumina社のHiseq 2000で生成された実データ（SRR385773）をNCBIのSRA（Sequence Read Archive）よりダウンロードして使用する．この実データは，ショートリード長が，100塩基であり，本実験では，先頭から64M個のショートリードを利用する．

表17に，各手法の実行時間とマッピング率，BWA-MEMと比較した時の高速化率を示す．Olson等のFPGAシステム [37] は，提案手法と同じBFASTに基づいたアルゴリズムを利用するFPGAシステムであり，最も高性能なFPGAシステムの1つである．

ソフトウェアの実行環境は，CPUはIntel(R) Core(TM) i7-6700K CPU @ 4.00GHz，メインメモリの容量は32GB，OSはUbuntu 16.04で実行している．また，全てのソフトウェアは8スレッドで実行している．BFASTのindexは，‘bfast index -f /ref/GRCh37/v37.fa -A 0 -m 1111111111111111111111 -w 14 -i 1 -n 8’というコマンドで作成し，これは，seed長が22塩基で$l_{idx} = 28$（14塩基）を意味する．他のソフトウェアについては，デフォルトオプションで実行している．表17に示すマッピング率は，単に，ショートリードがマッピング場所を持つかどうかであり，そのマッピング場所が正しい位置かどうかは不明である．したがって，マッピング率については，図21に示したものが正確な値である．

本研究では，FPGAを用いたシステムとGPUを用いたシステムの構築を行ったが，まず，FPGAシステムついて性能を検討する．表17に示すように，提案手法のFPGAシステムは，ソフトウェアマッピングツールであるBWA-MEMと比べて約17.6倍の高速化を達成している．同じFPGAシステムとして，提案手法のFPGAシステムとOlson等のFPGAシステムと比較を行った．Olson等のFPGAシステムは，Virtex-6 XC6VLX240Tを8個使用しており，また，性能評価を76塩基長のショートリード50M個で行っているが，表17では，1個のVirtex-6 XC6VLX240T，100塩基長のショートリード64M個という条件に換算している．異なるデバイスに実装されたFPGAシステムを比較するのは難しいが，なるべく公平に比較するために，CLBの数を考慮して比較を行った．提案手法のFPGA実装の実行時間は，Olson等のFPGAシステムの約2.8であり，使用するFPGAのCLBは，約2.9であるため，同等の効率であると言える．また，

Olson 等の FPGA システムのマッピング率は，BFAST と同じと考えられるが，図 21 に示すように，提案手法の fast-mode は BFAST よりマッピング率が高いため，提案手法の FPGA システムの方が高いマッピング率を持つことがわかる．

次に，GPU システムについて検討する．提案手法の GPU システムは，BWA-MEM と比べ，fast mode で約 8 倍，accurate mode で約 6 程度の高速化を達成している．提案手法の GPU システムの accurate mode と CUSHAW2-GPU が同程度の性能を示し，提案手法の GPU システムの fast mode が，CUSHAW2-GPU の 1.2 倍程度の性能を示す．一方で，図 21 に示すように，マッピング率については accurate mode では，提案手法が大きく上回り，fast mode ですら CUSHAW2-GPU より優れた結果を示している．以上を考慮すると，GPU システムについても十分優れた結果が得られたと言える．

表 17: 各手法の実行時間とマッピング率

	実行時間 [秒]	マッピング率 [%]	高速化率
BWA-MEM [1]	3074	99.60	1.0
BOWTIE2 [17]	2404	98.25	1.3
BFAST [20]	19026	98.99	0.2
Olson et al. [37](Virtex-6 XC6VLX240T)†	458	98.99	6.7
barraCUDA [47](GTX Titan Xp)	1351	95.67	2.3
CUSHAW2-GPU [46](GTX Titan Xp)	486	97.65	6.3
提案手法（FPGA） (fast mode) (Virtex-7 XC7VX690T)	175	98.23	17.6
提案手法（GPU) (fast mode) (GTX Titan Xp)	393	98.23	7.8
提案手法（GPU) (accurate mode) (GTX Titan Xp)	486	98.74	6.3

† の 8 個の Virtex-6 XC6VLX240T，76 塩基長のショートリード 50M 個で 34 秒という結果 [37] から，1 個の Virtex-6 XC6VLX240T，100 塩基長のショートリード 64M 個 に換算している．

第8章
結論

8.1 まとめ

DNA シーケンシングに必要な情報処理であるショートリードマッピングは，高速化が課題であり，本研究では，書き換え可能な LSI である FPGA と画像処理用の演算ユニットである GPU を用い，FPGA または GPU を付加したパーソナルコンピュータという比較的小規模な環境で，ショートリードマッピング高速計算システムの構築を行った．本論文では，ソートと並列比較に基づくショートリードマッピング並列処理アルゴリズムを提案し，FPGA および GPU を用いたシステム構築を行い，その性能を明らかにした．FPGA システムは，Xilinx 社の Virtex-7 XC7VX690T に実装し，最も優れたソフトウェアマッピングツールである BWA-MEM と比べて，約 18 倍の高速化を達成した．GPU システムでは，CUDA を利用し NVIDIA 社の GTX Titan Xp に実装し，BWA-MEM に対して，約 8 倍の高速化を達成した．また，マッピング率も，BWA-MEM と遜色ない程度に優れており，特に変異が多いショートリードに対しては BWA-MEM を凌駕している．

本研究で提案した手法は，従来のハードウェアシステムとは異なり，メモリ転送速度の制約をほとんど受けずに，ハードウェアの規模に従って性能向上可能である．これは，提案手法が，本論文で示したハードウェア構成において他の手法に対して優れているだけでなく，将来開発されるであろうより大規模なハードウェアを用いれば，より優れた性能が実現できることを意味する．将来の DNA シーケンサーの性能（ショートリードの生成速度）向上，DNA シーケンシングの需要拡大が予想されているが，CPU の性能向上が鈍化しているため，CPU の性能に依存するソフトウェアアプローチでは，将来求められる性能に対応できない．提案手法を用いれば，より大規模なハードウェアを用いることで将来においても十分な性能が達成できると考えられる．

ショートリードマッピングは，処理の潜在的な並列性が非常に高い一方で，大規模なデータに対して局所性が低い参照を頻繁に行う問題である．このような問題は，メモリ転送速度がボトルネックとなるため，FPGA

やGPUを用いた並列処理で大幅な高速化を実現することはできないと考えられて来た．本研究は，その定説に挑戦したものであり，本研究の成果を通して，他の同様の問題に対しても，突破口のヒントを与えうるものである．

本論文では，対象を参照配列をヒトゲノムに限定して議論した．その長さが4G塩基を超えない場合は，別の参照配列についてもそのまま応用できる．ただし，参照配列が特に短い場合（例えば，FPGAのオンチップメモリに保存できる程度）は，異なる手法の方が優れていると考えられる．これは，本手法がオフチップメモリにおけるメモリアクセスボトルネックに注目したものであるため，そもそもメモリアクセスがボトルネックにならないケースでは，本手法が有効に働かないためである．また，長さが4Gを越える参照配列は，参照配列中の位置を示すために8B必要になり，データ転送の問題によって性能が悪化するだろう．この場合における性能低下率は，未検討である．

8.2 今後の課題

NGSの技術の進歩は著しく，その性能向上とコスト低下が著しく進んでいる．それらの進歩速度と比べては，非常にゆっくりであるものの，DNAシーケンサーのリード長（一度に読みだすDNA断片の長さ）も長くなっている．これは，より長いリードの方が，より複雑な遺伝的特徴を検出しやすいためである．

本論文では，リード長を100塩基長に限定して議論した．この長さは，Illumina社のHi Seq 2000が対応している一番長いリード長である．Illumina社の最先端のDNAシーケンサーでは，リード長が150から300塩基長程度まで対応しているものがある．更に将来的には，より長いリード長が実現されると思われる．

本手法では，seed-and-extend法に基づくものであり，その処理は大きく2段階に別れる．seeding段階では，ショートリード全体ではなく一部を用いて処理を行うため，処理する総塩基数に依存し，処理時間はリード長の影響を受けない．一方で，extension段階では，ショートリード全体を利用するため処理時間がリード長の影響を受ける．本論文中の評価では，常にseeding段階の方がボトルネックになる状況を考え，seeding段階の高速化を中心に議論しているが，リード長が長くなるにつれて，そのバランスは変わってくるだろう．リード長が500から1000塩基長を超

えてくると，現在の構成では性能が出にくく，新たな検討が必要になると思われる．

また，本論文で対象としたFPGAボードは，DDR3メモリを搭載しているものであるが，今後，より帯域幅が優れたHBM（High Bandwidth Memory）を搭載したFPGAボードが，高性能計算の分野で主流になって行くことが考えられる．本論文で提案した手法が，HBMを搭載したFPGAボードでどれぐらい有効であるか，また，その場合において他の手法と比較したときの優位性等は，検討されていない．

これらの問題に対する対応が今後の課題である．

謝辞

筑波大学　システム情報系　教授であられる丸山勉先生には，大学 4 年次の研究室配属時から 6 年間に渡りご指導とご教鞭をいただきました．FPGA や GPU を用いたショートリードマッピングの高速計算という大変興味深いテーマを研究する機会を与えていただきました．今の私があるのは，丸山先生と出会えたことが大きかったと思っています．ここに深く感謝申し上げます．

筑波大学　システム情報系　准教授であられる延原肇先生には，本論文の審査ともに，学士および修士の副指導教員も務めていただきました．何度も丁寧に話を効いて頂き，ご助言等いただきました．深く感謝を申し上げます．

筑波大学　システム情報系　教授　和田耕一先生，同　教授　安永守利先生，同　准教授　山口佳樹先生には，本論文の審査を快くお引き受けいただくと共にに本研究についての貴重なご助言をしていただきました．深く感謝いたします．

筑波大学　システム情報系　教授をご定年退職され，現在は名誉教授であられる白川友紀先生には，学士および修士の副指導教員も務めていただきました．深く感謝申し上げます．

筑波大学　システム情報系のリコンフィギュラブルコンピューティングシステム研究室（丸山研）のメンバーの皆様には，ゼミ等を通して有意義な議論をさせていただきました．ここに感謝申し上げます．

最後に，博士課程に進学することを許していただいた両親に，心から感謝いたします．

参考文献

[1] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*, 2013.

[2] Sam Behjati and Patrick S Tarpey. What is next generation sequencing? *Archives of Disease in Childhood-Education and Practice*, Vol. 98, No. 6, pp. 236–238, 2013.

[3] Kris A Wetterstrand. DNA sequencing costs: data from the NHGRI genome sequencing program (GSP), 2017. `https://www.genome.gov/sequencingcostsdata/` 2017年12月1日に閲覧.

[4] Peter JA Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research*, Vol. 38, No. 6, pp. 1767–1771, 2009.

[5] Samtools. `https://samtools.github.io/hts-specs/SAMv1.pdf` 2017年12月1日に閲覧.

[6] David J Lipman and William R Pearson. Rapid and sensitive protein similarity searches. *Science*, Vol. 227, No. 4693, pp. 1435–1441, 1985.

[7] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, Vol. 11, No. 5, pp. 473–483, 2010.

[8] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, Vol. 147, No. 1, pp. 195–197, 1981.

[9] Stephan Pabinger, Andreas Dander, Maria Fischer, Rene Snajder, Michael Sperk, Mirjana Efremova, Birgit Krabichler, Michael R Speicher, Johannes Zschocke, and Zlatko Trajanoski. A survey of tools for variant analysis of next-generation genome sequencing data. *Briefings in bioinformatics*, Vol. 15, No. 2, pp. 256–278, 2014.

[10] Jing Shang, Fei Zhu, Wanwipa Vongsangnak, Yifei Tang, Wenyu Zhang, and Bairong Shen. Evaluation and comparison of multiple

aligners for next-generation sequencing data analysis. *BioMed research international*, Vol. 2014, , 2014.

[11] Ahmad Al Kawam, Sunil Khatri, and Aniruddha Datta. A survey of software and hardware approaches to performing read alignment in next generation sequencing. *IEEE/ACM transactions on computational biology and bioinformatics*, 2016.

[12] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 2000.

[13] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. In *Digital Equipment Corporation, Technical report 124*, 1994.

[14] Tak Wah Lam, Wing-Kin Sung, Siu-Lung Tam, Chi-Kwong Wong, and Siu-Ming Yiu. Compressed indexing and local alignment of dna. *Bioinformatics*, Vol. 24, No. 6, pp. 791–797, 2008.

[15] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome Biology*, Vol. 10, No. 3, 2009.

[16] Heng Li and Richard Durbin. Fast and accurate short read alingment with burrows-wheeler transform. *Bioinformatics*, Vol. 25, No. 4, pp. 1754–1760, 2009.

[17] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with bowtie 2. *Nature Methods*, Vol. 9, No. 4, pp. 357–359, 2012.

[18] Heng Li and Richard Durbin. Fast and accurate long-read alignment with burrows–wheeler transform. *Bioinformatics*, Vol. 26, No. 5, pp. 589–595, 2010.

[19] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. Soap2: an improved ultrafast tool for short read alignment. *Bioinformatics*, Vol. 25, No. 15, pp. 1966–1967, 2009.

[20] N. Homer, B. Merriman, and S. F. Nelson. Bfast: An alignment tool for large scale genome resequencing. *PLoS ONE*, Vol. 4, No. 11, 2009.

[21] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. Soap: short oligonucleotide alignment program. *Bioinformatics*, Vol. 24, No. 5, pp. 713–714, 2008.

[22] Hui Jiang and Wing Hung Wong. Seqmap: mapping massive amount of oligonucleotides to the genome. *Bioinformatics*, Vol. 24, No. 20, pp. 2395–2396, 2008.

[23] Davide Campagna, Alessandro Albiero, Alessandra Bilardi, Elisa Caniato, Claudio Forcato, Svetlin Manavski, Nicola Vitulo, and Giorgio Valle. Pass: a program to align short sequences. *Bioinformatics*, Vol. 25, No. 7, pp. 967–968, 2009.

[24] Guillaume Rizk and Dominique Lavenier. Gassst: global alignment short sequence search tool. *Bioinformatics*, Vol. 26, No. 20, pp. 2534–2540, 2010.

[25] Yangho Chen, Tade Souaiaia, and Ting Chen. Perm: efficient mapping of short sequencing reads with periodic full sensitive spaced seeds. *Bioinformatics*, Vol. 25, No. 19, pp. 2514–2521, 2009.

[26] Wan-Ping Lee, Michael P Stromberg, Alistair Ward, Chip Stewart, Erik P Garrison, and Gabor T Marth. Mosaik: a hash-based algorithm for accurate next-generation sequencing short-read mapping. *PloS one*, Vol. 9, No. 3, p. e90581, 2014.

[27] Novocraft. `http://www.novocraft.com` 2017年12月1日に閲覧.

[28] Stephen F Altschul and Bruce W Erickson. Optimal sequence alignment using affine gap costs. *Bulletin of mathematical biology*, Vol. 48, No. 5-6, pp. 603–616, 1986.

[29] 天野英晴, 尼崎太樹, 飯田全広, 泉知論, 長名保範, 佐野健太郎, 柴田裕一郎, 末吉敏則, 中原啓貴, 張山昌論, 丸山勉, 密山幸男, 本村真人 山口佳樹, 渡漫実. FPGAの原理と構成. オーム社, 2016.

[30] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 0–560, 2006.

[31] IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-1987*, pp. 0–, 1988.

[32] Edward Fernandez, Walid Najjar, Elena Harris, and Stefano Lonardi. Exploration of short reads genome mapping in hardware. In *FPL*, pp. 360–363, 2010.

[33] Knodel O, Preusser T, and Spallek R. Next-generation massive parallel short-read mapping on fpgas. In *IEEE International Conference on Application-Specific Systems; Architecture and Processors (ASAP)*, pp. 195–201, 2011.

[34] E. Fernandez, W. Najjar, and S. Lonardi. String matching in hardware using the fm-index. In *FCCM*, pp. 218–225, 2011.

[35] Wen TANG, Wendi WANG, Bo DUAN, Chunming ZHANG, Guangming TAN, Peiheng ZHANG, and Ninghui SUN. Accelerating millions of short reads mapping on a heterogeneous architecture with fpga accelerator. In *FCCM*, pp. 184–187, 2012.

[36] Thomas B Preuber, Oliver Knodel, and Rainer G Spallek. Short-read mapping by a systolic custom fpga computation. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pp. 169–176. IEEE, 2012.

[37] C. B. Olson, M. Kim, C. Clauson, B. Kogon, C. Ebeling, S. Hauck, and WL. Ruzzo. Hardware acceleration of short read mapping. In *FCCM*, pp. 161–168, 2012.

[38] Yupeng Chen, Bertil Schmidt, and Douglas Leslie Maskell. A hybrid short read mapping accelerator. *Bioinformatics*, Vol. 14, No. 67, 2013.

[39] Saul B Needleman and Christian D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, Vol. 48, No. 3, pp. 443–453, 1970.

[40] J. Arram, K. H. Tsoi, Wayne Luk, and P. Jiang. Hardware acceleration of genetic sequence alignment. In *ARC*, pp. 13–24, 2013.

[41] Hasitha Muthumala Waidyasooriya, Masanori Hariyama, and Michitaka Kameyama. Fpga-accelerator for dna sequence alignment based on an efficient data-dependent memory access scheme. In *Proc. of the 5th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*, pp. 127–130, 2014.

[42] Edward B Fernandez, Jason Villarreal, Stefano Lonardi, and Walid A Najjar. Fhast: Fpga-based acceleration of bowtie in hardware. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, Vol. 12, No. 5, pp. 973–981, 2015.

[43] デイビッド・A・パターソン, ジョン・L・ヘネシー, 成田光彰. コンピュータの構成と設計 第5版 下. 日経BP社, 2014.

[44] Programming guide :: Cuda toolkit documentation. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` 2017年12月1日に閲覧.

[45] Yongchao Liu, Bertil Schmidt, and Douglas L Maskell. Cushaw: a cuda compatible short read aligner to large genomes based on the burrows–wheeler transform. *Bioinformatics*, Vol. 28, No. 14, pp. 1830–1837, 2012.

[46] Yongchao Liu and Bertil Schmidt. Cushaw2-gpu: empowering faster gapped short-read alignment using gpu computing. *IEEE Design & Test*, Vol. 31, No. 1, pp. 31–39, 2014.

[47] Petr Klus, Simon Lam, Dag Lyberg, Ming Sin Cheung, Graham Pullan, Ian McFarlane, Giles SH Yeo, and Brian YH Lam. Barracuda-a fast short read sequence aligner using graphics processing units. *BMC research notes*, Vol. 5, No. 1, p. 27, 2012.

[48] Chi-Man Liu, Thomas Wong, Edward Wu, Ruibang Luo, Siu-Ming Yiu, Yingrui Li, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, et al. Soap3: ultra-fast gpu-based parallel alignment tool for short reads. *Bioinformatics*, Vol. 28, No. 6, pp. 878–879, 2012.

[49] Richard Wilton, Tamas Budavari, Ben Langmead, Sarah J Wheelan, Steven L Salzberg, and Alexander S Szalay. Arioc: high-throughput read alignment with gpu-accelerated exploration of the seed-and-extend search space. *PeerJ*, Vol. 3, p. e808, 2015.

[50] Yoko Sogabe and Tsutomu Maruyama. An acceleration method of short read mapping using fpga. In *Field-Programmable Technology (FPT), 2013 International Conference on*, pp. 350–353. IEEE, 2013.

[51] Yoko Sogabe and Tsutomu Maruyama. Fpga acceleration of short read mapping based on sort and parallel comparison. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–4. IEEE, 2014.

[52] Yoko Sogabe and Tsutomu Maruyama. A variable length hash method for faster short read mapping on fpga. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pp. 1–6. IEEE, 2015.

[53] National center for biotechnology information. `https://www.ncbi.nlm.nih.gov/` 2017年12月1日に閲覧.

[54] イルミナ株式会社. `https://jp.illumina.com/` 2017年12月1日に閲覧.

[55] Ryan E Mills, W Stephen Pittard, Julienne M Mullaney, Umar Farooq, Todd H Creasy, Anup A Mahurkar, David M Kemeza, Daniel S Strassler, Chris P Ponting, Caleb Webber, et al. Natural genetic variation caused by small insertions and deletions in the human genome. *Genome research*, pp. gr–115907, 2011.

[56] Daichi Shigemizu, Akihiro Fujimoto, Shintaro Akiyama, Tetsuo Abe, Kaoru Nakano, Keith A Boroevich, Yujiro Yamamoto, Mayuko Furuta, Michiaki Kubo, Hidewaki Nakagawa, et al. A practical method to detect snvs and indels from whole genome and exome sequencing data. *Scientific reports*, Vol. 3, p. 2161, 2013.

[57] Yoko Sogabe and Tsutomu Maruyama. A fast and accurate FPGA system for short read mapping based on parallel comparison on hash table. *IEICE Transactions*, Vol. 100-D, No. 5, pp. 1016–1025, 2017.

[58] Yoshiki Yamaguchi, Tsutomu Maruyama, and Akihiko Konagaya. High speed homology search with fpgas. In *PSB*, pp. 271–282, 2002.

[59] Isaac TS Li, Warren Shum, and Kevin Truong. 160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga). *BMC bioinformatics*, Vol. 8, No. 1, p. 185, 2007.

[60] Peiheng Zhang, Guangming Tan, and Guang R Gao. Implementation of the smith-waterman algorithm on a reconfigurable supercomputing platform. In *Proceedings of the 1st international workshop on High-performance reconfigurable computing technology and applications: held in conjunction with SC07*, pp. 39–48. ACM, 2007.

[61] Chi Wai Yu, KH Kwong, Kin-Hong Lee, and Philip Heng Wai Leong. A smith-waterman systolic cell. In *International Conference on Field Programmable Logic and Applications*, pp. 375–384. Springer, 2003.

[62] Pankaj Gupta. Swift: A gpu-based smith-waterman sequence alignment program, 2012.

[63] Yongchao Liu, Douglas L Maskell, and Bertil Schmidt. Cudasw++: optimizing smith-waterman sequence database searches for cuda-enabled graphics processing units. *BMC research notes*, Vol. 2, No. 1, p. 73, 2009.

[64] Panagiotis D Vouzis and Nikolaos V Sahinidis. Gpu-blast: using graphics processors to accelerate protein sequence alignment. *Bioinformatics*, Vol. 27, No. 2, pp. 182–188, 2010.

[65] William B Langdon, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. Improving cuda dna analysis software with genetic programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pp. 1063–1070. ACM, 2015.

研究業績

査読付雑誌論文

- Yoko Sogabe, and Tsutomu Maruyama. ”A fast and accurate FPGA system for short read mapping based on parallel comparison on hash table.” IEICE Transactions on Information and Systems 100.5 (2017): 1016-1025.

査読付国際会議論文

- Yoko Sogabe, and Tsutomu Maruyama. ”An acceleration method of short read mapping using FPGA.” Field-Programmable Technology (FPT), 2013 International Conference on. (4 pages) IEEE, 2013.
- Yoko Sogabe, and Tsutomu Maruyama. ”FPGA acceleration of short read mapping based on sort and parallel comparison.” Field Programmable Logic and Applications (FPL), 2014 24th International Conference on. (4 pages) IEEE, 2014.
- Yoko Sogabe, and Tsutomu Maruyama. ”A variable length hash method for faster short read mapping on FPGA.” Field Programmable Logic and Applications (FPL), 2015 25th International Conference on. (6 pages) IEEE, 2015.

査読なし（技術報告等）

- 曽我部 陽光, 丸山 勉「FPGA を用いたショートリードマッピングの高速化」, 電子情報通信学会技術研究報告 = IEICE technical report : 信学技報 113(52), 19-24, 2013-05-20 電子情報通信学会