B. Juurlink, W. Karl (Hrsg.)
Proceedings 28th PARS Workshop

GPU-beschleunigte Time Warping-Distanzen

Jörg P. Bachmann[1] Kevin M. Trogant[2] Johann-C. Freytag[3]

Abstract: Immer mehr Algorithmen konnten durch Implementierung auf GPUs um mehrere Größenordnungen beschleunigt werden. Insbesondere existieren hochparallele Implementierungen des im Bereich der Zeitreihenanalyse weit verbreiteten Algorithmus' Dynamic Time Warping (`DTW`). Dieser Algorithmus berechnet einen Ähnlichkeitswert zweier Zeitreihen (z. B. Temperaturverläufe) unter Berücksichtigung zeitlicher Variationen wie z. B. zeitliche Verschiebungen. Leider können die existierenden GPU-Implementierungen von `DTW` nicht beliebige zeitliche Variationen berücksichtigen.

In dieser Arbeit stellen wir Implementierungen für GPUs vor, die dieser Einschränkung nicht unterliegen. In unserer Evaluierung zeigen wir, dass sie einen Geschwindigkeitsvorteil von ca. zwei Größenordnungen gegenüber einer CPU-Implementierung erreichen.

Keywords: Dynamic Time Warping; GPU; Algorithmen

1 Einleitung

Viele Algorithmen konnten durch Parallelisierung auf moderner Hardware (z. B. GPUs) erfolgreich beschleunigt werden. Dazu gehören u. a. Sortieralgorithmen [Ar17], Algorithmen aus der linearen Algebra und dem Machine Learning [Ch14; TDB10] und selbstverständlich unzählige Algorithmen aus der Computergrafik.

Zahlreiche Beispiele von Adaptionen der Algorithmen auf die GPU kommen aus der Ähnlichkeitssuche, in der aus einer großen Menge von Objekten (der *Kandidatenmenge*) diejenigen gesucht werden, die einem *Anfrageobjekt* ähnlich sind: Bereits mit älterer Hardware aus 2009 konnte der für Sequenzalignment häufig verwendete Smith-Waterman Algorithmus um etwa eine Größenordnung beschleunigt werden [SA09]; Das im Bereich der Ähnlichkeitssuche häufig verwendete Dynamic Time Warping (`DTW`) [SC90] konnte durch GPUs um ein bis zwei Größenordnungen beschleunigt werden [HSS14; Sa10]; Durch Implementierung des R-Baumes [Be90] auf die GPU konnten die Ausführungszeiten von Ähnlichkeitssuchen um ca. eine Größenordnung beschleunigt werden [YZG13].

`DTW` berechnet ein Maß für die Ähnlichkeit (bzw. Distanz) zweier Zeitreihen, das robust gegen zeitliche Verzerrungen ist. Existierende Implementierungen des Algorithmus' beschränken die möglichen zeitlichen Verzerrungen, um die Berechnung zu beschleunigen [HSS14] oder

[1] Joerg.Bachmann@informatik.hu-berlin.de
[2] Kevin.Trogant@informatik.hu-berlin.de
[3] Freytag@informatik.hu-berlin.de

spezialisieren sich auf die Suche von Subsequenzen ausschließlich fester Länge innerhalb einer langen Zeitreihe [Sa10].

In dieser Arbeit stellen wir neue Implementierungen von DTW für GPUs vor, die von keiner der eben genannten Einschränkungen betroffen sind. Beginnend mit einer Variante für Zeitreihen beschränkter Länge führen wir anschließend Varianten für Sub- und Supersequenzsuche sowie die Möglichkeit, beliebig lange Zeitreihen zu verarbeiten, ein. Abschließend evaluieren wir die Implementierungen ausführlich und zeigen, dass mit Hilfe der GPU Geschwindigkeitsvorteile gegenüber einer CPU-Implementierung von ca. zwei Größenordnungen erreicht werden können. Um den Vergleich nachvollziehbar und möglichst unabhängig von der konkreten CPU zu gestalten, haben wir uns für eine sequentielle Implementierung auf der CPU entschieden. Zwar lassen sich Zeitreihen nicht unter DTW mittels metrischer Indexstrukturen [BKL06; CPZ97; NB09] indizieren, doch wir haben mit dieser Arbeit eine Grundlage geschaffen, Zeitreihen unter DK[4] [Fr06] mittels metrischer Indexstrukturen GPU-beschleunigt zu indizieren.

In Kapitel 2 stellen wir die DTW- und DK-Distanzfunktion ausführlich vor. Die Implementierungen werden in Kapitel 3 beschrieben und in Kapitel 4 evaluiert. Wir schließen mit einem kurzen Fazit in Kapitel 5 ab.

2 Time Warping Distanzfunkionen

Dynamic time warping (DTW) ist eine Funktion, die zwei Zeitreihen einen Abstandswert zuordnet [SC90]. Sie wurde ursprünglich im Bereich der Spracherkennung eingeführt und findet noch heute weite Verbreitung in der Analyse von Zeitreihen. Die Fréchet-Distanz (DK) ist eine Alternative zu DTW, welche die Dreiecksungleichung erfüllt[5].

Einfach formuliert versucht DTW (bzw. DK), den Abstand der Trajektorien (also die Menge der besuchten Punkte im Raum) zweier Kurven zu messen. Betrachtet man zum Beispiel den Weg zweier Personen durch eine Stadt entlang der selben Route (Trajektorie), so gibt es möglicherweise zeitliche Unterschiede durch unterschiedliches Lauftempo und unterschiedlicher Ampelschaltungen. Während sich die Kurven also stark unterscheiden können (wenn man die Punkte der Wege zu jeweils gleichen Zeiten vergleicht), können die Trajektorien identisch sein.

DTW ist für zwei Zeitreihen $S = (s_1, \cdots, s_m)$ und $T = (t_1, \cdots, t_n)$ wie folgt rekursiv definiert:

$$\begin{aligned} \texttt{DTW}(S, ()) &= \infty \\ \texttt{DTW}((), T) &= \infty \\ \texttt{DTW}((s), (t)) &= d(s,t)^2 \end{aligned} \qquad \texttt{DTW}(S,T) = d(s_1,t_1)^2 + \min \begin{cases} \texttt{DTW}(\texttt{Tail}(S), \texttt{Tail}(T)) \\ \texttt{DTW}(S, \texttt{Tail}(T)) \\ \texttt{DTW}(\texttt{Tail}(S), T) \end{cases}$$

[4] DK ist eine Alternative zu DTW, welche die Dreiecksungleichung erfüllt, vollständig invariant unter zeitlichen Verzerrungen ist und sehr ähnlich zu DTW implementiert wird [BF17].

[5] Dadurch eignet sich die DK-Distanz zur Indizierung von Zeitreihen mittels metrischer Indexstrukturen [BKL06; CPZ97; NB09]

wobei $\texttt{Tail}(T) := (t_2, \cdots, t_n)$. Die Abstandsfunktion $d(s,t)$ wird vom Nutzer definiert und entspricht im einfachsten Fall dem Betrag der Differenz zweier Zahlen. Die DK-Distanz ändert sich nur in der Art der Kombination des rekursiven Ergebnisses mit dem aktuellen Abstand beider Zeitreihenelemente:

$$\begin{aligned} \texttt{DK}(S, ()) &= \infty \\ \texttt{DK}((), T) &= \infty \\ \texttt{DK}((s), (t)) &= d(s,t)^2 \end{aligned} \qquad \texttt{DK}(S,T) = \max\left(d(s_1,t_1)^2, \min\begin{Bmatrix} \texttt{DK}(\texttt{Tail}(S), \texttt{Tail}(T)) \\ \texttt{DK}(S, \texttt{Tail}(T)) \\ \texttt{DK}(\texttt{Tail}(S), T) \end{Bmatrix} \right)$$

Da beide Funktionen so ähnlich in der Berechnung sind, erläutern wir im Rest dieses Kapitels nur DTW. Alle Aussagen gelten analog für die DK-Distanz.

Algorithmen, die DTW berechnen, werden üblicherweise mittels dynamischer Programmierung implementiert [SC90]. Abbildung 1 zeigt eine Beispielrechnung. Es wird das Kreuzprodukt beider Zeitreihen S und T gebildet, sodass eine Matrix entsteht, welche die paarweisen Abstände $\|s_i - t_j\|$ der einzelnen Elemente der Zeitreihen enthält. Die Matrix wird so angeordnet, dass sich die Zelle mit dem Abstand $\|s_1 - t_1\|$ links unten befindet.

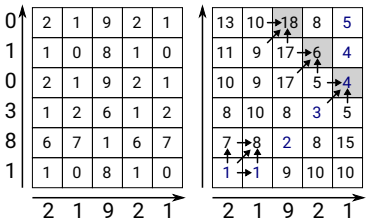

Abb. 1: Beispielmatrizen während der Berechnung von DTW (links das Kreuzprodukt der am Rand stehenden Zeitreihen und rechts die DTW-Matrix mit einem fett markierten optimalen Pfad). Die Pfeile repräsentieren exemplarisch die Abhängigkeiten der Berechnung der Zellen.

Anschließend wird ein Pfad von links unten nach rechts oben gesucht, der die Summe im Fall der besuchten Einträge minimiert. Dabei muss der Pfad zusammenhängend sein und darf niemals nach links oder unten gehen, d. h. innerhalb des Pfades muss der Nachfolger der Zelle mit den Indizes (i, j) entweder $(i+1, j)$, $(i, j+1)$ oder $(i+1, j+1)$ sein. Die Summe der Zellen entlang eines solchen *minimalen* Pfades ergibt das Ergebnis der DTW-Funktion.

Zur Berechnung eines minimalen Pfades wird jede Zelle durch die Summe ihres Inhalts und der kleinsten Vorgängerzelle ersetzt. Um eine Zelle zu ersetzen, müssen die Vorgängerzellen bereits nach diesem Schema angepasst worden sein, sodass eine Abhängigkeit von links unten nach rechts oben entsteht. Es ist leicht zu sehen, dass dieser Algorithmus eine quadratische Komplexität hat (genauer $m \times n$, wenn m und n die Längen der Zeitreihen sind). Mehr noch wurde gezeigt, dass es keine Algorithmen geben kann, die eine bessere Komplexität haben [BK15; Br14]. Algorithmus 1 stellt den Pseudo-Code für eine auf

dynamischer Programmierung basierende Implementierung zur Verfügung. Im folgenden Kapitel 3 implementieren wir den Algorithmus auf der GPU.

Algorithmus 1 CPU-Implementierung von DTW

```
1  Eingabe: Zeitreihen A und B; Ausgabe: Distanz DTW(A, B)
2  M := m × n Matrix
3  for x = 1 to n
4    for y = 1 to m
5      if x = 1 and y = 1 then M_{1,1} = |A_1 − B_1|
6      else if x = 1 and 1 < y then M_{y,1} = M_{y−1,1} + |A_y − B_1|
7      else if x ≤ n and y = 1 then M_{1,x} = M_{1,x−1} + |A_1 − B_x|
8      else if x ≤ n and 1 < y then
9        M_{y,x} = min{M_{y−1,x}, M_{y,x−1}, M_{y−1,x−1}} + |A_y − B_x|
10 return M_{m,n}
```

3 Implementierung

In diesem Kapitel stellen wir verschiedene Implementierungen für die Berechnung von DTW bzw. DK vor. Da sich die Funktionen nur in einer Operation unterscheiden (DK wählt das Maximum zweier Werte, die von DTW addiert werden), erläutern wir nur die Implementierung von DTW.

Die in Kapitel 3.1 vorgestellte Implementierung liefert die Basis für die weiteren Implementierungen. Sie unterstützt nur die Eingabe kleiner Zeitreihen, da diese vollständig in den lokalen Speicher der GPU kopiert werden. In Kapitel 3.2 zeigen wir, wie die Implementierung so abgeändert werden kann, dass sie die Sub- und Supersequenzsuche unterstützt. In Kapitel 3.3 erweitern wir die Basis-Implementierung, sodass eine der beiden Eingabezeitreihen beliebig lang sein kann. Diese Implementierung ist orthogonal mit der Sub- bzw. Supersequenz-Implementierung kombinierbar, sodass ein Algorithmus für GPU-beschleunigte Sub- bzw. Supersequenzsuche mit DTW bzw. DK zur Verfügung gestellt wird. Aus Platzgründen gehen wir darauf nicht weiter ein.

3.1 Basis-Implementierung

Aus Abbildung 1 ist zu erkennen, dass die Berechnung der Zellen innnerhalb einer Diagonalen unabhängig voneinander sind. Zerlegt man die DTW-Matrix in solche Diagonalen von links unten nach rechts oben beginnend (vgl. Abb. 2), so ist die Berechnung einer Diagonalen nur von den Werten der beiden Vorgängerdiagonalen abhängig. Basierend auf dieser Erkenntnis berechnen alle hier vorgestellten GPU-Implementierungen die DTW-Matrix sukzessive von links unten nach rechts oben, halten stets nur die letzten drei Diagonalen im lokalen Speicher und parallelisieren innerhalb der aktuell zu berechnenden Diagonalen.

Algorithmus 2 enthält den relevanten Ausschnitt des CUDA C Codes für die Berechnung von DTW[6].

Algorithmus 2 Basis Implementierung

```
1  // t: Kandidat; q: Anfrage; jeweils vom globalen in den lokalen Speicher kopiert
2  // c: Aktuelle−, p; Letzte−, p2: Vorletzte Diagonale; jeweils im lokalen Speicher
3  // R: Rückgabewert im globalen Speicher
4  // TL: Kandidatenlänge, QL: Anfragelänge,
5  if (threadIdx.x == 0) p2[0] = 0.0f;
6  for (int i = 0; i < QL + TL - 1; ++i) {
7    const int x = threadIdx.x, y = i - x;
8    if (y>=0 && y<TL) c[x+1] = abs(t[y]-q[x])+min(p[x+1],p[x],p2[x]);
9    if (threadIdx.x == 0) p2[0] = INFINITY;
10   __syncthreads();
11   volatile float *_t = p2; p2 = p; p = c; c = _t;
12 }
13 if (threadIdx.x == 0) R[blockIdx.x] = p[QL];
```

Da die Zugriffszeiten auf den lokalen Speicher um Größenordnungen geringer sind als auf den globalen Speicher, kopiert unsere Implementierung beide Zeitreihen zunächst in den schnellen lokalen Speicher (Variablen t und q) und berechnet die Diagonalen der DTW-Matrix ebenfalls im lokalen Speicher (Variablen c, p und p2). Ferner werden Diagonalen in Arrays abgelegt, sodass die Threads jegliche Speicherzugriffe ohne Verzögerung durch Bankkonflikte durchführen.

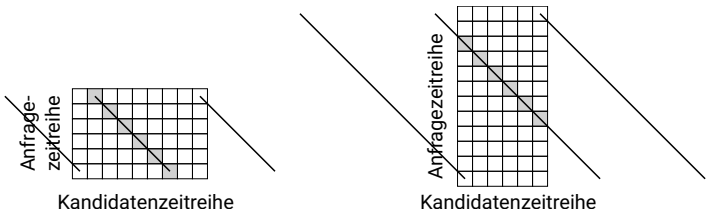

Abb. 2: Skizze zur Basis-Implementierung von DTW: Diagonale Linien repräsentieren Threads, die auf den Zellen der DTW-Matrix arbeiten.

Die maximale Länge der Diagonalen wird durch die Länge der kürzeren Zeitreihe bestimmt. Um jedoch den Programmcode innerhalb der Schleife (Zeile 6 bis 12) möglichst einfach, instruktionsarm und damit schnell zu gestalten, setzen wir die Länge einer Diagonalen mit der Länge der Anfragezeitreihe gleich (vgl. Abbildung 2). Diese Herangehensweise ist effizient, falls die Anfragezeitreihe kürzer ist, als der zu vergleichende Kandidat. Falls die Anfragezeitreihe länger ist, so reserviert die Implementierung für jede Diagonale mehr

[6] Eine vollständige Implementierung ist hier zu finden: http://www.dbis.informatik.hu-berlin.de/fileadmin/projects/GPUAlgorithms/dtw_on_gpus.tar.gz

Speicher, als notwendig ist. Da DTW jedoch symmetrisch bzgl. der Eingabeparameter ist, können im zweiten Fall die Parameter einfach vertauscht werden, um wiederum auf den ersten effizienten Fall zurückzugreifen.

Wie bereits erwähnt, sind die Zellen innerhalb einer Diagonalen unabhängig voneinander, sodass die Länge der Diagonalen gleichzeitig den Grad der Parallelität innerhalb eines Blocks angeben. Darüber hinaus kann durch die Anzahl der zu berechnenden DTW-Instanzen (also die Anzahl der CUDA-Blöcke) parallelisiert werden. Die hohe Geschwindigkeit unserer Implementierung resultiert also aus dem Arbeiten im lokalen Speicher sowie der maximal möglichen Parallelisierung bei minimaler und bankkonfliktfreier Speicherverwaltung.

3.2 Sub- und Supersequenzanfragen

Um auch Sub- und Supersequenzanfragen beantworten zu können, verfolgen wir den Ansatz von S-DTW [AF13; Mü07], wobei im Gegensatz zu DTW der minimale Pfad in einer beliebigen Zelle am linken Rand der DTW-Matrix startet und (rechts davon) in einer beliebigen Zelle des oberen Randes der DTW-Matrix endet (siehe Abbildung 3).

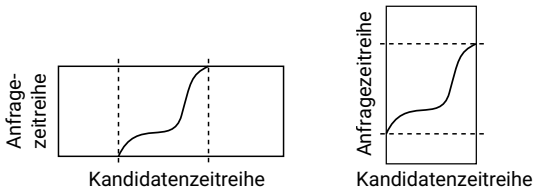

Abb. 3: Skizze der DTW-Matrix zur Berechnung von S-DTW (links: Sub-, rechts: Supersequenzsuche).

Die Implementierung ändert sich dazu nur geringfügig: Jede Zelle der untersten Zeile der DTW-Matrix wird behandelt, als wäre sie die Zelle der linken unteren Ecke in der Basis-Implementierung, d. h. ihre Berechnung ist nicht mehr abhängig von ihrem linken Nachbarn. Das Ergebnis der Berechnung entspricht dem kleinsten Wert der obersten Zeile statt des Wertes der Zelle der rechten oberen Ecke. Analog dazu wird in der Supersequenzsuche ein minimaler Pfad von linken Spalte zur rechten Spalte berechnet.

3.3 Implementierung für lange Zeitreihen

Die Größe des lokalen Speichers bestimmt die maximale Länge der verarbeiteten Zeitreihen bei der Basis-Implementierung. Wir ändern die Implementierung derart, dass kurze Anfragezeitreihen gegen beliebig lange Kandidatenzeitreihen verglichen werden können.

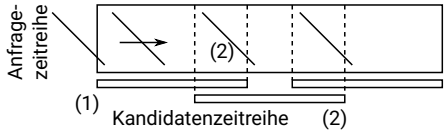

Abb. 4: Skizze der DTW-Matrix bei langen Kandidatenzeitreihen (oben); der überlappenden Fenster (unten); und der berechnenden Threads (Diagonalen oben).

Abbildung 4 skizziert den Ablauf: Wir laden stets nur ein Fenster der langen Kandidatenzeitreihe in den lokalen Speicher (1). Sobald die zu berechnende Diagonale Daten benötigt, die nicht im aktuellen Fenster liegen, wird das nächste Fenster geladen (2). Auf diese Weise laden wir die Zeitreihe sukzessiv in sich überlappende Fenster in den lokalen Speicher.

Um die Daten des überlappenden Bereichs nicht mehrfach aus dem globalen Speicher zu kopieren, verschieben wir sie innerhalb des Puffers im lokalen Speicher. Beim Verschieben der Daten können Race Conditions beim Lesen bzw. Schreiben verschiedener Threads auf die selbe Speicherstelle auftreten. Diese Konflikte treten nicht auf, sobald das Fenster mindestens doppelt so lang wie der überlappende Bereich (d. h. der Länge der Anfragezeitreihe) ist. Wir verlangen im Folgenden solche Fenstergrößen, um die Kosten entsprechender Synchronisierungsbefehle einzusparen.

4 Evaluierung

In diesem Kapitel evaluieren wir die in Kapitel 3 vorgestellten Implementierungen von `DTW` auf der GPU und zeigen, dass unsere GPU-Implementierungen bis zu zwei Größenordnungen schneller sind, als ihr CPU-Pendant. Da sich die Laufzeiten der Sub- und Supersequenzvarianten kaum messbar von der Basisimplementierung unterschieden, verzichten wir hier auf eine Präsentation der Ergebnisse.

In unseren Experimenten haben wir alle GPU-Algorithmen auf einer NVIDIA GeForce GTX 980 Ti ausgeführt. Der CPU-Algorithmus wurde auf einem AMD Ryzen 7 1700X Prozessor ausgeführt. Vereinzelte Experimente wurden auf einer NVIDIA GeForce GTX 780 Ti ausgeführt.

Datensatzeigenschaften Da die Anzahl der Rechenoperationen in den Algorithmen offensichtlich unabhängig von den konkreten Inhalten der Zeitreihen ist, haben wir ausschließlich synthetische Daten benutzt. Die einzigen Größen, von denen die Laufzeit abhängig ist, sind die Länge der Zeitreihen, die Fenstergröße bei der GPU-Implementierung für lange Zeitreihen sowie die Größe des Datensatzes. Da die GPU-Implementierungen von `DTW` kein symmetrisches Verhalten bezüglich der Längen beider Eingabezeitreihen haben, untersuchen wir den Einfluss beider Parameter getrennt voneinander.

Beschreibung der Experimente Innerhalb eines Experiments berechnen wir N verschiedene Abstände mittels `DTW` auf jeweils zufällig erzeugte Zeitreihen A und B vorgegebener Längen $\#A$ und $\#B$. Wir berechnen die Werte von `DTW` mit jeder vorgestellten Implementierung und messen jeweils die kumulierten Laufzeiten. Abbildung 5 zeigt repräsentative Beispielmessungen für die Laufzeiten der Implementierungen bei variierender Länge der Anfragezeitreihe (links), Kandidatenzeitreihe (mitte) sowie bei steigender Anzahl der zu berechnenden `DTW`-Instanzen (rechts).

Analyse der Messwerte Abbildung 5 zeigt, dass die GPU-Implementierungen in fast allen Fällen ca. zwei Größenordnungen schneller sind, als ihr CPU-Pendant. Dagegen zeigt das Diagramm rechts in der Abbildung, dass die CPU bei der Berechnung von nur einer

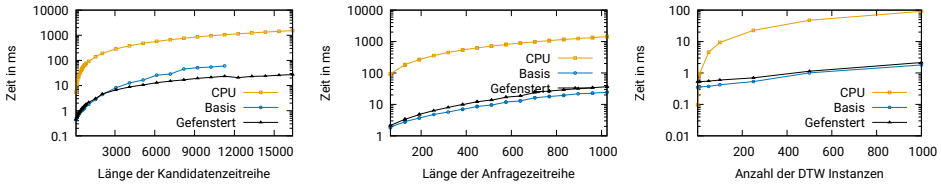

Abb. 5: Laufzeit von DTW. bei variierender Länge der Kandidatenzeitreihe (links); Länge der Anfragezeitreihe (mitte); Anzahl der DTW-Instanzen (rechts). Konstante Paramter: Anzahl DTW-Instanzen: 1000; Länge der gepufferten Zeitreihen: 1024; Länge der ungepufferten Zeitreihen: 64; Pufferlänge: doppelte Länge der Anfragezeitreihe

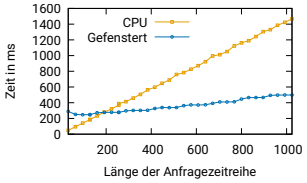

Abb. 6: Laufzeit von DTW. bei variierender Länge der Anfragezeitreihe. Anzahl DTW-Instanzen: 1; Länge der Kandidatenzeitreihe: 1024000;Pufferlänge: doppelte Länge der Anfragezeitreihe

DTW-Instanz schneller ist, als die GPU. Steigt die Anzahl der zu berechnenden DTW-Instanzen jedoch, so profitiert die GPU-Implementierung von der Parallelisierung der Berechnungen.

Abbildung 6 zeigt, dass das gleiche Phänomen bei höherer Parallelisierung einzelner DTW-Instanzen zu beobachten ist: Der Grad der Parallelisierung ist durch die Länge der Diagonalen, d. h. laut Algorithmus 2 durch die Länge der Anfragezeitreihe bestimmt. Ferner beobachten wir, dass die Laufzeit der GPU-Implementierung bei wachsender Länge der Anfragezeitreihe sprunghaft ansteigt. Wir vermuten, dass dieser Effekt auf Schedulingentscheidungen innerhalb der GPU zurückzuführen ist.

Abbildung 5 (links) zeigt, dass die gepufferte Implementierung wider Erwarten schneller als die Basis-Implementierung ist. Wir vermuten den Grund dafür in der Konkurrenz verschiedener DTW-Instanzen um den lokalen Speicher: Durch den wachsenden lokalen Speicherverbrauch der Basis-Implementierung (bei wachsender Länge der Anfragezeitreihe) können weniger DTW-Instanzen parallel ausgeführt werden, während die gepufferte Implementierung gegen diesen Effekt immun ist. Weitere Experimente haben diese Vermutung bestätigt: Abbildung 7 zeigt, dass die gefensterte Implementierung langsamer wird, je größer wir die Fensterlänge wählen. Darüber hinaus beobachten wir, dass die modernere Hardware (NVIDIA GeForce GTX 980 Ti) gegen diesen Effekt robuster als ihr Vorgängermodell (NVIDIA GeForce GTX 780 Ti) ist.

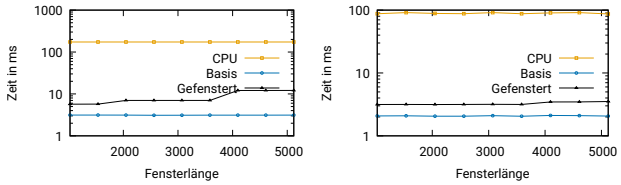

Abb. 7: Laufzeit von `DTW`. bei variierender Pufferlänge. Anzahl `DTW`-Instanzen: 500; Länge der Kandidatenzeitreihe: 256; Länge der Anfragezeitreihe: 512; Berechnung mit einer NVIDIA GeForce GTX 780 Ti (links) und einer 980 Ti (rechts)

5 Fazit

In dieser Arbeit haben wir speichereffiziente GPU-Implementierungen von `DTW` bzw. `DK` inklusive Versionen für Sub- und Supersequenzsuche vorgestellt. Wir haben die Implementierungen um die Möglichkeit erweitert, beliebig lange Kandidatenzeitreihen verabeiten zu können. Unsere Evaluierung ergab einen stabilen Geschwindigkeitsvorteil von ein bis zwei Größenordnungen gegenüber einer CPU-Implementierung. Beobachtungen zufolge wird der Geschwindigkeitsgewinn nicht durch Speicherzugriffe beschränkt, sondern durch den Parallelitätsgrad, den wir durch Optimierung des lokalen Speicherverbrauchs maximieren konnten. Der Flaschenhals wird folglich durch die Größe des lokalen Speichers bestimmt.

Damit wurde eine Grundlage geschaffen, metrische Indexstrukturen mittels GPU-beschleunigter Abstandsmaße zu verbessern. Eine mögliche Anwendung ist, die in dieser Arbeit vorgestellten Implementierungen für andere auf dynamischer Programmierung basierende Algorithmen zu adaptieren (z. B. Levenshtein-Distanz, Hidden-Markov-Chains).

Literatur

[AF13] Anguera, X.; Ferrarons, M.: Memory efficient subsequence DTW for Query-by-Example Spoken Term Detection. In: 2013 IEEE ICME. S. 1–6, Juli 2013.

[Ar17] Arkhipov, D. I.; Wu, D.; Li, K.; Regan, A. C.: Sorting with GPUs: A Survey. CoRR abs/1709.02520/, 2017, arXiv: 1709.02520.

[Be90] Beckmann, N.; Kriegel, H.-P.; Schneider, R.; Seeger, B.: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Rec. 19/2, S. 322–331, Mai 1990, ISSN: 0163-5808.

[BF17] Bachmann, J. P.; Freytag, J.-C.: Dynamic Time Warping and the (Windowed) Dog-Keeper Distance. In (Beecks, C.; Borutta, F.; Kröger, P.; Seidl, T., Hrsg.): Similarity Search and Applications. Springer International Publishing, Cham, S. 127–140, 2017, ISBN: 978-3-319-68474-1.

[BK15] Bringmann, K.; Künnemann, M.: Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping. CoRR abs/1502.01063/, 2015.

[BKL06] Beygelzimer, A.; Kakade, S.; Langford, J.: Cover trees for nearest neighbor. In: Proceedings of the 23rd ICML. ICML '06, ACM, Pittsburgh, Pennsylvania, S. 97–104, 2006, ISBN: 1-59593-383-2.

[Br14] Bringmann, K.: Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless SETH fails. CoRR abs/1404.1448/, 2014.

[Ch14] Chetlur, S.; Woolley, C.; Vandermersch, P.; Cohen, J.; Tran, J.; Catanzaro, B.; Shelhamer, E.: cuDNN: Efficient Primitives for Deep Learning. CoRR abs/1410.0759/, 2014, arXiv: 1410.0759.

[CPZ97] Ciaccia, P.; Patella, M.; Zezula, P.: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In: Proceedings of the 23rd International Conference on Very Large Databases. VLDB '97, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, S. 426–435, 1997, ISBN: 1-55860-470-7.

[Fr06] Fréchet, M. R.: Sur quelques points du calcul fonctionnel. Rendiconti del Circolo Mathematico di Palermo, 1906.

[HSS14] Hundt, C.; Schmidt, B.; Schömer, E.: CUDA-Accelerated Alignment of Subsequences in Streamed Time Series Data. In: 2014 43rd ICPP. S. 10–19, Sep. 2014.

[Mü07] Müller, M.: Information Retrieval for Music and Motion. Springer-Verlag, Berlin, Heidelberg, 2007.

[NB09] Novak, D.; Batko, M.: Metric Index: An Efficient and Scalable Solution for Similarity Search. In: 2009 Second SISAP. S. 65–73, Aug. 2009.

[SA09] Striemer, G. M.; Akoglu, A.: Sequence alignment with GPU: Performance and design challenges. In: 2009 IEEE SPDP. S. 1–10, Mai 2009.

[Sa10] Sart, D.; Mueen, A.; Najjar, W.; Keogh, E.; Niennattrakul, V.: Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs. In: 2010 IEEE ICDM. S. 1001–1006, Dez. 2010.

[SC90] Sakoe, H.; Chiba, S.: Readings in Speech Recognition. In (Waibel, A.; Lee, K.-F., Hrsg.): Readings in Speech Recognition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Kap. Dynamic Programming Algorithm Optimization for Spoken Word Recognition, S. 159–165, 1990, ISBN: 1-55860-124-4.

[TDB10] Tomov, S.; Dongarra, J.; Baboulin, M.: Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. Parallel Comput. 36/5-6, S. 232–240, Juni 2010, ISSN: 0167-8191.

[YZG13] You, S.; Zhang, J.; Gruenwald, L.: Parallel Spatial Query Processing on GPUs Using R-trees. In: Proceedings of the 2Nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data. BigSpatial '13, ACM, Orlando, Florida, S. 23–31, 2013, ISBN: 978-1-4503-2534-9.