Technische Universität Dresden
Fakultät Informatik
Institut für Technische Informatik
Professur für VLSI-Entwurfssysteme, Diagnostik und Architektur

Diplomarbeit

Implementierung des Genom-Alignments auf modernen hochparallelen Plattformen

Oliver Knodel
geboren am 11. Dezember 1984 in Herford
(Mat.-Nr.: 3111110)

Betreuender Hochschullehrer:
Prof. Dr.-Ing. habil. Rainer G. Spallek

Betreuer:
Dipl.-Inf. Thomas B. Preußer

Dresden, 29. Mai 2011

Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Diplomarbeit zum Thema

Implementierung des Genom-Alignments auf modernen hochparallelen Plattformen

selbstständig verfasst und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt, nur die angegebenen Quellen benutzt und die in den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Oliver Knodel, Dresden, 29. Mai 2011

Wettbewerbsrechtlicher Hinweis

Die bloße Nennung von Namen, Produkten, Herstellern und Firmennamen dient lediglich als Information und stellt keine Verwendung des Warenzeichens sowie keine Empfehlung des Produktes oder der Firma dar.

Kurzfassung

Durch die wachsende Bedeutung der DNS-Sequenzierung wurden die Geräte zur Sequenzierung weiterentwickelt und ihr Durchsatz so erhöht, dass sie Millionen kurzer Nukleotidsequenzen innerhalb weniger Tage liefern. Moderne Algorithmen und Programme, welche die dadurch entstehenden großen Datenmengen in akzeptabler Zeit verarbeiten können, ermitteln jedoch nur einen Bruchteil der Positionen der Sequenzen in bekannten Datenbanken. Eine derartige Suche ist eine der wichtigsten Aufgaben in der modernen Molekularbiologie. Diese Arbeit untersucht mögliche Übertragungen moderner Genom-Alignment Programme auf hochparallele Plattformen wie FPGA und GPU.

Die derzeitig an das Problem angepassten Programme und Algorithmen werden untersucht und hinsichtlich ihrer Parallelisierbarkeit auf den beiden Plattformen FPGA und GPU analysiert. Nach einer Bewertung der Alternativen erfolgt die Auswahl eines Algorithmus. Anschließend wird dessen Übertragung auf die beiden Plattformen entworfen und implementiert. Dabei stehen die Geschwindigkeit der Suche, die Anzahl der ermittelten Positionen sowie die Nutzbarkeit im Vordergrund.

Der auf der GPU implementierte reduzierte Smith & Waterman-Algorithmus ist effizient an die Problemstellung angepasst und erreicht für kurze Sequenzen höhere Geschwindigkeiten als bisherige Realisierungen auf Grafikkarten. Eine vergleichbare Umsetzung auf dem FPGA benötigt eine deutlich geringere Laufzeit, findet ebenfalls jede Position in der Datenbank und erreicht dabei ähnliche Geschwindigkeiten wie moderne leistungsfähige Programme, die aber heuristisch arbeiten. Die Anzahl der gefundenen Positionen ist bei FPGA und GPU damit mehr als doppelt so hoch wie bei sämtlichen vergleichbaren Programmen.

Abstract

Further developments of DNA sequencing devices produce millions of short nucleotide sequences. Finding the positions of these sequences in databases of known sequences is an important problem in modern molecular biology. Current heuristic algorithms and programs only find a small fraction of these positions. In this thesis genome alignment algorithms are implemented on massively parallel platforms as FPGA and GPU.

The next generation sequencing technologies that are currently in use are reviewed regarding their possible parallelization on FPGA and GPU. After evaluation one algorithm is chosen for parallelization. Its implementation on both platforms is designed and realized. Runtime, accuracy as well as usability are important features of the implementation.

The reduced Smith & Waterman algorithm which is realized on the GPU outperforms similar GPU programs in speed and efficiency for short sequences. The runtime of the FPGA approach is similar to those of widely used heuristic software mappers and much lower than on the GPU. Furthermore the FPGA guarantees to find all alignment positions of a sequence in the database, which is more than twice the number that is found by comparable software algorithms.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BLAST	Basic Local Alignment Search Tool
BLAT	**B**LAST-**L**ike **A**lignment **T**ool
bp	base pairs
BWA	Burrows-Wheeler Aligner
CLB	Configurable Logic Block
CUDA	Compute Unified Device Architecture
DNS	Desoxyribonukleinsäure (engl. DNA - Deoxyribonucleic Acid)
FANGS	Fast Algorithm for Next Generation Sequencers
FASTA-Format	Format zur Darstellung einer Sequenz der DNS-Primärstruktur
FASTA	Algorithmus zur Suche in Datenbanken - Abkürzung für *fast all*
FASTQ-Format	Format zur Darstellung einer Sequenz mit zusätzlichen Qualitätsinformationen
FIFO	First In First Out
FLOPs	Floating Point Operations per second
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
HMR	High Mapping Reads
HSP	High Scoring Pair
IOB	Input/Output Block
LLNL	Lawrence Livermore National Laboratory
LUT	Look Up Table

MAC-Adresse	Media-Access-Control-Adresse
NCBI	National Center for Biotechnology Information
OpenCL	**Open C**omputing **L**anguage
OSI	Open Systems Interconnection Reference Model
PET	Paired-End Tag
PSM	Programmable Switch Matrix
PST	Precomputed Score Table
RNA	Ribonukleinsäure
SAM	Sequence Alignment/Map
SSD	Solid-State Drive
SRAM	Static Random Access Memory
SRL	Shift Register Latch
TCP	Transmission Control Protocol
VHDL	Very High Speed Integrated Circuit Hardware Description Language

1 Einleitung

Die Entschlüsselung der menschlichen DNS[1] war eines der großen wissenschaftlichen Ziele des vergangenen Jahrhunderts. Seit dem Jahr 2003 ist das menschliche Genom vollständig bekannt. Das erlaubt es, Erbkrankheiten zu erforschen und neue Krebsarten besser zu verstehen. Da zu den Gendatenbanken immer neue Organismen und Viren hinzugefügt werden, wachsen diese Datenbanken mittlerweile exponentiell an. [Böc03, TS09]

Mit dem Beginn einer schnellen und einfachen Sequenzierung in den 1950er Jahren durch Fred Sanger begannen die Datenbanken zu wachsen und enthalten neben den eigentlichen Sequenzen auch deren Eigenschaften. Diese Eigenschaften werden im Labor ermittelt und analysiert, was oftmals mehrere Jahre in Anspruch nimmt. [SNC77]

Um Zeit und Kosten zu sparen, werden heutzutage neue Sequenzen zu Beginn mit den bekannten Datenbanken abgeglichen, um ähnliche Sequenzen zu finden und so die Eigenschaften der neuen Sequenzen einfacher ermitteln zu können. Bei dieser Suche handelt es sich allerdings nicht um eine exakte Suche, da Fehler durch die Sequenzierung, aber auch Mutationen enthalten sein können. [RB01]

Eine der wichtigsten Anwendungen in der modernen Molekularbiologie besteht in der Suche dieser neu gewonnenen Sequenzen in Datenbanken. Die ersten, in den 1970er Jahren entwickelten Algorithmen, arbeiteten mit einer exakten Suche. Um in akzeptabler Zeit Ergebnisse zu erzielen, werden aber seit Ende der 1980er Jahre heuristische, hoch parallele Verfahren wie BLAST zur Suche langer Sequenzen in Datenbanken eingesetzt. Diese Algorithmen arbeiten auf großen Computer-Clustern, die für eine Suche mehrere Tage voll ausgelastet sind. [Mar08, TS09]

Neue *Next-Generation Sequencing* Technologien haben das Sanger-Verfahren abgelöst und somit die Anforderungen an die Suche verändert. Indem der erzielte Durchsatz gesteigert wurde, entstand eine völlig neuen Ausgangslage. Innerhalb eines Durchlaufes von wenigen Tagen werden mehrere Millionen Sequenzen mit einer Länge von wenigen Basenpaaren erzeugt. Die bisherigen Algorithmen, die an wenige lange Sequenzen angepasst sind, sind nicht mehr nutzbar und neue so genannte Short-Read Mapper wurden entwickelt. [Mar08, TS09]

Da die neuen Sequenzierungstechniken aufgrund ihrer geringen Anschaffungs- und Betriebskosten weit verfügbar sind, gewinnen kostengünstige und trotzdem schnelle Verfahren zur Suche immer mehr an Bedeutung. Software-Algorithmen, die nur einen Bruchteil der Ergebnisse liefern, werden dadurch verstärkt eingesetzt. Trotzdem ist es notwendig, Suchergebnisse hoher Qualität zu liefern, was einer hohen Anzahl gefundener und auch korrekter Positionen entspricht. Daher

[1] Desoxyribonukleinsäure (engl. DNA - Deoxyribonucleic Acid)

verspricht der Einsatz moderner hochparalleler Plattformen wie FPGA[2] und GPU[3] Vorteile in Bezug auf Geschwindigkeit und Qualität der Suche, da diese Systeme eine hohe Rechenleistung bei vergleichsweise geringen Anschaffungskosten bieten.

Diese Arbeit untersucht die Möglichkeit einer Übertragung moderner Genom-Alignment Algorithmen auf hochparallele Plattformen. Wichtig ist dabei neben der Geschwindigkeit eine hohe Qualität der Ergebnisse. Eine Herausforderung besteht dabei speziell im Umgang mit den großen Datenmengen. Ziel ist es, ein System zu entwickeln, welches von typischen Nutzern bedient und eingesetzt werden kann. Dazu ist es von Bedeutung, besonders beim FPGA eine stabile und leistungsfähige Datenkommunikation zu implementieren.

Zu Beginn der Arbeit werden in Kapitel 2 die Grundlagen des Sequenzalignments erläutert und die wichtigsten Algorithmen vorgestellt, bevor anschließend das Short-Read Mapping Problem und bisherige Lösungsansätze vorgestellt werden. In Kapitel 3 werden zunächst die Anforderungen an das zu entwerfende System erläutert und anschließend versucht, die zuvor erläuterten Algorithmen auf die zur Verfügung stehenden parallelen Plattformen zu übertragen. Hierbei wird abgewägt, mit welchem Algorithmus eine hohe Geschwindigkeit erzielt werden kann sowie anschließend ein Entwurf vorgestellt.

Der für eine Übertragung eines Algorithmus auf die Plattformen ausgewählte Entwurf wird schließlich in Kapitel 4 implementiert. Dabei werden nach einer Erweiterung der Anforderungen die wichtigsten Punkte der Umsetzung erläutert.

Kapitel 5 gibt einen Überblick über die zum Funktionstest verwendeten Testdaten und spezielle Testfälle. Des Weiteren erfolgt eine detaillierte Analyse der Messergebnisse und ein Vergleich zu ausgewählten modernen Programmen aus Kapitel 2. Abschließend erfolgt in Kapitel 6 eine Zusammenfassung der Arbeit mit einer Analyse der Ergebnisse sowie einem Ausblick für mögliche Weiterentwicklungen.

[2]Field Programmable Gate Array
[3]Graphics Processing Unit

2 Grundlagen

Dieses Kapitel fasst die wesentlichen Grundlagen und Algorithmen des Sequenz-Alignments zusammen, um anschließend einen Algorithmus für eine spezielle Plattform auswählen zu können.

In Abschnitt 2.1 wird zunächst die Problemstellung des Sequenz-Alignment erläutert. Anschließend werden in Abschnitt 2.2 die gebräuchlichen Algorithmen vorgestellt, die über viele Jahre für das Alignment weniger langer Sequenzen eingesetzt wurden und auch heute noch gebräuchlich sind. Abschnitt 2.3 stellt das in den letzten Jahren aufgetretene Short-Read Mapping Problem vor. Dessen bisherige Lösungsansätze und Algorithmen werden in Abschnitt 2.4 vorgestellt.

Die Möglichkeiten und Besonderheiten der zur Verfügung stehenden parallelen Plattformen, werden im Abschnitt 2.5 aufgezeigt. Anschließend folgt in Abschnitt 2.6 eine Analyse bisheriger Lösungsansätze auf Parallelen Plattformen.

2.1 Sequenzalignment

2.1.1 Einführung in das Sequenzalignment

Unter dem Begriff Sequenz-Alignment werden Algorithmen zusammengefasst, welche die Ähnlichkeit zweier Genom-Sequenzen zueinander nachweisen. Die ursprüngliche Anwendung besteht im Vergleich von zwei oder mehreren DNS-Sequenzen verschiedener Organismen. Die Ähnlichkeit der Sequenzen zueinander ist ein wichtiges Maß um eine Aussage über den Verwandschaftsgrad treffen zu können.

Das Alphabet besteht bei Nukleotidsequenzen aus den vier Basen Adenin, Guanin, Thymin und Cytosin. Ein Nukleotid wird mit dem Anfangsbuchstaben der in ihm enthaltenen Base A, G, T oder C bezeichnet. Bei der RNA[1] ersetzt Uracil das Thymin. [Mun01, HS04]

Im Gegensatz zu einfachen Suchalgorithmen wie dem Algorithmus von Knuth-Morris-Pratt oder dem Boyer-Moore Algorithmus, die nur exakte Treffer ermitteln, werden beim Alignment auch *Mismatches* und *Gaps* akzeptiert. Ein *Match* liegt immer dann vor, wenn zwei Nukleotide identisch sind, ansonsten ergeben sie ein *Mismatch*. Bei der Arbeit mit *Gaps* können Positionen in eine Sequenz eingefügt (*Insertion*) oder entfernt (*Deletion*) werden. Abbildung 2.1 zeigt ein einfaches Alignment zweier Sequenzen mit mehreren Matches, einem Mismatch und einem Gap am Beispiel zweier Nukleotidsequenzen. [Han06, OW90]

[1] Ribonukleinsäure

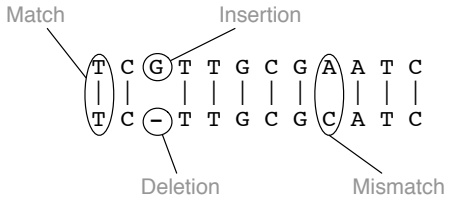


Abbildung 2.1: Definition von Match, Mismatch, Insertion und Deletion am Beispiel zweier Nukleotidsequenzen über dem Alphabet $\Sigma = \{A, G, C, T(\equiv U)\}$. Nach [Han06].

Um eine Aussage über die Qualität eines Alignments treffen zu können, wird ein *Score* berechnet. Der höchste Score, den zwei Sequenzen erreichen können, entspricht demnach dem optimalen Alignment. Gleichung 2.1 zeigt, wie der Score eines Alignments berechnet werden kann. Der Parameter n entspricht dabei der Länge des Alignments der beiden Sequenzen s und t. Die Gleichungen 2.2, 2.3 und 2.4 zeigen eine mögliche Definition der Funktion $\delta(a, b)$ zur beispielhaften Berechnung des Scores zweier Nukleotide a und b. Über die Gapkosten g in den Gleichungen 2.3 und 2.4 kann die Qualität des Alignments verändert werden. So ist es möglich, Gaps stärker zu bewerten als Mismatches. Das Alignment aus Abbildung 2.1 erreicht einen Score von Acht.

$$score = \sum_{i=0}^{n-1} \delta(s_i, t_i) \tag{2.1}$$

$$\delta(a, b) = \begin{cases} 1 & a = b \quad Match, \\ 0 & a \neq b \quad Mismatch \end{cases} \tag{2.2}$$

$$\delta(a, -) = g = -2 \qquad Deletion \tag{2.3}$$

$$\delta(-, b) = g = -2 \qquad Insertion \tag{2.4}$$

Bevor in Abschnitt 2.2 einige Alignment-Algorithmen erläutert werden, wird zunächst kurz auf die üblichen Sequenzierungsformate eingegangen. Diese Formate stellen den Ausgangspunkt für die Algorithmen dar, die immer mindestens zwei Sequenzen für ein Alignment benötigen. In vielen Fällen sind mehrere Sequenzen in einer Datei gespeichert, die dann in einer Datenbank eines Genoms, welches ebenfalls aus mehreren Sequenzen bestehen kann, gesucht werden.

2.1.2 Ausgangsformate

Die Sequenzen werden üblicherweise im FASTA-Format[2] oder FASTQ-Format[3] gespeichert. Dabei handelt es sich um ein einfaches Format, bei dem eine Datei mehrere Sequenzen enthalten

[2]Format zur Darstellung einer Sequenz der DNS-Primärstruktur
[3]Format zur Darstellung einer Sequenz mit zusätzlichen Qualitätsinformationen

kann, die mit Kommentarzeilen voneinander getrennt sind. Das Zeichen für den Beginn eines Kommentars ist eine spitze Klammer (>). In der folgenden Zeile beginnt die Sequenz, die mit den ASCII[4]-Zeichen *A*, *G*, *T* und *C* kodiert ist. Unbekannte Nukleotide werden mit dem Zeichen *N* kodiert. Listing 2.1 zeigt eine einfache Sequenz im FASTA-Format. [Han06]

Listing 2.1: Beispiel für eine Sequenz im FASTA-Format.

```
1 >gi|170079663|ref|NC_010473.1| Escherichia coli str. K-12, complete
2 AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
```

Eine Referenzdatenbank besteht üblicherweise aus mehreren sehr langen Sequenzen. Tabelle 2.1 zeigt wichtige Daten zweier Genome, um einen Überblick über die Größenordnungen zu vermitteln. Die Größe einer Datenbank oder die Länge einer Sequenz wird üblicherweise mit der Anzahl der Basenpaare (bp[5]) angegeben. Mittlerweile wächst die Anzahl der bekannten Sequenzen und somit die Größe und Anzahl der bekannten Datenbanken exponentiell an. [NCB10]

Tabelle 2.1: Aktuelle Genom-Datenbanken. Datenbanken von [NCB10]

Datenbank	Sequenzen	Basenpaare	Datenvolumen
Homo sapiens (AC000044.1)	73	9.011.418.319	8,4 GByte
Drosophila melanogaster (AE003845.2)	1170	124.326.318	119 MByte

2.2 Klassische Alignment-Algorithmen

Die Alignment-Algorithmen werden in der Regel in zwei Gruppen eingeteilt. Zum Einen gibt es die exakten Verfahren, die in Abschnitt 2.2.1 vorgestellt werden und die aufgrund ihrer hohen Laufzeit hauptsächlich zum Alignment zweier langer Sequenzen eingesetzt werden.

Durch das exponentielle Wachstum der Referenzdatenbanken stellt die Suche von neu sequenzierten Genomen durch die hohe Laufzeit der exakten Algorithmen ein Problem dar. Abschnitt 2.2.2 stellt die wesentlichen heuristischen Verfahren zur Suche von langen Sequenzen in Datenbanken vor, die in akzeptabler Zeit Ergebnisse liefern.

2.2.1 Exakte Verfahren

2.2.1.1 Überblick

Die beiden exakten Alignment-Algorithmen sind der Needleman & Wunsch-Algorithmus für ein *globales* Alignment und der Smith & Waterman-Algorithmus, der ein *lokales* Alignment ermöglicht. Das Ziel des globalen Alignments besteht darin, zwei Sequenzen über ihre gesamte Länge

[4] American Standard Code for Information Interchange
[5] base pairs

aneinander anzugleichen. Das lokale Alignment hingegen bestimmt die Ähnlichkeit zweier Sequenzen nur über einzelne Teilsequenzen. Nur die Bereiche der größten Sequenzähnlichkeit werden näher untersucht. Beide Algorithmen liefern immer ein exaktes Alignment für die Position mit dem höchstmöglichen Score.

Abbildung 2.2 zeigt den Unterschied zwischen globalem und lokalem Alignment. Beide Sequenzen enthalten identische Bereiche, die aber aufgrund ihrer Verschiebung zueinander beim globalen Alignment nicht berücksichtigt werden. Das lokale Alignment erkennt die beiden Teilsequenzen und richtet nur diese Teile mit der größten Ähnlichkeit zueinander exakt aus. Ein solches optimales lokales Alignment wird als *Sub-Alignment* bezeichnet. Die Suche nach Sequenzen in einer Datenbank entspricht somit dem Problem des lokalen Alignments. [Han06]

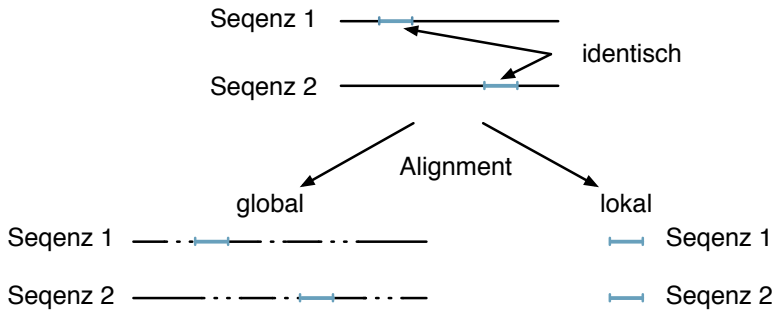


Abbildung 2.2: Unterschied zwischen lokalem und globalem Alignment. Nach [Han06].

Die exakten Algorithmen basieren auf dem System der dynamischen Programmierung. Das Optimierungsproblem wird in viele gleichartige Teilprobleme zerlegt, wobei sich die optimale Lösung aus den optimalen Lösungen der Teilprobleme ergibt. Beim Alignment ist ein Teilproblem ein optimales Alignment zweier Teilsequenzen, welches Schritt für Schritt um weitere Nukleotide erweitert wird. [Böc03]

2.2.1.2 Globales Alignment

Den ersten Algorithmus zur Berechnung eines globalen Alignments zweier Nukleotidsequenzen entwickelten Needleman und Wunsch [NW70] im Jahr 1970. Ausgangspunkt ist eine zweidimensionale Matrix der beiden Sequenzen. Die Felder der Matrix werden Schritt für Schritt in Abhängigkeit der aktuellen Position in den beiden Sequenzen berechnet. Ein optimales Alignment ist dabei der Pfad durch die Matrix, der den höchsten Score erreicht. Der Pfad berechnet sich durch die Summe der Matches unter Berücksichtigung der Gaps in Anlehnung an die δ-Funktion aus den Gleichungen 2.2 bis 2.4. [NW70, Han06]

Für zwei Sequenzen s und t mit einer Länge von m beziehungsweise n wird schrittweise eine Ähnlichkeitsmatrix der Größe $(m+1) \times (n+1)$ aufgebaut. Die zusätzliche Spalte und Zeile enthalten das leere Wort. Dieses leere Wort entspricht einem Alignment, welches nur Gaps enthält. Dementsprechend werden die erste Spalte und die erste Zeile mit einem Vielfachen der Gapkosten g initialisiert.

Die weiteren Felder der Matrix werden rekursiv nach Gleichung 2.5 zeilen- oder spaltenweise berechnet. Ein Schritt nach rechts entspricht einer Insertion in s und ein Schritt nach unten einer Deletion in t. Ein diagonaler Schritt entspricht nach der δ-Funktion aus Gleichung 2.2 einem Mismatch oder Match. In das aktuelle Feld wird immer der Wert übernommen, der den höchsten Score erzeugt.

$$M(i,j) = max \begin{cases} \underbrace{M(i-1,j)+g}_{\text{Insertion}}, \\ \underbrace{M(i,j-1)+g}_{\text{Deletion}}, \\ \underbrace{M(i-1,j-1)+\delta(s_i,t_j)}_{\text{Match / Mismatch}} \end{cases} \tag{2.5}$$

Das eigentliche Alignment entsteht, indem in einem nächsten Schritt, der *Pfadrekonstruktion*, der Weg von Feld (m,n) bis zum Feld $(0,0)$ zurückverfolgt wird. Anfangs- und Endpunkt sind durch das globale Alignment fest vorgegeben. Grundlage ist dabei ebenfalls Gleichung 2.5.

Der Nachteil des Needleman & Wunsch-Algorithmus besteht in seiner hohen Zeitkomplexität von $O(m \cdot n)$. Die Komplexität zur Bestimmung des optimalen Pfades liegt bei $O(m+n)$. Ein weiteres Problem liegt in der hohen Speicherkomplexität für die Matrix welche ebenfalls $O(m \cdot n)$ beträgt.

2.2.1.3 Lokales Alignment

Der Algorithmus von Smith und Waterman [SW81] wurde im Jahr 1981 vorgestellt und ist im wesentliche eine Erweiterung des Needleman & Wunsch-Algorithmus, die ein lokales Alignment ermöglicht. Im Gegensatz zum Needleman & Wunsch-Algorithmus wird keine vollständige Diagonale durch die gesamte $(m+1) \times (n+1)$ Matrix gesucht. Der Weg kann an einer beliebigen Stelle beginnen und enden. Ziel ist es, für eine Teilsequenz einen möglichst hohen Score zu erreichen. [SW81, Böc03]

Die Ränder der Matrix werden beim Smith & Waterman-Algorithmus mit Null und nicht mit dem vielfachen der Gapkosten initialisiert. Dadurch kann das eigentliche Alignment an einer beliebigen Stelle im Text beginnen, ohne das zuvor Gaps eingefügt werden müssen. Die rekursive Berechnung der Matrixelemente erfolgt nach Gleichung 2.6. Der Unterschied zu Gleichung 2.5 besteht lediglich in der letzten Zeile, die verhindert, dass negative Scores in die Matrix eingefügt werden. Ein Weg kann somit an einer beliebigen Stelle, unabhängig von seinem vorherigen Verlauf, mit einem Match neu starten und eine Teilsequenz kann unabhängig von der Gesamtsequenz einen hohen Score erzielen.

$$M(i,j) = max \begin{cases} \underbrace{M(i-1,j)+g}_{\text{Insertion}}, \\ \underbrace{M(i,j-1)+g}_{\text{Deletion}}, \\ \underbrace{M(i-1,j-1)+\delta(s_i,t_j)}_{\text{Match / Mismatch}}, \\ 0 \end{cases} \tag{2.6}$$

In der Pfadrekonstruktion wird nicht vom Feld (m, n) ausgegangen, sondern vom Eintrag mit dem höchsten Score in der Matrix. Um den Weg von diesem Feld bis zum Beginn des Alignments, also einem Feld mit einem Score von Null, zu verfolgen, wird zusätzlich zum Score in jedem Feld gespeichert, welcher der vier Fälle in Gleichung 2.6 das Maximum ergeben hat.

Abbildung 2.3 zeigt ein Alignment für die beiden Sequenzen s und t. Die Pfeile kennzeichnen welches der benachbarten Felder das Maximum lieferte. Der höchste Score hat im Beispiel den Wert Vier und liegt in Feld $(6, 10)$.

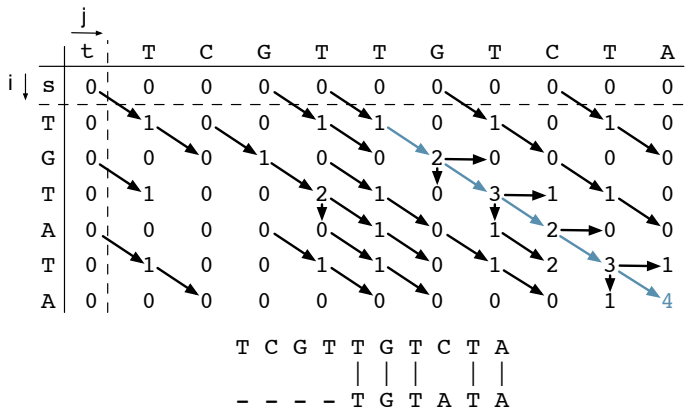


Abbildung 2.3: Smith & Waterman-Algorithmus am Beispiel zweier Nukleotidsequenzen mit einem Mismatch.

Die Zeit- und Speicherkomplexität des Smith & Waterman-Algorithmus liegt mit $O(m \cdot n)$ so hoch wie die des Needleman & Wunsch-Algorithmus und ist daher ebenfalls ein großes Problem. [SW81, Böc03]

Wird ein exaktes Alignment zweier Teilsequenzen benötigt oder liefern die im folgenden beschriebenen heuristischen Verfahren keine Lösung, wird immer noch auf den Smith & Waterman-Algorithmus zurückgegriffen. Das Programm *SSEARCH* des FASTA-Paketes (siehe Abschnitt 2.2.2.1) ist die am weitesten verbreitete Implementierung des Smith & Waterman-Algorithmus. Das Programm wurde über Jahre optimiert und dient in dieser Arbeit bei Vergleichsmessungen als Referenzprogramm für einen exakten Algorithmus.

2.2.2 Heuristische Verfahren

Aufgrund der hohen Zeit- und Speicherkomplexität der exakten Verfahren aus Abschnitt 2.2.1 sind heuristische Verfahren zur Suche langer Sequenzen von üblicherweise 1.000 Basenpaaren in Datenbanken notwendig, um in akzeptabler Zeit eine Lösung zu finden. Diese Verfahren arbeiten wesentlich schneller, wobei die Erhöhung der Geschwindigkeit zu Einbußen in der Qualität der Ergebnisse führt. Heuristische Verfahren liefern somit nicht immer das optimale Ergebnis.

Die beiden wichtigsten Ansätze zur schnellen Suche einer Sequenz in einer Datenbank bieten die beiden Programme FASTA[6] und BLAST[7], die in den beiden folgenden Abschnitten näher betrachtet werden.

2.2.2.1 FASTA

Mit der aufkommenden Suche von langen Proteinsequenzen in großen Datenbanken wurde im Jahr 1985 mit *FASTP* einer der ersten heuristischen Alignment Algorithmen für die Datenbanksuche entwickelt. Das Programm FASTA [PL88] selbst ist eine Erweiterung auf Nukleotidsequenzen aus dem Jahr 1988 und arbeitet nach dem selben System. Mittlerweile gibt es eine Vielzahl von unterschiedlichen Varianten und Ablegern, deren Vorgehensweise sich aber immer noch an den grundlegenden vier Schritten orientiert. Dieser grundsätzliche Ablauf wird im folgenden näher betrachtet. [Böc03, Han06, PL88]

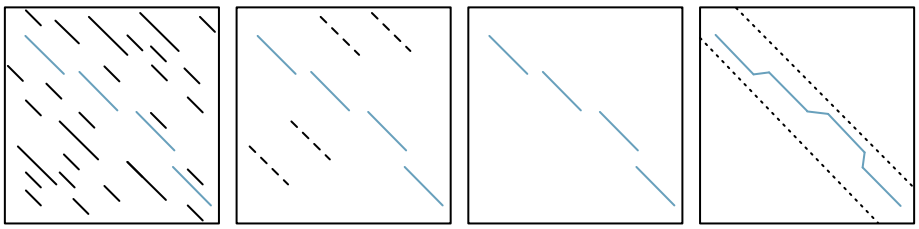


Abbildung 2.4: Die vier Schritte des FASTA-Algorithmus. Nach [Han06].

1. **Finden identischer Positionen in der Datenbank - Indexsuche**
   Der erste Schritt des FASTA-Algorithmus besteht in einer Indexsuche. Dazu wird von jeder Suchsequenz ein Index erstellt. Dabei wird die Sequenz in alle möglichen Teilsequenzen der Länge k zerlegt, die mit ihren Positionen inerhalb der Sequenz gespeichert werden. Mit diesem Index wird die Datenbank nach exakten Treffern der Länge k durchsucht. Ein exakter Treffer wird dabei als *Hot-Spot* bezeichnet. Aufgrund der kurzen Teilsequenzen, die üblicherweise eine Länge von sechs Basenpaaren haben, ist eine schnelle Suche möglich.

[6] Algorithmus zur Suche in Datenbanken - Abkürzung für *fast all*
[7] Basic Local Alignment Search Tool

Abbildung 2.4(a) zeigt beispielhaft wie eine Datenbank nach passenden Hot-Spots gefiltert wird.

2. **Berechnung des Scores - Ungapped Alignment**
   Im zweiten Schritt wird versucht, die Hot-Spots schrittweise ohne das Einfügen von Gaps auszudehnen. Der Score der einzelnen Teilsequenzen wird als *Init1-Score* bezeichnet. Für die weiteren Schritte werden lediglich diejenigen Ergebnisse berücksichtigt, deren Init-1 Score über einem bestimmten Schwellwert liegt. Abbildung 2.4(b) und 2.4(c) zeigen die Sequenzen, deren Score hoch genug ist. Da keine Gaps eingefügt wurden, wird dieser Teilschritt als *Ungapped Alignment* bezeichnet.

3. **Verknüpfen der Treffet - Gapped Alignment**
   Der dritte Teilschritt versucht, durch das Einfügen von Gaps die Teilsequenzen aus dem vorherigen Schritt zu verknüpfen. Der sich daraus ergebende Score wird als *Initn-Score* bezeichnet. Liegen die entstanden Pfade über einem bestimmten Score, werden sie im nächsten Schritt weiter bearbeitet. Abbildung 2.4(b) zeigt einen durch Verknüpfen entstanden Pfad.

4. **Alternative Lösung**
   Im letzen Schritt wird für die gefundene Sequenz ein exaktes Alignment mit Hilfe des Smith & Waterman-Algorithmus (*SSEARCH*) durchgeführt. Berücksichtigt wird dabei nur ein schmaler Streifen um die gefundene Sequenz, um den Berechnungsaufwand gering zu halten.

Die Heuristik liegt beim FASTA-Algorithmus in der Auswahl eines Bereiches, der in einem späteren exakten Alignment näher betrachtet wird. So kann für lange Sequenzen ein hoher Geschwindigkeitsgewinn gegenüber den zuvor betrachteten exakten Verfahren erreicht werden.

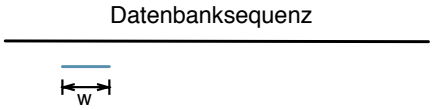
Das Problem besteht darin, dass in der ersten Phase nur Treffer der Länge k gesucht werden, wodurch auch Stellen der Datenbank herausgefiltert werden, die eine hohe Ähnlichkeit mit der zu suchenden Sequenz haben, aber in den folgenden Schritten des Algorithmus nicht weiter betrachtet werden. Über den Parameter k kann somit in hohem Maße Einfluss auf Geschwindigkeit und Genauigkeit des Algorithmus genommen werden.

2.2.2.2 BLAST

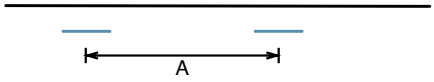
Ein anderer weit verbreiteter Algorithmus zur Datenbanksuche ist der BLAST-Algorithmus [AGM$^+$90] der 1990 am NCBI[8] entwickelt wurde. Das Ziel der Entwicklung bestand darin, ausgehend vom FASTA-Algorithmus die Suche weiter zu beschleunigen. Der Algorithmus teilt sich in drei Schritte, die im folgenden näher betrachtet werden. [AGM$^+$90, Han06]

1. **Finden eines ersten Hits**
   Im ersten Schritt wird, wie auch beim FASTA-Algorithmus, ein Index erstellt. Die zu suchende Sequenz wird in alle möglichen Teilsequenzen der Länge w (*word size*) zerlegt.

[8] National Center for Biotechnology Information



(a) Erster Hit der Länge w.



(b) Zweiter Hit der Länge w mit maximalen Abstand A.

(c) Ausdehnung der Hits.

Abbildung 2.5: Schritte des BLAST-Algorithmus. Nach [Han06].

Mit diesen Index wird in der Datenbank für Proteine nach ähnlichen und für Nukleotide nach exakten Treffern (*Hits*) gesucht. Abbildung 2.5(a) zeigt einen solchen Hit.

2. **Suchen nach einem zweiten Hit**
   Im folgenden Schritt wird versucht, in der Nähe eines Hits einen zweiten Hit auf der gleichen Diagonalen zu finden, damit möglichst wenige Gaps zwischen ihnen liegen. Abbildung 2.5(b) zeigt zwei Hits innerhalb des maximal zugelassenen Abstandes A, die im folgenden Schritt weiter berücksichtigt werden. Alle Hits mit einem größeren Abstand werden verworfen.

3. **Ausdehnen der Hits**
   Im letzen Schritt wird versucht, die Hits mit dem Abstand A in beide Richtungen auszudehnen, wie Abbildung 2.5(c) zeigt. Bei der Ausdehnung sind sowohl Mismatches als auch Gaps zugelassen und sie wird fortgesetzt, bis sich der Score nicht weiter erhöhen lässt. Liegt der erreichte Score über einem bestimmten Schwellwert S, werden die Hits als HSPs[9] bezeichnet und bilden die Ausgabe des BLAST-Algorithmus.

Da der BLAST-Algorithmus lediglich Hits weiter berücksichtigt, die den Abstand A zueinander haben, wird eine deutlich höhere Geschwindigkeit als mit dem FASTA-Algorithmus erreicht. Das Ausdehnen der Hits im letzten Schritt ist in Bezug auf die Laufzeit ebenfalls effektiver, als der Smith & Waterman-Algorithmus des FASTA-Verfahrens. Im Vergleich der Qualität des Alignments liefert FASTA allerdings bessere Ergebnisse. [Han06]

[9] High Scoring Pairs

Für die schnelle Suche langer Sequenzen in großen Datenbanken ist BLAST der am weitesten verbreitete Algorithmus. Es gibt verschiedene Weiterentwicklungen, wie beispielsweise BLAT[10] [Ken02] oder auch MPI-BLAST [DCF03] für Computer-Cluster mit einer Vielzahl von Prozessoren, um eine schnelle Suche in den immer größer werdenden Datenbanken zu gewährleisten. Um auch Nutzern ohne großen Computer-Cluster eine schnelle Suche zu ermöglichen, bieten viele große Forschungseinrichtungen ein Webformular zur Suche mittels BLAST in den eigenen Datenbanken an.

2.3 Short-Read Mapping Problem

In den vergangenen Jahrzehnten war das Sanger-Verfahren [SNC77] zur Sequenzierung gebräuchlich. Mit diesem ist es möglich, DNS-Stränge von bis zu 800 Nukleotiden zu verarbeiten und es wird ein Durchsatz von 80 Nukleotiden in der Stunde erreicht. Die bisher betrachteten Algorithmen wurden speziell für den Anwendungsfall einer Suche weniger, aber dafür sehr langer Sequenzen entwickelt. [HS04]

In den letzten Jahren haben große Projekte, wie das *1.000 Genomes Project* [Kai08], die Sequenzierung von Tumorzellen um unbekannte Krebsarten zu erforschen [MDD+09] oder das ChIP-Seq Verfahren [JMMW07], welches in der Molekularbiologie eingesetzt wird, zu großen technologischen Weiterentwicklungen geführt.

Mit neuen, so genannten *Next-Generation Sequencing* (auch bekannt als *Massively Parallel Sequencing*) Technologien, die eine automatisierte, massiv parallele Sequenzierung erlauben, ist es möglich, mehrere Millionen Basenpaare in wenigen Stunden zu sequenzieren. Der Durchsatz des Sanger-Verfahrens kann so um das Hundertfache erhöht und die Kosten dramatisch gesenkt werden.[11] Durch die sinkenden Kosten wird die Sequenzierung für viele kleinere Forschungseinrichtungen ermöglicht.

Tabelle 2.2 gibt einen Überblick über die aktuellen Sequenzierungstechniken und deren Durchsatz. Die sequenzierten kurzen DNS-Abschnitte werden als *Reads* bezeichnet. Durch das automatisierte Verfahren sind diese Reads mit einer Länge von 50 - 175 bp im Vergleich zu den durch das Sanger-Verfahren gewonnenen Sequenzen mit ihren 800 Basenpaaren relativ kurz. Durch massive Parallelisierung steigt aber die Anzahl dieser kurzen Reads, die pro Durchlauf gewonnen werden, auf bis zu 800.000.000 an. [RF09, MEA+05]

Die Problemstellung des Abgleichs mehrerer Millionen kurzer Reads mit einem Referenzgenom wird als *Short-Read Mapping Problem* bezeichnet. Die traditionellen Programme wie BLAST sind weder für derartig viele Suchanfragen ausgelegt, noch sind sie dazu in der Lage auf die kurzen Sequenzen herunterzuskalieren. [Mar08]

[10] **B**LAST-**L**ike **A**lignment **T**ool

[11] Das *Human Genome Project* dauerte 13 Jahre und kostete 3 Milliarden US$. Die neuen Sequenzierungstechniken ermöglichen eine komplette Sequenzierung des menschlichen Genomes innerhalb eines Durchlaufes von wenigen Tagen und die Kosten liegen zwischen 15.000 bis 20.000 US$.

Tabelle 2.2: Überblick über aktuelle Next-Generation Sequenzierungsgeräte. Nach [RF09].

Methode	Readlänge	Reads pro Durchlauf	Durchlaufzeit
ABI SOLiD	~ 50	85.000.000	6 Tage
Helicos Heliscope	30 - 38	800.000.000	8 Tage
Illumina Genome Analyzer	36 - 175	40.000.000	3 - 6 Tage

Programme, die speziell an diese neue Problemstellung angepasst sind, werden als *Short-Read Mapper* bezeichnet. Dabei rückt der bisher übliche Score in den Hintergrund und wird durch *Hamming-* oder *Edit-Distanz* ersetzt. Üblicherweise ist das entscheidende Kriterium für den Match eines Reads die Anzahl der Mismatches. Gaps rücken bei den Short-Read Mappern in den Hintergrund, da das Einfügen von Gaps bei so kurzen Reads zu zufälligen Treffern führt. Ebenso ist die Anzahl der zulässigen Mismatches auf wenige begrenzt[12]. [Mar08, LLKW08]

Das Ergebnis ist nicht unbedingt das Alignment selbst, sondern vielmehr die Positionen, an denen der Read in der Datenbank gefunden wird. Bei einer Resequenzierungsstudie des menschlichen Genoms wurden für Reads mit einer Länge von 36 Basenpaaren durchschnittlich 1.628 Positionen und für 75 Basenpaare 140 Positionen gefunden. Reads, die an vielen Positionen in der Datenbank vorhanden sind, werden im folgenden als High Mapping Reads (HMR) bezeichnet. [HHA+10]

In den folgenden beiden Abschnitten 2.3.1 und 2.3.2 wird auf *Paired-End Reads* und das SAM-Format eingegangen, da viele Short-Read Mapper diese beiden Punkte optional berücksichtigen.

2.3.1 Paired-End Reads

Eine Variante, das Problem der zu kurzen Sequenzen des Next-Generation Sequencing zu umgehen sind *Paired-End Reads* oder PETs[13]. Dabei werden beide Enden der DNS-Sequenz sequenziert. Beide so entstandene Reads werden zusammen mit ihrer Orientierung (Leserichtung) und ihrem Abstand zueinander abgespeichert. [FWLR09, LLKW08]

Damit ist es möglich, die Anzahl der Treffer zu minimieren, indem nur dann ein Hit ausgegeben wird, wenn beide Reads mit dem passenden Abstand zueinander gefunden wurden. Die Sensitivität kann trotz der kurzen Reads stark erhöht werden.

2.3.2 SAM-Ausgabeformat

Jeder der im folgenden Abschnitt vorgestellten Short-Read Mapper hat sein eigenes Ausgabeformat. Um einen einfachen Übergang zu den Nachbearbeitungsschritten zu ermöglichen, wurde das SAM[14]-Format entwickelt, welches von einer Vielzahl der Short-Read Mapper unterstützt wird. [LHW+09]

[12] Bei 36 bp sind zwei Mismatches üblich, bei 50 bp drei, bei 75 bp vier und bei 100 bp sechs Mismatches.
[13] Paired-End Tags
[14] Sequence Alignment/Map

Das SAM-Format ist ein einfaches Textformat, welches aus Header- und Alignment-Teil besteht. Der Header gibt die SAM-Version und Informationen über die Sortierung der Reads an. Der Alignment-Teil enthält den Namen der aktuellen Datenbanksequenz, gefolgt von den einzelnen Reads. Jede Zeile entspricht dabei einer neuen Position. Neben diesen sind weitere optionale Informationen möglich. Tabelle 2.3 listet die möglichen Informationen zu einem Alignment in der Reihenfolge der Spalten auf. Nicht vorhandene Informationen werden mit einem * markiert.

Tabelle 2.3: Felder im SAM Format. [LHW+09]

Spalte	Name	Beschreibung
1	QNAME	Name oder Bezeichnung des Reads
2	FLAG	Zusatzinformationen über den Read
3	RNAME	Name der Referenzsequenz
4	POS	Position des Reads in der Referenzsequenz
5	MAPQ	Mapping Qualität
6	CIGAR	Alignment im CIGAR-Format
7	MRNM	Name des zweiten Paired-End Reads
8	MPOS	Position des zweiten Paired-End Reads
9	ISIZE	Abstand zum zweiten Paired-End Reads
10	SEQ	Read in ASCII-Zeichen
11	QUAL	Qualität des Reads (aus FASTQ-Datei)

Listing 2.2 zeigt eine mögliche Darstellung eines Alignments mit einem Paired-End und einem einfachen Read. Die entsprechende Darstellung im SAM-Format wird in Listing 2.3 aufgezeigt. Die Positionen der Sequenzen werden im SAM-Format startend mit Eins angegeben (*1-based coordinat system*).

Listing 2.2: Einfache Darstellung mehrerer Alignments mit Referenzsequenz. [LHW+09]

```
1 Coor     12345678901234  5678901234567890123456789012345
2 ref      AGCATGTTAGATAA**GATAGCTGTGCTAGTAGGCAGTCAGCGCCAT
3
4 +r001/1        TTAGATAAAGGATA*CTG
5 +r002         aaaAGATAA*GGATA
6 -r001/2                                        CAGCGCCAT
```

Listing 2.3: Reads aus Listing 2.2 im SAM-Format. [LHW+09]

```
1 @HD VN:1.3 SO:coordinate
2
3 @SQ SN:ref LN:45
4 r001 163 ref 7  30 8M2I4M1D3M = 37 39 TTAGATAAAGGATACTG *
5 r002 0   ref 9  30 3S6M1P1I4M * 0   0 AAAAGATAAGGATA    *
6 r001 83  ref 37 30 9M         = 7 -39 CAGCGCCAT         *
```

Das SAMtools Software Paket [LHW+09] erleichtert die Arbeit mit dem SAM-Format und bietet unter anderem die Möglichkeit, bestimmte Informationen zu extrahieren, Positionen zu sortieren und SAM-Dateien in andere Formate umzuwandeln.

2.4 Software Short-Read Mapper

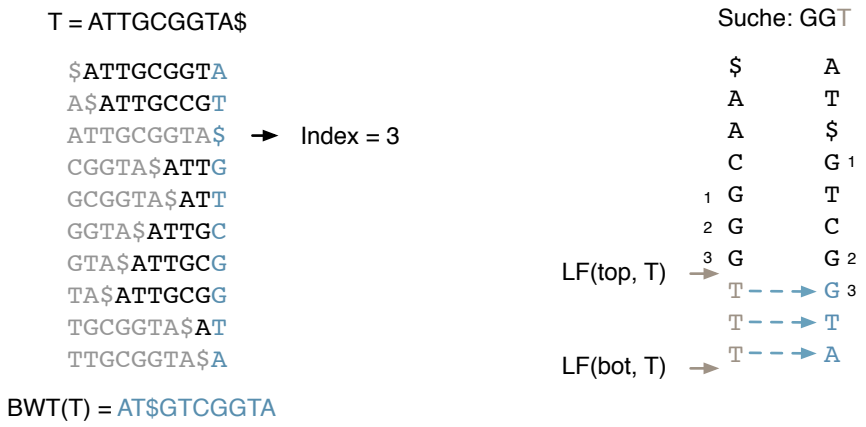
In den folgenden Abschnitten werden einige Software Short-Read Mapper vorgestellt und auf ihre unterschiedlichen Funktionsweisen eingegangen. In Abschnitt 2.4.5 folgt eine Zusammenfassung und ein Vergleich von Laufzeit und Qualität in Form von der Anzahl gefundener Positionen.

2.4.1 Bowtie

Bowtie [LTPS09] wurde an der Universität von Maryland im Jahr 2008 entwickelt und ist einer der am häufigsten eingesetzten Short-Read Mapper. Das Programm nutzt die Burrows-Wheeler Transformation [BW94], um die Datenbank zu komprimieren und den *FM Index*-Algorithmus [FM00], um in der komprimierten Datenbank exakte Matches zu finden.

Der Burrows-Wheeler Algorithmus selbst führt keine Datenkompression durch, sondern ist nur der erste Schritt. In der transformierten Datenbank stehen gleiche Zeichen häufiger hintereinander als in der ursprünglichen Datenbank, was die anschließende Kompression erleichtert und den zu durchsuchenden Bereich verkleinert. [TS09, BW94]

Um die transformierte Datenbank zu erstellen, werden die Zeichen der Datenbank zyklisch immer um eine Position verschoben, bis alle Permutationen aufgelistet sind. Anschließend werden die Permutationen alphabetisch sortiert. Die letzte Spalte ergibt die transformierte Datenbank. Zusätzlich gibt der Index an, in welcher Zeile sich die ursprüngliche erste Zeile befindet. Abbildung 2.6(a) zeigt eine Beispieltransformation.



(a) Burrows-Wheeler Transformation

(b) FM Index-Suche

Abbildung 2.6: Beispiel für eine Transformation der Datenbank und Suche mit Bowtie.

Um in der transformierten Datenbank etwas zu suchen, wird der FM Index-Algorithmus von Ferragina und Manzini eingesetzt. Dabei wird der Read von rechts nach links mit der Datenbank

verglichen. Notwendig für die Suche ist neben der transformierten Datenbank auch die erste Spalte der Permutationen, die aus der Anzahl jedes der vier Zeichen berechnet werden kann.

Abbildung 2.6(b) zeigt eine beispielhafte Suche. Die Funktionen $LF(top, x)$ und $LF(bot, x)$ suchen für ein Zeichen x die entsprechende Zeile in der ersten Spalte. Mögliche Vorgänger sind alle Zeichen zwischen diesen beiden Stellen in der transformierten Datenbank. Ist eines dieser Zeichen auch Vorgänger im Read, so wird der Suchraum durch die beiden Funktionen auf dieses Zeichen gesetzt. Im Beispiel ist der Vorgänger das dritte G in der Transformierten. Der Suchraum wird dementsprechend auf das dritte G gesetzt und somit eingegrenzt.

Durch dieses Vorgehen wird eine sehr schnelle exakte Suche ermöglicht, da es nicht notwendig ist, die gesamte Datenbank zu durchsuchen, sondern lediglich die Teile mit aktuellen Teilsequenzen. Eine inexakte Suche mit Mismatches erfordert Backtracking, da im Fall eines Mismatches ein anderes Zeichen eingesetzt werden muss, um mit der Suche fortzufahren. Da Mismatches und Backtracking zu einer hohen Laufzeitsteigerung führen, sind maximal drei Mismatches und 800 Backtracking Schritte zugelassen. [TS09, FM00]

Um die Position eines gefundenen Reads zu ermitteln, kommt ein Verfahren namens *Checkpointing* zum Einsatz. Dabei wird an bestimmten Checkpoints zusätzlich die Position aus der untransformierten Datenbank gespeichert. Wird ein Read gefunden, wird der Weg bis zum nächsten Zeichen, welches eine Positionsinformation besitzt, zurückverfolgt. [LTPS09]

Neben Bowtie nutzen weitere Short-Read Mapper wie BWA[15] [LD09] die Burrows-Wheeler Transformation und erreichen damit ebenfalls eine hohe Geschwindigkeit. Die fehlende Unterstützung von Gaps stellt allerdings ein Problem dieses Algorithmus dar.

2.4.2 RazerS

Das Programm RazerS [WER+09] der Freien Universität Berlin basiert auf der *Q-Gram Counting* Strategie [OM88]. Gleichung 2.7 zeigt das *Q-Gram* Lemma welches besagt, dass sich zwei Sequenzen t Teilsequenzen teilen, wobei n der Länge der betrachteten Datenbanksequenz, k der Hamming- oder Edit-Distanz und q der Länge der Teilsequenzen entspricht. Darauf aufbauend lässt sich eine effiziente Filterung implementieren. Der Programmablauf lässt sich in drei Schritte unterteilen, die im folgenden erläutert werden. [WER+09]

$$t = n + 1 - (k + 1) \cdot q \tag{2.7}$$

1. **Parameter ermitteln**
   Im ersten Schritt werden die Parameter für die Filterung ermittelt, um Laufzeit und Sensitivität zu optimieren. Dabei werden für kurze Reads vorberechnete Parameter genutzt, die sich an typischen Fehlerprofilen der Sequenzierungsgeräte orientieren.
2. **Filtern**

[15] Burrows-Wheeler Aligner

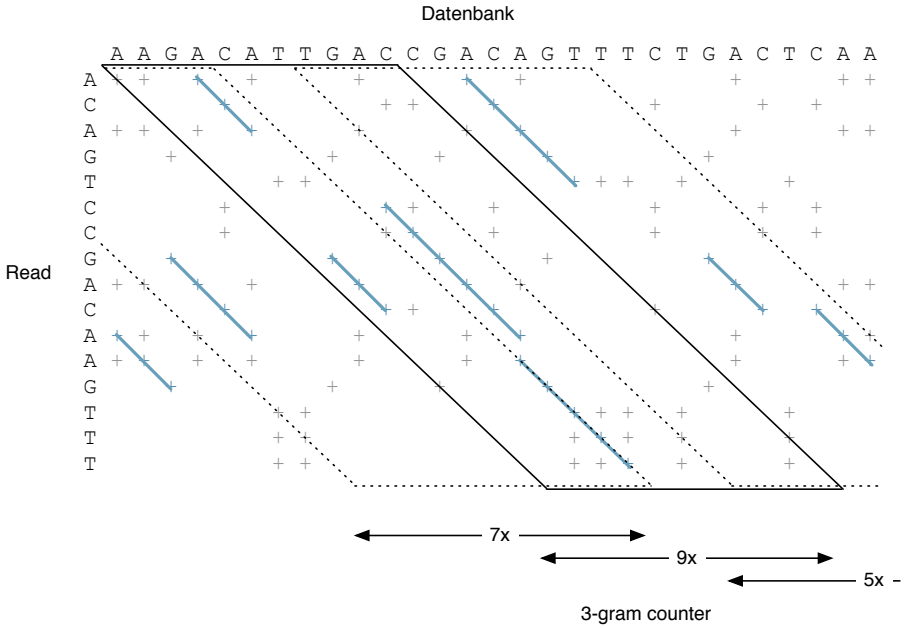


Abbildung 2.7: RazerS Dotplot zwischen einem Read und einem Datenbankausschnitt mit Breite $w = 11$, Überlappung $k = 4$ und Teilsequenzen der Länge $q = 3$. Mit einer Länge von $n = 23$ ergeben sich $t = 8$ erforderliche Teilsequenzen für einen Hit. Nach [Hau09].

Das Herausfiltern der potentiellen Regionen für einen Match erfolgt mit dem *SWIFT*-Algorithmus [RSM06]. Dabei wird davon ausgegangen, dass der Dotplot[16] in Form eines Parallelogramms t Diagonalen der Länge q, so genannte *q-Hits*, enthält. Jedes Parallelogramm der Breite w mit einer Überlappung k und t q-Hits enthält einen potentiellen Treffer. Die Länge des Datenbankausschnittes n wird dabei durch die Gleichung $n = r + w - k$ berechnet, wobei r der Länge des Reads entspricht. Abbildung 2.7 zeigt ein solches Parallelogramm.

Wenn der *Q-Gram Counter* eines Parallelogramms den Wert t erreicht, wird der Treffer verifiziert. Ein Parallelogramm wird mit jedem Schritt immer um $w - k$ Stellen weitergeschoben.

3. **Verifizieren**
   Um zu überprüfen, ob ein Parallelogramm einen richtigen Treffer mit k oder weniger Mismatches enthält, wird der Bit-Vector Algorithmus [Mye99] eingesetzt, um eine hohe Geschwindigkeit durch parallele Abarbeitung von Bit-Operationen zu erreichen.

RazerS ist einer der wenigen Short-Read Mapper, der Gaps zulässt und eine hohe Genauigkeit bei akzeptabler Geschwindigkeit erreicht. Die Anzahl der Positionen, die ausgegeben werden, ist jedoch auf 100 begrenzt.

[16] Ein Dotplot ist die einfachste Form eines Alignments. Hierbei werden ähnliche Abschnitte in Form einer Matrix graphisch aufbereitet, indem jeder Match zwischen beiden Sequenzen durch einen Punkt dargestellt wird.

2.4.3 Maq

Maq [LRD08] verfolgt einen einfachen Ansatz, der als *Spaced Seed Indexing* bezeichnet wird. Die ersten 28 Basenpaare jedes Reads werden dabei in vier gleich lange Segmente unterteilt, die als *Seeds* bezeichnet werden. Typischerweise sind die ersten Basenpaare eines Reads diejenigen, die die wenigsten Mismatches enthalten. Maq geht von dem Ansatz aus, dass in den ersten 28 Basenpaare maximal zwei Mismatches auftreten. Tritt idealerweise kein Mismatch auf, passen alle vier Seeds optimal zum aktuell betrachteten Ausschnitt der Datenbank. [LRD08]

Tritt ein Mismatch auf, passen drei Seeds und bei zwei Mismatches noch zwei. So ergeben sich für zwei zulässige Mismatches sechs Kombinationen der Seeds. Aus diesen Kombinationen wird für jeden Read genau eine Hashtabelle angelegt. Abbildung 2.8 zeigt dieses Vorgehen an einem Beispiel. Die reale Anzahl von Mismatches kann deutlich höher sein, als die festgelegte Anzahl von zwei, da jeder Seed mehrere Mismatches enthalten kann.

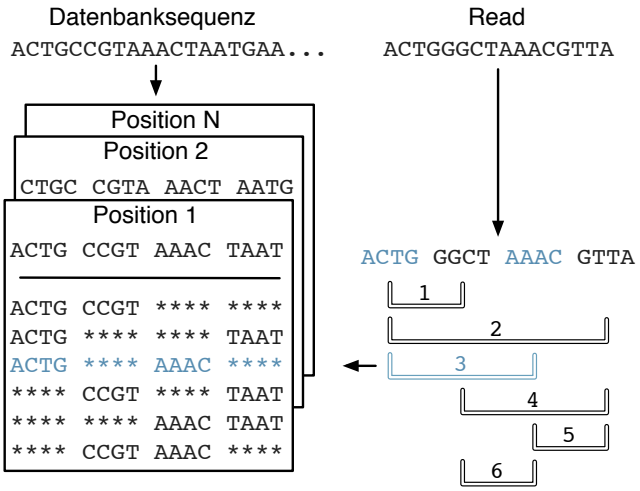


Abbildung 2.8: Beispiel für den Maq-Algorithmus. Nach [TS09].

Für jede Position in der Datenbank mit der Länge von 28 Basenpaaren werden ebenfalls je sechs Hashtabellen erzeugt. Der Speicherplatz des sich damit ergebenden Spaced Seed Index kann schnell um mehrere Größenordnungen im Vergleich zur eigentlichen Datenbank anwachsen. Eine Besonderheit von Maq besteht darin, dass Datenbanken und auch Reads in einem ersten Schritt in ein binäres Format umgewandelt werden, um Speicherplatz einzusparen. [TS09]

Die eigentliche Suche erfolgt, indem der Seed Index der Datenbank mit den Hashtabellen jedes Reads verglichen wird. Passen zwei Seeds eines Reads zueinander, wird die Position als potentieller Treffer markiert. Im folgenden Schritt werden die beiden Seeds, welche die Mismatches enthalten können und die restlichen, bisher nicht betrachteten Basenpaare des Reads mit der Datenbankposition verglichen, um ein vollständiges Alignment zu erhalten.

Indem sich die Suche zu Beginn nur auf die ersten 28 Basenpaare jedes Reads beschränkt sowie durch die schnelle Suche in den Hashtabellen erreicht Maq eine hohe Geschwindigkeit. Ein

Problem besteht darin, dass in jedem der beiden Seeds, die Mismatches enthalten, mehrere Mismatches auftreten können. Dadurch gibt Maq unter Umständen für eine vorgegebene Anzahl an Mismatches Positionen aus, die eine höhere Anzahl an Mismatches enthalten. [TS09]

Der entscheidende Unterschied zu anderen Short-Read Mappern besteht darin, dass Maq nur eine Position zurückgibt, die bei mehreren Treffern zufällig ausgewählt wird.

2.4.4 PASS

PASS [CAB+09] verfolgt einen ähnlichen Ansatz wie Maq und nutzt ebenfalls Seeds und einen Index der Datenbank. Zunächst wird ein Index aus Seeds der Datenbank aufgebaut, wobei ein Seed üblicherweise eine Länge von 12 Basenpaaren besitzt. Des Weiteren gibt es einen Index (PST[17]), der für sämtliche Kombinationen kurzer Seeds von 7-8 Basenpaaren mit dem Needleman und Wunsch-Algorithmus vorberechnete Scores enthält[18]. Die eigentliche Suche der Reads im Index unterteilt sich in drei Schritte. [CAB+09]

1. **Seeds abgleichen**
   Als erstes wird der Datenbank-Index mit den Seeds eines Reads abgeglichen. Wird ein exakter Treffer gefunden, folgt der nächste Schritt. Dieses Markieren einer Position läuft ähnlich ab, wie die Indexsuche der Algorithmen FASTA und BLAST.
2. **Alignment für Ränder**
   In diesen Schritt wird ähnlich wie bei BLAST versucht, die gefundenen Seeds in beide Richtungen um die Länge der Sequenzen im PST weiter auszudehnen. Dabei werden Datenbank- und Read-Seed genutzt, um mit Hilfe des PST einen Score zu liefern, der Mismatches und Gaps berücksichtigt.
3. **Exaktes Alignment**
   Wenn der Score in Schritt zwei hoch genug ist, wird im letzten Schritt, ähnlich wie beim FASTA-Algorithmus, ein exaktes Alignment durchgeführt, um eine hohe Sensitivität des Alignments zu erreichen.

PASS orientiert sich an einigen Punkten, wie beispielsweise dem Auffinden der Bereiche, an BLAST und FASTA, um aus den großen Datenmengen potentielle Treffer herauszufiltern. Mehrere Treffer mit vorgegebenen Abstand wie beim BLAST-Algorithmus sind aufgrund der kurzen Reads nicht sinnvoll, so dass dementsprechend viele Positionen für den zweiten Schritt gefunden werden. Durch den PST wird die Laufzeit hier aber so gering wie möglich gehalten, so dass eine hohe Geschwindigkeit und Sensitivität erreicht wird.

[17] Precomputed Score Table

[18] Für eine Länge von 6 Basenpaaren ergeben sich so beispielsweise 4.096 mögliche Seeds, was wiederum zu 16 Millionen unterschiedlichen Alignments führt und 16 MByte Speicher benötigt. Vorberechnete PSTs mit Seeds von bis zu 8 Basenpaaren sind im Programmpaket enthalten. [CAB+09]

2.4.5 Zusammenfassung

Die vorgestellten Short-Read Mapper geben einen Überblick über die verschieden Herangehensweisen an das Problem und lassen sich in drei Gruppen einteilen:

1. Spaced Seeds Indexing,
2. Burrows-Wheeler Transformation und
3. Q-Gram Counting.

Weit verbreitet sind Spaced Seed Ansätze mit Hashtabellen, wie sie bei den vorgestellten Programmen Maq, PASS aber auch dem Maq sehr ähnlichen Programm SOAP [LLKW08] eingesetzt werden. Die Programme unterscheiden sich stark in Geschwindigkeit und Genauigkeit. Ähnlich wie BLAST reduzieren die meisten Short-Read Mapper die Datenbank auf wenige Positionen, die in einem zweiten Schritt näher betrachtet werden.

Ein anderer weit verbreiteter Ansatz besteht in der Suche in einer Burrows-Wheeler transformierten Datenbank, wie bei Bowtie, SOAP2 [LYL+09] und BWA [LD09]. Diese Verfahren erzielen eine hohe Geschwindigkeit bei exakten Treffern, weisen aber zum Teil eine geringe Sensitivität auf, wodurch sie sich nicht problemlos auf längere Reads übertragen lassen.

Die dritte Herangehensweise ist das *Q-Gram Counting*, welches bei RazerS und auch SHRiMP [RLD+09] eingesetzt wird und dem Spaced Seed Ansatz ähnlich ist. Tabelle 2.4 fasst die wichtigsten Informationen zu den Short-Read Mappern nochmals zusammen.

Tabelle 2.4: Überblick über die wichtigsten Short-Read Mapper.

Programm	Readlänge (bp)	Mismatches	Gaps	Algorithmus
Bowtie	Unbegrenzt	3	Keine	Burrows-Wheeler
RazerS	Unbegrenzt	Beliebig	Beliebig	Q-Gram Counting
Maq	127	2	Keine	Spaced Seeds
SOAP2	1024	2	Ein ausgedehntes	Burrows-Wheeler
PASS	Unbegrenzt	Beliebig	Wenige	Seeds und PST

Alle vorgestellten Short-Read Mapper arbeiten nicht exakt, sondern sind lediglich Heuristiken. Daher ist es notwendig, neben der Laufzeit auch auf die Qualität zu achten, die in der Anzahl der Positionen und den gefundenen Reads besteht. Tabelle 2.5 zeigt Messungen von Laufzeit und die Anzahl der gefundenen Positionen für eine reale Datenbank. Die Short-Read Mapper werden zusätzlich mit dem Smith & Waterman-Algorithmus (*ssearch35*) und BLAST verglichen, um den Geschwindigkeitsgewinn gegenüber den klassischen Programmen zu verdeutlichen. In Abbildung 2.9 ist der Geschwindigkeitsgewinn in Bezug auf den Smith & Waterman-Algorithmus und die Anzahl der gefundenen Positionen für eine hohe Anzahl an Reads dargestellt.

Der Smith & Waterman-Algorithmus und BLAST finden beide deutlich mehr Positionen als die Short-Read Mapper, da sie auch kürze Teilsequenzen eines Reads als zusätzliches Alignment ausgeben. Ihre Laufzeit liegt aber deutlich höher als die der Short-Read Mapper, von denen

Tabelle 2.5: Vergleichsmessungen der Laufzeiten für die Transformation der Datenbank und die Suche verschiedener Alignment Programme und Short-Read Mapper.

Programm	Transformation (s)[a]	Suche (s)[a,b]	Reads mapped	Positionen
SWA (ssearch35)	—	1.390.00	100.000	> 2.000.000
BLAST	6	7.213	100.000	> 2.000.000
Maq	7	660	76.874	76.874
PASS	—	272	99.766	455.862
RazerS	—	248	100.000	552.573
SOAP2	255	21	76.873	163.913
Bowtie	463	223	76.776	1.188.046

[a]Testsystem: Intel Core 2 Duo 2,66 GHz, 64 Bit, 4 GByte RAM
[b]Datenbank: Homo sapiens chromosome 13 (59.606.055 bp)
Reads: 100.000 mit 50 bp und 0-3 Mismatches in diskreter Gleichverteilung

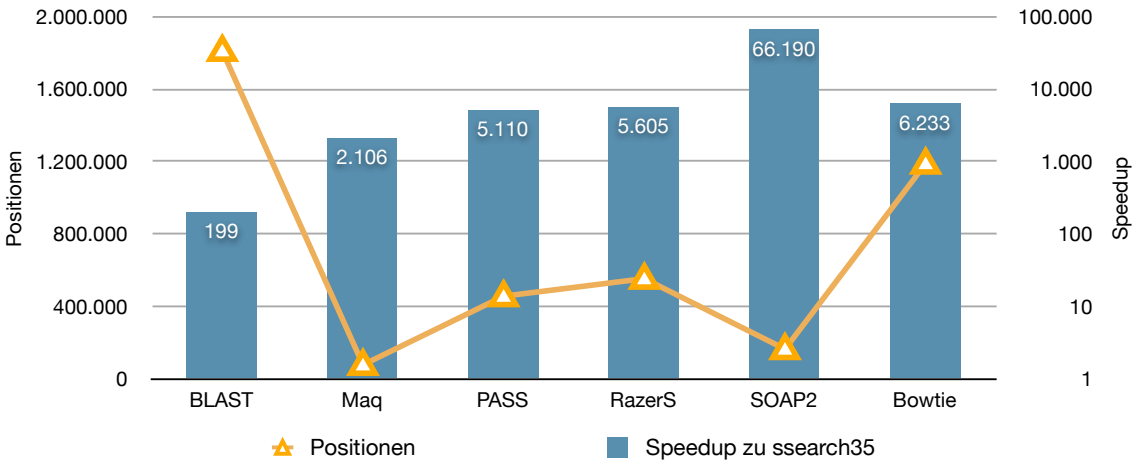


Abbildung 2.9: Vergleich zwischen Speedup und Anzahl der gefundenen Positionen unterschiedlicher Short-Read Mapper. Daten aus Tabelle 2.5.

Maq der langsamste ist und pro Read nur die beste Position ausgibt. PASS, RazerS und Bowtie unterscheiden sich lediglich in der Anzahl der gefunden Positionen, wobei Bowtie deutlich vorne liegt. Bei RazerS ist die Anzahl der gefundenen Positionen pro Read fest auf 100 beschränkt. Der mit Abstand schnellste Short-Read Mapper ist SOAP2, er findet aber auch weniger Positionen als die anderen Short-Read Mapper mit ausßnahme von Maq. Der Anteil der in der Datenbank gefundenen Reads ist bei PASS und RazerS mit 100 % am höchsten, da diese Programme eine ausreichende Zahl von Mismatches tolerieren.

2.5 Parallele Architekturen

Durch die in Abschnitt 2.3 angesprochenen geringen Kosten und die dadurch einfache Verfügbarkeit der neuen Sequenzierungstechniken ist es von großer Bedeutung, eine parallele Plattform zu

wählen die ebenfalls kostengünstig ist. Die in dieser Arbeit betrachteten parallelen Architekturen beschränken sich demnach auf FPGA und GPU. Auf die Übertragung auf Mehrprozessorsysteme beziehungsweise große Computer-Cluster wird demnach nicht weiter eingegangen. Da Multicorearchitekturen von den Entwicklern der meisten Short-Read Mapper bereits näher betrachtet wurden, wird auch darauf verzichtet.

Im Folgenden werden die Besonderheiten der beiden parallelen Architekturen FPGA in Abschnitt 2.5.1 und GPU in Abschnitt 2.5.2 näher betrachtet, um anschließend im Kapitel 4 eine Auswahl treffen zu können.

2.5.1 Field Programmable Gate Array

Ein FPGA ist ein programmierbarer integrierter Schaltkreis mit einer feingranularen Architektur. Die internen Funktionselemente können beliebig programmiert und verbunden werden, sodass von einfachen logischen Schaltungen, über komplexe Algorithmen, bis hin zu vollständigen Multicoreprozessoren anwendungsspezifische integrierte Schaltungen direkt in Hardware realisiert werden können. Die Beschreibung einer logischen Schaltung erfolgt mit den Hardwarebeschreibungssprachen VHDL[20] oder Verilog. [Flo97]

Abbildung 2.10 zeigt den internen Aufbau eines FPGAs. Ein CLB[21] ist das Element, in dem die eigentliche Funktion durch LUTs[22], Register, Multiplexer und Carry-Chains realisiert wird. Eine LUT ist ein SRAM[23], der durch das Programmieren des FPGAs beschrieben wird und zur Laufzeit gelesen werden kann. So lassen sich innerhalb einer LUT beliebige boolesche Funktionen realisieren. Des Weiteren besteht die Möglichkeit, eine LUT als *Distributed RAM* oder adressierbares Schieberegister zu verwenden. [Xil09, Xil10b, Xil10c]

Die einzelnen CLBs können über programmierbare Verbindungen mit Signalleitungen in den angrenzenden Kanälen verbunden werden. An Kreuzungspunkten dienen die PSM[24]s als programmierbare Verbindungsknoten. Die Kommunikation mit der Außenwelt erfolgt schließlich über die IOBs[25]. Neben den bisher betrachteten frei programmierbaren Elementen existieren weitere spezielle Bausteine, wie der schnelle interne *Block RAM* und Multiplizierer. Tabelle 2.6 zeigt eine Auflistung der relevaten Ressourcen eines Virtex-5 und eines Virtex-6. [Flo97]

Der Vorteil des FPGAs liegt darin, dass die Hardware direkt an das individuelle Problem angepasst werden kann. Trotz der geringen Taktrate von 100 - 200 MHz kann durch die schnellen internen Verbindungsleitungen und die hohe erreichbare Parallelität bei einfachen Algorithmen eine hohe Geschwindigkeit erreicht werden. Ein weiterer Vorteil des FPGAs besteht in seinem niedrigen Energieverbrauch.

[20] Very High Speed Integrated Circuit Hardware Description Language
[21] Configurable Logic Block
[22] Look Up Tables
[23] Static Random Access Memory
[24] Programmable Switch Matrix
[25] Input/Output Blocks

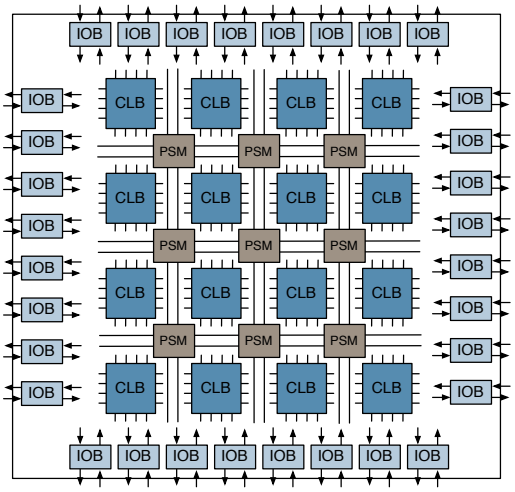


Abbildung 2.10: Prinzipieller Aufbau eines FPGAs.

Tabelle 2.6: Kennzahlen Virtex-5 [Xil09] und Virtex-6 [Xil10b].

FPGA	Virtex-5 XC5VLX50T	Virtex-6 XC6VLX240T
Slices	7.200	37.680
Register	28.800	301.440
LUTs	28.800	150.720
Block RAM (KBits)	2.160	14.976

2.5.2 Graphics Processing Unit

GPUs sind spezielle Prozessoren, die zur Unterstützung des Hauptprozessors bei Grafikberechnungen dienen. Eine GPU bestand ursprünglich aus mehreren einfachen Grafikpipelines, aufgebaut aus Pixel- und Virtexshader. Um die Auslastung zu optimieren wurden die beiden Shader in einem *Unified Shader* zusammengefasst, der beide Funktionen abdecken kann und sich zu einer fast vollständigen CPU entwickelt hat. Durch diese Veränderung der Grafikpipeline, die hohe Leistung durch die massive Parallelität und die hohe Speicherbandbreite im Vergleich zu einem einfachen Prozessor, werden GPUs aber auch zunehmend für das wissenschaftliche Rechnen interessant. Wird die GPU für Berechnungen genutzt, die nicht der direkten Darstellung eines Bildes dienen, spricht man von GPGPU[26]. [HK09]

2.5.2.1 Aufbau der Hardware

Nvidia bezeichnet seine Unified Shader als Stream- oder Threadprozessoren. Abbildung 2.11 zeigt eine Nvidia Tesla GPU mit 128 Streamprozessoren verteilt auf 8 Multiprozessoren. Jeder Threadprozessor kann bis zu 96 Threads verwalten. In Anlehnung an die Klassifikation nach Flynn wird von *Single Instruction Multiple Thread* gesprochen. Jeder Thread verfügt über einen

[26] General Purpose Graphics Processing Unit

eigenen Befehlszähler und eigene Register. Das Programm ist für alle gleich, nur Daten und Ausführungszeitpunkt sind für jeden Thread unterschiedlich. Daher ist es in sinnvoller von *Single Programm Multiple Data* zu sprechen. [HK09]

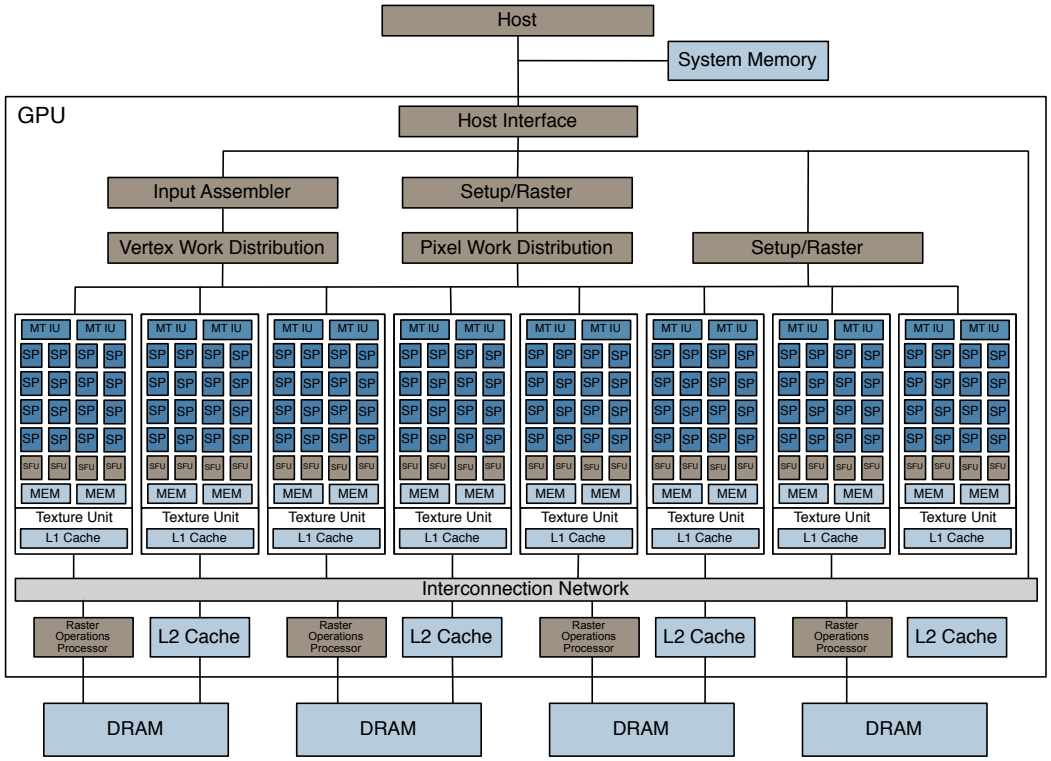


Abbildung 2.11: Aufbau einer Nvidia Tesla mit 128 Streamprozessoren. Nach [HK09].

Als Speicher dient der externe Grafikspeicher, auf den alle Streamprozessoren zugreifen können. Die Streamprozessoren innerhalb eines Multiprozessors erlauben den Zugriff auf einen gemeinsamen internen Speicher. Textur- und Konstantenspeicher können ebenfalls für Berechnungen genutzt werden. Die Vorteile einer GPU liegen in der hohen Speicherbandbreite von bis zu 144 GByte/s und der hohen Floating-Point Leistung von bis zu 515 GigaFLOPs[27] bei einer Nvidia Tesla der neusten Generation. Der Nachteil der GPU besteht in dem hohen Energieverbrauch, der zwischen 200 und 250 Watt liegt. [Hal08]

2.5.2.2 Programmiermodell

Die ursprünglichen GPGPU Programmiermodelle erforderten es, Berechnungen in Grafikprobleme zu überführen und erlaubten keinen Zugriff auf die internen Speicher. Dadurch arbeiten alle Threads auf dem externen Grafikspeicher, was zu einer Begrenzung der Leistung führt. Um diese Probleme zu lösen und um die Übertragung wissenschaftlicher Programme zu erleichtern, wurde in den letzten Jahren von Nvidia das CUDA[28] Programmiermodell für Nvidia GPUs und von

[27] Floating Point Operations per second
[28] Compute Unified Device Architecture

ATI das FireStream Modell entwickelt. OpenCL[29] ist ein vom Hersteller unabhängiges Modell, welches sich an CUDA orientiert. [NBGS08, SK10]

Da CUDA zum gegenwärtigen Zeitpunkt eine höhere Leistung erreicht als vergleichbare Programmiermodelle, stellt es die beste Wahl zur Entwicklung einer Anwendung dar. Eine spätere Portierung auf OpenCL sollte aufgrund des selben Programmiermodelles ebenfalls leicht möglich sein. Abbildung 2.12 gibt einen Überblick über die CUDA-API[30]. Der CUDA-Compiler erzeugt aus einem C-Quelltext mit CUDA-Bibliotheksaufrufen Code, der auf dem Host-System und einen *Kernel* der auf der GPU ausgeführt wird. Die eigentliche Grafikhardware wird hier vollständig verdeckt, so dass ein CUDA-Programm auf unterschiedlichen Nvidia-Architekturen, unabhängig von der realen Anzahl der Thread Prozessoren, abgearbeitet werden kann. [NBGS08, HK09]

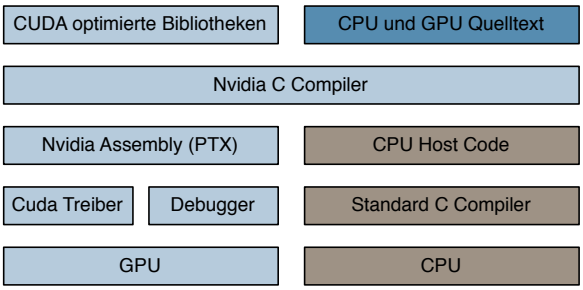


Abbildung 2.12: Aufbau der CUDA API. Nach [HK09].

Der Ablauf eines CUDA-Programmes unterteilt sich immer in drei Schritte. Zu Beginn müssen die Daten vom Host auf den externen Speicher der GPU übertragen werden. Anschließend wird der Kernel, das Programm welches auf der GPU ausgeführt wird, gestartet. Beim Start wird dem Kernel die Anzahl von *Blöcken* und *Threads* mitgegeben, die innerhalb eines *Grids* ausgeführt werden.

Ein Grid fasst alle Threads eines Kernelaufrufes zusammen. Ein Grid kann aus mehreren Blöcken bestehen, welche die Multiprozessoren repräsentieren. Auf einem Multiprozessor werden dementsprechend mehrere Threads ausgeführt, die sich einen lokalen Speicher, den *Shared Memory*, teilen können. Ist die Zahl der Blöcke größer als die der Multiprozessoren, werden Sie nacheinander ausgeführt. Ein Thread kann seine Position im Block und im Grid eindeutig ermitteln.

Durch das CUDA-Programmiermodell wird die Übertragung eines Algorithmus auf die GPU wesentlich vereinfacht. Das Problem besteht darin, die Daten so auf die unterschiedlichen Speicherhierarchien der GPU aufzuteilen, dass die Zugriffe möglichst effizient sind und eine hohe Leistung erzielt wird.

[29] **Open C**omputing **L**anguage
[30] Application Programming Interface

2.6 Alignment auf parallelen Architekturen - Stand der Technik

Dieser Abschnitt gibt einen kurzen Überblick über bisherige Entwicklungen und Versuche, das Alignment-Problem auf die in Abschnitt 2.5 vorgestellten parallelen Architekturen zu übertragen. Eine ausführliche Analyse der möglichen Parallelisierungen der einzelnen Algorithmen und Verfahren folgt im Kapitel 3.

2.6.1 Alignmentalgorithmen allgemein

Aufgrund seiner einfachen Struktur existieren für den Smith & Waterman-Algorithmus eine Vielzahl von GPU- und FPGA-Implementierungen. Bislang wurde aber hauptsächlich die Problemstellung des Alignments zweier langer Sequenzen zueinander untersucht. Der erreichbare Speedup auf der GPU liegt für lange Sequenzen bei 45 [KSPBP10] gegenüber einer einfachen Softwarelösung. Wird mit kurzen Sequenzen gearbeitet, wird lediglich ein Speedup von 2 bis 10 [MV08] erreicht.

Die GPU-Programme werden häufig mit eigenen Smith & Waterman Software Implementierungen verglichen, was einen Vergleich erschwert, da die GPU-Programme nicht immer zugänglich sind und häufig nur der eigentliche Algorithmus und nicht das Einlesen der Daten und die Ausgabe implementiert wurde. Der GPU Smith & Waterman-Algorithmus des LLNL[31] [LHJV06] erreicht verglichen mit dem stark optimierten *ssearch* lediglich einen Speedup von $1,5$ bis 2.

Die Ergebnisse auf einem FPGA reichen von Speedups von 64 bis zu 330 [HAAV, LST07]. Als Vergleichssysteme dienen zum Teil reale CPUs wobei aber oftmals eigene Implementierungen des Algorithmus als Vergleichswert dienen, was eine Einschätzung des realen Gewinns an Geschwindigkeit erschwert.

Für den BLAST-Algorithmus liegt der bisher erreichte Speedup auf einer GPU bei 3 bis 4 [VS11], wobei hier als Vergleich das BLAST-Programm des NCBI dient. Versuche, BLAST auf einem FPGA zu beschleunigen, erreichen einen Speedup für kurze Sequenzen im unteren einstelligen Bereich. [HMS+07]

2.6.2 Short-Read Mapping Problem

Bislang existieren wenige Versuche, die Short-Read Mapper auf parallele Architekturen zu überführen. Einen interessanten Versuch das Problem auf eine GPU zu übertragen stellt FANGS[32] [MNLC10] dar. Der verwendete Algorithmus basiert auf dem Q-Gram Counter Ansatz der auch bei RazerS verwendet wird. Im Vergleich mit SOAP und Bowtie ist FANGS langsamer als die Software Programme, findet aber deutlich mehr Positionen. Die Geschwindigkeit ist stark abhängig von den erlaubten Mismatches, Gaps und der Readlänge. Da bei Reads mit mehreren hundert Basenpaaren aber dementsprechend auch mehr Mismatches und Gaps eingefügt werden

[31] Lawrence Livermore National Laboratory
[32] Fast Algorithm for Next Generation Sequencers

müssen, ist selbst bei langen Reads kein deutlicher Geschwindigkeitsgewinn zu erzielen.

Einen weiterern Versuch, das Problem auf einer GPU zu beschleunigen, stellt GEAgpu [MAF+08] dar. Hier wird in einem ersten Schritt ein Index des Genomes nach kurzen Teilsequenzen der Reads von vier bis sechs Basenpaaren gesucht, ähnlich wie bei BLAST und dessen Weiterentwicklung BLAT (siehe Abschnitt 2.2.2.2). Im zweiten Schritt folgt ein lokales Alignment mit einem Smith & Waterman ähnlichen Algorithmus. Durch die nahe Verwandtschaft mit BLAST gibt es nur Vergleiche mit BLAST und BLAT, wobei aber kurze Reads von 30 Basenpaaren als Referenz dienen, für die BLAST keine guten Ergebnisse liefert. Auf Vergleiche zu Short-Read Mappern wurde verzichtet.

Im Zusammenhang mit dem Next-Generation Sequencing wird häufig das Programm MUMmerGPU [KPD+04] genannt, welches einen Speedup von 10 gegenüber der CPU Version [STDV07] erreicht. MUMmer arbeitet mit Suffix-Bäumen und erlaubt exakte Alignments beliebiger Länge. Die eigentliche Anwendung von MUMmer liegt im Vergleich zweier langer Genome miteinander, weshalb das Programm nicht als Short-Read Mapper betrachtet wird.

3 Analyse und Entwurf

Dieses Kapitel beschäftigt sich mit einer Bewertung der in Kapitel 2 vorgestellten Algorithmen und einem Entwurf für FPGA und GPU. Zunächst werden in Abschnitt 3.1 die Anforderungen an eine Umsetzung auf beiden Architekturen aufgezeigt. Anschließend werden in Abschnitt 3.2 die Algorithmen hinsichtlich ihrer Parallelisierbarkeit und Übertragung auf die Architekturen analysiert. In Abschnitt 3.4 wird eine Abbildung eines zuvor ausgewählten Algorithmus auf FPGA und GPU näher betrachtet. Der entstandene Entwurf wird in Kapitel 4 schließlich auf beiden Architekturen implementiert.

3.1 Anforderungen an den Entwurf

Eine wichtige Anforderung besteht darin, dass der größte Teil des Rechenaufwandes auf der parallelen Plattform anfallen sollte und nur geringe Nachbearbeitungen auf dem Host-System notwendig sind. Da für die Short-Read Mapper ein wichtiges Kriterium die zugeordneten Reads und die Anzahl der in der Datenbank gefunden Positionen sind, wird auf ein komplettes Alignment als Ausgabe verzichtet. Die Positionen der Reads in der Datenbank sind ausreichend.

Da nur wenige Short-Read Mapper die Möglichkeit eines Alignments mit Gaps bieten und bei kurzen Reads Gaps nicht notwendig sind (vgl. Abschnitt 2.3), wird bei der Auswahl des Algorithmus und dem Entwurf in erster Linie versucht, Mismatches effizient zu verarbeiten. Gaps sind nicht vollkommen auszuschließen, haben aber nicht die höchste Priorität.

Die angestrebte Länge der Reads sollte im für Short-Read Mapper üblichen Bereich zwischen 30 und 60 Basenpaaren liegen (vgl. Tabelle 2.2 in Abschnitt 2.3). Ohne das zusätzliche Einfügen von Gaps ist ein Alignment längerer Reads nicht sinnvoll. Die Anzahl der zulässigen Mismatches bei solchen kurzen Reads liegt üblicherweise bei maximal Sechs.

Aufgrund der heuristischen Ansätze, mit denen die bisherigen Programme arbeiten, ist der Versuch einer exakten Suche durchaus interessant und möglicherweise ebenfalls auf einer der beiden Architekturen zu realisieren. Diese Möglichkeit sollte daher nicht außer Acht gelassen werden.

Um neben einer Beschleunigung des eigentlichen Algorithmus eine hohe Geschwindigkeit zu erreichen, ist eine parallele Abarbeitung mehrerer Reads notwendig. Um dieses Ziel zu erreichen, ist es notwendig, dass die einzelnen Einheiten möglichst wenige Hardwareressourcen in Anspruch nehmen. Neben einer einfachen Untersuchung der möglichen Parallelisierung ist daher auch der Hardwareaufwand entscheidend, um sowohl auf FPGA als auch auf GPU mehrere Reads parallel abarbeiten zu können.

3.2 Analyse und Bewertung der Algorithmen

Die folgende Analyse beschränkt sich auf die wesentlichen Algorithmen, auf die sämtliche Short-Read Mapper aufbauen. Die einfachen Alignment-Algorithmen werden ebenfalls, mit Hinblick auf das Short-Read Mapping Problem untersucht, so dass sich folgende Auswahl ergibt:

1. Smith & Waterman-Algorithmus,
2. BLAST,
3. Burrows-Wheeler Transformation,
4. Q-Gram Counting und
5. Spaced Seeds Indexing.

Die erste Untersuchung beschränkt sich auf eine direkte Parallelisierung auf den genannten Plattformen. Eine parallele Abarbeitung durch Verteilung der Reads auf mehrere gleichartige Plattformen wird daher nicht betrachtet, ist aber prinzipiell immer möglich.

3.2.1 Smith & Waterman-Algorithmus

Der Smith & Waterman-Algorithmus ist durch die einfache, iterative Berechnung der Matrix sehr einfach zu parallelisieren. Jeder Rechenschritt besteht aus der gleichen Anzahl von Operationen. Die Abhängigkeiten zwischen den Matrixfeldern sind durch eine diagonale Berechnung der Felder leicht aufzulösen. Jedes Rechenelement kann unabhängig von den anderen Elementen ein Basenpaar des Reads bearbeiten.

Der Schritt auf die nächste Position in der Datenbank muss aufgrund der verbleibenden Abhängigkeiten zwischen den Rechenelementen, die den gleichen Read bearbeiten, synchron erfolgen. Sobald die Berechnungen aller Einheiten an einer Position der Datenbank abgeschlossen sind, kann die folgende Position berechnet werden. Die in Abschnitt 2.6.1 beschriebenen parallelen Ansätze für den Smith & Waterman-Algorithmus nutzen weitestgehend dieses Verfahren.

Da das Short-Read Mapping Problem in seiner eigentlichen Form nicht auf Gaps angewiesen ist, ist eine weitere Vereinfachung des Algorithmus möglich, indem nur die Diagonale berechnet wird. So müssen horizontale und vertikale Berechnungen nicht betrachtet werden und die Abfrage, welcher Vorgänger des aktuellen Matrixfeldes das Maximum liefert, entfällt.

Die Umsetzung der Pfadrekonstruktion wird von den in Abschnitt 2.6.1 erläuterten vorhandenen Ansätzen entweder auf die Ausgabe des erreichten Scores reduziert oder durch eine Auslagerung auf das Host-System erreicht. Dieser Schritt kann jedoch für das Short-Read Mapping Problem auf das Minimum reduziert und direkt an die Berechnung der Matrixfelder gekoppelt werden. Liegt der Score des untersten Feldes über dem zuvor definierten Score, wird die aktuelle Position gespeichert.

Die Anfangsposition des Reads kann durch seine bekannte Länge einfach errechnet werden. Dieses Verfahren bietet des Weiteren den Vorteil, dass nur derjenige Bereich der Matrix im Speicher gehalten werden muss, der im aktuellen Schritt benötigt wird.

Durch die erläuterten Vereinfachungen und Anpassungen an das Short-Read Mapping Problem kann der Smith & Waterman-Algorithmus auf beiden Plattformen theoretisch deutlich beschleunigt werden. Durch die einfachen Berechnungen sollte eine parallele Abarbeitung mehrerer Reads durch einen geringen Hardwareaufwand einer Einheit erreicht werden können. Durch synchrones Bearbeiten der gleichen Datenbanksequenz wird der Speicher nicht zum Engpass.

3.2.2 BLAST

Aufgrund der verschiedenen Teilschritte des BLAST-Algorithmus (vgl. Abschnitt 2.2.2.2) verspricht eine Übertragung des kompletten Algorithmus eine geringe erreichbare Parallelität. Eine Auslagerung des rechenintensivsten Teilschrittes, welcher dem Auffinden der Hits entspricht, auf die Hardware ist daher sinnvoll. Die gefundenen Hits können anschließend auf dem Host-System verknüpft werden. Das Auffinden eines Hits kann ähnlich aufgebaut und parallelisiert werden wie der zuvor betrachtete Smith & Waterman-Algorithmus ohne Gaps und Backtracking.

Da der BLAST-Algorithmus für lange Sequenzen effizient arbeitet und nicht an kurze Reads angepasst ist, verspricht eine Untersuchung des Q-Gram Countings oder des Spaced Seed Ansatzes, deren erster Schritt jeweils dem BLAST-Verfahren ähnlich aufgebaut ist, einen höheren Geschwindigkeitsgewinn.

3.2.3 Burrows-Wheeler Transformation

Die Suche in einer Burrows-Wheeler transformierten Datenbank erfolgt mit dem FM Index-Algorithmus (vgl. Abschnitt 2.4.1). Für eine CPU liegt der Vorteil bei dieser Suche darin, dass nicht die komplette Datenbank durchsucht werden muss, sondern nur bestimmte Teilabschnitte. Da diese Teilabschnitte nicht vorhersagbar sind, ist diese Suche auf FPGA und GPU schwer zu realisieren. Für eine hohe Parallelität sind mehrere Einheiten notwendig, die unterschiedliche Reads bearbeiten, wodurch es zwangsläufig zu vielen Speicherzugriffen an unterschiedlichen Stellen der transformierten Datenbank kommt. Das Speicherinterface wird somit zum Engpass.

Für die inexakte Suche kommt zusätzlich noch das Backtracking beim Einfügen von Mismatches hinzu, was zu einer ungleichen Lastverteilung zwischen den parallelen Einheiten führt und kein synchrones Laden neuer Reads ermöglicht. Dadurch wird die Komplexität auf den beiden Plattformen zusätzlich erhöht. Mit einem Vorsortieren der Reads, welches ermöglicht, dass möglichst gleichartige Anfragen zusammen abgearbeitet werden, kann das Problem zwar verringert, aber nicht komplett umgangen werden.

Auch wenn der Algorithmus eine hohe Geschwindigkeit bei der Suche erzielt, ist eine effiziente Parallelisierung auf FPGA und GPU schwer zu realisieren. Durch fehlende Cache-Speicher auf der GPU muss auf dem externen Grafikspeicher gearbeitet werden, der zwar über eine hohe

Speicherbandbreite verfügt, aber bei mehreren Einheiten dennoch zum Engpass wird. Auf dem FPGA wird ebenfalls der Zugriff auf einen externen Speicher zum Engpass. Der Algorithmus selbst erfordert durch das Backtracking ebenfalls einen höheren Bedarf an Hardware, wodurch es zu einer Verringerung der Parallelität kommt.

3.2.4 Q-Gram Counting

Der wichtigste Schritt des Q-Gram Countings, der auf die parallele Plattform übertragen werden muss, ist das Herausfiltern der Positionen, die einen potentiellen Treffer enthalten (vgl. Abschnitt 2.4.2). Dieser Schritt entspricht im Wesentlichen den ersten beiden Teilschritten des BLAST-Verfahrens, wobei beim Q-Gram Counting durch die kurzen Teilsequenzen der Reads von 3 bis 6 Basenpaaren die Sensitivität deutlich höher ist und das Verfahren somit optimal an das Short-Read Mapping Problem angepasst ist.

Im Gegensatz zur Suche in einer Burrows-Wheeler transformierten Datenbank erfolgen die Speicherzugriffe fortlaufend und für sämtliche parallele Einheiten an der selben Position. Dadurch werden die Speicherzugriffe nicht zum Engpass und die Einheiten können synchron arbeiten. Die Q-Grams für ein Parallelogramm können innerhalb der Einheit parallel mit der Datenbank verglichen werden, da keine Abhängigkeiten bestehen. Lediglich das Aktualisieren des Counters und Sichern der Position innerhalb einer Einheit muss sequentiell erfolgen.

Eine Parallelisierung der Suche und Realisierung mehrerer paralleler Einheiten ist einfach möglich. Das Problem besteht lediglich in einer notwendigen Nachbearbeitung und Verifizierung der Positionen auf dem Host-Rechner und den hohen Ressourchenaufwand innerhalb jeder Einheit für das Speichern eines kompletten Parallelogramms.

3.2.5 Spaced Seeds

Die Suche beim Spaced Seed Ansatz baut bei den Short-Read Mappern Maq und Pass auf Hashtabellen auf. Die Suche mit Seeds fester Größe, wie sie bei Maq zum Einsatz kommt, lässt sich auf den FPGA ideal durch Vergleicher realisieren, die bei einer zuvor definierten Anzahl passender Seeds die Position als Treffer ausgeben. Das Problem dieses Verfahrens besteht in seiner Ungenauigkeit. Durch einen geringen Hardwareaufwand sollte es jedoch möglich sein, eine große Anzahl von parallel arbeitenden Einheiten zu realisieren.

Weitere Nachbearbeitungen wie das Verifizieren der Position bei Maq oder das Alignment der Ränder wie bei PASS müssten in diesem Fall wieder auf den Host-Rechner ausgelagert werden. Durch die geringe Sensitivität der Verfahren im ersten Schritt ist der Anteil der Berechnungen, der nicht auf FPGA oder GPU durchgeführt wird, dementsprechend hoch.

3.3 Auswahl eines Algorithmus

Für eine hohe Anzahl paralleler Einheiten ist die Arbeit auf einem gemeinsamen Datenstrom von Vorteil und erlaubt einen Geschwindigkeitsgewinn durch die gleichmäßigen vorhersagbaren Speicherzugriffe. Gegenüber einer Suche in einer Burrows-Wheeler transformierten Datenbank, die schlecht parallelisierbar ist, können solche Ansätze durchaus eine hohe Leistung durch massive Parallelität erreichen. Algorithmen, die hierfür in Frage kommen, sind der Smith & Waterman-Algorithmus, das Q-Gram Counting und der Spaced Seed Ansatz, wobei die letzten beiden Verfahren aufgrund ihrer Komplexität auf einem FPGA oder einer GPU nicht ressourcensparend umgesetzt werden können.

Um eine möglichst hohe Sensitivität ohne eine aufwändige Nachbearbeitung auf dem Host-System zu erhalten, müssen die Algorithmen möglichst direkt bei der Suche exakte oder weniger heuristische Lösungen generieren. Werden lediglich Mismatches zugelassen, können mit einem breiten Vergleicher direkt in Hardware exakte Positionen auf dem FPGA berechnet werden. Auch auf der GPU lassen sich mit einem auf die Diagonale reduzierten Smith & Waterman-Algorithmus exakte Positionen ermitteln. Ein solcher Ansatz, der direkt an das Short-Read Mapping Problem angepasst wurde, kann als ein Smith & Waterman-Algorithmus ohne Gaps angesehen werden. Es bestehen Ähnlichkeiten mit dem Spaced Seed Verfahren, oder dem ersten Schritt des BLAST-Algorithmus, im Gegensatz dazu arbeitet das Verfahren jedoch exakt. Ein vollständig in Hardware umgesetzter Smith & Waterman-Algorithmus, welcher als systolisches Array aufgebaut ist, würde aufgrund des hohen Harwareaufwandes deutlich langsamer sein. Eine mögliche Erweiterung, die auf kurze Vergleicher aufbaut, stellt das Q-Gram Counting dar.

Somit wird nicht eine Vielzahl an Positionen markiert, die in einem nächsten Schritt nachbearbeitet werden, sondern unmittelbar exakte Ergebnisse geliefert. Neben einen möglichen Geschwindigkeitsgewinn gegenüber anderen Short-Read Mappern besteht eine Besonderheit des reduzierten Smith & Waterman-Ansatzes darin, für einen Read bei gegebener Anzahl Mismatches sämtliche Positionen in der Datenbank mit Sicherheit zu finden.

Ein solches Verfahren reduziert zwar die Geschwindigkeit der eigentlichen Suche, da im Gegensatz zu anderen Short-Read Mappern jede Position in der Datenbank betrachtet wird, kann aber durch die hohe Parallelisierung des Algorithmus selbst und direkte Anpassung an die Hardware beschleunigt werden. Durch die einfache Struktur wird der Verbrauch an Ressourcen pro Einheit gesenkt, was die Anzahl an parallelen Einheiten deutlich erhöht und somit die Geschwindigkeit steigert. Ein synchrones Arbeiten auf einem gemeinsamen Datenbank-Stream verhindert, dass der Zugriff auf die Datenbank zum Engpass wird.

3.4 Abbildung auf parallele Architekturen

Dieser Abschnitt beschreibt einen Entwurf des vorgestellten Ansatzes auf FPGA und GPU. Der FPGA verspricht durch die direkte Anpassung an die Hardware eine höhere Geschwindigkeit.

Zum Vergleich wird versucht, einen ähnlichen Ansatz auf die GPU zu übertragen, was trotz der geringeren Leistung eine weitere Möglichkeit bietet, sämtliche Positionen exakt zu berechnen.

3.4.1 Entwurf einer Abbildung für einen FPGA

Die einfachste Möglichkeit, einen derartigen reduzierten Smith & Waterman-Algorithmus auf dem FPGA umzusetzen, besteht in einem einfachen Register, welches einen Read enthält und mit einem Teil der Datenbanksequenz verglichen wird, die immer um eine Position weiter geschoben wird. Dabei werden die Anzahl der Mismatches gezählt und die aktuelle Position ausgegeben, sobald eine vorgegebene Zahl von Mismatches unterschritten wird. Abbildung 3.1 zeigt den Entwurf eines solchen Systems. Das Zählen der Mismatches kann über einen baumförmigen Zähler erfolgen, der eine hohe Geschwindigkeit durch größtmögliche Parallelität gewährleistet, so dass die Datenbank mit jedem Takt um eine Position weiter geschoben werden kann.

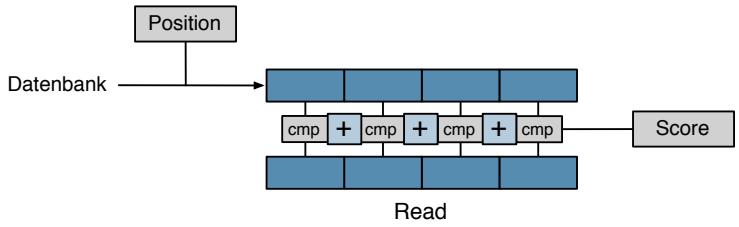


Abbildung 3.1: Vereinfachte Darstellung der Suche auf dem FPGA. Die Datenbank wird durch ein Schieberegister geleitet, der Read wird zu Beginn fest in das Register geladen.

Da der Hardwareaufwand einer solchen Such-Einheit relativ gering ist, kann eine hohe Leistung durch parallele Einheiten erreicht werden. Ein synchrones Arbeiten auf dem selben Datenbank-Stream verhindert, dass der Zugriff auf die Datenbank zum Engpass wird, wie es beispielsweise bei einer Suche in einer Burrows-Wheeler transformierten Datenbank der Fall wäre.

Um die Ergebnisdaten zu sichern, stehen grundsätzlich zwei Möglichkeiten zur Verfügung. Such-Einheiten können lokal ihre Ergebnisse für die Dauer einer kompletten Suche speichern, oder die Ergebnisse im laufenden Betrieb an einen zentralen Speicher senden, der sie weiter an das Host-System leitet. Da bei mehr als 100 Einheiten die Wahrscheinlichkeit von mehreren HMRs wächst und es so trotz Zwischenspeichern zu Konflikten beim zentralen Sichern der Daten kommen kann, sind in einem ersten Entwurf lokale Speicher an den einzelnen Such-Einheiten zu bevorzugen, um korrekte und vollständige Ergebnisse zu liefern. Da vor der Suche nicht bekannt ist, welche Reads HMRs sind, ist eine Sortierung auf dem Host-System im Vorfeld nicht möglich.

Die Übertragung von Reads, Datenbank und Ergebnissen erfolgt aufgrund der ausreichenden Datenrate mittels Gigabit-Ethernet. Aufgrund der Größe der Datenbank von bis zu 8 GByte ist das lokale Speichern am FPGA nicht möglich. Die Aufgabe des Host-Systems besteht darin, die Reads zu übertragen, ein stabiles Streaming der Datenbank zu gewährleisten, die Ergebnisse einzusammeln und in einem Format zu speichern, das einfach weiterverarbeitet werden kann.

3.4.2 Entwurf einer Abbildung für eine GPU

Den selben Ansatz wie auf dem FPGA auf der GPU zu realisieren, verspricht keine guten Ergebnisse, da das Vergleichen und Aufsummieren mit einer unterschiedlichen Zahl von Threads erfolgen würde, was zu einer schlechten Lastverteilung führt. Eine bessere Auslastung und leichtere Parallelisierung verspricht ein auf die Diagonale reduzierter Smith & Waterman-Algorithmus. Da es auf der GPU auch leicht möglich ist, einen vollständigen Smith & Waterman-Algorithmus umzusetzen, bildet dieser den Ausgangspunkt, kann jedoch optional deaktiviert werden, um die Geschwindigkeit zu steigern. Abbildung 3.2 zeigt ein schrittweises Auflösen der Abhängigkeiten.

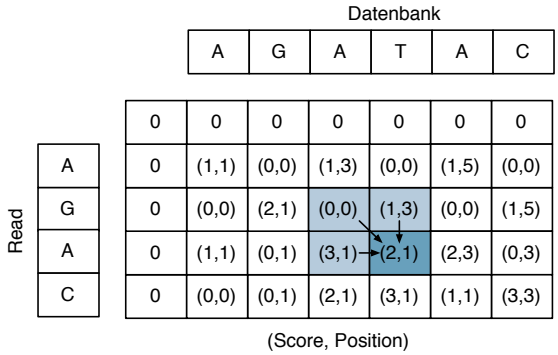
Die sequentielle Berechnung erfolgt zeilen- oder spaltenweise, wodurch die Abhängigkeiten (vgl. Abbildung 3.2(a)) umgangen werden. Eine mögliche Parallelisierung besteht in der parallelen Berechnung einer kompletten Spalte, wobei die Anzahl der Threads der Anzahl der Basenpaare des Reads entspricht. Um die Abhängigkeiten aufzulösen, müssen die Threads die Felder diagonal nach links versetzt berechnen, wie in Abbildung 3.2(b) angedeutet (vgl. [MV08, KSPBP10]). Um eine gleichmäßige Abarbeitung zu gewährleisten, müssen vor und hinter der Datenbank $r - 1$ ungültige Basenpaare angehangen werden, wobei r der Länge des Reads entspricht.

Eine weitere Veränderung zum eigentlichen Smith & Waterman-Algorithmus besteht darin, dass neben dem Score nicht die Richtung gespeichert wird, die das Maximum ergab. Da ein komplettes Alignment des gesamten Reads gefordert ist, kann beginnend mit einem Match des ersten Basenpaares die Position als Zusatzinformation gespeichert und weitergereicht werden. Dadurch ist es möglich, bei einem ausreichenden Score in der untersten Zeile der Matrix die Startposition des Reads auszugeben. Werden Gaps außer Acht gelassen, ist diese Zusatzinformation nicht mehr notwendig, da aus der Position des letzten Feldes und der bekannten Länge des Reads die Startposition berechnet werden kann.

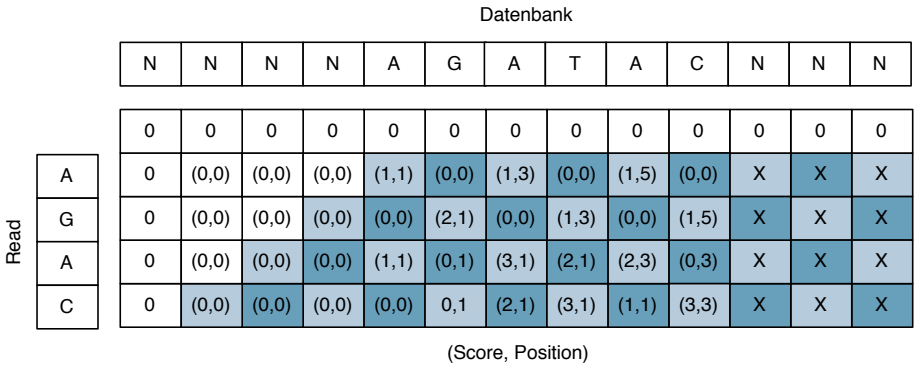
Durch die integrierte Ausgabe der Position ist keine aufwändige Pfadrekonstruktion wie im zugrundeliegenden Algorithmus mehr notwendig. Daraus ergibt sich als weiterer Vorteil, dass nicht die gesamte Matrix im Speicher gehalten werden muss, sondern nur ein Bereich von drei Diagonalen. Die Speicherkomplexität des Algorithmus wird so von $O(m \cdot n)$ auf $O(3 \cdot n)$ reduziert, wobei die Länge einer Datenbanksequenz m mehrere Millionen Basenpaare betragen kann. Durch diese Reduktion ist es möglich, interne Speicher der GPU zu nutzen. Abbildung 3.2(c) zeigt eine reduzierte Matrix, wobei jeder Zeile ein anderer Thread zugeordnet ist.

Werden keine Gaps zugelassen, entspricht der Score der Summe der Matches. Werden aber Gaps zugelassen, kann aus dem Score nicht eindeutig berechnet werden, ob Mismatches oder Gaps in dem Alignment vorhanden sind.

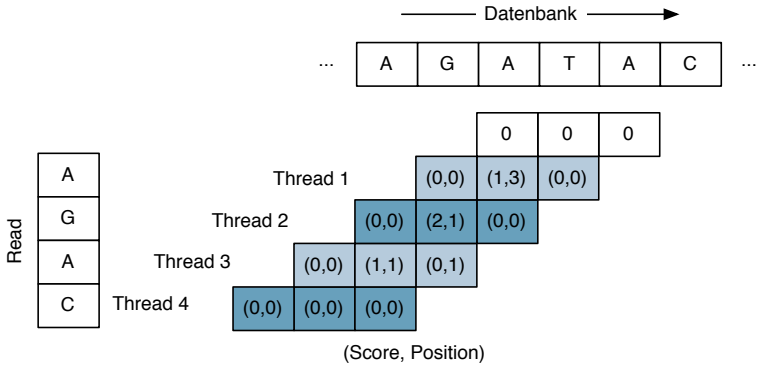
Aufgrund der gemeinsamen Matrix, auf der die für einen Read zuständigen Threads arbeiten, ist es sinnvoll, jeweils einen Multiprozessor der GPU genau einen Read abarbeiten zu lassen. Die erreichbare Anzahl der parallelen Einheiten liegt bei der zur Verfügung stehenden Nvidia GeForce GTX 480 bei 16.



(a) Vollständige Matrix mit gekennzeichneten Abhängigkeiten.



(b) Vollständige Matrix mit diagonaler Berechnung der Matrixelemente.



(c) Reduktion auf für aktuelle Berechnung notwendige Felder.

Abbildung 3.2: Auflösen der Abhängigkeiten des Smith & Waterman-Algorithmus.

4 Implementierung

Nachdem sich das vorherige Kapitel mit einer Analyse der Algorithmen und den Entwürfen auf einer FPGA- und einer GPU-Architektur beschäftigt hat, werden in diesem Abschnitt die Implementierungen auf beiden Plattformen ausgearbeitet und erläutert. Zunächst werden in Abschnitt 4.1 die Anforderungen aus Abschnitt 3.1 in Hinblick auf Anforderungen typischer Nutzer erweitert. Danach folgen in den Abschnitten 4.2 und 4.3 Ausarbeitung und Dokumentation der Entwürfe.

4.1 Anforderungen an die Implementierung

Die bisherige Analyse der Anforderungen im Entwurf (vgl. Abschnitt 3.1) beschränkt sich auf den Algorithmus selbst, sowie die Länge der Reads und die Anzahl der Mismatches. Weitere wichtige Aspekte sind jedoch auch Leistung, Korrektheit der Ergebnisse, Zuverlässigkeit und Bedienbarkeit.

Die zu erwartende Leistung sollte für beide Plattformen deutlich über der des einzigen exakten Verfahrens, dem Smith & Waterman-Algorithmus, liegen. Wünschenswert wäre, dass die Leistung etwa derer der Short-Read Mapper Bowtie, PASS und RazerS entspricht. Für die GPU ist die erreichbare Geschwindigkeit schwer vorhersagbar. Um mit einem Virtex-6 bei einer angestrebten Taktrate von 200 MHz eine mit Bowtie vergleichbare Geschwindigkeit zu erreichen, sind ungefähr 600 parallele Such-Einheiten notwendig (berechnet mit Daten aus Tabelle 2.5). Die einzelnen Such-Einheiten müssen dementsprechend wenig Hardware beanspruchen, damit das Ziel von 600 Einheiten erreicht werden kann. Des Weiteren ergibt sich die Anforderung an die Datenschnittstelle, ein Streaming mit ausreichender Geschwindigkeit zu ermöglichen, um eine gleichmäßige Abarbeitung mit 200 MHz zu ermöglichen.

Da es sich um ein exaktes Verfahren handelt, welches alle möglich Positionen findet, muss außerdem sichergestellt werden, dass selbst bei HMRs keine Position verlorengeht. Daher spielt ebenso die Zuverlässigkeit der Systeme eine große Rolle. Gerade die Kommunikation und insbesondere das Streaming der Datenbank beim FPGA müssen hier äußerst zuverlässig arbeiten.

Eine weitere wichtige Anforderung besteht darin, Systeme zu entwickeln, die von typischen Nutzern einfach bedient und konfiguriert werden können. Wichtig ist in diesem Zusammenhang auch, dass Standardformate bei den Eingabe- und Ausgabedaten unterstützt werden. Das bedeutet, dass das übliche FASTA-Format für Reads und Datenbanken gelesen werden kann und die Ausgabe im SAM-Format erfolgt.

4.2 Umsetzung auf dem FPGA

Im folgenden wird der Entwurf aus Abschnitt 3.4.1 hinsichtlich der zuvor erweiterten Anforderungen detailliert ausgearbeitet. Zu Beginn wird in Abschnitt 4.2.1 der Aufbau einer einzelnen Such-Einheit erläutert. Die Zusammenschaltung der einzelnen Einheiten folgt in Abschnitt 4.2.2 und auf die erforderliche Ablaufsteuerung wird in Abschnitt 4.2.3 eingegangen. In Abschnitt 4.2.4 wird die Realisierung der Datenkommunikation mittels Gigabit-Ethernet beschrieben. Anschließend folgt in Abschnitt 4.2.5 eine Auflistung der erforderlichen Hardwareressourcen. Die beiden letzten Abschnitte 4.2.6 und 4.2.7 behandeln die Implementierung auf Seiten des Hosts und die notwendigen Datenformate.

4.2.1 Such-Algorithmus

Der prinzipielle Aufbau einer *Such-Einheit* wurde bereits in Abschnitt 3.4.1 erläutert und in Abbildung 3.1 skizziert. Die aktuelle Teilsequenz der Datenbank wird in einem Schieberegister außerhalb der Einheit gehalten. Jedes der vier Nukleotide ist mit zwei Bit kodiert und in der Suchphase werden mit jeden Takt zwei neue Basenpaare in das Schieberegister geladen. Tabelle 4.1 zeigt die Kodierung der Nukleotide.

Tabelle 4.1: Kodierung der Nukleotide.

Nukleotid	Kodierung
A	00_2
G	01_2
T	10_2
C	11_2

Zum lokalen Sichern der Positionsdaten wird für jede Einheit ein Port eines Dual Port Block RAMs zur Verfügung gestellt, um Konflikte durch den parallelen Zugriff mehrerer Einheiten zu vermeiden. Das Laden der Reads in die Einheiten erfolgt ebenfalls über den Block RAM, so dass keine weiteren Datenleitungen zur Kommunikation mit der Außenwelt notwendig sind.

Die Einheit selbst besitzt vier grundlegende Funktionen:

1. Sichern des aktuellen Reads und des Schwellwertes,
2. Vergleichen des Reads mit der aktuellen Teilsequenz der Datenbank,
3. Zählen der auftretenden Mismatches und
4. Speichern der aktuellen Position bei Unterschreitung des Schwellwertes.

Abbildung 4.1 zeigt den Aufbau einer einzelnen Such-Einheit. Der Vergleich der Basenpaare des Reads mit den Basenpaaren der Datenbank im externen Schieberegisters erfolgt über einen verteilten LUT RAM. Um das Speichern einer 32-Bit Position möglichst schnell durchzuführen,

hat der Datenbus des Block RAM eine Breite von ebenfalls 32 Bit. Jeder Block RAM verfügt über einen Speicher von 1.024 Worten zu je 4 Byte.

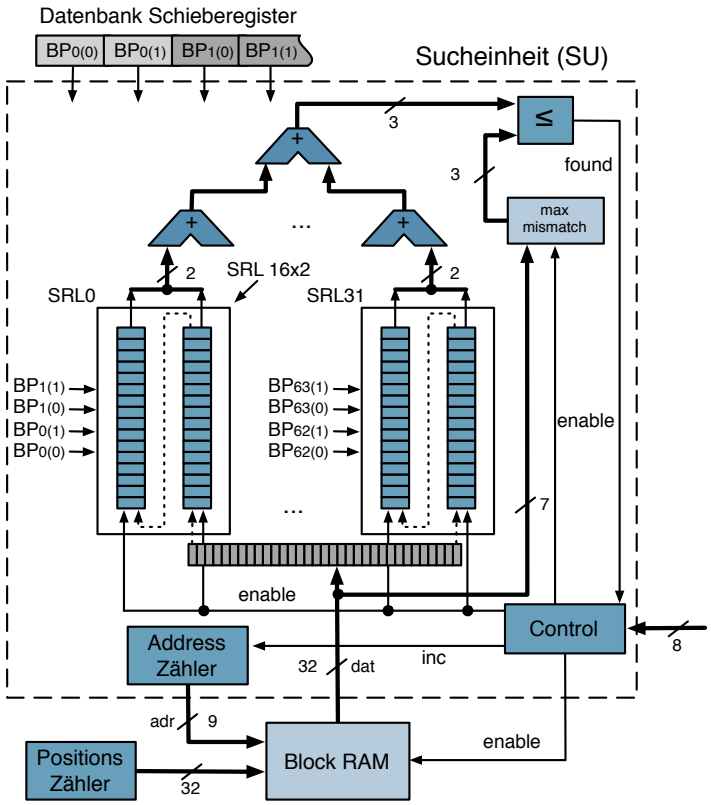


Abbildung 4.1: Such-Einheit, bestehend aus 32 verketteten LUT-RAMs, die parallel geladen werden, und einem baumförmigen saturierenden Zählers für die Mismatches.

Um die Anzahl der benötigten Register zu reduzieren, wird der Read in verteilten LUT RAMs gehalten, die als verkettetes serielles Schieberegister (SRL[1]) geladen werden. Die SRLs arbeiten als 16×2-ROMs, in denen nicht der Read selbst gespeichert ist, sondern die Anzahl der Mismatches zwischen zwei Basenpaaren der Datenbank mit dem Read. Die Basenpaare der Datenbank dienen hierbei als 4-Bit Adresse und adressieren beide Blöcke parallel, so dass zwei Bit ausgegeben werden, was den maximal möglichen zwei Mismatches entspricht die von zwei Basenpaaren erzeugt werden können. Die zwei Ausgabebits sind einfach binär kodiert.

Durch die Breite des Block RAMs von 32-Bit ist ein paralleles bitweises Laden von 32 verketteten 16×2-ROMs direkt aus dem Block RAM möglich und es kann ein Read mit einer Länge von 64 Basenpaaren abgeglichen werden. Dadurch ergibt sich für das Schieberegister, in dem sich der aktuelle Teil der Datenbank befindet, eine Länge von 128 Bit für 64 Basenpaare. Die Transformation der Reads in das notwendige Format erfolgt im Vorfeld per Software (vgl. Abschnitt 4.2.6). Die SRLs werden für kürzere Reads mit Nullen aufgefüllt, was bei Anlegen beliebiger Basenpaare keine Mismatches verursacht.

[1]Shift Register Latch

Abbildung 4.2 zeigt die Konfiguration und das Auslesen der Mismatches an einem Beispiel für einen einzelnen 16 × 2-ROM. Die Basenpaare *AG* des Reads werden im Vorfeld per Software in die hexadezimale Sequenz *0xDDD0 222D* übersetzt (vgl. Listing 7.2) und dann seriell in die verketteten LUTs geschrieben. Die Datenbank adressiert beide LUTs parallel entsprechend ihrer binären Kodierung. Die Ausgabe ist die ebenfalls binär kodierte Anzahl der Mismatches.

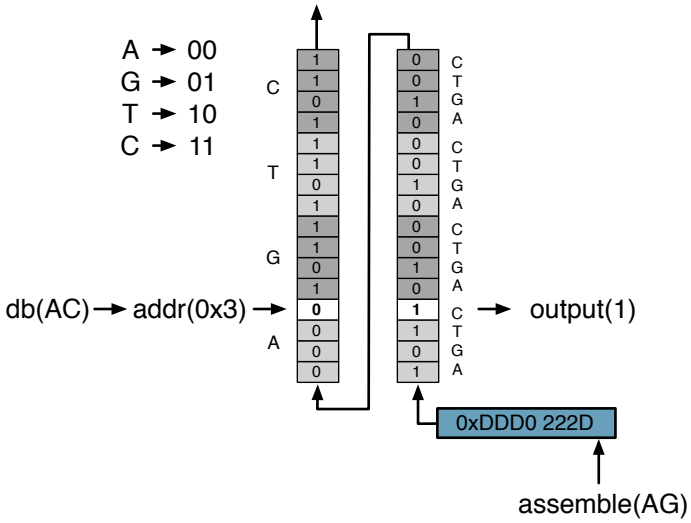


Abbildung 4.2: Beispiel des Vergleiches zwischen den Basenpaaren AG eines Reads und AC der Datenbank.

Realisiert werden die 16 × 2 SRLs in einer einzelnen LUT mit 6 Eingängen direkt über die `CFGLUT5` Primitive, die auf Virtex-5 und Virtex-6 Architekturen vorhanden ist. [Xil10c]

Dabei wird kein normales Slice Register benötigt und der Vergleich sowie die erste Stufe des Zählers komplett in den SRLs realisiert. Durch die Auslagerung der Vorberechnung auf den Host-Rechner und die effiziente Bündelung der unterschiedlichen Funktionen, kann der Ressourcenbedarf einer Einheit stark reduziert werden. Die Datenmenge, die in die SRLs geladen werden muss, ist mit 128 Byte um den Faktor acht größer ist als bei einer reinen Implementierung mit Registern. Durch das serielle Laden und die höhere Datenmenge sind 32 Takte notwendig, um die 32 SRLs mit jeweils 32 Bit zu laden.

Die von den SRLs bereitgestellten, 2-Bit binär kodierten Ausgangsleitungen, werden mit mehreren saturierenden 3-Bit Zählern aufsummiert. Um eine größtmögliche Geschwindigkeit durch hohe Parallelität zu erreichen, ist der Zähler baumförmig aufgebaut. Die erste Stufe besteht aus einzelnen LUTs, von denen jede drei 2 Bit breite Eingänge und einen 3 Bit breiten Ausgang besitzt. Die folgenden Stufen haben je zwei 3-Bit Eingänge und einen 3-Bit Ausgang. Durch die effiziente Auslastung der LUTs ist so eine komplette Berechnung der Mismatches mit nur einer Pipelinestufe bei einer Taktrate von 200 MHz möglich.

Anschließend werden die gefundenen Positionen in den Speicher geschrieben. Dazu ist der Dateneingang des Block RAMs direkt mit dem Positionszähler verbunden. Es muss bei einem Treffer nur das Signal zum Schreiben gesetzt und die Adresse des internen 10-Bit Adresszählers in-

krementiert und somit auf die folgende Speicherstelle gesetzt werden. Der aktuelle Wert des Adresszählers entspricht ebenfalls der Anzahl der bisher gefundenen Positionen.

Der Automat zur Steuerung einer Einheit wurde möglichst einfach gehalten, um Ressourcen zu sparen. Die wesentlichen Arbeitsabläufe werden über eine 8 Bit breite Signalleitung von der zentralen Ablaufsteuerung aus gesteuert. In einem ersten Zustand werden die Daten, die zuvor von außen in den Block RAM geschrieben wurden, in die LUT RAMs und das Mismatch Register geladen (Abbildung 4.3(a) zeigt die Formatierung der Daten). Sobald über ein Steuersignal der Beginn der Suche signalisiert wird, werden automatisch die gefunden Positionen, beginnend mit der zweiten Adresse, in den Block RAM geschrieben.

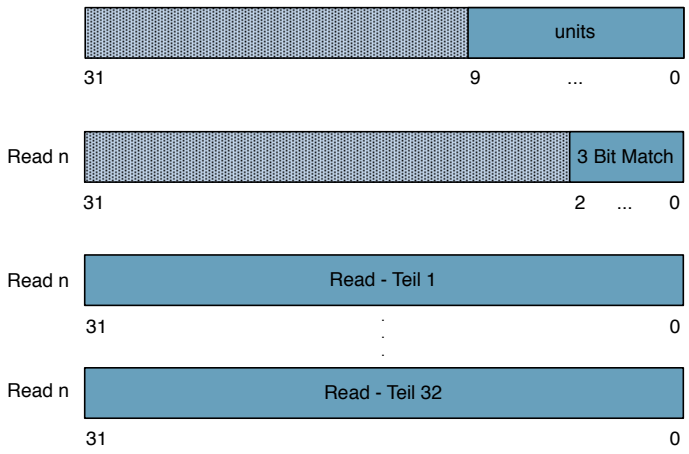
Wird über ein weiters Steuersignal das Ende des Durchlaufes bekannt gegeben, werden automatisch die Anzahl der gefunden Positionen und die minimale Zahl der Mismatches an die erste Adresse des Block RAMs geschrieben. Da die Anzahl der gefundenen Positionen der ursprünglich anliegenden Adresse des Block RAMs entspricht, sind keine weiteren Register oder Zähler notwendig. Werden keine Übereinstimmungen gefunden, wird lediglich die Zahl der notwendigen Mismatches für einen Treffer in den Speicher geschrieben. Abbildung 4.3(b) zeigt die Belegung des Speichers am Ende der Suche. Falls mehr als 1.023 Positionen gefunden werden, erfolgt die Vermeidung eines Überlaufes im Speicher über eine externe Kontrolle, auf die im folgenden Abschnitt eingegangen wird.

4.2.2 Zusammenschaltung aller Elemente (Top-Level)

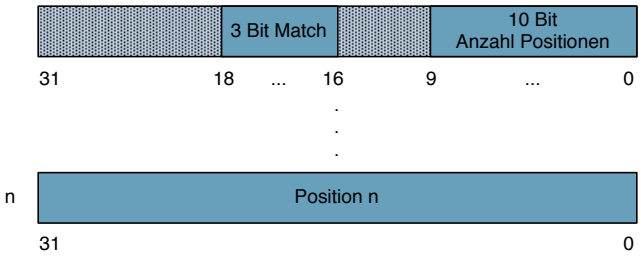
Die bisher betrachteten einfachen Such-Einheiten verfügen lediglich über ein einfaches Interface zum Zugriff auf ihren Block RAM, einen 8 Bit breiten Eingang für Steuersignale und den 128 Bit breiten Datenbus des Schieberegisters, in dem sich die aktuellen 64 Basenpaare der Datenbank befinden.

Um 600 Such-Einheiten zu realisieren, sind insgesamt 600 Block RAMs notwendig. Da der Virtex-6 lediglich über 416 einzelne Block RAMs verfügt, ist es notwendig, dass sich jeweils zwei Einheiten einen Dual Port Block RAM teilen. Diese beiden Einheiten sind gemeinsam mit dem Speicher in 300 *Such-Module* gegliedert. Durch die zwei unabhängigen Ports des Block RAMs entstehen keine Konflikte beim Speichern der Positionen.

Abbildung 4.4 zeigt den Aufbau eines Moduls. Die beiden Such-Einheiten sind in *Control-* und *Search*-Einheit unterteilt, wobei beide Einheiten wie in Abbildung 4.1 beschrieben aufgebaut sind, die Control-Einheit aber über zusätzliche Funktionen verfügt. Die Unterteilung des Block RAMs in zwei für die Einheiten getrennte Bereiche erfolgt, indem die Control-Einheit während der Suche auf den höherwertigen und die Search-Einheit auf den niederwertigen Speicherbereich zugreift. Zeigen die Adresszähler auf die selbe Folgeadresse, wird automatisch ein Überlaufsignal (Overflow) gesetzt, welches die Suche anhält, bis die Ergebnisse an den Host übertragen wurden. So wird der Speicherbereich optimal ausgenutzt, da relativ selten zwei HMRs in einem Modul auftreten (vgl. Abbildung A.1).



(a) Ausgangsdaten



(b) Ergebnisdaten

Abbildung 4.3: Formatierung der Ein- und Ausgabedaten.

Ein weiterer Unterschied zwischen den unterschiedlichen Einheiten besteht darin, dass die Control-Einheit als Zusatzfunktion die Adressierung des Block RAMs für einen externen Schreib- oder Lesezugriff übernimmt und für diesen Fall auf beide Bereiche des Speichers zugreifen kann. Durch dieses Vorgehen wird ein Adressbus der vom zentralen Steuerautomaten an jedes der 300 Module geleitet wird, vermieden. Der Positionszähler kann nicht als Adresszähler dienen, da bei einem Überlauf die Position im Zähler erhalten bleiben muss. Des Weiteren erfolgt ein Schreibzugriff über einen Multiplexer, über den sich die Control-Einheit einen Dateneingang des Block RAMs mit der Außenwelt teilt.

Abbildung 4.5 zeigt einen Überblick über alle notwendigen Elemente des Gesamtsystems mit 300 Such-Modulen und 600 Such-Einheiten. Der Ethernet-Controller, welcher in Abschnitt 4.2.4 erläutert wird, ist durch seine geringere Taktrate von 125 MHz über FIFO[2]-Speicher mit sämtlichen anderen Komponenten, die mit einer Taktrate von 200 MHz arbeiten, verbunden. Gesteuert werden die Verteilung der Daten und die einzelnen Such-Module mit ihren Such-Einheiten über einen zentralen Steuerautomaten (Control), der Informationen zur Steuerung mit dem Ethernet-Controller austauschen kann und auf den in Abschnitt 4.2.3 näher eingegangen wird.

[2]First In First Out

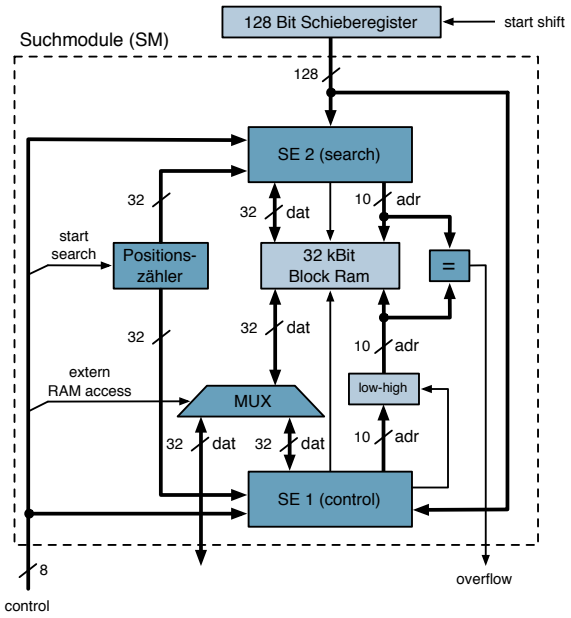


Abbildung 4.4: Aufbau eines Moduls, bestehen aus zwei Such-Einheit.

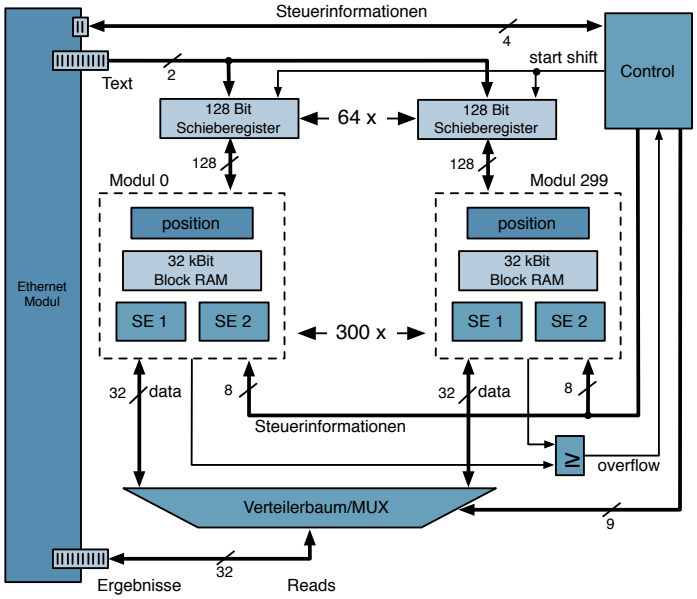


Abbildung 4.5: Oberste Ebene mit Ethernet-Controller, Steuereinheit und 300 Such-Modulen, von denen jedes zwei Such-Einheiten beinhaltet.

Das Verteilen und Einsammeln der Daten erfolgt über binäre Bäume, die zur Vereinfachung nicht dargestellt wurden. An jedem Knoten eines solchen Baumes befindet sich ein Register, um die einzelnen Module auf dem FPGA zu gruppieren und das Routing durch die Pipelinestufen zu vereinfachen. Der Datenausgang des Ethernet-Controllers wird auf eine Breite von 8 Bit serialisiert, über einen Scatter-Baum an die Dateneingänge aller Einheiten verteilt und dort wieder auf eine Breite von 32 Bit umgewandelt um das Routing durch kleinere Busse zu vereinfachen. Über ein separates Enable-Signal kann über einen baumförmigen Scatter-Multiplexer jedes Such-Modul ausgewählt werden.

Der Dateneingang des Ethernet-Controllers ist über einen baumförmigen Gather-Multiplexer, der ebenfalls binär aufgebaut ist und in jedem Knoten ein Register enthält, mit den Datenausgängen der einzelnen Such-Module verbunden. Die Breite von 32 Bit wird hier beibehalten, um den Hardwareaufwand für das Serialisieren in jedem Modul zu vermeiden sowie eine hohe Datenrate beim Auslesen der Ergebnisse zu gewährleisten. Da das Auslesen der Daten wesentlich häufiger vorkommt als das Schreiben, bietet sich hier ein breiterer Bus an. Durch die Kommunikation über Ethernet ist zwar eine Begrenzung der Datenrate gegeben, für mögliche Weiterentwicklungen hinsichtlich PCI-Express oder SATA steht aber dennoch ein 32-Bit Bus zur Verfügung.

Steuersignale und die 2 Bit breite Datenleitung zur Übertragung der einzelnen Zeichen der Datenbank an die Schieberegister werden ebenfalls über einen Scatter-Baum verteilt. Wichtig ist hierbei, dass alle Bäume gleich aufgebaut sind, um eine strukturierte Verteilung über den gesamten FPGA zu erreichen. Die Anzahl der Knoten von der Wurzel bis zu jedem Blatt ist für alle Blätter gleich, um für die Kommunikation mit jedem Modul die gleiche Latenz zu gewährleisten.

Wie in Abbildung 4.5 angedeutet, wird ein Schieberegister immer von vier Modulen genutzt, um die benötigten Hardwareressourcen zu verringern. Würde jedes Modul ein eigenes Schieberegister mit 128 Bit besitzen, wären deutlich weniger Einheiten auf dem Virtex-6 realisierbar. Die vier Module die auf das gleiche Schieberegister zugreifen, liegen auf dem FPGA sehr nahe beieinander und bilden ein *Cluster*. Die sich so ergebenden 75 Schieberegister bilden den optimalen Kompromiss zwischen Hardwareaufwand und Routingressourcen. Abbildung 4.2.2 zeigt beispielhaft den Verteilungsbaum für die Datenbankzeichen mit den Schieberegistern und den Clustern.

4.2.3 Zentraler Steuerautomat

Um die Verteilung der Daten und die unterschiedlichen Phasen eines Durchlaufes zu koordinieren, ist ein zentraler Steuerautomat notwendig. Der Funktionsablauf gliedert sich in die Teilschritte:

1. Laden der transformierten Reads in den Block RAM der Such-Module,
2. Paralleles Laden der Reads aus den Block RAMs in die SRLs der Such-Einheiten,
3. Laden der Datenbank in das 128 Bit Schieberegister,
4. Suche mit konstantem streaming der Datenbank und
5. Auslesen der Ergebnisse und sofortiges Senden an den Host.

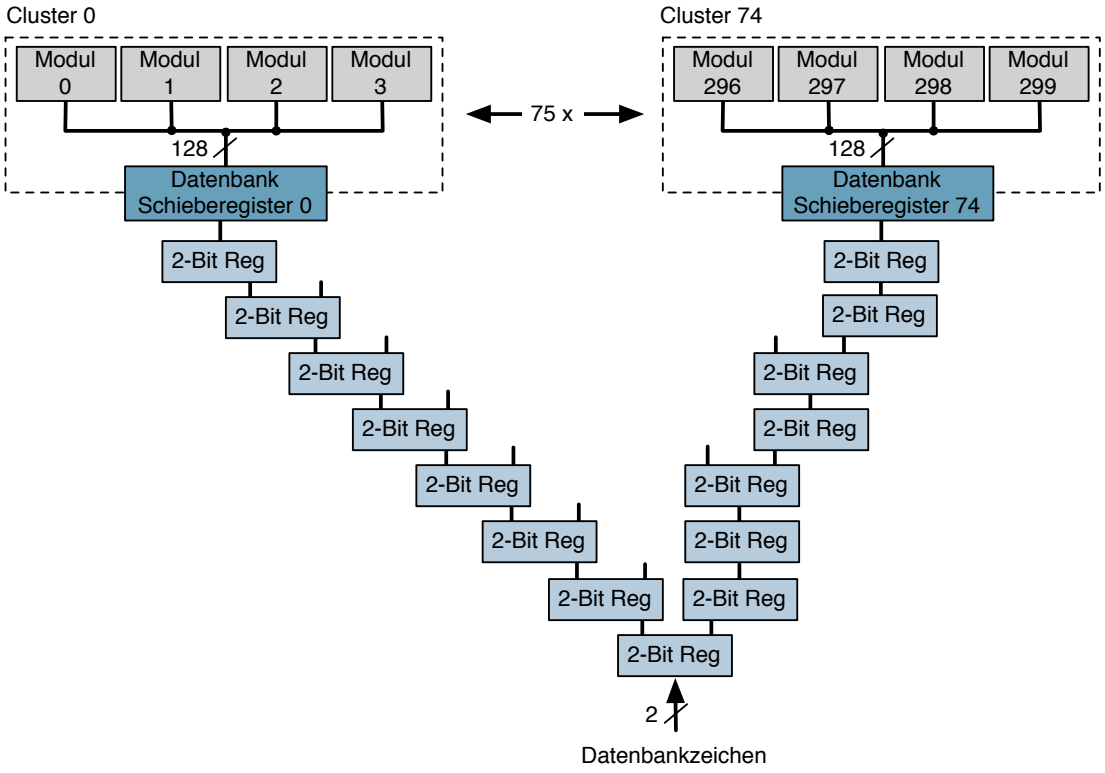


Abbildung 4.6: Verteilung der Datenbankzeichen auf die verteilten Such-Module. Vier Module werden über das gemeinsame Schieberegister zu einem Cluster zusammengefügt.

Der vierte Schritt, die eigentliche Suche, kann durch das Überlauf-Signal unterbrochen werden, um die Speicher zu entleeren und anschließend wieder mit der Suche fortzufahren. Abbildung 4.7 zeigt die wichtigsten Zustände und Zustandsübergänge des Steuerautomaten. Zum besseren Überblick wurden zum Teil Zustände zusammengefasst und nur die wichtigsten Signale dargestellt. Der Automat besitzt zwei unabhängige Zähler für die Anzahl der Einheiten (*unit*) sowie für Datenpakete (*counter*).

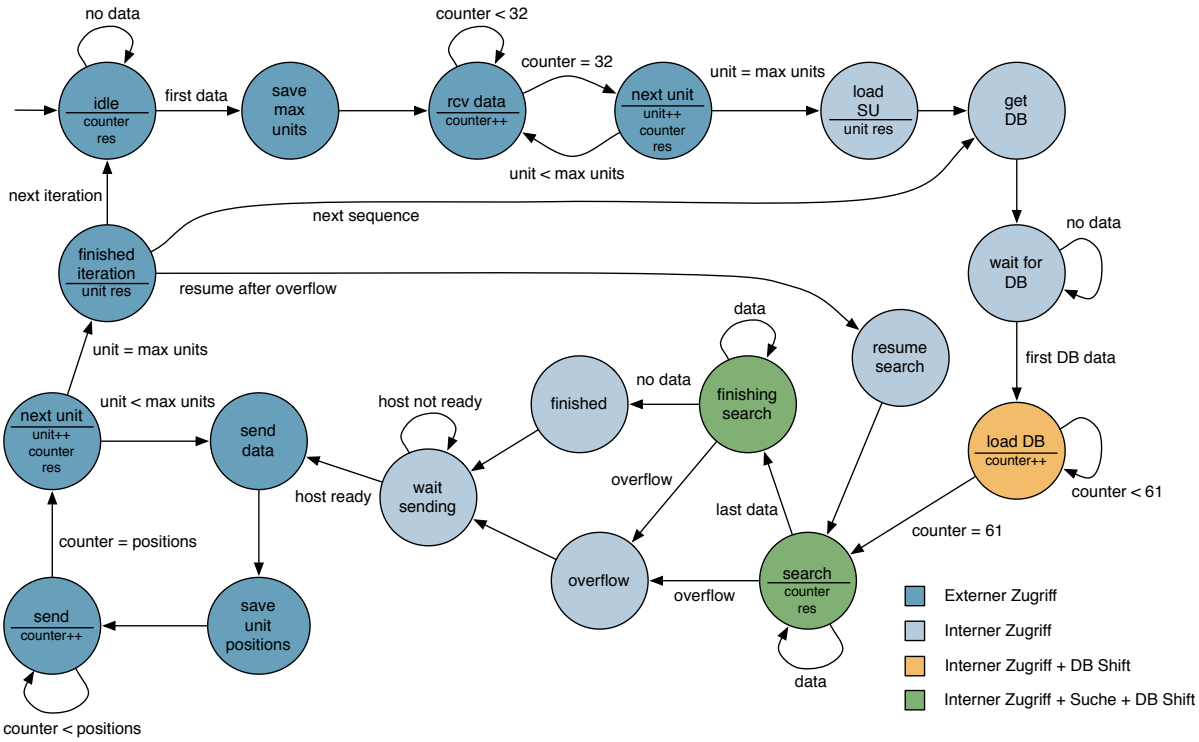


Abbildung 4.7: Zustandsautomat zur Ablaufkontrolle. Zur vereinfachten Darstellung sind Zustände zusammengefasst und nur die wichtigsten Signale und Zustandsübergänge aufgelistet. Ein Reset führt immer in den Startzustand.

Sobald der Ethernet-Controller das Eintreffen des ersten Datenpaketes signalisiert, wird die Anzahl der aktiven Such-Einheiten, die sich auf den niederwertigen Bytes des ersten 4-Byte Wortes befinden, in einem Register gesichert (vgl. Abbildung 4.3(a)). Anschließend werden Blöcke von 32×4 Byte in die einzelnen Bereiche der Block RAMs geladen, bis die maximale Anzahl der Einheiten erreicht ist. Der Multiplexer in den Such-Modulen ist auf externen Datenzugriff geschaltet und die Control-Einheit jedes Moduls inkrementiert automatisch die Adressen. Für den Steuerautomaten erscheinen die unterschiedlichen Speicherbereiche als getrennte Einheiten.

Anschließend werden die Such-Module auf internen Datenzugriff umgeschaltet und laden die Reads aus dem Block RAM in ihre LUTs. Sobald der Ethernet-Controller signalisiert, dass das erste Paket der Datenbank eingetroffen ist, wird zunächst das Schieberegister mit 64 Basenpaaren geladen, bis anschließend die Suche beginnt. Mit jedem Takt wird ein neues Zeichen geladen. Liegen im FIFO für eingehende Datanbankdaten keine Daten vor, wird die Suche kurzzeitig angehalten, bis wieder neue Daten verfügbar sind.

Signalisiert der Ethernet-Controller das Ende der Datenbank, schreiben die Such-Einheiten automatisch die Anzahl der gefundenen Positionen auf die erste Speicherstelle ihres Speicherbereiches (vgl. Abschnitt 4.2.1) und es wird auf externen Datenzugriff umgeschaltet. In den darauf folgenden Zuständen wird von jeder Such-Einheit die Anzahl der Positionen gelesen und die dementsprechende Menge an Daten gesendet, bis die letzte Einheit erreicht wurde.

Am Ende einer Iteration wird in Abhängigkeit des Steuersignales vom Ethernet-Controller entweder auf eine neue Übertragung von Reads oder einen weiteren Durchlauf mit den alten Reads gewartet. Wurde die Suche durch einen Überlauf unterbrochen, signalisiert der Host nach erfolgreicher Entgegennahme der Ergebnisse, dass die Suche fortgesetzt werden kann. Dabei wird mit dem aktuellen Inhalt des Schieberegisters und des Positionszählers weitergearbeitet.

4.2.4 Datenkommunikation

Zur Datenkommunikation zwischen Host und FPGA kommt aufgrund der ausreichenden Datenrate und einfachen Implementierung Gigabit-Ethernet zum Einsatz. Für das Streaming der Datenbank wird in jedem Takt ein neues Basenpaar benötigt. Daraus ergibt sich bei einer Taktrate von 200 MHz und einer Kodierung mit 2 Bit eine notwendige Datenrate von 400 MBit/s, die problemlos erreicht werden kann.

Wie bereits in Abbildung 4.5 gezeigt wurde, stellt der Ethernet-Controller die Daten in unterschiedlichen FIFO-Speichern bereit. Die Kommunikation mit dem Host wird zum größten Teil intern geregelt und vor dem Steuerautomaten verborgen. Nur für den eigentlichen Ablauf notwendige Signale werden an den Steuerautomaten weitergeleitet.

Als Ethernet-Schnittstelle für den Virtex-6 dient der Embedded Tri-Mode Ethernet MAC Wrapper von Xilinx [Xil10a], der eine Kommunikation auf Basis des LocalLink Interfaces [Xil05] mit den internen FIFOs bereitstellt. Der Beginn eines eingehenden Ethernet-Frames wird mit einem *Start of Frame* angekündigt und mit einem *Destination Ready* byteweise entnommen. Das Ende wird mit einem *End of Frame* gekennzeichnet. Das so bereitgestellte Paket ist ein RAW-Ethernet Paket, bestehend aus Quell- und Ziel-MAC-Adresse[3] und Ethernet-Type[4] (vgl. Abbildung 4.8).

Abbildung 4.8: Aufbau eines RAW-Ethernet Paketes mit zusätzlichen Feldern für Steuerinformationen und Paket-ID.

Abbildung 4.9 zeigt die Komponenten des Ethernet-Controllers. Zwei Zustandsautomaten (FSM[5]) regeln jeweils das Verknüpfen und Aufteilen der einzelnen Bestandteile der Pakete. Die Kommunikation erfolgt ausschließlich über die FIFOs, die damit die reale Kommunikation mit Paketen

[3] Media-Access-Control-Adresse
[4] Der Ethernet-Type gibt das eingesetzte Protokoll an. $0x0801$ steht hierbei für das X.75 Protokoll, eine einfache Erweiterung des X.25 Protokolls für einfache Paketvermittlung [BH01].
[5] Finite State Machine

vollständig nach außen hin verbergen.

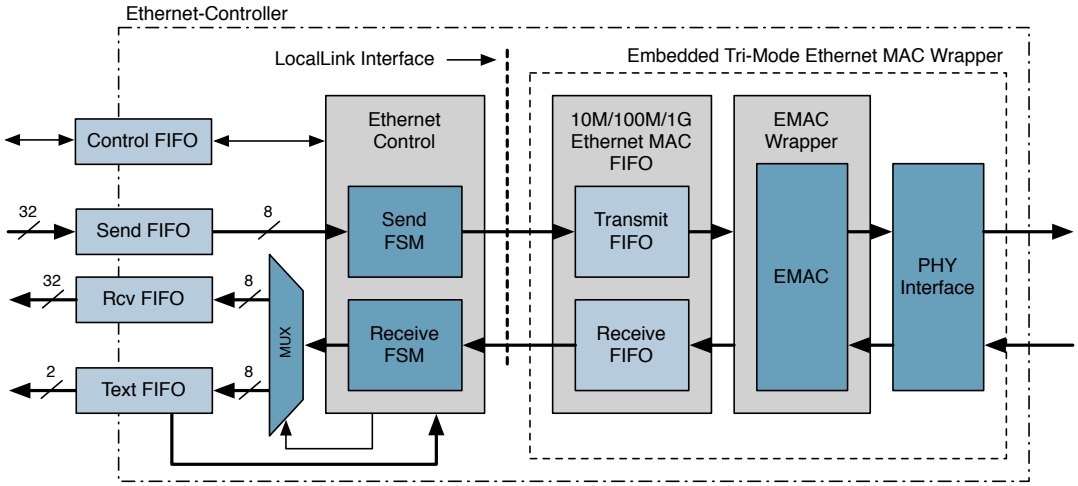


Abbildung 4.9: Aufbau des Ethernet-Controllers mit Wrapper, LocalLink Interface und Zustandsautomaten. Wrapper nach [Xil05].

Die Pakete wurden so erweitert, dass sie zusätzlich eine Steuerinformation und eine Paketnummer enthalten. Um die Paketkommunikation komplett vor den nachfolgenden Komponenten zu verbergen, werden Pakete mit Steuerinformationen auf einfache Signale reduziert und die reinen Daten aus den Datenpaketen in Daten-FIFOs geschoben, auf die einfach zugegriffen werden kann.

Die Steuerinformationen dienen einerseits dazu, direkte Befehle und Statusinformationen zwischen FPGA und Host auszutauschen, kennzeichnen andererseits aber auch reine Datenpakete. Pakete zur reinen Steuerung enthalten nur die 48 Byte Nutzdaten die für die minimale Paketlänge notwendig sind. Abbildung 4.10 zeigt den vollständigen Kommunikationsablauf zwischen Host und FPGA eines kompletten Durchlaufes mit den unterschiedlichen Steuerinformationen.

Daten können schneller in den Block RAM geschrieben, als vom Ethernet bereitgestellt werden. Durch die Taktrate des Ethernet-Controllers von 125 MHz und der Datenbreite von genau einem Byte beträgt die maximale Datenrate 1.000 MBit/s. Da die Taktrate für die Entnahme der Daten bei 200 MHz liegt und die maximale Datenmenge nur 75 kByte beträgt, ist keine Flusskontrolle für das Empfangen der Reads notwendig. Die eingehenden Daten werden direkt in den Daten-FIFO geschrieben, aus dem der Steuerautomat sie entnehmen kann. Das Senden von Daten beginnt, indem der Steuerautomat das Vorliegen von Daten im Ausgangs-FIFO signalisiert. Sobald für einen definierten Zeitraum keine Daten mehr in den FIFO geschrieben werden, signalisiert der FPGA dem Host mit einem Steuerpaket das Ende der Übertragung.

4.2.4.1 Streaming der Datenbank

Für das Streaming der Daten ist aufgrund der verminderten Datenrate von 400 MBit/s eine Flusskontrolle notwendig. Diese setzt ein, sobald Datenbankpakete empfangen werden und schreibt die Daten umgehend in den Datenbank FIFO, der zur Entnahme der Daten ein 2-Bit Interface zur

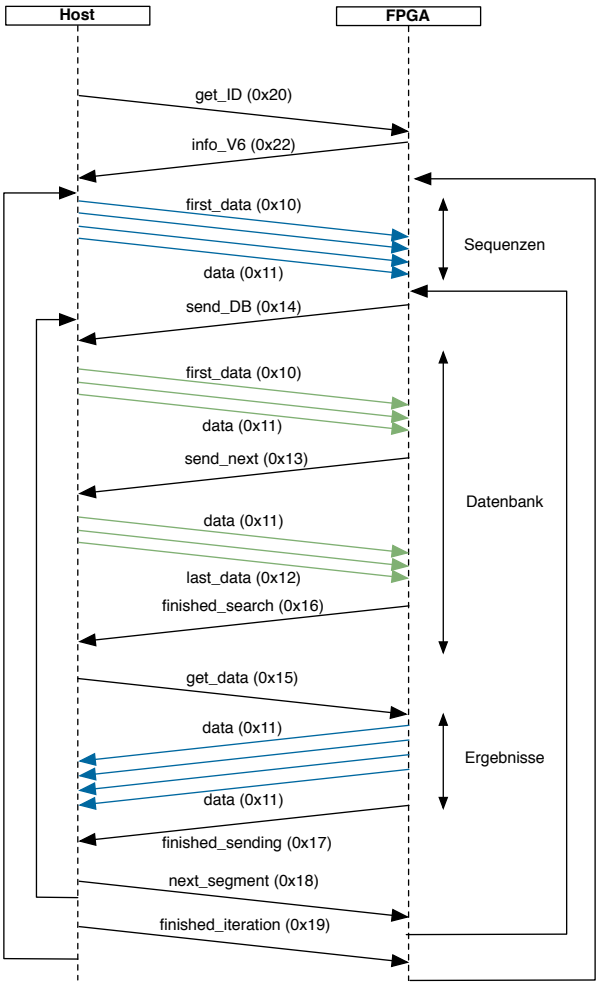


Abbildung 4.10: Übersicht der Kommunikation zwischen Host und FPGA mit Kodierung der Steuerinformationen.

Verfügung stellt. Sobald die Datenrate höher ist, gehen Pakete verloren, ist sie niedriger, muss während der Suche auf Daten gewartet werden und die Laufzeit erhöht sich dementsprechend. Ein einfaches Streaming durch Messung der Datenrate für eine statische Anzahl von Paketen auf Seiten des Hosts und einer daraus dynamisch berechneten Wartezeit garantiert eine Datenrate, die unter 400 MBit/s liegt, ist aber stark abhängig von der Größe der Datenbank und zeigt eine schlechte Leistung bei kleinen Datenbanken (vgl. Abbildung A.3).

Als effizienter hat sich ein Streaming erwiesen, welches nicht statisch eine vorgegebene Anzahl von Paketen sendet und wartet, bis diese verarbeitet sind, sondern adaptiv neue Pakete übermittelt. Dabei sendet der FPGA zu Beginn des Streamings die Anzahl von Bytes, die der Datenbank-FIFO aktuell aufnehmen kann. Diese Anzahl wird vom Host gesendet, der anschließend den Thread schlafen legt. Sobald der FPGA für einen Zeitraum, der der Round-Trip Time entspricht, keine Daten empfangen hat, sendet er erneut ein Paket mit der Anzahl der freien Bytes im FIFO und der Thread beginnt erneut mit dem Senden. So wird ein Streaming ermöglicht, welches im Bereich von 400 MBit/s arbeitet, ohne dass Pakete verloren gehen (vgl. Abbildung A.4).

Diese Art der Flusskontrolle entspricht dem einfachen Kredit-Verfahren ohne Bestätigung des Empfangs.

Das Streaming wird durch den FPGA gesteuert, der die Bandbreite so in Abhängigkeit seiner Verarbeitungsgeschwindigkeit steuert. Wird eine Datenrate von 400 MBit/s erreicht, arbeitet der FPGA die Daten ohne Wartezeit ab und erreicht die maximal mögliche Geschwindigkeit.

4.2.4.2 Fehlerbehandlung

Durch die Paketnummern kann lediglich das Fehlen eines Paketes erkannt werden. Auf Seiten des FPGAs wird dafür gesorgt, dass die Daten schnell genug verarbeitet werden und durch die Flusskontrolle können niemals mehr Pakete an den FPGA gesendet werden, als auch im FIFO Platz haben. Um eine hohe Datenrate und eine geringe Latenz zu ermöglichen, sind FPGA und Host ohne Router oder Switch direkt miteinander verbunden.

Ein Paketverlust oder eine falsche Reihenfoge der Pakete tritt daher in der Regel nicht auf und auf eine entsprechende Reaktion wurde in der aktuellen Version verzichtet. Sollte dennoch ein Paketverlust vom FPGA signalisiert werden, ist ein schwerwiegenderer Fehler aufgetreten und der gesamte FPGA wird über ein Reset-Paket, welches nicht auf eine korrekte Paketnummer überprüft wird, komplett neu gestartet.

4.2.4.3 Einordnung

Die Kommunikation über MAC-Adressen befindet sich auf Schicht 2 (Sicherungsschicht) des OSI[6]-Referenzmodells. Durch die Flusskontrolle zum Streaming der Datenbank und die Möglichkeit, verlorene Pakete zu erkennen, sind Teile von Schicht 4 (Transportschicht) und somit in Teilen ein TCP[7]-Ähnliches Protokoll realisiert worden. [BH01]

[6] Open Systems Interconnection Reference Model
[7] Transmission Control Protocol

4.2.5 Benötigte Hardwareressourcen

Tabelle 4.2 listet die benötigten Ressourcen der einzelnen Komponenten und des vollständigen Designs mit 600 Einheiten für einen Virtex-6 auf. Auf einem Virtex-5 lassen sich 100 Einheiten bei einer Taktrate von 100 MHz realisieren, was einer Leistung von 10 Giga Vergleiche/s entspricht.

Tabelle 4.2: Überblick über die benötigten Hardwareressourcen.

Modul	Suchmodul	Ethernet	Vollständiges Design (600 Einheiten)
Register	165	1.161	76.768 (25 %)
LUTs	405	964	127.921 (85 %)
Slices	162	400	35.362 (93 %)
Unbenutztes Flip Flop	61 %	24 %	47 %
Unbenutzte LUT	3 %	22 %	3 %
Benutzte LUT-FF Paare	35 %	53 %	48 %
Block-RAM	1	20	320 (76 %)
Maximale Taktrate			200 MHz
Vergleiche/s 64 bp			120 Giga
Erforderliche Datenrate			400 MBit/s

4.2.6 Host-Schnittstelle

Die Software auf Seiten des Hosts hat die Aufgabe, Datenbank und Reads in die für den FPGA notwendige Kodierung umzuwandeln und den in Abbildung 4.10 gezeigten Kommunikationsablauf mit dem FPGA zu ermöglichen. Daraus ergibt sich für einen kompletten Durchlauf einer Suche folgender Ablauf, der durch die Software ermöglicht werden muss:

1. Transformieren der Datenbank,
2. Aufbau einer Kommunikation mit dem FPGA,
3. Transformation von 600 Reads,
4. Übertragen der Reads auf den FPGA,
5. Beginn der Suche, die sich unterteilt in:
   a) Streamen einer Datenbanksequenz,
   b) Reagieren auf ein Overflow,
   c) Empfangen und Sichern der Ergebnisse und
   d) Erneutem Streaming, solange noch Sequenzen vorhanden sind.
6. Sind weitere Reads vorhanden, fortfahren mit Schritt drei, andernfalls
7. Ausgabe von statistischen Daten nach erfolgreicher Suche aller Reads.

Eine Gruppe von 600 Reads wird dabei nur einmal geladen, um anschließend die komplette Datenbank mit sämtlichen Sequenzen zu streamen. So muss das rechenintensive Transformieren der Reads und deren Übertragung nur ein Mal durchgeführt werden. Neben den Reads müssen auch die zusätzliche Informationen, wie die Anzahl der erlaubten Mismatches und die Anzahl der aktiven Einheiten, übertragen werden.

In der gegenwärtigen Version wird für sämtliche Reads eine einheitliche Zahl von tolerierten Mismatches übertragen. Mit einer Erweiterung des Eingabeformates wäre aber auch eine spezifische Zahl für jeden Read möglich, da auf Seiten des FPGAs in jeder Einheit mit unterschiedlichen Werten gearbeitet werden kann.

Ein wichtiger Schritt der Nachbearbeitung ist das Herausfiltern ungültiger Positionen in der Datenbank. Zur Kodierung eines weiteren Zeichens müsste die Kodierung der Datenbankzeichen auf 3 Bit erhöht werden, was zu Problemen bei der erreichbaren Datenrate und des Weiteren zu einem größeren Hardwareaufwand führen würde.

In der aktuellen Version werden ungültige Datenbankbereiche durch eine entsprechende Anzahl des Nukleotides Adenin ersetzt, wodurch die nachfolgenden Positionsdaten erhalten bleiben. Da Reads üblicherweise nicht komplett aus Adenin-Nukleotiden bestehen, wird kein Treffer aus dem unbekannten Bereich ausgegeben. Falls es gerade an den Rändern zu Treffern kommen sollte, werden diese üblicherweise durch die nachfolgenden Softwareschritte herausgefiltert.

4.2.6.1 Gebrauchstauglichkeit

Eine der wichtigsten Anforderungen aus Abschnitt 4.1 besteht darin, dass das System von typischen Nutzern bedienbar und konfigurierbar sein muss. Um dieser Forderung nachzukommen, wurde ein einfaches Kommandozeilen Programm entwickelt, welches über ähnliche Parameter und Einstellungen verfügt wie übliche Short-Read Mapper und mit den selben Datenformaten arbeitet.

Tabelle 4.3 gibt eine Übersicht über die implementierten Programmparameter. Es wurde weitestgehend versucht, unzulässige Kombinationen und fehlende Eingaben abzufangen. Es besteht die Möglichkeit, eine Datenbank im Vorfeld zu transformieren und anschließend immer mit der binären Datenbank zu arbeiten, da die Transformation unabhängig von den Reads ist.

Im Verzeichnis des ausführbaren Programms befindet sich die Datei `mac.config`, die die MAC-Adressen von Host, FPGA und den Namen des Netzwerkinterfaces enthält, um eine einfache dauerhafte Konfiguration zu ermöglichen.

Die folgenden Teilabschnitte gehen auf spezielle Teile des Programms und die Dateiformate ein. Eine Auswertung der Funktion der Software in Verbindung mit dem FPGA erfolgt schließlich in Kapitel 5.

Tabelle 4.3: Übergabeparameter und Optionen für das Host-Programm des FPGAs.

Parameter	Kurzform	Beschreibung
`--query <filename>`	`-q`	Datei mit Reads im FASTA- oder FASTQ-Format
`--database <filename>`	`-d`	Datenbank im FASTA-Format
`--bindb <filename>`	`-b`	Bereits transformierte binäre Datenbank
`--output <filename>`	`-o`	Ausgabe in Datei
`--sam`	`-s`	Ausgabe im SAM-Format
`--unmap`	`-u`	Zusätzliche Ausgabe einer Übersicht von gefundenen und nicht gefundenen Reads
`--transform`	`-t`	Nur Transformation der ASCII-Datenbank
`--mismatch [int]`	`-m`	Anzahl erlaubter Mismatches
`--status`	`-i`	Statusinformationen auf Konsole ausgeben
`--help`	`-h`	Hilfetext anzeigen

4.2.6.2 Transformation der Datenbank

Die vier Zeichen in den Nukleotid Datenbanken sind üblicherweise im ASCII-Format kodiert. Um eine einfache Verarbeitung auf dem FPGA zu realisieren und die erforderliche Datenrate zu reduzieren, bietet sich eine 2-Bit Kodierung an (siehe Tabelle 4.1). Über einfache Bitoperationen werden 4 Basenpaare mit genau einem Byte kodiert (vgl. Listing 7.1).

Die Transformation der Datenbank ist nur einmalig notwendig und daher ein vollkommen von der eigentlichen Suche unabhängiger Schritt. Um einen effizienten Programmablauf zu gewährleisten und die Arbeit mit Datenbanken zu ermöglichen, die mehrere Sequenzen enthalten, sind zusätzliche Informationen notwendig. Diese werden in einer separaten Datei gespeichert, um für spätere Suchen bereitzustehen. Abbildung 4.11 zeigt die beiden Dateien, die bei der Transformation entstehen.

Die `.info`-Datei enthält die Anzahl der Sequenzen und der gesamten Basenpaare. Zu jeder Sequenz beinhaltet die Datei die Startposition und die Anzahl der Basenpaare. Für Datenbanken mit vielen kurzen Sequenzen besteht hier der Vorteil, dass mehrere Sequenzen fortlaufend ohne Unterbrechung der Suche übertragen werden können und anschließend die Positionen für die einzelnen Sequenzen angepasst werden. Da die Datenbanken aber in der Regel nur große Sequenzen enthalten, ist diese Option in der aktuellen Programmversion nicht vollständig implementiert.

4.2.6.3 Transformation der Reads

Zur Transformation der Reads werden jeweils zwei Basenpaare durch die Anzahl von Mismatches mit der entsprechenden Speicherstelle (Datenbankzeichen) innerhalb des LUT-RAMs ersetzt, wie bereits in Abschnitt 4.2.1 erläutert. Dieser Vorgang wird durch einfache Tabellen realisiert, welche die Kodierung der entsprechenden Basenpaare enthalten (vgl. Listing 7.2 und 7.3). Zwei Basenpaare werden auf diese Weise mit genau 32 Bit kodiert und in einer Zeile abgespeichert.

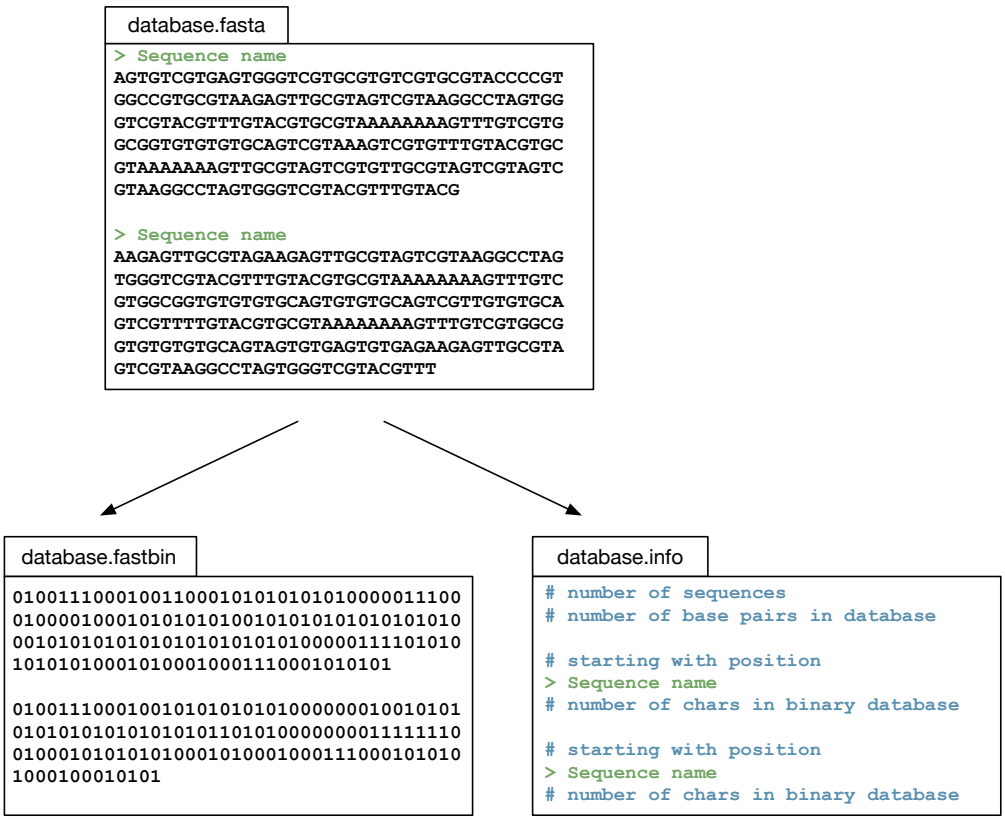


Abbildung 4.11: Übersicht der aus der Datenbank generierten Datenformate.

Ein Read kann aus maximal 64 Basenpaaren bestehen, sodass 32 Zeilen notwendig sind. Für jeden Read entsteht eine aus 4×4 Bytes bestehende Matrix. Ist ein Read kürzer, werden die verbleibenden Zeilen mit Nullen aufgefüllt, sodass für die nicht vorhandenen Zeichen mit beliebigen Datenbankzeichen kein Mismatch ausgegeben wird. Somit sind ungültige Basenpaare ebenso möglich wie solche, die niemals einen Mismatch verursachen.

Anschließend muss die Matrix noch bitweise transponiert werden, um ein paralleles serielles Laden der LUTs zu ermöglichen (vgl. Abschnitt 4.2.1). Dieser Schritt wird über einfache Bitoperationen realisiert.

4.2.7 Datenformate

Im Folgenden werden die unterschiedlichen Datenformate erläutert, die zum Einsatz kommen. Hierbei ist eine Gliederung in Eingabe-, Zwischen- und Ausgabeformate möglich.

4.2.7.1 Eingabeformate

Als Eingabeformate für die Reads und die Datenbank sind Dateien im FASTA-Format möglich (vgl. Abschnitt 2.1.2). Die Datei, welche die Datenbank enthält, darf hierbei unterschiedliche,

durch Sequenznamen getrennte, Sequenzen enthalten. In der Datei für die Reads müssen die einzelnen Sequenzen dabei mit einem Kommentar beginnen. Die Kodierung der Basenpaare erfolgt mit ASCII-Zeichen.

4.2.7.2 Zwischenformate

Die notwendigen Zwischenformate sind die ins binäre transformierte Datenbank und die Datei mit den zusätzlichen Informationen zur binären Datenbank, die beide bei der Transformation erzeugt werden (vgl. Abschnitt 4.2.6.2).

4.2.7.3 Ausgabeformate

Das bei Short-Read Mappern gebräuchliche SAM-Format (vgl. Abschnitt 2.3.2) wird weitestgehend unterstützt. Dabei wird für jede Position genau eine Zeile in die Ausgabedatei geschrieben.

Daneben gibt es noch ein weiteres Ausgabeformat, welches speziell die Informationen enthält, welche der Host-Software beim Auslesen der Ergebnisse vom FPGA zur Verfügung stehen und dadurch mit weniger Rechenleistung generiert werden kann. Im Prinzip und Aufbau ähnelt es dem SAM-Format und wird daher als PAM bezeichnet (Endung `.pam`). Die Zählung der Positionen beginnt wie beim SAM-Format mit Eins (*1-based coordinate system*).

Daneben ist es möglich, die Reads, für die Positionen gefunden wurden, in eine `.map` und diejenigen Reads, die nicht in der Datenbank gefunden wurden, in eine `.unmap`-Datei zu schreiben. Diese beiden Dateien enthalten für jeden Read spezielle Zusatzinformationen. Bei gefundenen Reads werden für jeden Read die Anzahl von gefundenen Positionen und die minimale Anzahl der Mismatches aufgelistet. Bei nicht gefundenen Reads wird die Zahl der minimal notwendigen Mismatches aufgelistet. Keiner der anderen Short-Read Mapper kann eine derartige Information erzeugen.

4.3 Umsetzung auf der GPU

Im folgenden wird der Entwurf aus Abschnitt 3.4.2 hinsichtlich der zuvor in Abschnitt 4.1 erweiterten Anforderungen detailliert ausgearbeitet. Als erstes wird in Abschnitt 4.3.1 der reduzierte Smith & Waterman-Algorithmus vorgestellt und anschließend in Abschnitt 4.3.2 mittels CUDA parallelisiert. In Abschnitt 4.3.3 wird das CUDA-Programm optimiert.

4.3.1 Anpassung des Algorithmus

4.3.1.1 Vollständiger Smith & Waterman-Algorithmus

Der zu entwerfende Smith & Waterman-Algorithmus mit aufgelösten Abhängigkeiten orientiert sich an Abbildung 3.2(c). Dabei wird die Matrix auf drei diagonale Spalten reduziert, die dann zeilenweise berechnet werden. Eine weitere Änderung besteht im sofortigen Sichern einer Position, sobald sie einen zuvor definierten Score ($minScore$) erreicht. Listing 4.1 zeigt den sich ergebenden Algorithmus in Pseudocode.

Listing 4.1: Auf drei Spalten reduzierter Smith & Waterman-Algorithmus mit direkter Berechnung der Positionen.

for $j = (readLength + 1) \rightarrow (dbLength + (readLength - 1))$ **do**
  $positionMatrix[(j \ mod \ 3)] \leftarrow j - readLength - 1$
  for $i = 1 \rightarrow readLength$ **do**
    $actual \leftarrow ((j - i) \ mod \ 3 + (i \cdot 3))$
    $vertical \leftarrow ((j - i) \ mod \ 3 + ((i - 1) \cdot 3))$
    $horizontal \leftarrow ((j - i - 1) \ mod \ 3 + (i \cdot 3))$
    $diagonal \leftarrow ((j - i - 1) \ mod \ 3 + ((i - 1) \cdot 3))$
    if $db[j - i - 1] = read[i - 1]$ **then**
      $diagonalScore \leftarrow scoreMatrix[diagonal] + matchScore$
    else
      $diagonalScore \leftarrow scoreMatrix[diagonal] + misScore$
    end if
    $verticalScore \leftarrow scoreMatrix[virtical] - gapScore$
    $horizontalScore \leftarrow scoreMatrix[horizontal] - gapScore$
    $writeMaxScoreAndPosition(diagonalScore, verticalScore, horizontalScore)$
    $i \leftarrow i + 1$
  end for

  if $(scoreMatrix[actual] \geq minScore)$ **then**
    if $(positionMatrix[actual] \neq positionMatrix[horizontal])$ **then**
      print $positionMatrix[actual]$
    end if
  end if
  $j \leftarrow j + 1$
end for

Wie im original Smith & Waterman-Algorithmus sind zwei Schleifen notwendig. Die äußere Schleife durchläuft die Datenbank, die innere durchläuft den Read. Durch die diagonale Verschiebung müssen zu Beginn vor und hinter die Datenbank leere Sequenzen in der Länge des Reads angehangen werden. Die Berechnung des ersten Zeichens des Reads beginnt um $readLength + 1$ Zeichen versetzt (vgl. Abbildung 3.2(c)).

Zu Beginn der Berechnung müssen beide Matrizen mit Nullen initialisiert werden. Die eigentliche Matrix, in die die Scores geschrieben werden, beginnt wiederum in der zweiten Zeile (vgl. Abschnitt 2.2.1.3). Auf diese Weise wird eine Spaltenweise Berechnung realisiert. Die diagonale Verschiebung erfolgt über eine Indexberechnung, welche in der inneren Schleife gleich zu Beginn durchgeführt wird. Dabei ist die Modulo-Rechnung notwendig, um die Matrizen auf drei Spalten zu reduzieren.

Die Berechnungen der unterschiedlichen Scores für ein Feld erfolgt nach Gleichung 2.6 aus Abschnitt 2.2.1.3, wobei hier nicht die Richtung in einer zweiten Matrix gespeichert wird, sondern die Position, an der das aktuelle Alignment begonnen hat. Um das zu erreichen wird vor Beginn der inneren Schleife bei einem Match die aktuelle Position in die Positionsmatrix geschrieben und danach aus der Richtung des maximalen Scores übernommen (im Algorithmus nicht dargestellt).

Im letzten Teil des Algorithmus wird die Position gesichert, wenn sie über einem zuvor definierten Score liegt und sich von der vorherigen unterscheidet. So wird vermieden, das die selbe Position durch das Einfügen von Gaps am hinteren Ende des Reads mehrmals ausgegeben wird.

4.3.1.2 Reduzierter Smith & Waterman-Algorithmus

Um die Implementierung an das Short-Read Mapping Problem anzupassen, sind nach der Analyse der Anforderungen aus Abschnitt 3.1 keine Gaps notwendig, sodass eine weitere Vereinfachung des Algorithmus möglich ist. Diese Vereinfachung dient jedoch lediglich als Zusatzoption für eine zusätzliche Steigerung der Geschwindigkeit.

Werden keine Gaps zugelassen, können sämtliche horizontalen und vertikalen Zugriffe auf die Matrix ebenso entfallen, wie die Überprüfung auf den maximalen Score. Auf das Mitführen der Positionen in der zweiten Matrix kann ebenfalls verzichtet werden, da sich die Anfangsposition aus der Endposition des Reads ergibt.

4.3.2 Parallelisierung des Smith & Waterman-Algorithmus

4.3.2.1 Überführung des Algorithmus in einen parallelen CUDA-Kernel

In dem zuvor betrachteten Algorithmus wurden bereits sämtliche Abhängigkeiten durch die zeilenweise diagonal verschobene Berechnung der Matrix aufgelöst. Eine Parallelisierung mittels CUDA erfolgt wie in Abbildung 3.2(c) gezeigt, indem jeder Thread genau ein Zeichen des Reads berechnet. Ein Read wird immer von einem CUDA-Block abgearbeitet, der sich wiederum aus den einzelnen Threads zusammensetzt.

Um für beliebige Readlängen eine optimale Abarbeitung zu gewährleisten, wird ein Block für einen Read bis zu einer Länge von 32 immer mit Threads von der Anzahl der nächst größeren Zweierpotenz abgearbeitet. Für Reads mit einer höheren Anzahl Zeichen entspricht die Zahl der Threads einem Vielfachen von 32, da die Größe eines Warps[8] 32 entspricht. Die Länge der Reads wird dem entsprechen erweitert, indem ungültige Zeichen angehangen und der erforderliche Schwellwert angepasst wird.

Um eine Abarbeitung eines langen Reads mit wenigen Threads zu ermöglichen, ist eine Verteilung von mehreren Basenpaaren des Reads auf einen Thread notwendig. Dazu wird der Read in *Chunks* unterteilt, wobei bei kurzen Reads immer ein Thread ein Zeichen des Reads verarbeitet. Listing 4.2 zeigt die Erweiterung des in Abschnitt 4.3.1 entworfenen Algorithmus hin zu einem parallelen CUDA-Kernel.

Listing 4.2: Parallelisierung des Smith & Waterman-Algorithmus aus Listing 4.1.

```
Require: readLength mod 32 == 0
  threadID    ← threadIdx.x
  blockID     ← blockIdx.x
  matrixBlock ← blockID · matrixSize
  chunk       ← readLength/blockDim

  for j = (readLength + 1) → (dbLength + (readLength − 1)) do
    positionMatrix[(j mod 3) + matrixBlock] ← j − readLength − 1

    for i = ((threadID · chunk) + 1) → ((threadID + 1) · chunk) do
      actual     ← ((j − i) mod 3     + (i · 3))       + matrixBlock
      vertical   ← ((j − i) mod 3     + ((i − 1) · 3)) + matrixBlock
      horizontal ← ((j − i − 1) mod 3 + (i · 3))       + matrixBlock
      diagonal   ← ((j − i − 1) mod 3 + ((i − 1) · 3)) + matrixBlock

      if db[j − i − 1] = (read[i − 1]+(blockID · readLength)) then
          ⋮
      end if
        ⋮
      i ← i + 1
    end for

    syncthreads()
    if (scoreMatrix[actual] ≥ minScore) then
        ⋮
    end if
    syncthreads()
    j ← j + 1
  end for
```

[8] Ein Warp ist eine Gruppe von Threads, die immer zusammen innerhalb eines Multiprozessors abgearbeitet wird. Weicht die Anzahl von einem Vielfachen von 32 ab, wird das Scheduling erschwert.

Die innere Schleife kann mit einer beliebigen Zahl von Threads abgearbeitet werden. Die gesamte Zahl der Threads ergibt sich aus der Blockgröße, mit der der Kernel gestartet wurde. Da die Threads fortlaufende Thread-IDs besitzen, wird jedem Thread ein bestimmter Bereich oder ein einzelnes Zeichen des Reads zugeteilt. Durch die feste Zuordnung der Threads zu einem Threadblock ist auch die parallele Abarbeitung unterschiedlicher Reads möglich. Jeder Block arbeitet durch seine Block-ID auf anderen Matrizen und mit einem anderen Read.

Die Synchronisation der Threads muss vor und nach dem Schreiben der gefundenen Position erfolgen, da es sonst aufgrund der unterschiedlichen Laufzeit durch die bedingten Programmteile zu einer ungleichen Lastverteilung kommt, die die Abarbeitung ausbremst und unter Umständen zu falschen Ergebnissen führen kann.

Der CUDA-Kernel kann ebenfalls auf Mismatches reduziert und so erheblich vereinfacht werden, wie schon in Abschnitt 4.3.1.2 beschrieben. Um eine hohe Leistungs zu erreichen wurde ein Kernel mit und einer ohne Gaps implementiert.

4.3.2.2 Erweiterung zum vollständigen CUDA-Programm

Das CUDA-Programm, in welches die beiden Kernel eingebettet werden, gliedert sich in die Teilschritte

1. Berechnung der notwendigen Parameter in Abhängigkeit der Readlänge,
2. Übertragung der Datenbanksequenz und der Reads auf die GPU,
3. Starten einer der beiden Kernel,
4. Übertragen der Ergebnisse an das Host-System,
5. Ausgabe der Ergebnisse in einem lesbaren Format und
6. Sind weitere Reads oder Datenbanksequenzen vorhanden, fortfahren mit Schritt drei.

Es sind mehrere Durchläufe notwendig, da aufgrund des begrenzten Grafikspeichers und der begrenzten Anzahl der Blöcke in Abhängigkeit der Grafikkarte eine unterschiedliche Anzahl von Reads abgearbeitet werden kann. Im ersten Schritt wird auf dem Host-System die erforderliche Zahl von Threads eines Blocks für die Reads ermittelt. Alle Blöcke haben dabei die selbe Anzahl von Threads. Welche Anzahl an Threads optimal ist, wird in Abschnitt 4.3.3 untersucht.

Die maximale Zahl der Blöcke ergibt sich aus der Speichergröße, der Datenbankgröße, der Anzahl der Reads und der durchschnittlichen Menge der zu erwartenden Ergebnisse und wird zu Beginn berechnet, um eine optimale Auslastung zu gewährleisten.

Anschließend folgt die Übertragung der Daten an den Grafikspeicher. In Abhängigkeit davon, ob der Benutzer eine Suche mit oder ohne Gaps wünscht, wird der entsprechende Kernel mit den zuvor ermittelten Parametern gestartet. Abbildung 4.12 zeigt einen Überblick über den Programmaufbau und einen Kernelaufruf. Es folgt das Einsammeln der Ergebnisse und die Ausgabe.

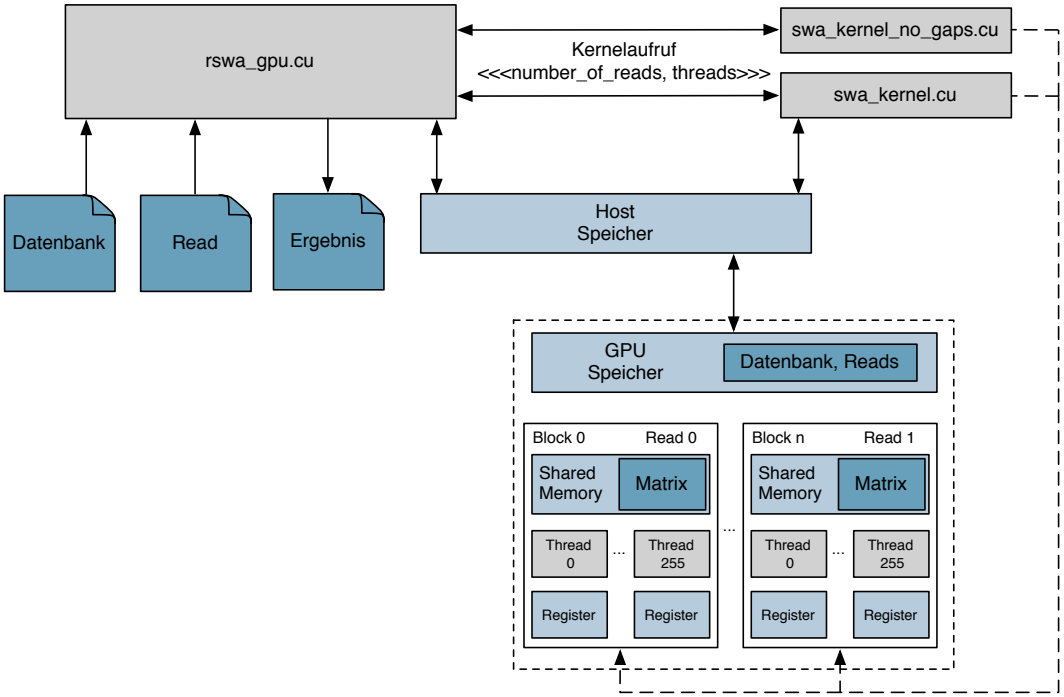


Abbildung 4.12: Aufbau des CUDA Programms mit zwei unterschiedlichen Kerneln (mit und ohne Gaps), deren Aufrufe und die Aufteilung der Daten auf die internen Speicher.

4.3.2.3 Gebrauchstauglichkeit

Eine der wichtigsten Anforderungen aus Abschnitt 4.1 besteht darin, dass das Programm wie auch die FPGA-Implementierung von typischen Nutzern bedienbar sein muss. Tabelle 4.4 zeigt eine Auflistung der Kommandozeilen Übergabeparameter. Durch die Möglichkeit einer Suche mit Gaps und keiner Begrenzung der Länge der Reads ist das Programm vielseitig einsetzbar und nicht an das Short-Read Mapping Problem gebunden.

Sind keine Gaps zugelassen, wird mit dem Parameter *MisGaps* die Zahl der zulässigen Mismatches festgelegt. Sind Gaps zugelassen, muss der Parameter als Abweichung vom maximal möglichen Score angesehen werden.

Die Ein- und Ausgabeformate entsprechen denen der FPGA-Implementierung, die in Abschnitt 4.2.7 erläutert werden, wobei aber keine Zwischenformate erforderlich sind.

4.3.3 Optimierung und Anpassung an die Hardware

Die bisher vorgestellte Implementierung wurde noch nicht hinsichtlich der verschiedenen internen Speicher der GPU optimiert und die Anzahl der Threads ist aufgrund der Größe eines Warps auf ein Vielfaches von 32 ohne obere Grenze festgelegt. Abbildung 4.13 zeigt den Speedup für unterschiedlich lange Reads mit einer steigenden Zahl von Threads. Hat der Read mehr Basenpaare als Threads zur Verfügung stehen, wird die Last automatisch auf die Threads verteilt.

Tabelle 4.4: Übergabeparameter und Optionen für den reduzierten Smith & Waterman-Algorithmus auf der GPU.

Parameter	Kurzform	Beschreibung
`--query <filename>`	`-q`	Datei mit Reads im FASTA- oder FASTQ-Format
`--length <filename>`	`-l`	Maximale Länge der Reads (max: 512)
`--database <filename>`	`-d`	Datenbank im FASTA-Format
`--mismatch [int]`	`-m`	Anzahl erlaubter *MisGaps*
`--gaps`	`-g`	Zulassen von Gaps
`--output <filename>`	`-o`	Ausgabedatei
`--sam`	`-s`	Ausgabe im SAM-Format
`--unmap`	`-u`	Zusätzliche Ausgabe einer Übersicht von gefundenen und nicht gefundenen Reads
`--gpuinfo`	`-i`	Informationen zur vorhandenen GPU
`--help`	`-h`	Hilfetext anzeigen

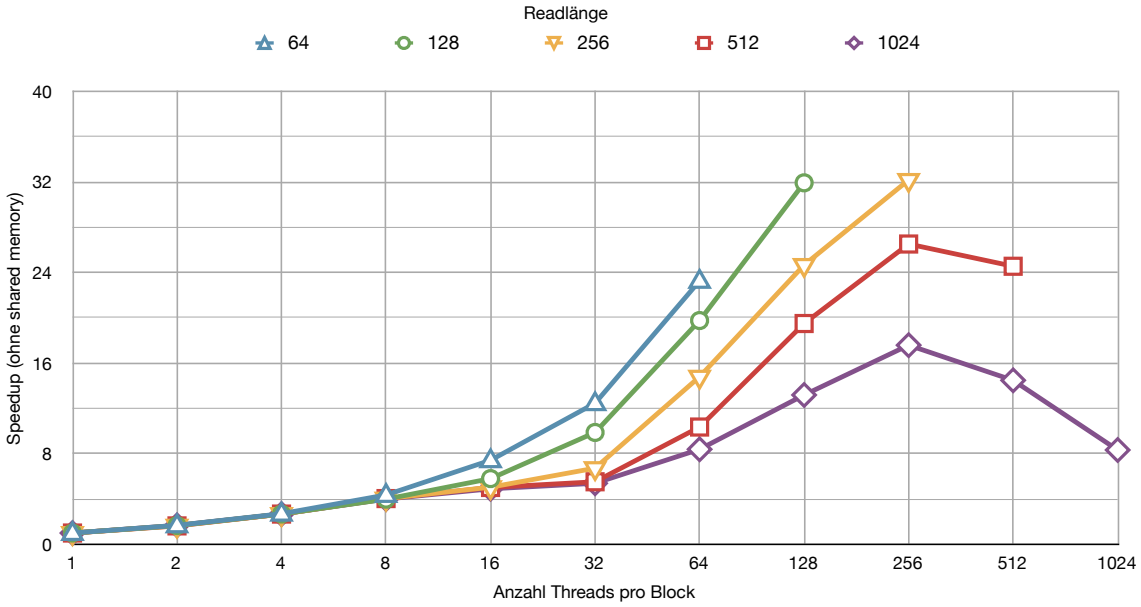


Abbildung 4.13: Erreichter Speedup in Abhängigkeit der Anzahl der Threads.

Die Messung zeigt, dass Reads nicht mit einer höheren Anzahl von Threads als Basenpaare vorhanden sind abgearbeitet werden sollten und dass ein maximaler Speedup bei einer Anzahl von 256 Threads erreicht wird. Selbst bei Reads mit 1.024 Basenpaaren liegt das Optimum bei 256 Threads, danach bricht die Leistung ein. Grund dafür ist ein aufwändigeres Scheduling und die geringere Zahl von internen Registern, die den Threads zur Verfügung stehen [SK10]. Um eine maximale Leistung zu erreichen, werden nie mehr als 256 Threads innerhalb eines Blockes genutzt.

Weitere Ansätze zur Optimierung bestehen in den unterschiedlichen internen Speichern. Abbildung 4.14 zeigt den erreichbaren Speedup im Vergleich zur einfachen Version mit nur einem Thread. Ein erheblicher Speedup wird erreicht, wenn die Matrizen nicht im globalen Speicher, sondern im internen Shared Memory eines Thread Blocks gehalten werden.

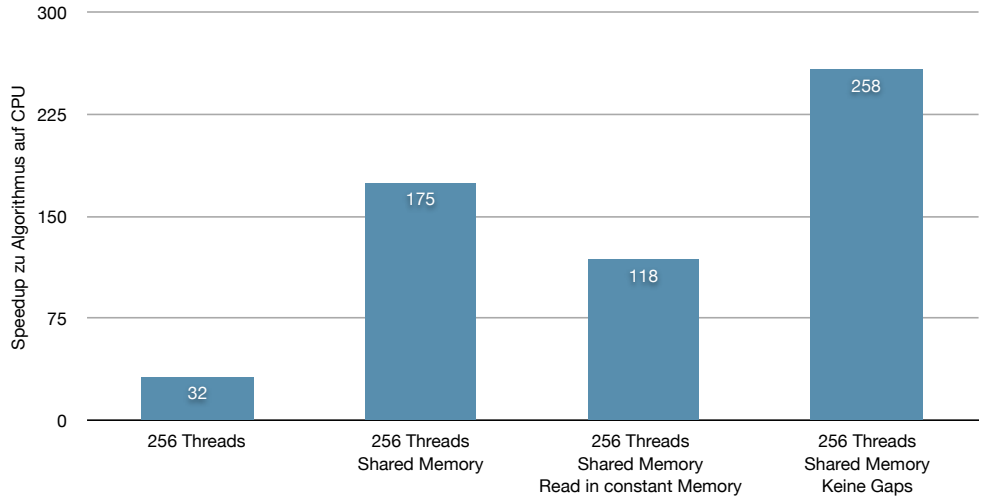


Abbildung 4.14: Erreichter Speedup durch unterschiedliche Optimierungsschritte.

Eine weitere Optimierung mit Textur- oder Konstantenspeicher für den Read brachte keinen weiteren Geschwindigkeitsgewinn, sondern eine Verringerung des Speedups. Der Grund dafür ist, dass Daten, welche sich in diesen Speichern befinden, immer von dort gelesen und nicht in internen Registern der Threads gehalten werden. Liegt der Read aber im globalen Speicher, behält bei kurzen Sequenzen jeder Thread sein Basenpaar in einem lokalen Register und benötigt so eine geringe Zugriffszeit.

Der Ausschluss von Gaps führt zu einer weiteren deutlichen Erhöhung der Geschwindigkeit und ist daher eine Option, die für kurze Reads eingesetzt werden sollte.

Ein weiterer Ansatzpunkt besteht in der Optimierung des Zugriffes auf die Datenbank. Aufgrund ihrer Größe kann sie aber nicht im Konstanten- oder im Texturspeicher gehalten werden. Ein teilweises Übertragen der Datenbank in diese Speicher ist nur von Seiten des Hosts mit einem anschließenden erneuten Kernelaufruf möglich, der aber einen hohen Overhead verursacht (vgl.

Abbildung A.12). Daher ergibt sich eine Verteilung der Daten wie bereits in Abbildung 4.12 gezeigt. Matrizen befinden sich im Shared Memory des Multiprozessors, Datenbank und Reads im globalen Grafikspeicher, wobei die Reads während der Suche in den lokalen Registern der Threads gehalten werden.

Weitere Messungen haben gezeigt, dass durch den Zugriff mehrerer paralleler Blöcke auf das selbe Datenbankzeichen aufgrund der hohen Bandbreite zum Grafikspeicher keine Engpässe entstehen, da ein Block die selbe Laufzeit benötigt wie 16 Blöcke (vgl. Abbildung A.12). Eine Verwendung von CUDA-Streams, welche während der Ausführung eines Kernels das Übertragen der Daten vom und zum Host ermöglichen, ist aufgrund des Verhältnisses von Übertragungszeit zu Suchzeit nicht erforderlich, um die Laufzeit weiter zu reduzieren (vgl. Abbildung A.10).

5 Test, Ergebnisse und Vergleich

Dieses Kapitel beschäftigt sich mit dem Funktionsnachweis und dem Vergleich der beiden in Kapitel 4 implementierten Systeme. Neben einfacher Funktionstests dienen die verschiedenen Testdaten ebenfalls dazu, das Verhalten und die Leistungsfähigkeit der Systeme, deren Korrektheit und die Zuverlässigkeit zu ermitteln.

Nach einer Erläuterung der Testdaten in Abschnitt 5.1 folgt eine Auswertung und Analyse der Ergebnisse in Abschnitt 5.2. Dabei sind die Analyse des Verhaltens der Systeme sowie die Identifizierung von Engpässen und möglichen Optimierungen von großer Bedeutung. Schließlich folgen in Abschnitt 5.3.1 Vergleichsmessungen gegen leistungsfähige, moderne Software Heuristiken und Short-Read Mapper.

5.1 Testdaten und Testfälle

Die Testdatenbanken, die während der Entwicklung zur Überprüfung der Funktionsweise dienten, werden zufällig mit zuvor definierter Länge generiert. Aus zufälligen Positionen in den Datenbanken werden Reads festgelegter Länge ohne Mismatches kopiert. An die Reads wird als Kommentar die Position aus der Datenbank angehängt, um die später ermittelte Position einfach kontrollieren zu können.

Da durch die zufällige Generierung der Datenbank die Wahrscheinlichkeit sich wiederholender Teilsequenzen gering ist, tritt jeder Read üblicherweise an genau einer Position auf. Für einen Testlauf mit 1.000 Reads sind somit genau 1.000 Positionen in der Datenbank enthalten. Mit diesen Datenbanken und Reads ist eine sichere Kontrolle der Korrektheit der Positionsberechnung und Zuverlässigkeit der Systeme möglich.

Durch gezieltes Einfügen von Mismatches in die Reads ist des Weiteren eine Kontrolle auf inexakte Treffer und deren Positionen möglich. Werden durch eine höhere Zahl der zulässigen Mismatches mehr Positionen gefunden, erfolgt die Kontrolle mit dem jeweils anderen System oder einem Short-Read Mapper.

Für die Vergleichsmessungen der Laufzeiten beider Systeme und unterschiedlicher Short-Read Mapper dienen reale Testdatenbanken mit generierten Reads. Die Reads werden dabei zufällig aus einer Datenbank extrahiert, anschließend wird eine vorgegebene Anzahl an Mismatches an verschiedenen Stellen eingefügt. Die Anzahl der Mismatches wird dabei einer diskreten Gleichverteilung innerhalb eines definierten Bereiches entnommen. Untersuchungen zeigen, dass unter den Reads auch HMRs enthalten sind, wie sie in realen Daten auftreten (vgl. Abbildung A.1).

Die Messergebnisse der Short-Read Mapper entsprechen veröffentlichten Ergebnissen mit realen Reads (vgl. [LD09, LTPS09, WER+09]). Daher kann davon ausgegangen werden, dass die generierten Reads realen Reads ähneln.

Im Folgenden werden spezielle Testdaten und Datenbanken vorgestellt, die notwendig sind, um spezielle Eigenschaften der beiden Systeme im Detail zu testen und zu untersuchen.

5.1.1 Spezielle Testfälle für den FPGA

Durch die Komplexität des Systems ist einerseits die Funktionsweise der einzelnen Einheiten zu überprüfen, andererseits muss aber auch die Korrektheit und Zuverlässigkeit der Datenkommunikation und des Streamings der Datenbanken nachgewiesen werden. Im Detail sind die durchgeführten Tests in Abschnitt C aufgeführt.

Mit Hilfe von Testdatenbanken, die Reads an speziellen Positionen enthalten, kann die korrekte Übertragung untersucht werden.[1] Des Weiteren wurde der Überlauf des Speichers eines Such-Moduls mit einer Datenbank untersucht, die einen Read oft genug enthält, um mehrere Überläufe zu generieren.

5.1.2 Spezielle Testfälle für die GPU

Für die GPU sind deutlich weniger Testfälle zur Überprüfung des Systems notwendig, da beispielsweise keine Datenverbindung auf Fehler untersucht werden muss. Bei der Entwicklung des parallelen Kernels wurden die vollständigen Matrizen berechnet und an den Host übertragen, wo sie mit den auf der CPU berechneten Matrizen verglichen wurden.

Nach korrekter Implementierung des einfachen parallelen Kernels erfolgt die Überprüfung auf die korrekte Funktionsweise mit Hilfe realer Testdaten.

5.2 Ergebnisse und Analysen

Im Folgenden werden die Ergebnisse der Tests der beiden Systeme ausgewertet und analysiert. Es werden Engpässe und Optimierungsansätze aufgezeigt und bewertet. Diese werden in Kapitel 6 für einen Ausblick herangezogen. Die Messungen zum Vergleich mit anderen Systemen erfolgen zuvor in Abschnitt 5.3.1.

5.2.1 FPGA-Implementierung

Der FPGA und die Host-Software haben sämtliche in Abschnitt C aufgelisteten Testfälle und den Test mit der generierten sowie der realen Datenbank bestanden. Im Folgenden wird auf die

[1] Solche speziellen Positionen sind beispielsweise zu Beginn und am Ende der Datenbank sowie zwischen zwei Paketen.

Leistungsmessungen und die dabei entdeckten Engpässe des Systems eingegangen. Aufgrund von Spannungsschwankungen auf dem Entwicklungsboard und den resultierenden Verlust der Programmierung des FPGAs wurden die nachfolgenden Messungen lediglich mit 512 Such-Einheiten durchgeführt. Bei dieser Auslastung des FPGAs tritt das Problem seltener auf.

Im Folgenden wird zunächst das Verhalten bei wachsenden Problemgrößen in Abschnitt 5.2.1.1 und einer stark wachsenden Anzahl der Ergebnisse in Abschnitt 5.2.1.2 analysiert. In Abschnitt 5.2.1.3 wird die Auslastung des Host-Systems untersucht, um zu zeigen, dass der größte Teil des Rechenaufwandes auf dem FPGA anfällt. Anschließend werden in Abschnitt 5.2.1.4 die entdeckten Engpässe aufgelistet und es werden Möglichkeiten der Optimierung erläutert.

5.2.1.1 Untersuchung zur Skalierung

Testmessungen mit einer unterschiedlichen Anzahl von Reads haben gezeigt, dass das System wie erwartet skaliert (vgl. Abbildung A.5). Die Laufzeit verdoppelt sich jeweils durch das Hinzufügen von 512 Reads, da in diesem Fall ein erneutes Streaming der Datenbank notwendig ist. Bei kleinen Datenbanken von wenigen tausend Basenpaaren ist deutlich zu erkennen, dass mit jedem Read die Laufzeit minimal ansteigt, da die zu übertragenden Datenmengen entsprechend anwachsen und der Anteil der Laufzeit, der für die reine Suche benötigt wird sinkt (vgl. Abbildung A.6).

Um durch Überläufe verursachte Geschwindigkeitseinbrüche in längeren Datenbanken zu vermeiden, wurden HMRs im Vorfeld entfernt sowie nur zwei Mismatches zugelassen. Die Messungen zeigen, dass sich die Laufzeit durch zusätzliche Reads kaum ändert. Der Zeitanteil der Suche bleibt nahezu konstant, wenn die Datenbank groß genug ist. Bei üblichen Datenbankgrößen von einer Million Basenpaaren wird der größte Anteil der Laufzeit durch die Suche auf dem FPGA in Anspruch genommen, was einer der wichtigsten Anforderungen aus Abschnitt 3.1 entspricht.

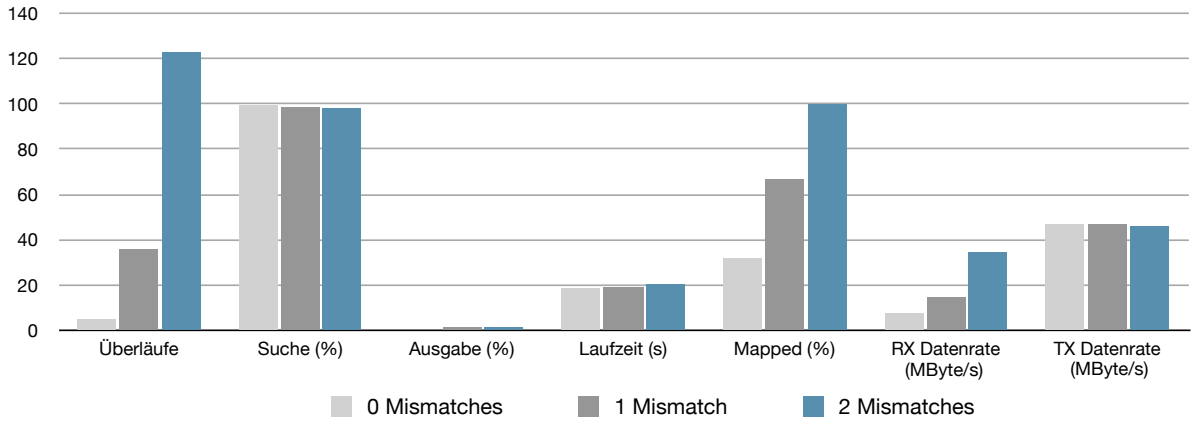
Bei einer hohen Anzahl von Reads verhält sich die Laufzeit mit wachsender Anzahl der Reads durch die stufenförmigen Schritte annähernd linear (vgl. Abbildung A.7). Ebenfalls linear verhält sich die Laufzeit mit steigender Anzahl der Basenpaare in einer Datenbank (vgl. Abbildung A.8). Da bei großen Datenbanken eine Bandbreite von annähernd konstant 400 MBit/s erreicht wird, welche von Seiten des FPGAs gesteuert wird, kann davon ausgegangen werden, dass die Abarbeitung der Datenbank lückenlos erfolgt und der FPGA voll ausgelastet ist (vgl. Abbildung A.4).

Die Laufzeit ist durch die feste Länge der Such-Einheiten für beliebige Längen der Reads gleich. Lediglich die größere Zahl von gefunden Positionen bei kurzen Reads führt zu einer dementsprechend höheren Laufzeit und wird im Folgenden näher betrachtet.

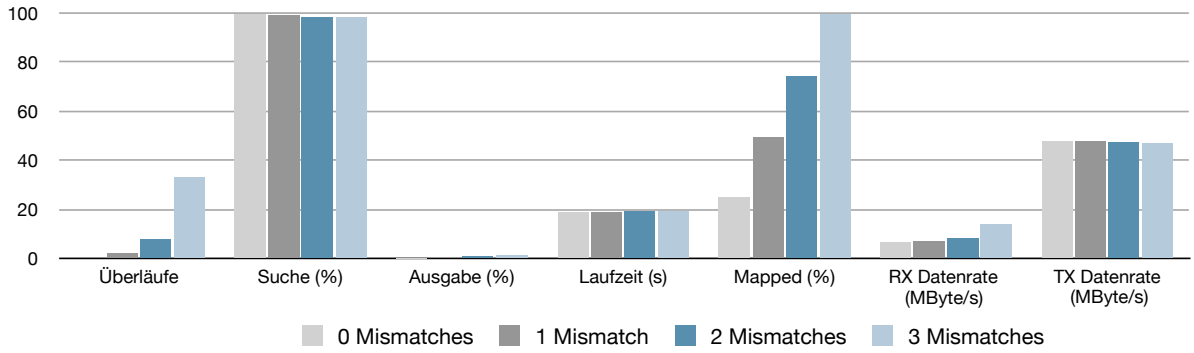
5.2.1.2 Verhalten bei variabler Readlänge und unterschiedlicher Zahl von Mismatches

Bisher wurde lediglich das Verhalten des Systems in Bezug auf die Anzahl der Reads und die Länge der Datenbank analysiert. Dieser Abschnitt beschäftigt sich mit der Untersuchung des Verhaltens bei realen Daten und steigender Anzahl von Positionen durch Veränderung der zulässigen Anzahl Mismatches, welche zu Überläufen der lokalen Speicher führen.

Abbildung 5.1 zeigt die Ergebnisse der Messungen, die an einer möglichst großen Datenbank durchgeführt wurden, um viele Positionen zu generieren. Die Suche wurde sowohl mit kurzen Reads mit einer Länge von 35 Basenpaaren als auch mit langen Reads mit 50 Basenpaaren durchgeführt. Die Zahl der maximalen Mismatches ist bei kurzen Reads auf Zwei und bei langen Reads auf Drei begrenzt, was den gebräuchlichen Suchanfragen entspricht (vgl. Abschnitt 2.3). Gemessen wurden die Überläufe, die gefundenen Positionen, die daraus resultierenden Datenraten[2] sowie die prozentualen Anteile ausgewählter Teile des Programms an der gesamten Laufzeit.



(a) 10.000 Reads mit einer Länge von 35 Basenpaaren und 0 bis 2 Mismatches in diskreter Gleichverteilung.



(b) 10.000 Reads mit einer Länge von 50 Basenpaaren und 0 bis 3 Mismatches in diskreter Gleichverteilung.

Abbildung 5.1: Abhängigkeit der Laufzeiten und Ergebnisse von der Anzahl der Mismatches für eine reale Datenbank (Homo sapiens chromosome 1).

Aufgrund der Ergebnisse anderer Short-Read Mapper (vgl. Tabelle 2.5) wurde erwartet, dass

[2]Die Datenraten beziehen sich nicht auf die Gesamtlaufzeit, sondern lediglich auf die Laufzeit der jeweiligen Funktion zur Suche oder zum Einsammeln der Daten.

für 38 Basenpaare maximal 300.000 und für 50 Basenpaare etwa 150.000 Positionen ermittelt werden und Überläufe daher selten auftreten. Abbildung 5.2 zeigt die ermittelten Positionen zur Messung aus Abbildung 5.1. Es wird deutlich, dass mit dem exakt arbeitenden FPGA deutlich mehr Positionen als erwartet ermittelt werden. Überläufe und HMRs treten häufiger auf, als angenommen. Bowtie kann aufgrund seines Such-Algorithmus nicht alle Positionen ermitteln und andere Short-Read Mapper wie Maq und RazerS brechen bei einer bestimmten Anzahl gefundener Positionen automatisch ab. Aufgrund dessen fällt das häufige Auftreten von HMRs bei Nutzung der Short-Read Mapper nicht auf. Die unzureichende Genauigkeit dieser Programme zeigt die Notwendigkeit eines exakten Verfahrens, welches alle möglichen Positionen ermittelt.

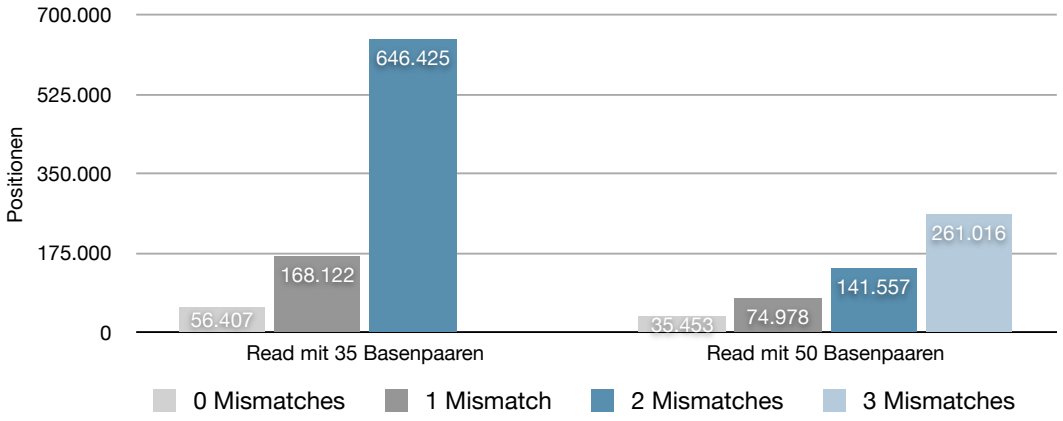


Abbildung 5.2: Ermittelte Positionen in Abhängigkeit der Anzahl der Mismatches für 100.000 Reads in einer großen Datenbank (Erweiterung zu Abbildung 5.1).

Weiterhin wird durch die Messungen deutlich, dass der Geschwindigkeitsverlust beim Auftreten von Überläufen deutlich geringer ausfällt als erwartet. Der Anteil der Zeit, die für die reine Suche benötigt wird, sinkt von 99 % auf 97 %, da das Empfangen und Sichern der Ergebnisse mehr Zeit in Anspruch nimmt. Durch die höhere Anzahl zulässiger Mismatches und die damit ebenfalls wachsende Anzahl gefundener Positionen steigt die Laufzeit an. Das Senden der Ergebnisse erhöht auch die Datenrate, sie befindet sich jedoch noch immer unter dem maximal möglichen Wert und stellt somit keinen Engpass dar.

Im Vergleich von Abbildung 5.1(a) und 5.1(b) wird deutlich, dass bei kurzen Sequenzen eine Suche mit drei zulässigen Mismatches zu einem deutlichen Anwachsen der Anzahl von Positionen führt. Dieses Ergebnis kommt zustande, da die hohe Anzahl zulässiger Mismatches bei kurzen Reads einen Anstieg zufälliger Positionen bewirkt. Die Laufzeit selbst ändert sich in Abhängigkeit der Länge der Reads nur minimal durch die wachsende Menge an Ergebnisdaten.

5.2.1.3 Auslastung des Host-Systems

Wie bereits durch Messungen gezeigt, wird bei großen Datenbanken der größte Teil der Laufzeit für die Suche auf dem FPGA benötigt. Messungen der CPU-Auslastung des Host-Systems bestä-

tigen dieses Ergebnis (vgl. Abbildung A.9). Für eine große Datenbank und einer Suche mit drei Mismatches, die Überläufe erzeugt, liegt die CPU-Auslastung durchschnittlich bei 15 %. Eine Bandbreite von durchschnittlich 400 MBit/s kann ebenfalls aufrecht erhalten werden.

5.2.1.4 Engpässe und Optimierungsmöglichkeiten

Die bisherigen Untersuchungen in Abschnitt 4.3.3 haben keinen gravierenden Engpass der Implementierung gezeigt. Das System skaliert mit der Anzahl der Reads und zunehmender Größe der Datenbank. Eine Erhöhung der Anzahl von Mismatches führt zu einer geringen Erhöhung der Laufzeit durch Überläufe. Bei einer großen Anzahl von Reads und den somit benötigten zahlreichen Durchläufen ist der Einbruch der Geschwindigkeit, verursacht durch die Überläufe, gravierender und darf nicht vernachlässigt werden.

Die Verarbeitung eines Überlaufes benötigt aufgrund der komplexen Steuerung ihres Ablaufes einen erhöhten Rechenaufwand auf Seiten des Hosts, ist anfällig für Fehler und führt außerdem zu einer starken Auslastung der Hardwareressourcen auf dem FPGA. Bei einer Erweiterung zu einem Multi-FPGA-System führt ein globales Einsammeln der Daten dazu, dass das System deutlicher als bisher ausgebremst wird und an Geschwindigkeit verliert.

Der Geschwindigkeitsverlust kann vermieden werden, indem die Ergebnisse nicht lokal gespeichert, sondern unverzüglich an den Host übertragen werden, ohne das Streaming zu unterbrechen. Die für das Einsammeln der Ergebnisse benötigte Datenrate liegt über die gesamte Laufzeit betrachtet nur bei wenigen MBit/s (vgl. Abbildung 5.1) und reicht für eine kontinuierliche Übertragung der Ergebnisse aus. Die Untersuchungen haben gezeigt, dass bei kurzen Reads, einer hohen Anzahl Mismatches und 600 aktiven Einheiten durchschnittlich maximal 1.000 Positionen in der Sekunde ermittelt werden, sodass ein konfliktfreies Einsammeln der Daten gewährleistet ist.

Das Verteilen und Einsammeln der Daten über die binären Bäume kann durch ein einfacheres System ersetzt werden. Eine ringförmige Kommunikation, bei der jede Einheit ihre Eingangsdaten an die nächste Einheit weiterreicht und die Ergebnisse beginnend mit der letzten Einheit zur ersten durchgeleitet werden, würde sich effektiv und mit geringem Hardwareaufwand auf dem FPGA realisieren lassen. Somit stellt auch das Routing auf einem größeren FPGA kein Problem dar, sodass die frei werdenden Ressourcen für weitere Such-Einheiten eingesetzt werden können.

Ein weiterer Engpass besteht in der festen Länge der Reads, mit der die Such-Einheiten arbeiten. Für Reads mit einer Länge von 30 Basenpaaren ist die Laufzeit genau so hoch wie für Reads mit 64 Basenpaaren. Bei kurzen Reads ist das System somit immer nur zur Hälfte ausgelastet. Die Lösung des Problems durch eine dynamische Anpassung der Hardware zur Laufzeit benötigt mehr Ressourcen und verlangsamt das System zusätzlich. Eine Nachbearbeitung auf dem Host-System stellt ebenfalls keine effiziente Lösung dar, da sie den Rechenaufwand deutlich erhöht.

Um eine effiziente Abarbeitung unterschiedlicher Reads zu ermöglichen, ist ein generisches Design notwendig, welches direkt an die Länge der Reads angepasst werden kann. Aufgrund der zuvor

erwähnten notwendigen Veränderungen zur kontinuierlichen Übertragung der Ergebnisse wird auf die lokalen Speicher verzichtet. Dadurch wird ebenfalls das Laden der Reads erleichtert und kann somit einfacher an eine generische Readlänge angepasst werden. Da die Länge der Reads von dem Sequenzierungsgerät abhängig ist und sich dementsprechend nicht ändert, verspricht ein direkt an die Länge der Reads angepasstes Design die höchste Leistung.

5.2.2 GPU-Implementierung

Die Ergebnisse einiger Testmessungen auf der GPU wurden bereits in Abschnitt 4.3.3 vorgestellt und für die Optimierung herangezogen. Mit realen Datenbanken wurde nach jedem Optimierungsschritt die korrekte Funktion des Systems nachgewiesen. Im folgenden Abschnitt 5.2.2.1 wird das Verhalten bei wachsenden Problemgrößen genauer betrachtet und in Abschnitt 5.2.2.2 folgt eine Auswertung mit weiteren Optimierungsansätzen.

5.2.2.1 Untersuchung zur Skalierung

Das System skaliert linear mit der Anzahl der Reads (vgl. Abbildung A.13) und der Länge der Datenbank (vgl. Abbildung A.10). Die Länge der Reads hat bis zu einer Länge von 256 Basenpaaren keinen wesentlichen Einfluss auf die Laufzeit, da genügend Threads zu Verfügung stehen, um alle Basenpaare eines Reads parallel abzuarbeiten (vgl. A.11). Für alle Reads, die mehr als 256 Basenpaare aufweisen, erfolgt der Anstieg der Laufzeit jeweils linear für ein Vielfaches von 32. Die Laufzeit wird demnach immer auf die eines Vielfachen von 32 aufgerundet.

5.2.2.2 Engpässe und Optimierungsmöglichkeiten

Die bisherigen Untersuchungen in Abschnitt 4.3.3 haben gezeigt, dass die internen Speicher der GPU einen deutlichen Geschwindigkeitsgewinn ermöglichen. Da die Datenbank aufgrund ihrer Größe im externen Grafikspeicher gehalten wird, ist kein weiterer Geschwindigkeitsgewinn möglich. Der Zugriff von mehreren parallel arbeitenden Multiprozessoren auf das selbe Datenbankzeichen führt jedoch nicht zu einem Einbruch der Leistung (vgl. Abbildung A.12).

5.3 Vergleichsmessungen

5.3.1 Short-Read Mapping Problem

Dieser Abschnitt beschäftigt sich mit einem Vergleich der Laufzeit und Ergebnisqualität beider implementierten Systeme im Vergleich zu den in Abschnitt 2.4 vorgestellten Short-Read Mappern. Das Ermitteln der Leistungsfähigkeit im Vergleich zu modernen Software-Heuristiken stellt einen wesentlichen Punkt für die Nutzbarkeit der Systeme dar.

Um einen Vergleich der beiden implementierten exakten Systeme mit dem einzigen bisherigen

exakten Ansatz zu ermöglichen, wird auch der Smith & Waterman-Algorithmus (*ssearch35*) zum Vergleich herangezogen. Beim Vergleich mit diesem Algorithmus ist jedoch lediglich die Laufzeit von Bedeutung, da die Anzahl der ermittelten Positionen durch das Alignment von Teilsequenzen beim Smith & Waterman-Algorithmus deutlich höher ist. Verglichen werden neben der Laufzeit auch der Anteil der Reads, für die eine Position gefunden wird, sowie die gesamte Anzahl an ermittelten Positionen.

Tabelle 5.1 enthält die Messergebnisse der Short-Read Mapper, Tabelle 5.2 zeigt die Ergebnisse der beiden implementierten Systeme FPGA und GPU. Die Messungen erfolgen mit zwei unterschiedlich langen Datenbanken und unterschiedlicher Länge der Reads. Die Zahl der Mismatches orientiert sich an den Werten für übliche Suchanfragen des Short-Read Mapping Problems (vgl. Abschnitt 2.3). Als Testdaten dienen reale Datenbanken.

Die beiden implementierten Ansätze versuchen, neben einer schnellen Suche auch eine hohe Qualität der Ergebnisse durch das Auffinden einer großen Anzahl Positionen zu garantieren. Um vergleichbare Ergebnisse zu erhalten, wurden die Short-Read Mapper mit Optionen ausgeführt, welche zum Auffinden möglichst vieler Positionen führen, ohne dabei die Suche zu stark zu verlangsamen[3]. Die Zeit für die Transformation der Datenbank wurde ebenfalls gemessen, ist aber für jede Datenbank nur einmalig notwendig.

Die Messungen zeigen, dass die Suche in der längeren Datenbank die Laufzeit aller Programme etwa um den Faktor drei gegenüber der kurzen Datenbank erhöht. Für unterschiedliche Readlängen liefern die Programme deutlich unterschiedliche Ergebnisse. Maq und SOAP2 finden bei längeren Reads nur für etwa drei Viertel der Suchanfragen eine Position, da sie maximal zwei Mismatches zulassen. Bowtie findet für lange Reads trotz den drei zulässigen Mismatches und der hohen Zahl von Backtracking Schritten nicht immer eine passende Position und eignet sich daher wie auch Maq und SOAP2 nur für kurze Reads mit maximal zwei Mismatches. Wie PASS und RazerS finden die beiden implementierten exakten Systeme für jeden Read mindestens eine passende Position. Abbildung 5.3 stellt den gegenüber dem Smith & Waterman-Algorithmus erreichten Speedup sowie die Anzahl der gefundenen Positionen grafisch dar.

Der Short-Read Mapper SOAP2 erreicht die mit Abstand höchste Geschwindigkeit, findet aber auch vergleichsweise wenig Positionen. Bei der Suche von kurzen Reads mit maximal zwei Mismatches erreicht der FPGA einen geringeren Speedup als Bowtie, findet aber deutlich mehr Positionen. Grund für die geringere Geschwindigkeit ist die fehlende Anpassung der Hardware an kurze Reads und eine daraus resultierende ineffiziente Ausnutzung der verfügbaren Ressourcen. Durch die Länge der Reads von 38 Basenpaaren sind die Such-Einheiten nur zur Hälfte ausgelastet, da sie in der Lage sind, 64 Basenpaare zu verarbeiten. Mit einer direkten Anpassung des Designs an die aktuelle Länge der Reads kann der Speedup verdoppelt werden, wodurch eine mit Bowtie vergleichbare Geschwindigkeit erreicht werden kann.

[3] Bowtie wurde für lange Reads mit drei zulässigen Mismatches und maximal 400 Backtracking Schritten ausgeführt, um die Geschwindigkeit nicht zu stark einzuschränken. Für kurze Reads wurde mit der Option `--nomaqround` eine Maq-ähnliche Rundung der Ergebnisse deaktiviert, die ansonsten durch Ungenauigkeiten zu einer größeren Anzahl gefundener Positionen führt. Durch die Option `--best` sind 800 Backtracking Schritte zugelassen, um dennoch viele Positionen zu ermitteln.

Tabelle 5.1: Vergleichsmessungen der Laufzeit und Qualität verschiedener Short-Read Mapper für zwei unterschiedliche Datenbanken und Reads unterschiedlicher Länge.

Programm	ssearch35[a]	Maq[b]	SOAP2[b]	PASS	RazerS	Bowtie
Datenbank	**Homo sapiens chromosome 13 (59.605.541 bp)**					
Transformation[c] (h:m:s)	—	00:00:02	00:01:01	—	—	00:01:34
Reads	**100.000 × 38 bp – maximal 2 Mismatches[d]**					
Laufzeit[c] (h:m:s)	83:20:00	00:00:58	00:00:04	00:00:49	00:01:06	00:00:18
Reads mapped (%)	100	100	100	99	100	82
Positionen	—[e]	100.000	307.484	441.457	510.968	531.764
Reads	**100.000 × 50 bp – maximal 3 Mismatches[d]**					
Laufzeit[c] (h:m:s)	97:13:00	00:03:07	00:00:08	00:01:59	00:01:06	00:01:29
Reads mapped (%)	100	76	76	99	100	75
Positionen	—[e]	76.742	102.981	327.523	208.996	354.023
Datenbank	**Homo sapiens chromosome 1 (222.388.987 bp)**					
Transformation[c] (h:m:s)	—	00:00:08	00:04:21	—	—	00:07:48
Reads	**100.000 × 38 bp – maximal 2 Mismatches[d]**					
Laufzeit[c] (h:m:s)	305:23:00	00:03:26	00:00:12	00:03:32	0:03:38	00:00:57
Reads mapped (%)	100	100	100	99	100	85
Positionen	—[e]	100.000	476.991	601.683	707.067	1.967.427
Reads	**100.000 × 50 bp – maximal 3 Mismatches[d]**					
Laufzeit[c] (h:m:s)	333:20:00	00:11:03	00:00:16	00:04:32	00:04:12	00:03:45
Reads mapped (%)	100	76	76	99	100	76
Positionen	—[e]	76.874	163.913	455.862	552.573	1.188.046

[a] Wert für 100.000 Reads aus gemessenen Daten für 1.000 Reads berechnet.
[b] Maximal zwei Mismatches möglich.
[c] Testsystem: Intel Core 2 Duo 2,66 GHz, 64 Bit, 4 GByte RAM.
[d] Mismatches in diskreter Gleichverteilung.
[e] Zu hohe Anzahl von Positionen durch Alignment von Teilsequenzen eines Reads.

Bei längeren Sequenzen mit mehreren Mismatches erreicht der FPGA einen mit Bowtie vergleichbaren Speedup und findet fast die doppelte Anzahl an Positionen (vgl. Abbildung 5.3(b)). Die FPGA-Implementierung ist bei der Suche längerer Sequenzen mit mehreren Mismatches herkömmlichen Short-Read Mappern in der Anzahl der gefundenen Positionen und der erreichten Geschwindigkeit deutlich überlegen. Damit ist der FPGA auch anderen in Abschnitt 2.6 vorgestellten Implementierungen auf parallelen Architekturen überlegen, wobei konkrete Vergleichswerte fehlen. Eine Weiterentwicklung des Systems hin zu längeren Reads führt zu einem weiteren Geschwindigkeitsgewinn gegenüber anderen Short-Read Mapper und ist daher erforderlich[4].

Zum besseren Vergleich der Leistungsfähigkeit zeigt Abbildung 5.4 das Verhältnis von gefundenen Positionen zur Laufzeit für die Suche langer Reads in einer großen Datenbank. Es wird deutlich, dass Bowtie zwar eine vergleichbare Geschwindigkeit erreicht, aber aufgrund der vergleichsweise geringen Anzahl gefundener Positionen eine geringe Gesamtleistung erzielt. SOAP2 hingegen erreicht aufgrund seiner sehr geringen Laufzeit trotz der wenigen Treffer eine dem FPGA ähnliche

[4] Durch längere Reads sinkt zwar die Zahl der parallelen Such-Einheiten, der Speedup, den eine Einheit gegenüber den Software-Algorithmen erreicht, steigt aber an.

Tabelle 5.2: Vergleichsmessungen der Laufzeit und Qualität der implementierten Systeme.

Programm	FPGA (Virtex-6[a])	GPUrSWA[d] (GTX 480)
Datenbank	**Homo sapiens chromosome 13 (59.605.541 bp)**	
Transformation[b] (h:m:s)	00:00:01	—
Reads	**100.000 × 38 bp – maximal 2 Mismatches[c]**	
Laufzeit (h:m:s)	00:01:01	11:54:00
Reads mapped (%)	100	100
Positionen	1.520.794	1.520.794
Reads	**100.000 × 50 bp – maximal 3 Mismatches[c]**	
Laufzeit (h:m:s)	00:01:01	13:53:00
Reads mapped (%)	100	100
Positionen	763.364	763.364
Datenbank	**Homo sapiens chromosome 1 (222.388.987 bp)**	
Transformation[b] (h:m:s)	00:00:05	—
Reads	**100.000 × 38 bp – maximal 2 Mismatches[c]**	
Laufzeit (h:m:s)	00:03:46	43:39:00
Reads mapped (%)	100	100
Positionen	5.475.284	5.475.284
Reads	**100.000 × 50 bp – maximal 3 Mismatches[c]**	
Laufzeit (h:m:s)	00:03:41	47:46:00
Reads mapped (%)	100	100
Positionen	2.306.253	2.306.253

[a] Virtex-6 XC6VLX240T - ML605 Prototyping Board.
[b] Testsystem: Intel Core 2 Duo 2,66 GHz, 64 Bit, 4 GByte RAM.
[c] Mismatches in diskreter Gleichverteilung.
[d] Wert für 100.000 Reads aus gemessenen Daten für 1.000 Reads berechnet.

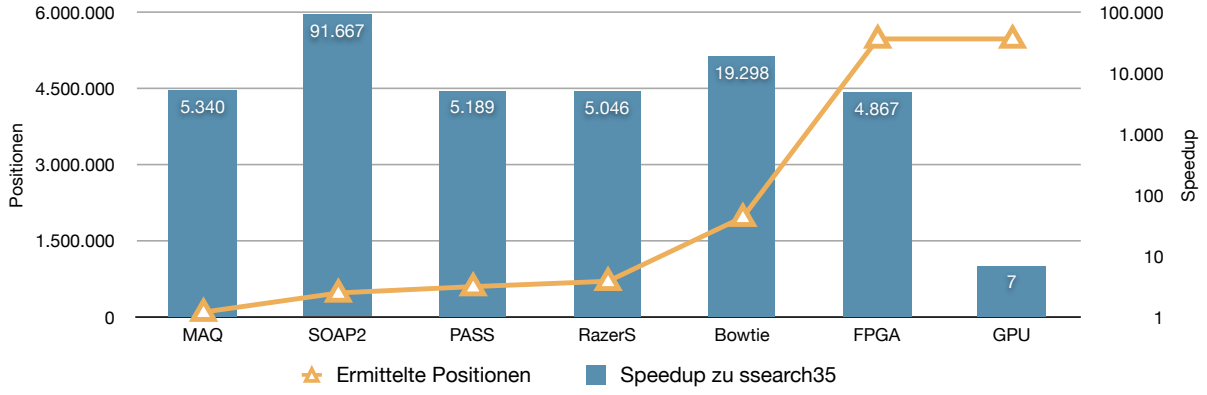
Gesamtleistung.

Der GPU-Ansatz ist zwar um eine Größenordnung schneller als die Software-Implementierung und findet ebenfalls sämtliche Positionen, erreicht aber nicht die hohen Geschwindigkeiten der anderen Short-Read Mapper und des FPGAs. Die GPU ist daher nicht als Short-Read Mapper geeignet. Da aber die erreicht Geschwindigkeit die des einfachen Smith & Waterman-Algorithmus übertrifft wird im Folgenden die Leistungsfähigkeit der GPU bei längeren Reads untersucht.

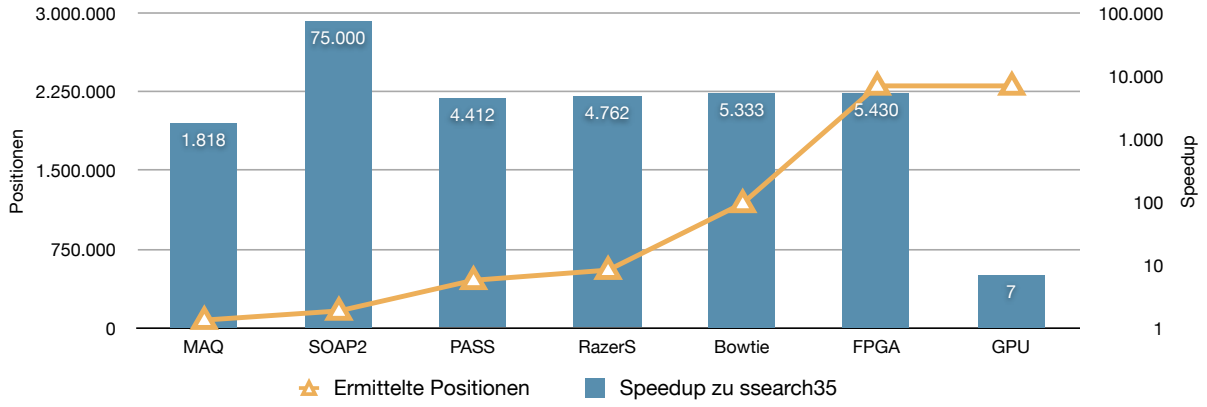
5.3.2 Alignment langer Sequenzen

Die bisherigen Untersuchungen haben sich lediglich auf kurze Reads beschränkt. Abbildung 5.5 zeigt hingegen den erreichten Speedup der GPU-Implementierung gegenüber des einfachen Smith & Waterman-Algorithmus bis zu einer Readlänge von 1.024 Basenpaaren. Bei Reads bis zu 256

Basenpaaren ist durch die zusätzlichen Threads eine weitere Geschwindigkeitssteigerung möglich. Ab 256 Threads erfolgt durch die Lastverteilung und die höhere Anzahl von Rechenoperationen je Datenbankzeichen ein weiterer, wenn auch geringerer Geschwindigkeitsgewinn.



(a) 100.000 Reads mit einer Länge von 38 Basenpaaren mit maximal zwei Mismatches.



(b) 100.000 Reads mit einer Länge von 50 Basenpaaren mit maximal drei Mismatches.

Abbildung 5.3: Erreichter Speedup unterschiedlicher Short-Read Mapper und der beiden Implementierung gegenüber des Smith & Waterman-Algorithmus (*ssearch35*) für eine lange Datenbanksequenz (Homo sapiens chromosome 1). Daten aus Tabelle 5.1 und 5.2.

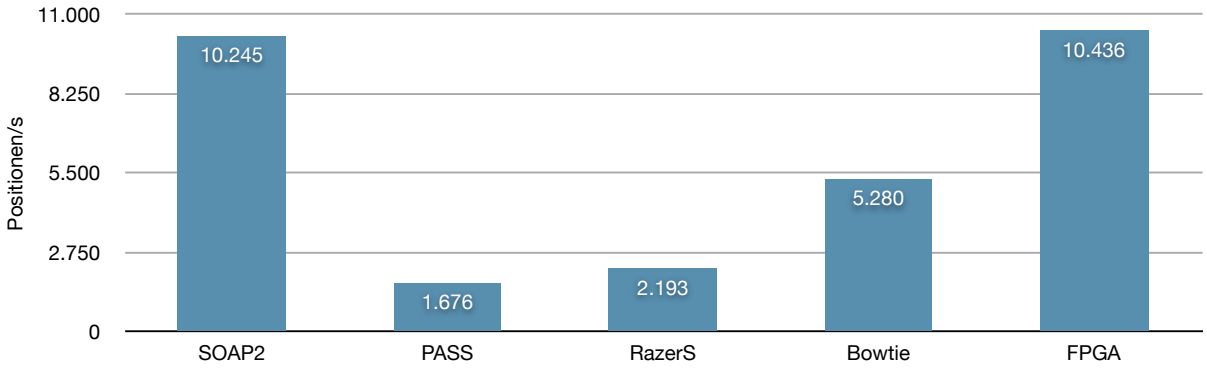


Abbildung 5.4: Anzahl der gefundenen Positionen pro Sekunde für Reads mit einer Länge von 50 Basenpaaren und einer langen Datenbanksequenz (Homo sapiens chromosome 1). Daten aus Tabelle 5.1 und 5.2.

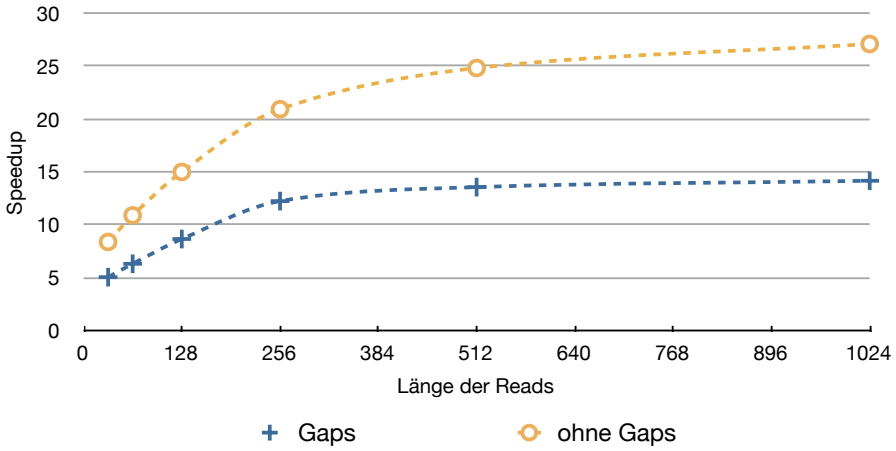


Abbildung 5.5: Erreichter Speedup der GPU-Implementierung gegenüber des Smith & Waterman-Algorithmus auf einer CPU[5].

Bei einzelnen Sequenzen werden nicht alle zur Verfügung stehenden Threads zur Lösung des Problems eingesetzt, so dass der Geschwindigkeitsgewinn geringer ausfällt. Die zur Berechnung des Speedups verwendete Laufzeit der GPU-Implementierung enthält die komplette Ausführungszeit des Programms einschließlich der Datenkommunikation und des Schreibens der Ergebnisdaten in eine Datei.

[5]Datenbank: Homo sapiens chromosome 1 (59.606.055 bp)
Reads: 1.000 mit 0-3 Mismatches in diskreter Gleichverteilung
Testsystem: Intel Core 2 Duo 2,66 GHz, 64 Bit, 4 GByte RAM

6 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit bestand in der Untersuchung unterschiedlicher Genom-Alignment Algorithmen und ihrer möglichen Übertragung auf moderne hochparallele Plattformen wie FPGA und GPU. Die Analyse hat ergeben, dass ein Großteil der Algorithmen, welche für das Mapping kurzer Sequenzen existieren, nicht effizient auf parallele Plattformen übertragen werden können. Ein auf die Diagonale reduzierter Smith & Waterman-Algorithmus kann hingegen auf beiden gewählten Plattformen implementiert werden. Auf dem FPGA entspricht ein solcher Algorithmus dem naiven Suchalgorithmus mit Mismatches und auf der GPU wird er zu einem reduzierten Smith & Waterman-Algorithmus. Beide Plattformen versprechen durch die Parallelisierung dieses einfachen Ansatzes einen großen Geschwindigkeitsgewinn.

Die Anforderungen an die Systeme ergeben sich aus dem Short-Read Mapping Problem. Eine weitere wichtige Anforderung besteht darin, dass der größte Anteil der Berechnungen direkt auf den parallelen Plattformen durchgeführt wird, dass die Systeme von typischen Nutzern verwendet werden können und mit den üblichen Datenformaten arbeiten. Die Implementierung erfolgt in Hinblick auf diese Anforderungen. Bei der Entwicklung der beiden Systeme steht des Weiteren die effiziente Auslastung von Hardwareressourcen im Vordergrund, um möglichst viele parallele Einheiten auf den Plattformen zu realisieren und somit eine hohe Geschwindigkeit zu erreichen. Beim FPGA ist auch eine leistungsfähige Schnittstelle zur Kommunikation und Übertragung der Daten von großer Bedeutung.

Der FPGA-Entwurf verfügt über die Besonderheit, für alle Reads, die nicht mit der vorgegebenen Anzahl von Mismatches in der Datenbank gefunden werden, automatisch die notwendige Anzahl von Mismatches auszugeben. Bei einer folgenden Suche kann somit unter Berücksichtigung dieser Anzahl von Mismatches mit Sicherheit eine Position für den Read ermittelt werden. Aufgrund der getrennten Funktionseinheiten ist es außerdem möglich, für jeden Read eine individuelle Anzahl von Mismatches zu definieren.

Es wurden zwei Systeme implementiert, welche im Gegensatz zu anderen Short-Read Mappern exakte Ergebnisse liefern, da sämtliche Positionen eines Reads in der Datenbank gefunden werden. Der FPGA erreicht bei der Suche von Reads mit einer Länge von 50 Basenpaaren eine Geschwindigkeit, die im Bereich leistungsfähiger Short-Read Mapper mit heuristischer Suche liegt und die doppelte Anzahl an Positionen liefert. Eine Besonderheit besteht darin, dass die Laufzeit von der Anzahl der Mismatches nur unwesentlich beeinflusst wird. Die Auslastung des Host-Systems während einer Suche ist dabei sehr gering, was eine Integration in ein bestehendes Arbeitsplatzsystem oder in ein System zur Sequenzierung ermöglicht. Der geringe Energieverbrauch des FPGAs und die dadurch niedrigen Betriebskosten sind ein weiterer wesentlicher Vorteil.

Die Implementierung auf der GPU ist aufgrund ihrer niedrigen Geschwindigkeit nicht als Short-Read Mapper geeignet. Sie kann jedoch als Beschleunigung des Smith & Waterman-Algorithmus verwendet werden. Bisherige Smith & Waterman-Implementierungen auf parallelen Plattformen erreichen eine derartige Leistung nur für sehr lange Sequenzen. Aufgrund der im Vergleich zum FPGA deutlich höheren Laufzeit sowie des hohen Energieverbrauches, stellt die GPU keinen sinnvollen Ausgangspunkt für weitere Entwicklungen eines exakt arbeitenden Short-Read Mappers dar.

Beide Systeme wurden auf Korrektheit und Zuverlässigkeit untersucht und die Ergebnisse wurden hinsichtlich sich ergebender Engpässe analysiert. Daraus haben sich Ansätze zur Optimierung ergeben. Der FPGA bildet aufgrund seiner hohen Leistung einen guten Ausgangspunkt für weitere Entwicklungen. Durch ein kontinuierliches Auslesen der Ergebnisse ist ein großer Geschwindigkeitsgewinn durch einen reduzierten aufwand an Ressourcen und somit zusätzliche Einheiten zu erwarten. Die Laufzeit wird vollkommen von der Anzahl der Mismatches und der dadurch steigenden Zahl der Positionen gelöst. Eine Erhöhung der Taktrate sowie der Anzahl der Such-Einheiten macht eine Verdopplung der Geschwindigkeit durchaus realistisch.

So ist auch das Zusammenschalten mehrerer FPGAs zur Reduzierung der Laufzeit durch eine Steigerung der Parallelität eine denkbare Weiterentwicklung. Um den FPGA völlig vom Host-System zu trennen, stellt des Weiteren ein Streaming der Datenbank von einer lokalen SSD[1]-Festplatte einen interessanten Ansatz dar. Für längere Reads und Paired-End Reads ist ein generisches Design und somit eine direkte Anpassung der Hardware an die Länge der Reads notwendig. Eine Architektur, welche flexibel sowohl kurze als auch lange Reads unterstützt oder die Ergebnisse mittels Software anpasst, erreicht nicht die Geschwindigkeit eines direkt an das Problem angepassten Designs.

Für längere Reads sollte des Weiteren die Möglichkeit des Einfügens von Gaps in Erwägung gezogen werden, da eine Suche, welche ausschließlich Mismatches zulässt, für Reads mit mehr als 100 Basenpaaren wenig Erfolg verspricht. Durch eine mit der Umsetzung auf der GPU vergleichbaren Parallelisierung des Smith & Waterman-Algorithmus oder der Implementierung als systolisches Array könnte bei der Suche eine hohe Geschwindigkeit erreicht werden.

[1] Solid-State Drive

7 Anhang

A Messwerte

A.1 FPGA-Implementierung

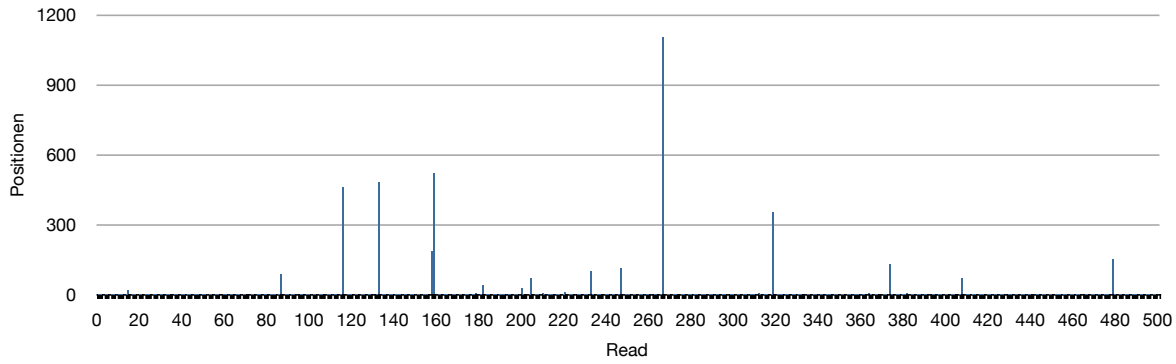


Abbildung A.1: Gemessene Verteilung der gefunden Positionen auf die SUs am Beispiel einer realen Datenbank und 500 Reads (0-3 Mismatches).

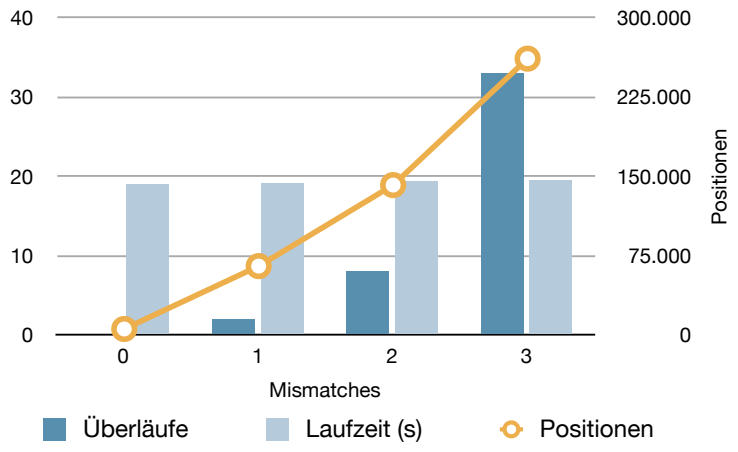


Abbildung A.2: Überläufe und Laufzeit in Abhängigkeit von der Anzahl der Mismatches.

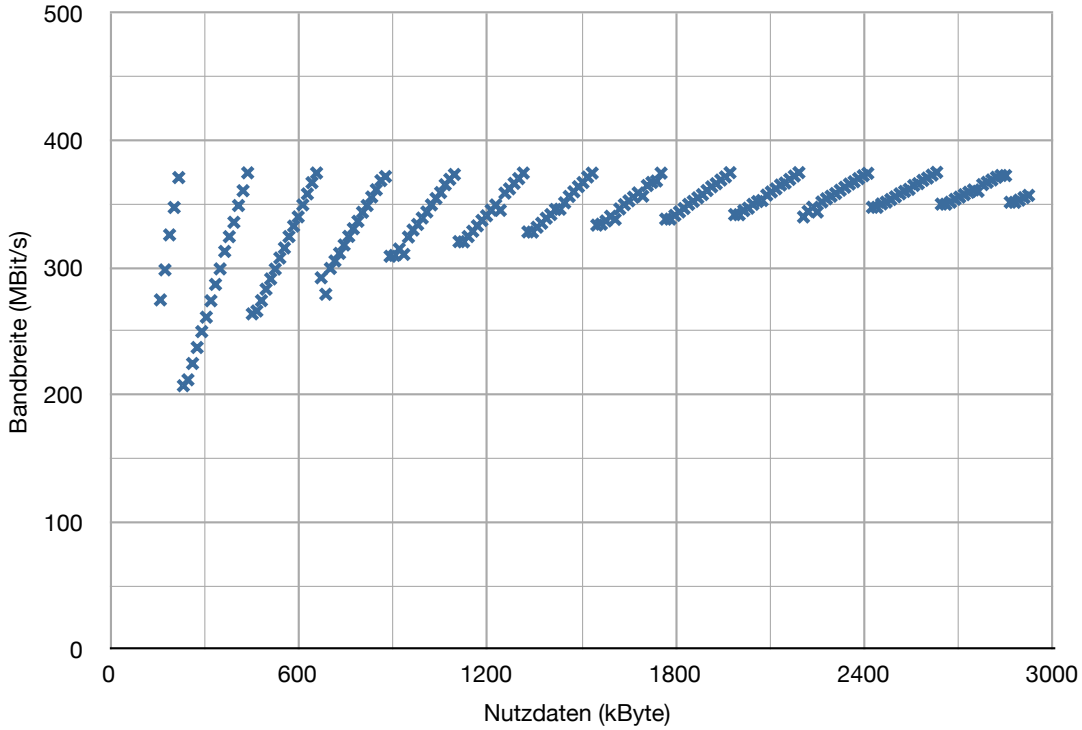


Abbildung A.3: Streaming mit statischer Flusskontrolle.

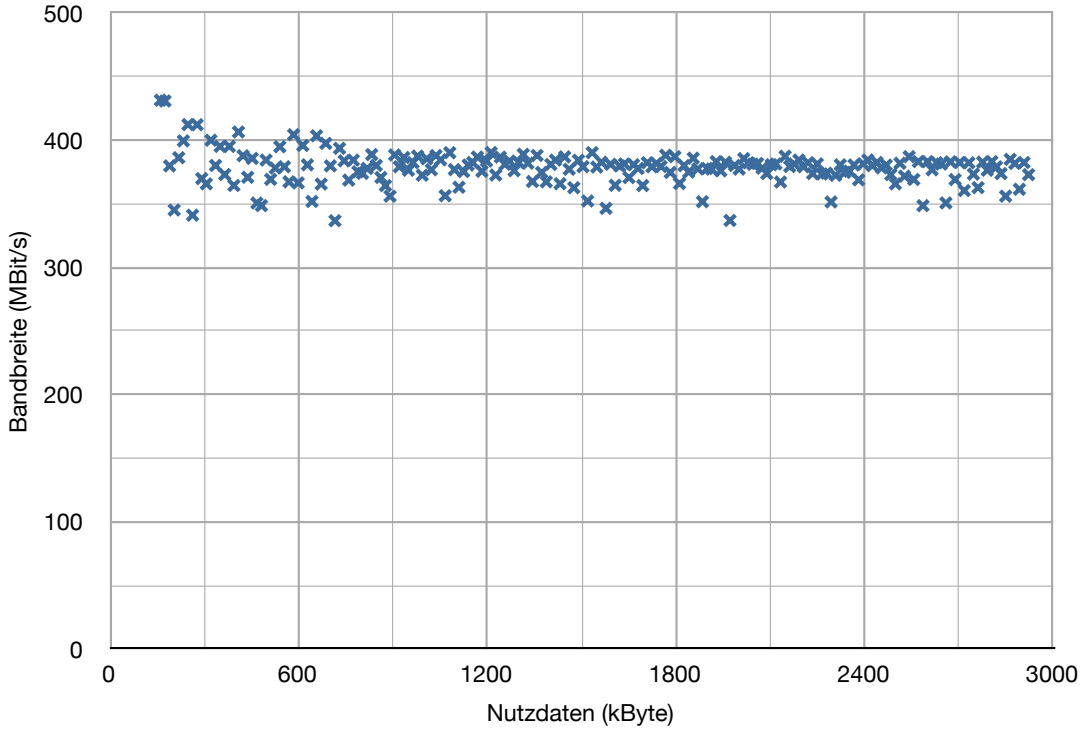


Abbildung A.4: Streaming mit adaptiver Flusskontrolle.

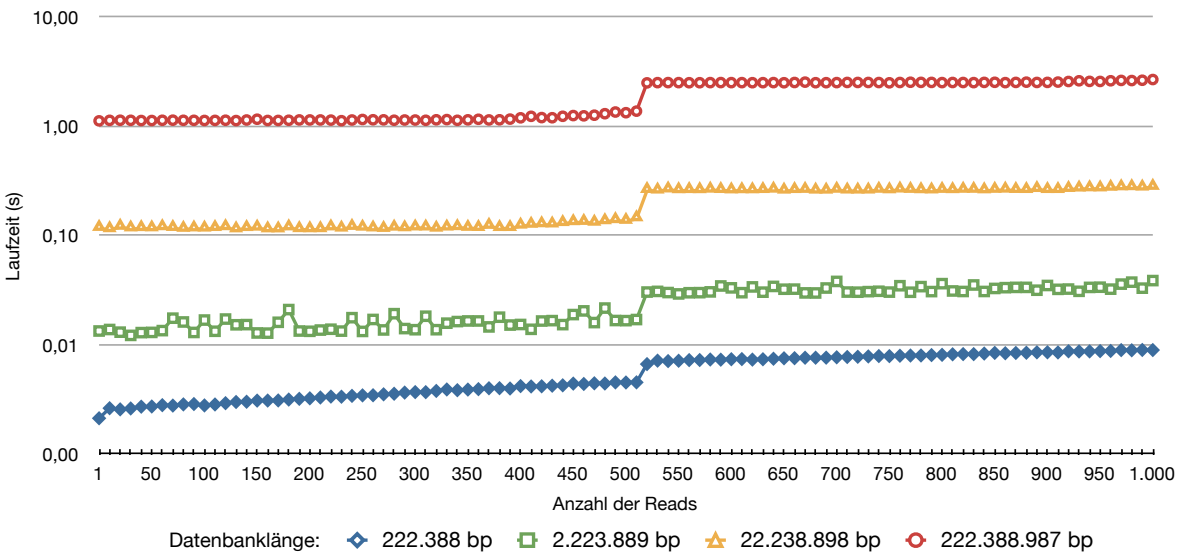


Abbildung A.5: Auswirkung der Anzahl der Reads und der Länge der Datenbank auf die Laufzeit (Messung durchgeführt mit 512 Such-Einheiten).

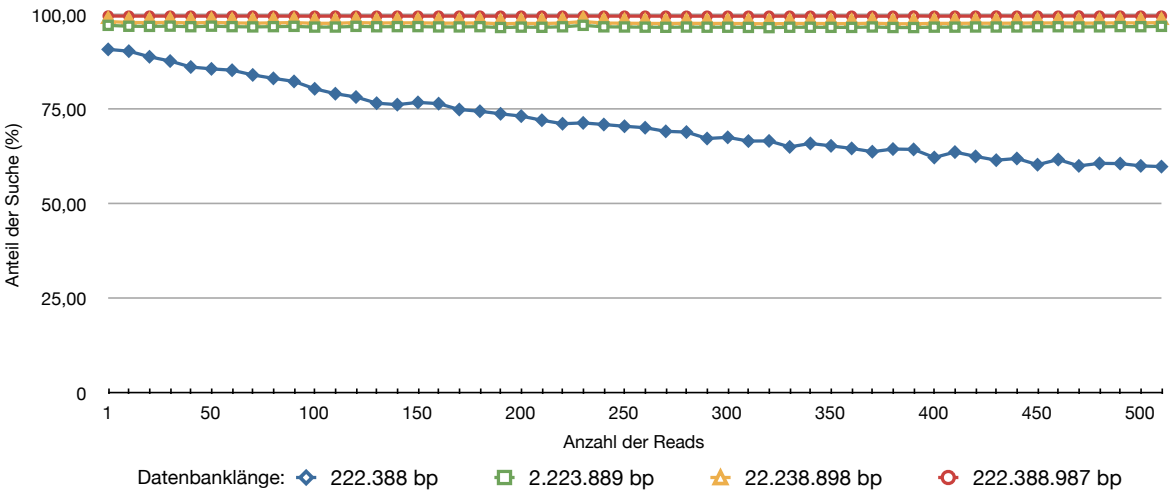


Abbildung A.6: Prozentualer Anteil der Suche in Abhängigkeit der Anzahl der Reads und der Länge der Datenbank (Messung durchgeführt mit 512 Such-Einheiten).

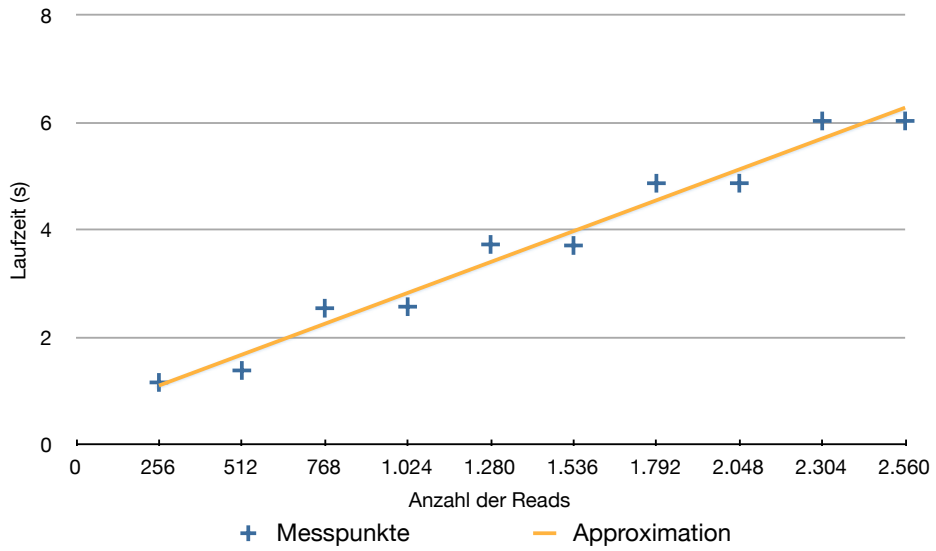


Abbildung A.7: Skalierung bei einer großen Anzahl von Reads mit einer Datenbank von 222.388.987 Basenpaaren (Messung durchgeführt mit 512 Such-Einheiten).

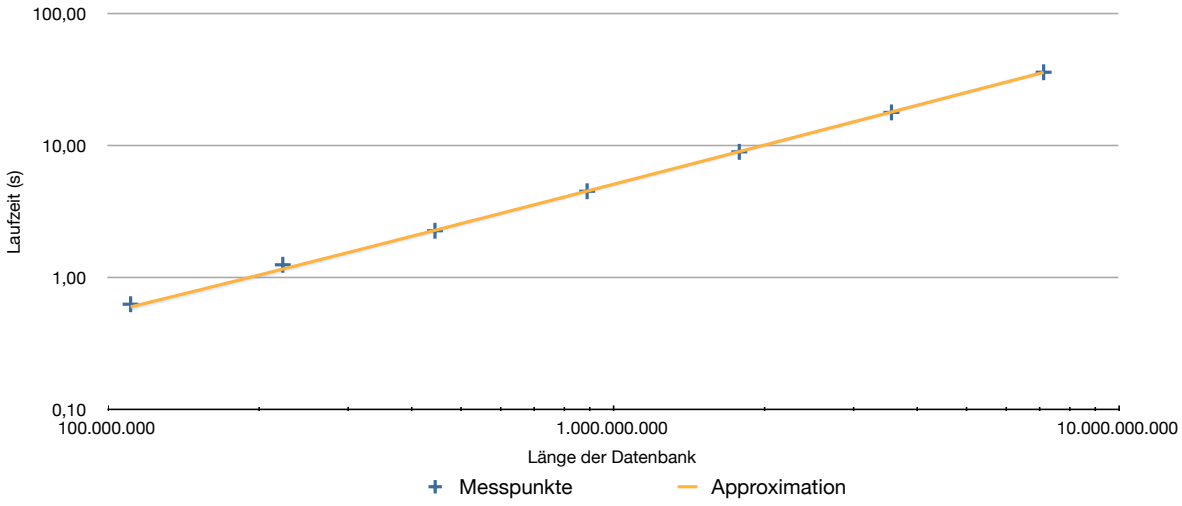


Abbildung A.8: Laufzeit in Abhängigkeit der Länge der Datenbank.

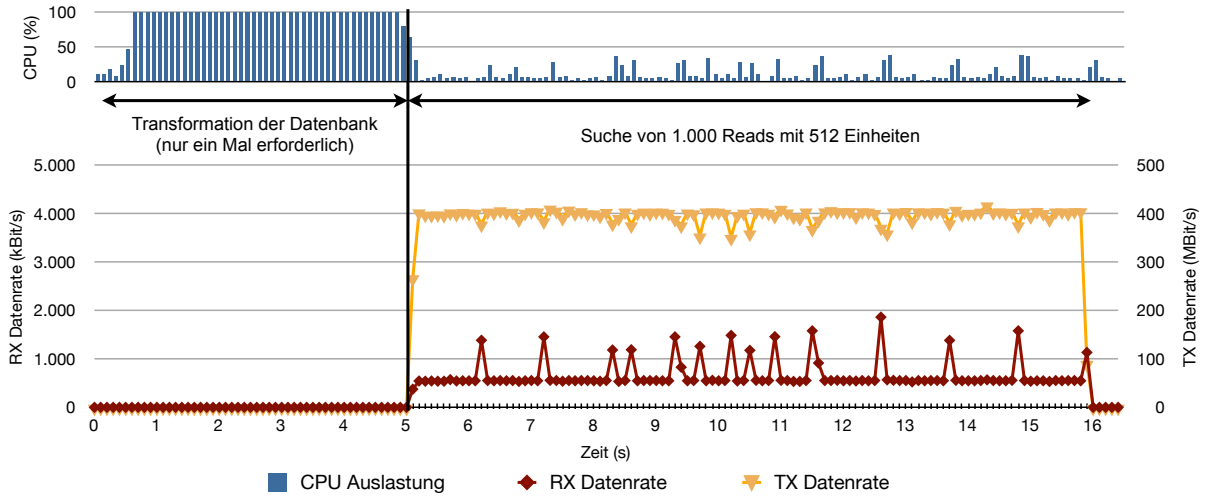


Abbildung A.9: Auslastung der CPU bei Transformation der Datenbank und Suche[0].

[0]Datenbank: Homo sapiens chromosome 1 (59.606.055 bp)
Reads: 1.000 mit 50 bp und 0-3 Mismatches in diskreter Gleichverteilung
Testsystem: Intel Core 2 Duo 2,66 GHz, 64 Bit, 4 GByte RAM

A.2 GPU-Implementierung

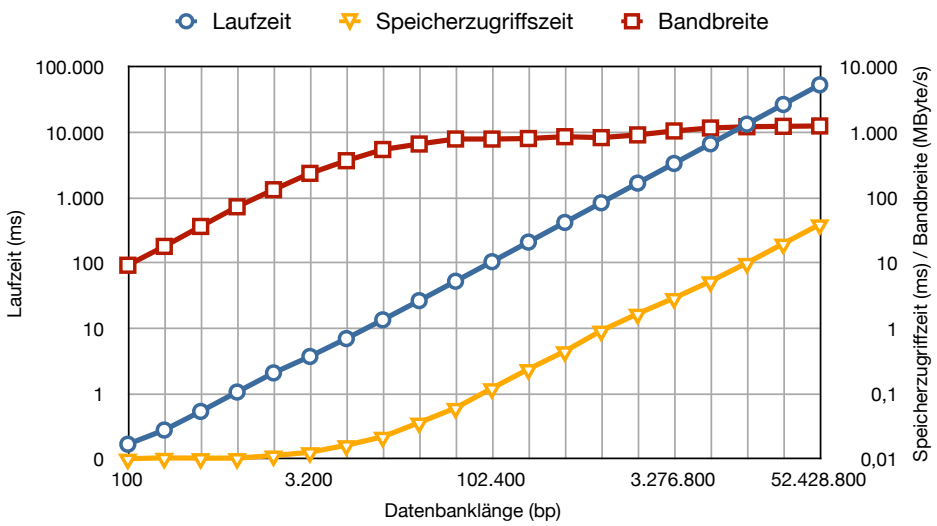


Abbildung A.10: Darstellung von Laufzeit, Speicherzugriffszeit und Bandbreite zwischen Host- und GPU Speicher, in Abhängigkeit der Datenbankgröße.

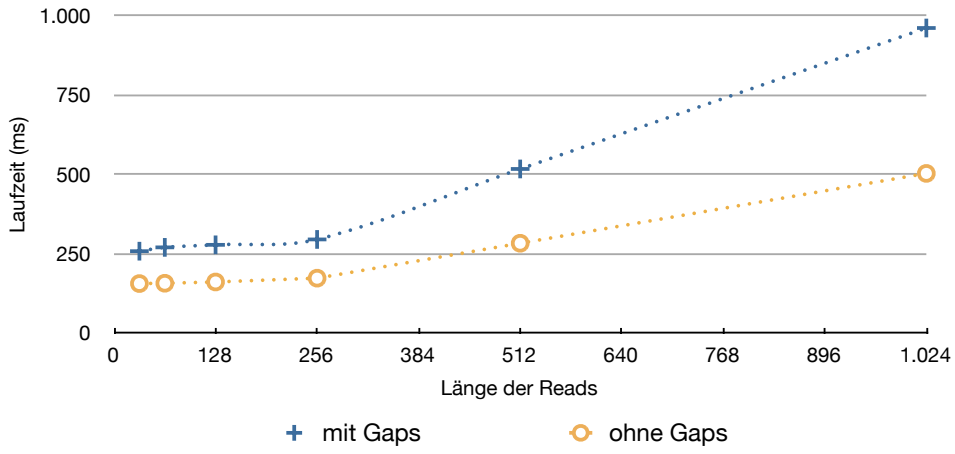


Abbildung A.11: Laufzeit in Abhängigkeit der Länge der Reads. Linearer Anstieg erfolgt Stufenförmig, wobei die Laufzeit der des nächst größeren Vielfachen von 32 entspricht.

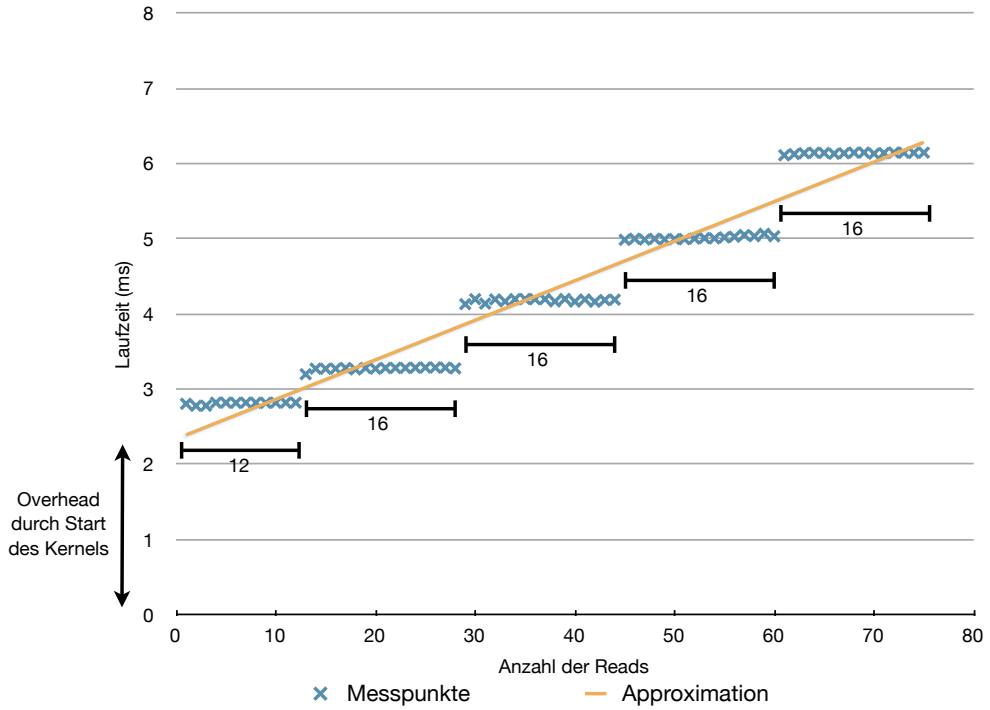


Abbildung A.12: Laufzeit in Abhängigkeit von Reads bei einer geringen Anzahl Reads.

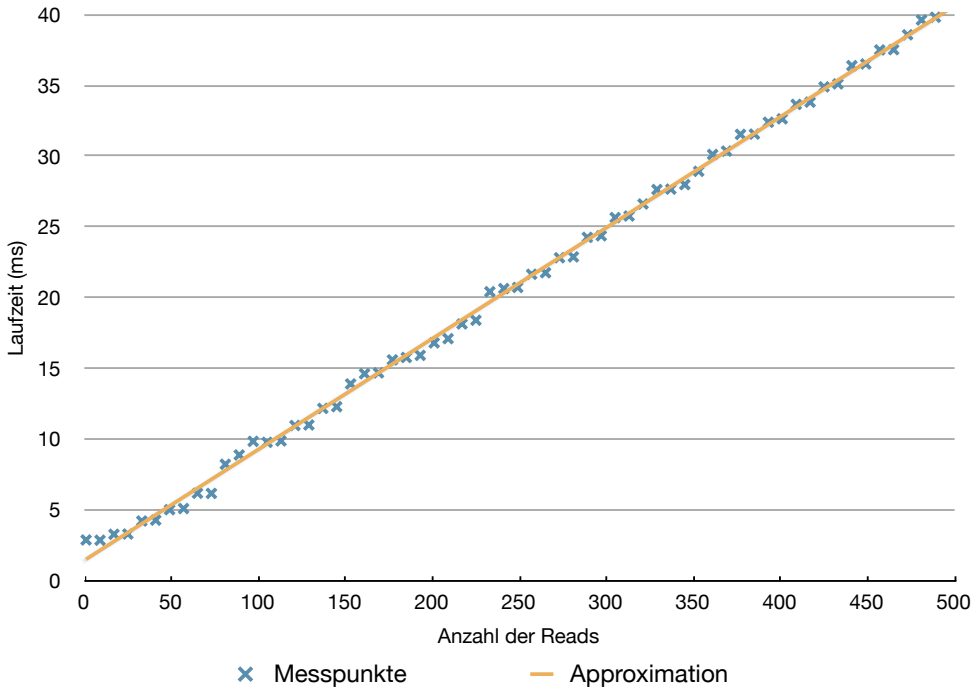


Abbildung A.13: Laufzeit in Abhängigkeit von Reads bei einer hohen Anzahl Reads.

B Ausgewählte Quelltexte

Listing 7.1: Tabelle zur Übersetzung der Datenbank.

```
char nucA[4] = {0x00, 0x00, 0x00, 0x00};
char nucG[4] = {0x40, 0x10, 0x04, 0x01};
char nucT[4] = {0x80, 0x20, 0x08, 0x02};
char nucC[4] = {0xC0, 0x30, 0x0C, 0x03};
```

Listing 7.2: Tabelle zur Transformation von zwei Zeichen eines Reads.

```
unsigned long const LUT5_CFG_B2[16] = {
  0xEEE0111E, // 0x0 -> AA
  0xDDD0222D, // 0x1 -> AG
  0xBBB0444B, // 0x2 -> AT
  0x77708887, // 0x3 -> AC

  0xEE0E11E1, // 0x4 -> GA
  0xDD0D22D2, // 0x5 -> GG
  0xBB0B44B4, // 0x6 -> GT
  0x77078878, // 0x7 -> GC

  0xE0EE1E11, // 0x8 -> TA
  0xD0DD2D22, // 0x9 -> TG
  0xB0BB4B44, // 0xA -> TT
  0x70778788, // 0xB -> TC

  0x0EEEE111, // 0xC -> CA
  0x0DDDD222, // 0xD -> CG
  0x0BBBB444, // 0xE -> CT
  0x07777888, // 0xF -> CC
};
```

Listing 7.3: Tabelle zur Transformation eines einzelnen Zeichens eines Reads.

```
unsigned long const LUT5_CFG_B1[4] = {
  0x0000FFF0, // 0x0 -> A
  0x0000FF0F, // 0x1 -> G
  0x0000F0FF, // 0x2 -> T
  0x00000FFF, // 0x3 -> C
};
```

C FPGA Testfälle

1. Einzelne Positionen mit unterschiedlicher Anzahl von Mismatches,
2. Mehrere Positionen eines Reads,
3. Korrektheit einer Position im ersten Datenpaket,
4. Korrektheit einer Position im folgenden Datenpaket,
5. Korrektheit einer Position Read auf der Grenze zwischen zwei Datenpaketen,
6. Flusskontrolle und Paketverlust bei 1.000 Paketen,
7. Streaming einer realer Datenbank mit 12.343.212 bp,
8. Erneutes Laden einer Such-Einheit nach einem Reset-Paket,
9. Suche mit zwei Suche-Einheiten innerhalb eines Such-Moduls,
10. Überlauf und Korrektheit der Positionen wenn nur eine Such-Einheit schreibt,
11. Überlauf und Korrektheit der Positionen wenn beide Such-Einheiten schreiben,
12. Suche mit insgesamt 600 Such-Einheiten und Testdatenbank zur Ermittlung der Positionen,
13. Suche in Testdatenbank sowie in realer Datenbank mit 1.000 Reads und
14. Streaming mehrerer Sequenzen.

D Inhalt der CD-ROM

Das Wurzelverzeichnis der beiliegenden CD-ROM enthält folgende Verzeichnisse:

⇒ **Dokumentation** Diese Arbeit als PDF, Dokumente, sämtliche Bilder und Tabellen

⇒ **FPGA-Alignment** Die VHDL-Quelltexte und die C-Quelltexte der Host-Software

⇒ **GPU-Alighnment** Die C-Quelltexte des CUDA Programmes

⇒ **Testdaten** Testdatenbanken für spezielle Funktionstests auf FPGA und GPU

⇒ **Hilfsprogramme** Hilfsprogramme und Skripte

Literaturverzeichnis

[AGM+90] Atschul, S.F. ; Gish, W. ; Miller, W. ; Myers, EW ; Lipman, DJ: Basic local alignment search tool. In: *J. Mol. Biol* 215 (1990), S. 403–410

[BH01] Badach, A. ; Hoffmann, E.: *Technik der IP-Netze*. Hanser, 2001. – ISBN 3446215018

[Böc03] Böckenhauer, H.: *Algorithmische Grundlagen der Bioinformatik*. Teubner, 2003

[BW94] Burrows, M. ; Wheeler, D.J.: A block-sorting lossless data compression algorithm. (1994)

[CAB+09] Campagna, D. ; Albiero, A. ; Bilardi, A. ; Caniato, E. ; Forcato, C. ; Manavski, S. ; Vitulo, N. ; Valle, G.: PASS: a program to align short sequences. In: *Bioinformatics* 25 (2009), Nr. 7, S. 967. – ISSN 1367–4803

[DCF03] Darling, A.E. ; Carey, L. ; Feng, W.: The design, implementation, and evaluation of mpiBLAST. In: *Proceedings of ClusterWorld* 2003 (2003)

[Flo97] Floyd, T.L.: *Digital fundamentals*. Pearson Education, 1997

[FM00] Ferragina, P. ; Manzini, G.: Opportunistic data structures with applications. In: *focs* Published by the IEEE Computer Society, 2000, S. 390

[FWLR09] Fullwood, M.J. ; Wei, C.L. ; Liu, E.T. ; Ruan, Y.: Next-generation DNA sequencing of paired-end tags (PET) for transcriptome and genome analyses. In: *Genome research* 19 (2009), Nr. 4, S. 521. – ISSN 1088–9051

[HAAV] Hasan, L. ; Al-Ars, Z. ; Vassiliadis, S.: Hardware acceleration of sequence alignment algorithms-an overview. In: *Design & Technology of Integrated Systems in Nanoscale Era, 2007. DTIS. International Conference on* IEEE, S. 92–97

[Hal08] Halfhill, T. R.: Parallel Processing with CUDA: Nvidia's High-Performance Computing Platform Uses Massive Multithreading. In: *Microprocessor Journal* (2008)

[Han06] Hansen, Andrea: *Bioinformatik: Ein Leitfaden für Naturwissenschaftler*. Birkhäuser Basel, 2006. – ISBN 3764362537

[Hau09] Hauswedell, H.: BLAST-like Local Alignments with RazerS. (2009)

[HHA+10] Hach, F. ; Hormozdiari, F. ; Alkan, C. ; Hormozdiari, F. ; Birol, I. ; Eichler, E.E. ; Sahinalp, S.C.: mrsFAST: a cache-oblivious algorithm for short-read mapping. In: *Nature Methods* 7 (2010), Nr. 8, S. 576–577. – ISSN 1548–7091

[HK09] Hwu, W. ; Kirk, D.: Programming massively parallel processors. In: *Special Edition* (2009), S. 92

[HMS+07] Herbordt, M.C. ; Model, J. ; Sukhwani, B. ; Gu, Y. ; VanCourt, T.: Single pass streaming BLAST on FPGAs. In: *Parallel computing* (2007)

[HS04] Hinze, T. ; Sturm, M.: *Rechnen mit DNA: Eine Einführung in Theorie und Praxis.* Oldenbourg Wissenschaftsverlag, 2004

[JMMW07] Johnson, D.S. ; Mortazavi, A. ; Myers, R.M. ; Wold, B.: Genome-wide mapping of in vivo protein-DNA interactions. In: *Science* 316 (2007), Nr. 5830, S. 1497. – ISSN 0036–8075

[Kai08] Kaiser, J.: A plan to capture human diversity in 1000 genomes. In: *Science* 319 (2008), Nr. 5862, S. 395. – ISSN 0036–8075

[Ken02] Kent, W.J.: BLAT-the BLAST-like alignment tool. In: *Genome research* 12 (2002), Nr. 4, S. 656. – ISSN 1088–9051

[KPD+04] Kurtz, S. ; Phillippy, A. ; Delcher, A. ; Smoot, M. ; Shumway, M. ; Antonescu, C. ; Salzberg, S.: Versatile and open software for comparing large genomes. In: *Genome biology* 5 (2004), Nr. 2, S. R12. – ISSN 1465–6906

[KSPBP10] Khajeh-Saeed, A. ; Poole, S. ; Blair Perot, J.: Acceleration of the Smith-Waterman algorithm using single and multiple graphics processors. In: *Journal of Computational Physics* 229 (2010), Nr. 11, S. 4247–4258. – ISSN 0021–9991

[LD09] Li, H. ; Durbin, R.: Fast and accurate short read alignment with Burrows–Wheeler transform. In: *Bioinformatics* 25 (2009), Nr. 14, S. 1754. – ISSN 1367–4803

[LHJV06] Liu, Y. ; Huang, W. ; Johnson, J. ; Vaidya, S.: Gpu accelerated smith-waterman. In: *LECTURE NOTES IN COMPUTER SCIENCE* 3994 (2006), S. 188

[LHW+09] Li, H. ; Handsaker, B. ; Wysoker, A. ; Fennell, T. ; Ruan, J. ; Homer, N. ; Marth, G. ; Abecasis, G. ; Durbin, R.: The sequence alignment/map format and SAMtools. In: *Bioinformatics* 25 (2009), Nr. 16, S. 2078. – ISSN 1367–4803

[LLKW08] Li, R. ; Li, Y. ; Kristiansen, K. ; Wang, J.: SOAP: short oligonucleotide alignment program. In: *Bioinformatics* 24 (2008), Nr. 5, S. 713. – ISSN 1367–4803

[LRD08] Li, H. ; Ruan, J. ; Durbin, R.: Mapping short DNA sequencing reads and calling variants using mapping quality scores. In: *Genome research* 18 (2008), Nr. 11, S. 1851. – ISSN 1088–9051

[LST07] Li, I.T.S. ; Shum, W. ; Truong, K.: 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array(FPGA). In: *BMC bioinformatics* 8 (2007), Nr. 1, S. 185

[LTPS09] Langmead, B. ; Trapnell, C. ; Pop, M. ; Salzberg, S.: Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. In: *Genome Biology* 10 (2009), Nr. 3, S. R25

[LYL+09] Li, R. ; Yu, C. ; Li, Y. ; Lam, T.W. ; Yiu, S.M. ; Kristiansen, K. ; Wang, J.: SOAP2: an improved ultrafast tool for short read alignment. In: *Bioinformatics* 25 (2009), Nr. 15, S. 1966. – ISSN 1367–4803

[MAF+08] Manavski, S.A. ; Albiero, A. ; Forcato, C. ; Vitulo, N. ; Valle, G.: GEAgpu: Improved alignment of spliced DNA sequences to genomic data using Graphics Processing Units. In: *European Conference on Computational Biology* (2008)

[Mar08] Mardis, E.R.: The impact of next-generation sequencing technology on genetics. In: *Trends in Genetics* 24 (2008), Nr. 3, S. 133–141

[MDD+09] Mardis, E.R. ; Ding, L. ; Dooling, D.J. ; Larson, D.E. ; McLellan, M.D. ; Chen, K. ; Koboldt, D.C. ; Fulton, R.S. ; Delehaunty, K.D. ; McGrath, S.D. u. a.: Recurring mutations found by sequencing an acute myeloid leukemia genome. In: *New England Journal of Medicine* 361 (2009), Nr. 11, S. 1058–1066. – ISSN 0028–4793

[MEA+05] Margulies, M. ; Egholm, M. ; Altman, W.E. ; Attiya, S. ; Bader, J.S. ; Bemben, L.A. ; Berka, J. ; Braverman, M.S. ; Chen, Y.J. ; Chen, Z. u. a.: Genome sequencing in microfabricated high-density picolitre reactors. In: *Nature* 437 (2005), Nr. 7057, S. 376–380. – ISSN 0028–0836

[MNLC10] Misra, S. ; Narayanan, R. ; Lin, S. ; Choudhary, A.: Fangs: High speed sequence mapping for next generation sequencers. In: *Proceedings of the 2010 ACM Symposium on Applied Computing* ACM, 2010, S. 1539–1546

[Mun01] Munk, K.: *Grundstudium Biologie. 4. Genetik*. Spektrum, Akad.-Verl., 2001

[MV08] Manavski, S. ; Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. In: *BMC bioinformatics* 9 (2008), Nr. Suppl 2, S. S10

[Mye99] Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. In: *Journal of the ACM (JACM)* 46 (1999), Nr. 3, S. 395–415. – ISSN 0004–5411

[NBGS08] Nickolls, J. ; Buck, I. ; Garland, M. ; Skadron, K.: Scalable parallel programming with CUDA. (2008)

[NCB10] NCBI: *GenBank Statistics*.
http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html, 2010

[NW70] Needleman, S.B. ; Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. In: *Journal of molecular biology* 48 (1970), Nr. 3, S. 443–453

[OM88] Owolabi, O. ; McGregor, DR: Fast approximate string matching. In: *Software: Practice and Experience* 18 (1988), Nr. 4, S. 387–393. – ISSN 1097–024X

[OW90] Ottmann, T. ; Widmayer, P.: *Algorithmen und Datenstrukturen*. BI-Wiss.-Verl., 1990

[PL88] Pearson, WR ; Lipman, DJ: Improved tools for biological sequence comparison. In: *Proceedings of the National Academy of Sciences* (1988)

[RB01] Rashidi, H.H. ; B
ühler, L.K.: *Grundriss der Bioinformatik*. Spektrum, Akad. Verl., 2001

[RF09] Reis-Filho, J.: Next-generation sequencing. In: *Breast Cancer Research* 11 (2009), Nr. Suppl 3, S. S12. – ISSN 1465–5411

[RLD+09] Rumble, S.M. ; Lacroute, P. ; Dalca, A.V. ; Fiume, M. ; Sidow, A. ; Brudno, M. u. a.: SHRiMP: accurate mapping of short color-space reads. In: *PLoS Comput Biol* 5 (2009), Nr. 5, S. e1000386. – ISSN 1553–7358

[RSM06] Rasmussen, K.R. ; Stoye, J. ; Myers, E.W.: Efficient q-gram filters for finding all ε-matches over a given length. In: *Journal of Computational Biology* 13 (2006), Nr. 2, S. 296–308. – ISSN 1066–5277

[SK10] Sanders, J. ; Kandrot, E.: *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 2010

[SNC77] Sanger, F. ; Nicklen, S. ; Coulson, A.R.: DNA sequencing with chain-terminating inhibitors. In: *Proceedings of the National Academy of Sciences* 74 (1977), Nr. 12, S. 5463. – ISSN 0027–8424

[STDV07] Schatz, M.C. ; Trapnell, C. ; Delcher, A.L. ; Varshney, A.: High-throughput sequence alignment using Graphics Processing Units. In: *BMC bioinformatics* 8 (2007), Nr. 1, S. 474. – ISSN 1471–2105

[SW81] Smith, TF ; Waterman, MS: Identification of common molecular subsequences. In: *Journal of molecular biology* 147 (1981), Nr. 1, S. 195–197

[TS09] Trapnell, C. ; Salzberg, SL: How to map billions of short reads onto genomes. In: *Nature biotechnology* 27 (2009), Nr. 5, S. 455

[VS11] Vouzis, P.D. ; Sahinidis, N.V.: GPU-BLAST: using graphics processors to accelerate protein sequence alignment. In: *Bioinformatics* 27 (2011), Nr. 2, S. 182. – ISSN 1367–4803

[WER+09] Weese, D. ; Emde, A.K. ; Rausch, T. ; Döring, A. ; Reinert, K.: RazerS - fast read mapping with sensitivity control. In: *Genome research* 19 (2009), Nr. 9, S. 1646. – ISSN 1088–9051

[Xil05] Xilinx: *LocalLink Interface Specification*. Julie 2005

[Xil09] Xilinx: *Virtex-5 FPGA User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug190.pdf, November 2009

[Xil10a] Xilinx: *Virtex-6 FPGA Embedded Tri-Mode Ethernet MAC Wrapper v1.4*. http://www.xilinx.com/support/documentation/ip_documentation/v6_emac_ds710.pdf, April 2010

[Xil10b] Xilinx: *Virtex-6 FPGA User Guide*. http://www.xilinx.com/support/documentation/user_guides/ug360.pdf, Januar 2010

[Xil10c] Xilinx: *Virtex-6 Libraries Guide for HDL Designs*. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/virtex6_hdl.pdf`, September 2010