Smith-Waterman - Sequence Alignment mittels OpenCL

Laurence Bortfeld (l.bortfeld@gmail.com),
Wojciech Konitzer (w.konitzer@student.htw-berlin.de)
Prüfer: Sebastian Bauer

(Ausarbeitung)

***Abstract*—Abstract**

***Index Terms*—Smith-Waterman, OpenCL, Parallelität, Sequenz Aligment, DNA**

I. Einleitung

Seit einigen Jahren steigt die Taktfrequenz von Prozessoren (CPU) nicht weiter an, dies ist durch die hohe Wärmeabgabe bei hohen Taktfrequenzen des Prozessors bedingt. Die Entwicklung vom Mehrkernprozessoren erlaubt es durch Parallelisierung dies teilweise zu kompensieren. Mittlerweile ist nicht nur die Parallelisierung auf herkömmlichen CPUs von Bedeutung. Die rasante Entwicklung von Grafikartenprozessoren (GPU) macht diese für parallele Ausführung von Programmen immer interessanter. Verantwortlich ist die wesentlich höhere Anzahl an Prozessorkernen einer GPU im Vergleich zu einer herkömmlichen CPU. Auch die schneller ansteigende Leistung der GPUs im Bezug zu CPUs lassen der Grafikkarte für Parallelisierung mehr Bedeutung zukommen. [Bau13]

Diese Arbeit befasst sich mit der Parallelisierung eines Algorithmus zur Berechnung von Aligments in zwei Sequenzen. Anwendung finden Algorithmen zur Bestimmung von Aligments zum Großteil in der Bioinformatik, um beispielsweise DNA-Sequenzen zu analysieren. Diese Arbeit betrachtet den Smith-Waterman Algorithmus, welcher das optimale lokale Alignment zweier Zeichenketten A und B ermittelt. Das verwendete Framework für die Parallelisierung auf der GPU ist OpenCL. Die Open Computing Language (OpenCL) definiert einen plattformübergreifenden Standard zum Ausführen von parallelen Anwendungen auf Mehrkern CPUs und GPUs [1]. Ziel dieser Arbeit ist es die Nebenläufigkeit des Smith-Waterman Algorithmus zu identifizieren und diesen mittels OpenCL auf der GPU zu parallelisieren. Vergleiche zwischen der seriellen und parallelen Version des Algorithmus geben Aufschluss darüber, ob eine effektive Parallelisierung des Algorithmus auf der GPU, unter Berücksichtigung der Implementierung, möglich ist.

II. Smith-Waterman Algorithmus

Der Smith-Waterman Algorithmus ist konstruiert um das optimale lokale Alignment zweier Zeichenketten oder Sequenzen zu bestimmen, somit ermittelt er den ähnlichsten Abschnitt in einer Zeichenkette. T.F. Smith und M.S. Waterman veröffentlichten den Algorithmus 1981 in dem Paper: *Identification of common molecular subsequences*. Wie der zuvor entworfene Algorithmus von Needleman & Wunsch (1970) wird mit Hilfe einer Matrix das Alignment berechnet. Es gibt eine Vielzahl von heuristischen Algorithmen, die vor der Entwicklung des Smith-Waterman Algorithmus verfasst wurden, jedoch waren diese für biologische Untersuchungen nicht ausreichend oder nicht interpretierbar. 1982 verbesserte Gotoh den Algorithmus vom Smith & Waterman. Der Ursprüngliche Algorithmus benötigte M^2N Schritte, um das lokale Alignment zu erhalten. Gotoh reduzierte die benötigten Schritte auf MN, wobei M und N ($M \geq N$) die Längen der zu vergleichenden Zeichenketten bzw. Sequenzen sind. [SW81, Got82]

Bevor jedoch der Algorithmus vom Smith & Waterman beschrieben wird, soll der Unterschied zwischen lokalen und globalen Aligments geklärt werden. Lokale bzw. globale Alignments betrachten die zu untersuchenden Sequenzen unterschiedlich und ermitteln somit verschiedene Ergebnisse. Ein globales Alignment betrachtet das Alignment auf der gesamten Länge der Sequenzen (vgl. Listing 1). Hingegen betrachtet das lokale Alignment nur ähnliche Abschnitte in einer Sequenz (vgl. Listing 1). Nun ist es möglich, dass mehrere lokale Alignments in einer Sequenz vorkommen, um das optimale lokale Alignment zu bestimmen, wählt ein Algorithmus das Alignment mir der höchsten Wertigkeit aus. Der Needleman & Wunsch Algorithmus ermittelt ein globales Alignment, indessen der Smith-Waterman Algorithmus ein optimales lokales Alignment bestimmt. [KS13]

Listing 1 Beispiel für globales und lokales Alignment

```
Global alignment:
A: A——DDAAAA–XXX————A
B: AOPIODDAAAAZXXXASDASDASDAA

Local alignment:
A: DDAAAA–XXX
B: DDAAAAZXXX
```

A. Algorithmus

Der Smith-Waterman Algorithmus basiert auf dem Paradigma der dynamischen Programmierung. Hierbei ist "Programmierung" nicht im Sinne vom Schreiben von Code zu verstehen. Die dynamische Programmierung löst das Probleme durch das Ausfüllen einer Tabelle (Matrix). Wie auch bei bei der Methode von "teile und herrsche" zerlegt die

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

dynamische Programmierung ein Probleme in viele leichter zu lösende Teilprobleme, deren Ergebnisse in einer Tabelle hinterlegt werden. Jedoch sind die Teilprobleme untereinander voneinander abhängig, da ihre Berechnungen bzw. Lösungen auf denen der Vorgänger beruhen. Generell lässt sich dynamische Programmierung auf Optimierungsprobleme anwenden. Solche Probleme bestehen aus einer Vielzahl von korrekten Lösungen, wohingegen nur eine optimale Lösung des Problems (Minima, Maxima) von Interesse ist. [Cor01] Gegeben sind zwei Zeichenketten bzw. Sequenzen $A = a_1 a_2 \ldots a_n$ und $B = b_1 b_2 \ldots b_m$. Die Ähnlichkeit zweier Elemente einer Zeichenkette (Buchstaben) sind durch die Funktion $s(a,b)$ definiert. Das Entfernen von Elementen aus der Zeichenkette ist durch das Gewicht W_k bestimmt. Für das Ermitteln von gleichartigen Segmenten in den Zeichenketten wird eine Matrix H mit den folgenden Werten initialisiert: $H_{k0} = H_{0l} = 0$ für $0 \leq k \leq n \wedge 0 \leq l \leq m$, wobei m, n die Länge der Zeichenketten $|A|$ und $|B|$ sind. Die Werte in H ergeben sich aus den Operationen (siehe Formel 1) an der Stelle H_{ij} mit den Elementen a_i und b_j, hierbei ist $1 \leq i \leq n \wedge 1 \leq j \leq m$ zu beachten. Ist das Ergebnis einer Operation *negativ* ist es *Null* zu setzen. Um in der Zeichenkette das Segment zu finden, das die größte Ähnlichkeit aufweist, muss das Element in H gefunden werden, welches den größten Wert hat. Von diesem Element ausgehend, können die nachfolgenden Elemente in einem Rückverfolgungsprozess[1] (Traceback) bestimmt werden. Der Traceback endet sobald ein Matrix Element *Null* ist. Dieser Prozess führt zu dem ähnlichsten Segmenten und zu dem optimalen Alignment der Zeichenketten A und B.[SW81]

B. Beispiel

Im Folgenden soll ein Beispiel gegeben werden, welches den Algorithmus verdeutlichen soll. Anhand der Zeichenketten $A =$ ANANAS und $B =$ BANANE, wobei die Parameter für den Vergleich wie folgt gewählt sind:

$$s(\text{match}) = 2 \wedge s(a,-) = s(-,b) = s(mismatch) = -1$$

Die Abbildungen Fig. 1(a) und 1(b) zeigen die Initialisierung der Matrix M sowie die Berechnung der jeweiligen Elemente von M durch die vorher festgelegten Operationen aus der Formel 1. Die Pfeile in der Abbildung Fig. 1(b) zeigen auf das Element aus dem sich das Element an der Stelle H_{ij} ergibt.

Ist die Matrix wie in der Abbildung Fig. 1(b) ausgefüllt, kann mittels des Tracebacks, ausgehend von dem Element mit dem höchsten Wert, die ähnlichste Sequenz aus der Zeichenkette A und B bestimmt werden (siehe Fig. 2). Aus dem Traceback ergibt sich somit ein Alignment für A und B von *ANAN*.

[1]Der Weg aus dem sich das Element im Vergleich mit den Operationen aus Formel 1 ergeben hat.

/	–	A	N	A	N	A	S
–	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0
N	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0

(a) Initialisierung der Matrix M.

/	–	A	N	A	N	A	S
–	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0
A	0	2	1	2	1	2	1
N	0	1	4	3	4	3	2
A	0	2	3	6	5	6	5
N	0	1	4	5	8	7	6
E	0	0	3	4	7	7	6

(b) Berechnen der Elemente von M mit Operationen aus der Formel 1.

Fig. 1: Initialisierung und Berechnung der Matrix M.

C. Serieller Ansatz

Die Implementierung des seriellen Ansatzes hat im Kern zwei Matrizen. Die erste Matrix H enthält die Werte, die aus den Ergebnissen der Operationen bestehen und kann wie die Matrix aus dem Beispiel II-B vorgestellt werden. Die zweite Matrix M ist für das Traceback notwendig. Sie dient für die Speicherung aus welchem Element sich das Element Hij zusammensetzt. Der Pseudocode 1 soll die Implementierung der seriellen Version des Smith-Waterman Algorithmus verdeutlichen. Die beiden ersten Zeilen initialisieren die Matrix für die Berechnungen des Alignments und die Matrix für das Merken aus welchem Element sich das aktuelle Element ergibt. Das Ausfüllen der Matrix mittels der Operationen aus Formel 1 findet in den Zeilen 3-8 statt. Zeile 6 wendet die Operationen auf das aktuelle Element H_{ij} an. In der darauffolgenden Zeile findet das Speichern der benutzten Operation statt, aus welcher sich der gegangene Pfad im Traceback rekonstruieren lässt. Die in den Zeilen 9-11 definierten Variablen dienen für den Traceback. *Result*$_a$ und *Result*$_b$ halten die im Traceback ermittelten Sequenzen der Zeichenketten A und B. *Current* speichert den aktuellen Wert, $Next$ den Folgewert, welcher mittels M bestimmt werden

/	–	A	N	A	N	A	S
–	0	0	0	0	0	0	0
B	0	0	0	0	0	0	0
A	0	2	1	2	1	2	1
N	0	1	4	3	4	3	2
A	0	2	3	6	5	6	5
N	0	1	4	5	8	7	6
E	0	0	3	4	7	7	6

Fig. 2: Traceback in M um das optimale lokale Alignment zu bestimmen.

kann. In den Zeilen 12-25 wird das optimale lokale Alignment mittels H und M ermittelt und die ähnlichsten Segmente zurückgegeben.

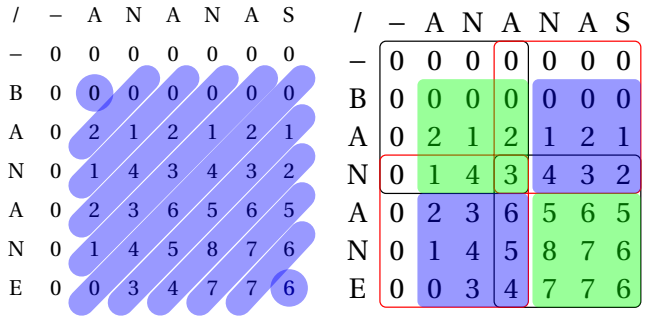
Pseudocode 1 Implementierung Smith-Waterman serieller Ansatz

```
Require: A, B ∈ [A − Z]
 1: H ← InitMatrix()
 2: M ← InitMatrix()
 3: for all 1 ≤ i ≤ n do                          ▷ Scoring the matirx
 4:     for all 1 ≤ j ≤ m do
 5:         H_ij ← FindMaximumWith(a_i, b_j)
 6:         M_ij ← MemomorizeMaxDirection()
 7:     end for
 8: end for
 9: Result_a, Result_b
10: Current ← MaximumValueIn(H)
11: Next ← GetNextWith(M, Current)
12: while Current ≠ Next ∧ Next ≠ 0 do            ▷ Traceback
13:     if Next_i = Current_i then                ▷ Deletion in A
14:         Result_a ← '-'
15:     else                                      ▷ Match/Mismatch in A
16:         Result_a ← A[Current]
17:     end if
18:     if Next_j = Current_j then                ▷ Deletion in B
19:         Result_b ← '-'
20:     else                                      ▷ Match/Mismatch in B
21:         Result_b ← B[Current]
22:     end if
23:     Current = Next
24:     Next ← GetNextWith(M, Current)
25: end while
26: return Result_a, Result_b
```

D. Paralleler Ansatz mittels OpenCL

Für eine parallele Implementierung des Smith-Waterman Algorithmus ist das Erkennen der Abhängigkeiten der Operatoren, die in der Formel 1 definiert sind, notwendig. Aus den Indizes der benutzten Operatoren ($i-1, j-1$, $i-1, j$ und $i, j-1$), um auf Elemente aus H zuzugreifen, geht hervor, dass die Operationen von Vorgängerwerten in der Matrix abhängen. Dies macht es nicht möglich H Zeile für Zeile parallel auszufüllen. Eine andere Herangehensweise ist es die Matrix in Antidiagonalen[2] zu berechnen (siehe Fig. 3(a)). Die Herausforderung hierbei ist, dass sich die Längen der Antidiagonalen in Abhängigkeit von n und m ändern. Somit müssen die Antidiagonalen dementsprechend berechnet werden. Ein weiteres Problem bei diesem Ansatz ist, dass es zu einem Overhead bei der parallelen Ausführung kommet, da die jeweiligen Antidiagonalen mit den Elementen von denen Sie abhängen auf die entsprechenden Prozessoren verteilt werden müssen. Eine bessere Vorgehensweise ist es die Matrix H in mehrere Blöcke zu unterteilen (siehe Fig. 3(b)). Jeder Block ist nur von seinem Vorgänger abhängig. Die schwarzen und roten Rahmen in der Abbildung Fig. 3(b) zeigen den gesamten

[2]Die Addition von i und j gibt pro Antidiagonale immer den selben Wert.



(a) Jedes Element in einer Antidiagonalen kann parallel berechnet werden.

(b) Jede Submatrix kann pro Antidiagonale (grün, blau) parallel berechenet werden.

Fig. 3: Parallele Ausführung des Smith-Waterman Algorithmus mittels Antidiagonalen. Antidiagonale müssen seriel berechnete werden. Jedoch sind ihre Element voneinander unabhängig

Block, der für eine Berechnung betrachtet wird. Die grün und blau ausgefüllten Blöcke sind die Werte, die in einem Block berechnet werden. In der Abbildung Fig. 3(b) ist zu erkennen, dass sobald der erste grüne Block berechnet ist, sich die beiden nachfolgenden blauen Blöcke voneinander unabhängig berechnen lassen. Zu beachten ist, dass für jede Berechnung eines Blockes die erste horizontale und vertikale Zeile der vorherigen Berechnung mitgeführt werden muss. Dies führt aber zu einem geringeren Overhead als die Herangehensweise mit Antidiagonalen. [MCU+01]

1) OpenCL[3]: Die Open Computing Language ist ein Plattform übergreifender Standard, der für das Entwickeln von Anwendungen auf hoch parallelen Prozessoren entworfen ist. Basierend auf der C99 Programmiersprache können Kernel auf der CPU oder GPU ausgeführt werden. Die OpenCL Spezifikation bezeichnet Geräte auf denen das Programm ausgeführt wird im allgemeinen als *Device*. Jedes *Device* besteht aus einer Reihe von *Compute Units*. Die *Compute Unit* besteht aus mehreren Prozessorelementen und einem lokalen Speicher. Eine *Workgroup* wird auf einer *Compute Unit* ausgeführt, die *Compute Units*, welche den Prozessorkernen entsprechen limitieren die gleichzeitige Ausführung. Das dabei auf der GPU oder CPU ausgeführte Programm, heißt *Kernel*. Der *Kernel* ist eine Funktion, die in der OpenCL Sprache verfasst ist und zur Laufzeit für das entsprechende *Device* kompiliert wird. Während der gleichzeitigen Ausführung rechnen eine Vielzahl von "Kernel Instanzen" auf verschiedenen Bereichen der zu bearbeitenden Daten. Der *Host*, das Programm, welches den *Kernel* aufruft, alloziert Speicher für die benötigten Argumente der *Kernel* Funktion und übergibt sie dieser beim Aufruf. Nach der Abarbeitung der Daten, können diese wieder ausgelesen und der vorher allozierte Speicher freigegeben werden. [1]

2) SimpleOpenCL: Aus dem OpenCL Beispiel des Apple Developer Guide [2] geht hervor, dass die Programmierung mittels OpenCL einen Großteil von Boilerplate Code erzeugt. Dieser ist für die Initialisierung von OpenCL, das Allozieren und Kopieren von Speicher auf die *Devices* notwendig, sowie

[3]Dieser Abschnitt soll lediglich einen Einblick in die Funktionsweise OpenCLs geben, jedoch keine Einführung in die Programmierung mittels OpenCL, da es eine Vielzahl davon im Internet zu finden ist.

das Ausführen des *Kernel*, das Abrufen von Ergebnissen und das Freigen und Beenden von OpenCL. Er macht den Quellcode unübersichtlich, den Umgang mit OpenCL aufwändig und schwer wartbar. SimpleOpenCL implementiert OpenCL, bietet jedoch über eine C Bibliothek eine sehr vereinfachte Schnittstelle zu OpenCL. Ausgerichtet für die Bedürfnisse wissenschaftlicher Untersuchungen und die Entwicklung von Prototypen, ermöglicht SimpleOpenCl einen einfacheren Umgang mit OpenCL und erleichtert die Entwicklung erheblich. Durch SimpleOpenCL lässt sich der "Host Code" um den Kernel auszuführen stark reduzieren. Dabei bleibt der "Kernel Code" exakt der gleiche wie bei einer regulären OpenCL Implementierung. [3]

3) Implementierung: Anstatt nativen OpenCL "Host Code" zu schreiben, verwendet die Implementierung für den Host SimpleOpenCL. Wie auch die serielle Implementierung des Smith-Waterman Algorithmus basiert auch die parallele Ausführung auf den Matrizen H und M. Wie in II-D vorgeschlagen, werden Teilblöcke (i.f. Submatritzen) entlang einer Antidiagonalen von H parallel berechnet. Für die Berechnung der Antidiagonalen kommt der aus [6] vorgeschlagene Algorithmus zum tragen. Der Algorithmus ist so angepasst, dass er die Startindizes der Submatritzen ermittelt. Mit der festgelegten Dimension d für jede Submatrix werden die Elemente jeder Submatrix gegeben. Listing 2 weist den Algorithmus für die Berechnung der Submatritzen aus einem Quellcodeabschnitt auf. *sub* entspricht hierbei d für jede Submatrix und *n* der Länge der Zeichenketten. Zur vereinfachten Berechnung wird vorausgesetzt, dass die Längen der Zeichenketten A und B gleichlang sind, also $m = n$. Des Weiteren muss d so gewählt sein, dass es m bzw. n ohne Rest teilt ($m \bmod d = 0 \wedge n \bmod d = 0$), ansonsten ist es nicht möglich H in gleiche Submatritzen zu zerteilen. Der Pseudocode 2 zeigt den Ablauf in vereinfachter Form für die parallele Ausführung. Die Operatoren aus Formel 1 können parallel in *computeSubmatrixsInParallel* auf die Submatritzen angewandt werden. Lediglich der Traceback, um auf die ähnlichen Teilsegmente und das Alignment der Zeichenketten A und B zukommen ist wieder seriell.

Listing 2 Berechnung der Startindizes für Submatritzen.

```
int max = n / sub;
int slice, j;
for (slice = 0; slice < 2 * max - 1; ++slice)
{
  int z = slice < max ? 0 : slice - max + 1;
  for (j = z; j <= slice - z; ++j)
  {
    int startI = j
    int startJ = (slice  - j) ;
     // ...
  }
  // ...
}
```

Pseudocode 2 Implementierung Smith-Waterman paralleler Ansatz

Require: $A, B \in [A - Z]$
1: $H \leftarrow$ InitMatrix()
2: $M \leftarrow$ InitMatrix()
3: **for all** $1 \leq i \leq n$ **do** ▷ Scoring the matirx
4: SubMatrixs $\leftarrow$ getSubmatrixFor(i)
5: computeSubmatrixsInParallel(SubMatrixes, A, B)
6: **end for**
7: doSerialTraceBackWith(H, M)
8: **return** Result$_a$, Result$_b$

III. DNA

Die DNA (Desoxyribonukleinsäure) ist ein Biomolekül und ist Bestandteil jedes Lebewesen und Viren. Es besteht aus vielen Bestandteilen, den sogenannten Nukleotiden. Jedes Nukleotid besteht aus Phosphorsäure bzw. Phosphat und Zucker (Desoxyribose) sowie einer einer Base. Bei der Base kann es sich um Adenin (A), Thymin (T), Cytosin (C) oder Guanin (G) handeln. Die Phosphorsäure und der Zucker sind immer gleich und bilden den Strang des DNA Moleküls. Dabei bilden immer zwei Nukleotide anhand ihrer Basen ein Basenpaar. Es können jedoch nur Basenpaare aus Adenin und Thymin oder Cytosin und Guanin gebildet werden (siehe Abbildung 4). Anhand der Komplexität der DNA Sequenz, die beim Menschen aus 3.101.788.170 Basenpaaren besteht, erfolgt die DNA Sequenzierung abschnittsweise. [4]

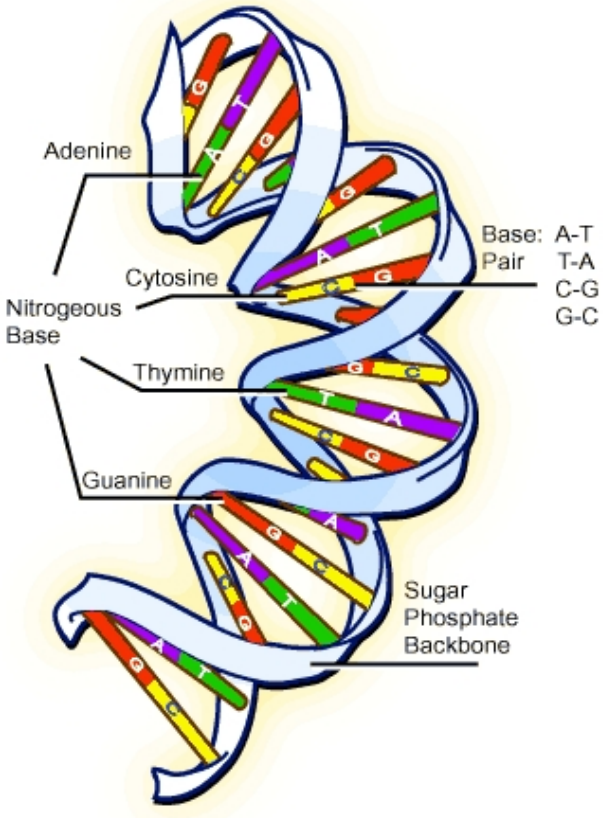


Fig. 4: Strukturmodell eines DNA-Moleküls [5]

A. Aligments in der DNA

Der Vergleich (Alignment) von DNA Sequenzen spielt eine wichtige Rolle in der Forschung, Medizin, Forensik und Bioinformatik. Es ist mit Hilfe von Algorithmen, wie dem

Smith-Waterman Algorithmus möglich zwei DNA Sequenzen miteinander zu vergleichen. Stellt man eine biologische Sequenz als einen eindimensionalen Zeichenkette dar, so kann der Vergleich der beiden Sequenzen als zeichenweiser Vergleich von diesen Zeichenketten verstanden werden. Dazu werden als Zeichen die Abkürzungen der Basenpaare (AT, TA, CG, GC) verwendet. Ein sehr kurzes Beispiel einer Sequenz könnte, wie folgt sein: "CGCGATCGATCGTACG". Ziel ist es den Grad Ähnlichkeit bzw. Unähnlichkeit zu bestimmen.

In der Forensik können DNA Sequenzen mit einer vorgegebenen Sequenz verglichen werden, um Täter zu identifizieren. In der Forschung können z.B. Spezies mit anderen Spezies oder defekte DNA Sequenzen mit korrekten DNA Sequenzen verglichen werden. [4]

IV. Ergebnisse

Für das Ermitteln der Performance, der seriellen bzw. parallelen Ausführung des Smith-Waterman Algorithmus wurden zufällige Zeichenketten generiert und untereinander verglichen. Dieser Prozess wurde einige Male wiederholt. Die zum Zeitpunkt der Messung vorhandene Hardware war ein MacBook Pro mit einem Intel Core 2 Duo @ 2,53 GHz, 8GB RAM und einer NVIDIA GeForce 9400M 256 MB. Tabelle IV.1 zeigt einige Messungen. Aus den Messergebnissen gehen zwei Schlussfolgerungen hervor. Zum einen scheint das Kopieren des Speichers auf die Grafikkarte wesentlich zeitintensiver als das Kopieren des Speicher für die CPU zu sein. Dies ist erkennbar daran, dass die Ausführung auf der GPU wesentlich länger dauert. Zum anderen erhöht sich die Ausführungszeit je kleiner die Dimension d der Submatrizen wird. Dies ist dadurch bedingt, dass öfter Speicher kopiert wird.

Nr	Durchläufe	Zeichenkettenlänge	Device	d	t_s
1	100	20	seriell	10	< 1
2	100	20	CPU	10	< 1
3	100	20	GPU	10	2
4	100	100	seriell	20	< 1
5	100	100	CPU	20	1
6	100	100	GPU	20	11
7	1000	50	seriell	10	≤ 1
8	1000	50	CPU	10	1-2
9	1000	50	GPU	10	42
10	1000	50	seriell	50	≤ 1
11	1000	50	CPU	50	1-2
12	1000	50	GPU	50	21
13	1000	30	GPU	15	16
14	1000	30	GPU	30	13
15	10000	40	seriell	40	1
16	10000	40	CPU	40	17
17	10000	40	CPU	20	50

TABLE IV.1: Auswertung der Messergebnisse seriell vs. parallel

V. Zusammenfassung

In der Arbeit konnte gezeigt werden, dass eine parallele Implementierung des Algorithmus von Smith & Waterman möglich ist. Jedoch ist die parallele Ausführung zum jetzigen Zeitpunkt nicht so performant wie der serielle Ansatz. Dies kann zum einen daran liegen, dass der Algorithmus auf dynamischer Programmierung basiert, dessen Grenze zum großen Teil der Speicher ist und nicht die Geschwindigkeit mit welcher berechnet wird. Zum anderen handelt es sich bei der benutzten Grafikkarte um ein Notebook Model. Dieses hatte bei hoher Last Displayflackern, Abstürze der Anwendung oder extrem lange Laufzeiten zur Folge. Trotz längerer Laufzeit sind diese Effekte in großen Ausmaß bei der CPU als OpenCL Device nicht aufgetreten. Im Rahmen dieser Arbeit konnte ebenfalls nicht festgestellt werden, ob eine bessere Performance in nativem OpenCL Host Code anstelle von SimpleOpenCL möglich ist.

VI. Ausblick

Aufgrund von SimpleOpenCL ist es erschwert möglich auf die Speicherverwaltung von OpenCL zu zugreifen. Von daher wäre eine Implementierung ohne SimpleOpenCL interessant. Jedoch hat diese nicht mehr in den Rahmen dieser Arbeit gepasst hat. Des Weiteren könnten Messungen unter aktuellerer und performanterer Hardware bessere Ergebnisse erzielen.

Eine andere Herangehensweise an die Parallelisierung hätte eventuell auch Auswirkungen auf eine schnellere Ausführung. Beispielsweise ließe sich der komplette Algorithmus als Kernel ausführen, nur die Aufrufe mit verschiedenen Zeichenketten würden parallel ausgeführt werden. Somit gäbe es keine Abhängigkeiten der Operationen des Algorithmus auf der Matrix H, da der Kernel Code selber seriell ausgeführt werden könnte.

Literatur

[Bau13] Bauer, Sebastian: *Parallel Systems: Mehrkern-Prozessoren*. Presented as Lecture SoSe 2013, 2013

[Cor01] Cormen, Thomas H.: *Introduction to algorithms*. 2. ed. Cambridge, Mass. [u.a.] : Cambridge, Mass. [u.a.] : MIT Press [u.a.], 2001

[Got82] Gotoh, Osamu: An Improved Algorithm for Matching Biological Sequences. In: *J. Mol. Biol*, 1982, S. 705–708

[KS13] Kohlbacher, Oliver ; Schmid, Steffen: *Bioinformatik für Lebenswissenschaftler: Paarweises Alignment – Teil II*. Presented as Lecture SoSe at Eberhard Karls Universität Tübingen, 2013

[MCU+01] Martins, W.S. ; Cuvillo, J.B. D. ; Useche, F.J. ; Theobald, K.B. ; Gao, G.R.: A MULTITHREADED PARALLEL IMPLEMENTATION OF A DYNAMIC PROGRAMMING ALGORITHM FOR SEQUENCE COMPARISON. In: *Communications of the ACM* (2001)

[SW81] Smith, T.F. ; Waterman, M.S.: Identification of Common Molecular Subsequences. In: *J. Mol. Biol*, 1981, S. 195–197

Internetquellen

[1] Apple Inc. Opencl programming guide for mac. Website, 2013. https://developer.apple.com; Abruf am 01.05.13.

[2] Apple Inc. Opencl programming guide for mac. Website, 2013. http://developer.apple.com/library/mac/samplecode/OpenCL_Hello_World_Example/Introduction/Intro.html; Abruf am 01.05.13.

[3] morousg@gmail.com. simple-opencl. Website, 2013. https://code.google.com/p/simple-opencl/; Abruf am 25.07.13.

[4] Wikipedia. Desoxyribonukleinsäure. Website, 2013. http://de.wikipedia.org/wiki/Desoxyribonukleins%C3%A4ure; Abruf am 01.05.13.28.13.

[5] Nuno Roma. Design of efficient co-processing structures for dna sequence alignment. Website, 2010. http://sips.inesc-id.pt/~nfvr/msc_theses/msc09b/; Abruf am 28.07.13.

[6] Mark Byers. Opencl programming guide for mac. stackoverflow.com, 2009. http://stackoverflow.com/questions/1779199/traverse-matrix-in-diagonal-strips; Abruf am 29.06.13.