Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Improving Google's Open-Source Machine Learning System TensorFlow

Carsten Bahnmüller

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Rothermel
Betreuer/in:	Dipl.-Inf. Ruben Mayer, Dipl.-Inf. Christian Mayer
Beginn am:	13. November 2017
Beendet am:	14. Mai 2018

Kurzfassung

Die künstliche Intelligenz und das maschinelle Lernen werden ein immer wichtigeres Thema. Viele von ihren Algorithmen haben eine Graphstruktur oder können als ein Datenflussgraph repräsentiert werden, wie z. B. ein neuronales Netz. Ein System, welches auf dem Konzept von Datenflussgraphen aufbaut, ist TensorFlow. Bei diesem baut der Nutzer einen Datenflussgraphen, welcher dann von TensorFlow ausgewertet wird. Da für das maschinelle Lernen ein einzelner Datenflussgraph sehr häufig ausgeführt werden muss, ist es nicht unüblich, dass zum Lernen eines Modells Wochen vergehen können. Eine weitere Beobachtung ist, dass die Graphen immer größer werden, und dadurch es vorkommen kann, dass ein Graph im Ganzen nicht mehr auf eine GPU passt. Aus diesen Gründen ist es interessant eine gute Partitionierung für die Graphen zu finden.

Momentan bietet TensorFlow noch keine Möglichkeit an, mit der die Graphen automatisch partitioniert werden können. Ein üblicher Ansatz ist daher eine Expertenplatzierung, bei welcher ein Experte eine gute Partitionierung findet.

In dieser Arbeit untersuchen wir Algorithmen für die automatische Partitionierung der Graphen. Dabei stellen wir drei Algorithmen vor, welche eine ähnliche oder bessere Geschwindigkeit als eine Expertenplatzierung erreichen. Eine wichtige Beobachtung, die wir in dieser Ausarbeitung gemacht haben ist, dass mehr GPUs nicht immer mehr bringen, sondern es sinnvoll ist den Graphen auf sowenig Geräte wie möglich zu verteilen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Algorithmen

1 Introduction

In den letzten Jahren wurde das Thema künstliche Intelligenz und maschinelles Lernen immer wichtiger. Das große Interesse an dem Thema zeigt sich in der Wissenschaft, wie die steigende Anzahl von Publikationen im Bereich Deep Learning zeigen [VMR], und auch in der Wirtschaft und dadurch im Alltag, indem immer mehr Produkte das maschinelle Lernen nutzen. Es lässt sich in Suchmaschinen, wie Google, welches die angezeigten Ergebnisse automatisch an den Nutzer anpasst, auf Verkaufsportalen wie Amazon, welches versucht die besten möglichen Produktvorschläge zu machen und auf Unterhaltungsplattformen wie YouTube, welches versucht für den Nutzer interessante Videos vorzuschlagen, finden. Aber auch Produkte wie Kameras und Smartphones nutzen maschinelles Lernen, z. B. für die Gesichtserkennung. Auch Haushaltshilfen wie Alexa oder Echo, lassen sich in immer mehr Haushalten finden, wessen Spracherkennung über maschinelles Lernen funktioniert[Bre18]. Mit AlphaGo konnte in den letzten Jahren auch ein großer Erfolg errungen werden, indem ein Computer zum ersten Mal in einem GO Spiel, gegen einer der weltbesten Spieler Lee Sedol gewonnen hatte [dpa16].

Einer der Gründe für den neueren Erfolg der künstlichen Intelligenz und der neuronalen Netze, obwohl das Konzept von mehrschichtigen neuronalen Netzen schon im Jahr 1975 [Gur14] entstanden ist, ist die zunehmende Rechenpower und die zunehmende Menge an gesammelten Daten. Das Trainieren eines neuronales Netz ist sehr Rechenintensiv, wodurch es erst durch die neueren stärkeren Computer lohnenswert ist [Cor17]. Ein weiterer wichtiger Punkt für die künstliche Intelligenz, ist die Anzahl der Daten. So sind Chun et al. [SSSG17] zum Beispiel der Meinung, dass der Fokus von der Entwicklung der Modelle für das maschinelle Lernen zurück auf das Sammeln der Daten gelegt werden sollte.

Durch diese sehr großen Mengen an Daten, die verarbeitet werden müssen, kann das Lernen von größeren Modellen Tage bis Wochen dauern. Durch die lange Dauer und die hohe Rechenintensität, ist es sehr wünschenswert das Verfahren zu parallelisieren. Eine der beiden Möglichkeiten wie man das erreichen kann, ist die Datenparallelität, bei welcher das gleiche Modell mit verschiedenen Daten gleichzeitig auf verschiedenen Rechnern, meist GPUs, ausgeführt werden. Die andere Möglichkeit ist die Modellparallelität, bei welcher mehrere GPUs, das gleiche Modell mit den gleichen Daten parallel verarbeiten. Der Graph wird hierbei aufgeteilt und die einzelnen Knoten, werden auf verschiedene GPUs verteilt. Die Modellparallelität wird auch dann wichtig, wenn das Modell zu groß für eine GPU wird und deswegen verteilt werden muss. Dadurch, dass die verschiedenen Modelle sehr unterschiedliche Strukturen und dadurch auch Eigenschaften haben, muss auf einen Experten oder für das jeweilige Modell vorhandene Heuristiken zurückgegriffen werden.

Die automatische Partitionierung und Verteilung auf die verschiedenen GPUs ist noch ein offenes Forschungsthema. In dieser Arbeit untersuchen wir verschiedene Möglichkeiten der automatischen Partitionierung für TensorFlow. TensorFlow ist ein von Google entwickeltes Framework für das maschinelle Lernen, wessen Grundidee die Programmierung von Datenflussgraphen ist. Dabei konnten wir Algorithmen finden, die ähnlich gute Partitionierungen wie eine Expertenplatzierung

finden. Eine wichtige Beobachtung, die wir gemacht haben ist, dass eine höher Anzahl an GPUs nicht immer vorteilhaft ist und dass eine Nutzung von so wenig GPUs wie möglich, die besten Ergebnisse liefert.

Diese Arbeit gliedert sich in im folgenden in 6 Kapitel:

- Im ersten Kapitel dieser Arbeit werden die Grundlagen erklärt, welche zum Verständnis der Arbeit benötigt werden. Das ist eine Übersicht über TensorFlow und eine kurze Übersicht über relevante Themen des Maschinellen Lernens.
- Im zweiten Kapitel wird die Problemstellung, welche wir zu lösen versuchen, genauer definiert.
- Im dritten Kapitel werden unsere Lösungsansätze vorgestellt Dazu werden kurz unsere Überlegungen und die existierenden Einschränkungen beschrieben und dann drei von uns entwickelte Algorithmen vorgestellt. Die Idee des ersten Algorithmus ist die Knoten so aufzuteilen das es auf jeder GPU immer parallel ausführbare Knoten gibt. Die Idee des zweiten Algorithmus ist, dass wir die Knoten Abhängig von einer Bewertung, welche sich aus der Lastverteilung und dem Kommunikationsaufwand berechnen lässt, platzieren. Die Idee des dritten und letzten Algorithmus ist, dass wir die Knoten nach Clustern platzieren.
- Im vierten Kapitel, werden wir die verschiedenen Algorithmen für zwei typische Graphen evaluieren. Dabei konnten wir ein ähnliche Performanz zur Expertenplatzierung erreichen und haben festgestellt, dass eine Erhöhung der GPUs nicht immer optimal ist.
- Im fünften Kapitel werden verwandte Arbeiten vorgestellt. Dabei werden wir auf Algorithmen eingehen, welche sich direkt mit dem Partitionierungsproblem für TensorFlow beschäftigen und auf verwandte Ansätze aus dem Graph Processing.
- Im letzten Kapitel werden die Ergebnisse für die Arbeit zusammengefasst und ein kurzer Zukunftsausblick gegeben.

2 Grundlagen

In diesem Kapitel werden für die Arbeit wichtige Grundlagen vorgestellt. Diese unterteilen sich in zwei Abschnitte. Als Erstes wird TensorFlow vorgestellt werden, welches das für diese Arbeit verwendete Framework ist. TensorFlow wurde von Google entwickelt und wird für das maschinelle Lernen und numerische Berechnungen genutzt. In dem Abschnitt werden die Architektur, die Funktionsweise und für die Arbeit wichtige Implementierungsdetails eingegangen. Als Zweites werden für die Arbeit relevante Algorithmen und Grundlagen des maschinellen Lernens vorgestellt. Die Grundlagen unterteilen sich in einen kurzen Überblick über das maschinellen Lernen und dessen Anwendungsgebiete. Die relevanten Algorithmen, sind die Klassifizierung, das Clustering und das Deep Learning.

2.1 TensorFlow

TensorFlow wurde 2015 von Google entwickelt. Es basiert auf dem ebenfalls von Google entwickelten Framework DistBelief[DCM+12]. In ihrem Paper haben die Entwickler vor allem drei Punkte hervorgehoben in welchen sie TensorFlow gegenüber DistBelief verbessern wollten [ABC+16]. Der erste Punkt ist, dass zur Definition von neuen Schichten C++ genutzt werden musste. Dies sahen sie als eine zu große Barriere für Forscher mit nur wenig Programmiererfahrung.
Der zweite Punkt ist, dass sie die Implementierung der Parameterserver für zu unflexibel empfunden haben, wodurch Änderungen die das updaten der Parameter beinhalteten zu mühsam waren.
Der dritte und letzte Punkt war, dass DistBelief zu stark auf einen Forward und Backward Pass der Netzwerke ausgelegt war und dadurch nicht geeignet war für andere Anwendungsgebiete, wie zum Beispiel reinforcement Learning(bestärkendes Lernen), bei welchem ein Agent in einer separaten Umgebung die Verlust-Funktion generiert.

2.1.1 Grundprinzip

Wie schon DistBelief, basieren auch in TensorFlow die Berechnungen auf Datenflussgraphen. Ein Datenflussgraph besteht aus zwei Grundeinheiten, Operationen und Tensoren. Die Operationen bilden dabei die Knoten des Graphen und die Tensoren die Kanten. Eine Operation hat keine, eine oder mehrere Tensoren als Eingabe und keinen, einen oder mehrere Tensoren als Ausgabe. Ein Beispiel für eine Operation mit einem Eingabe-Tensor, ist die "*reshape*-Operation, welche es dem Nutzer erlaubt die Dimensionen eines Tensors zu ändern. Ein Beispiel für eine Operation die mehrere Tensoren als Eingabe hat, ist die Addition.

Zusätzlich zu den Operationen gibt es noch Variablen und Placeholder. Eine Variable ist, wie bei herkömmlichen Programmiersprachen, eine Operation, welche sich verändernde Werte speichert. Dadurch werden Variablen genutzt, um Parameter umzusetzen, welche ein Algorithmus über

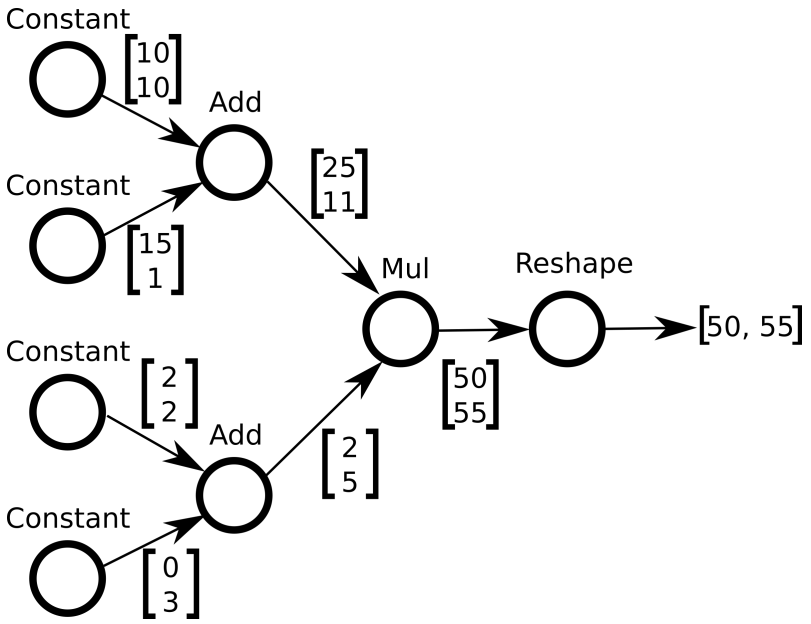


Abbildung 2.1: Beispiel für einen Datenflussgraphen. Er fängt links mit 4 Konstanten an, welche jeweils 2x1 Tensoren sind. Danach addiert er den 1. mit dem 2. Tensor und den 3. mit dem 4. Tensor. Dadurch sind nur noch 2 Tensoren im Datenfluss. Als nächstes werden die beiden Tensoren miteinander multipliziert. Der so enstandene Tensor hat die Form 2x1 und wird mit der „rehshape“ Operation zu einem 1x2 Tensor transformiert.

mehrere Graphausführungen lernt. Ein Platzhalter hingegen ist eine Operation im Graphen, für die man bei jeder Ausführung des Graphen einen Wert festlegen muss. Man kann sie sich also, wie Argumente vorstellen die einer Methode übergeben werden. Abb. 2.1 zeigt ein Beispiel für einen einfachen Datenflussgraphen. Ein Vorteil des Datenfluss Models ist, dass es die Kommunikation zwischen den Operationen explizit modelliert. Dadurch ist es möglich unabhängige Teilgraphen parallel auszuführen[ABC+16].

Damit ein Graph ausgewertet werden kann, muss der Nutzer eine Session definieren. Der Session wird bei der Initialisierung ein Graph zugewiesen, auf welchem sie arbeitet. Um den Graphen auszuwerten, muss der Nutzer die *Run*-Methode der Session aufrufen. Diese erwartet als Eingabe einen oder mehrere Tensoren und falls im Graph vorhanden, die Werte für die Placeholder. Der Graph wird dann so ausgeführt, dass nur Operationen, welche zur Berechnung der übergebenen Tensoren benötigt werden, ausgeführt werden.

2.1.2 Architektur

In Abb. 2.2 kann man die Architektur von TensorFlow sehen. Im groben teilt sich die Architektur in zwei Teile auf. Den Client, welcher für die Definition der Datenflussgraphen und die Initiation der Sessions zuständig ist, und den Master, welcher sich um die Ausführung der Graphen und den damit verbundenen Mehraufwand kümmert. Im folgenden werden die einzelnen Komponenten genauer beschrieben.

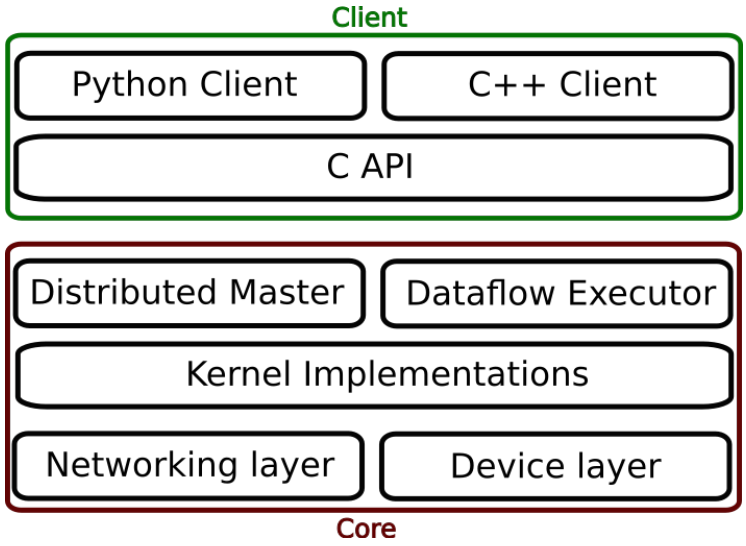


Abbildung 2.2: TensorFlow Architektur (angelehnt an Graphik von [ABC+16])

Client

Der Client ist die Schnittstelle zum Benutzer. Auf ihm baut der Nutzer die Datenflussgraphen und kann über Sessions die Auswertung des gesamten Graphen oder von Teilen des Graphen initiieren. Der Client bietet dem Nutzer die Möglichkeit die Platzierung der Operationen zu bestimmen. Dazu kann der Nutzer Constraints angeben mit welchen die Platzierung von Knoten auf ein bestimmtes Gerät oder eine Menge von Geräten restriktiert wird. Momentan bietet TensorFlow eine Python und eine C++ Implementierung an, wobei Python die verbreiterte Version ist. Dadurch hat die Python Version eine größere Auswahl an Bibliotheken. Die interne Kommunikation von Client zum Kern von TensorFlow findet unabhängig von der genutzten Sprache mit einer C-API statt.

Distributed Master

Die Hauptaufgabe des Distributed Masters ist die Verteilung des zu verarbeitenden Datenflussgraphen auf die verschiedenen Tasks. Da ein Nutzer beliebige Teilgraphen ausführen kann, muss der Distributed Master zuerst bestimmen welcher Operationen des Graphen benötigt werden. Nachdem er diese bestimmt hat entfernt er die nicht benötigten Operationen aus dem Graphen. Als Nächstes weist er den Operationen den entsprechenden Tasks zu. Ein Beispiel für diese Interaktion von Client, Master und Tasks bietet Abb. 2.3.

Um die Operationen den Tasks zuzuweisen, wertet er implizite Constraints und explizite aus. Implizite Constraints können zum Beispiel eine zustansbehafteter Knoten und sein Zustand sein. Explizite Constraints werden vom Clienten angegeben. Falls es zwei oder mehrere Constraints gibt welche verlangen, dass ein Knoten auf verschiedenen Geräten platziert werden muss, entscheidet ein vom Nutzer bestimmter Parameter über das weitere Verhalten. Erlaubt der Nutzer ein „Soft-Placement“, dann ingoriert der Distributed Master die vom Nutzer bestimmten Constraints und platziert die Operationen. Falls er kein „Soft-Placement“ erlaubt, bricht der Distributed Master die Berechnung ab und informiert den Nutzer über den Fehler.

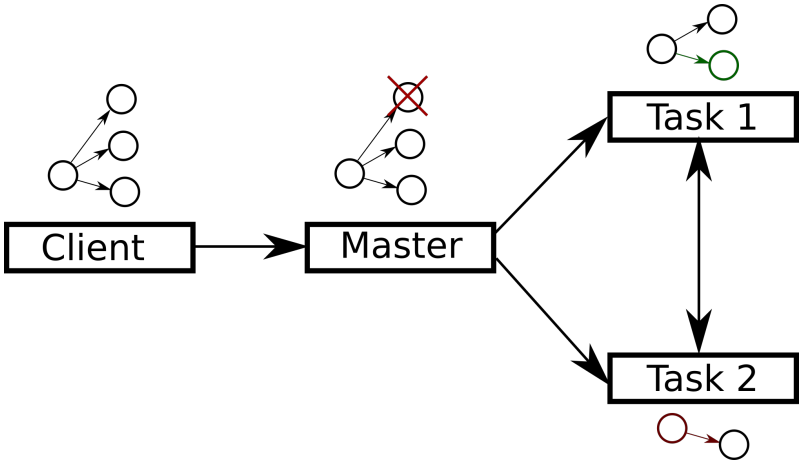


Abbildung 2.3: Auf dem Clienten wird der gesamte Datenflussgraph definiert und übergibt ihn dem Master. Der Master beschneidet den Graphen passend, so dass nur noch relevante Operationen vorhanden sind. Danach bestimmt er die passenden Partitionen und sendet die Partitionen an die Tasks. Dabei werden Sender(Grün) und Empfänger(Rot) Knoten hinzugefügt.

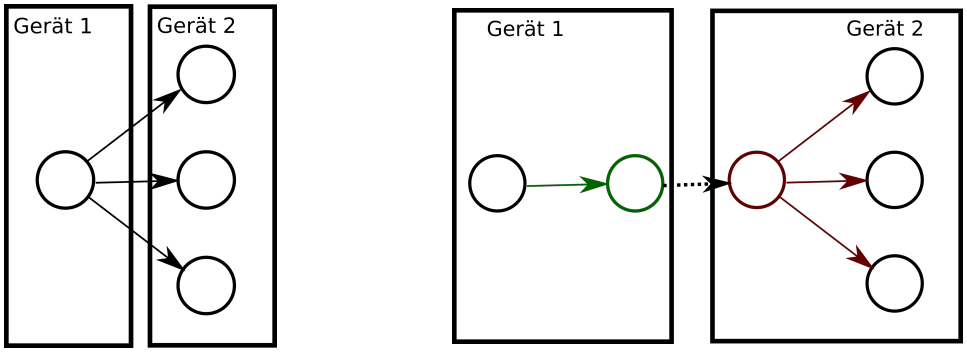
Werden zwei Knoten auf verschiedene Geräte platziert und es existiert eine Kommunikation zwischen den beiden Knoten, erzeugt der Distributed Master einen „Empfänger“-Knoten und einen „Sender“-Knoten. Falls ein Tensor von mehreren Operationen auf dem gleichen Gerät genutzt wird, erzeugt der Distributed Master nicht für jede Operation ein separates Empfänger und Sender Paar, sondern nur für jedes Tensor und Geräte Paar. Dies hat den Vorteil, dass der Tensor nur einmal von einem Gerät zum anderen gesendet werden muss, was an Kommunikation spart und dadurch eine höhere Performanz bietet. Ein Beispiel dafür bietet Abb. 2.4.

Dataflow Executor

Dem Dataflow Executor seine Aufgabe ist die Ausführung der ihm vom Master zugewiesenen Operatoren. Dazu kümmert er sich um das Sheduling der Operatoren und das senden der Kernel an die lokale GPU. Die Kernels werden so parallel, wie möglich ausgeführt. Das heißt, wenn es möglich ist, werden alle vorhandenen CPU-Kerne oder mehrere GPU-Streams genutzt. TensorFlow bietet von Haus an sehr viele Operationen an. Diese beinhalten Datenoperationen, wie die Addition und die Multiplikation von Tensoren, und Kontrolloflussperationen, wie eine For-Schleife.

Kernel Implementation

Für die meisten Operationen liegt eine CPU und eine GPU basierte Implementierung vor. Für den Nutzer ist es möglich sich eigene Kernel zuschreiben und in TensorFlow einzubinden, falls die gewünschte Operation mit den vorhandenen Kernels zu ineffizient oder nicht umsetzbar wäre. Die Kernels werden in C++ mit verschiedenen Bibliotheken umgesetzt, wie z. B. Eigen::Tensor oder cuDNN.



(a) Davor **(b)** Danach

Abbildung 2.4: Ein Graph auf zwei Partitionen, bevor und nachdem Sender(Grün) und Empfänger(Rot) Knoten hinzugefügt wurden.

Network/Device Layer

Der Network/Device Layer kümmert sich um die Übertragung der Tensoren zwischen den Geräten. Zur Kommunikation zwischen lokaler CPU und lokaler GPU nutzt TensorFlow die *cudaMemcpyAsync()*-API. Zur Kommunikation zwischen lokalen GPUs nutzt TensorFlow DMA. Für die Kommunikation zwischen verschiedenen Tasks nutzt TensorFlow verschiedene Protokolle wie z. B. gRPC über TCP.

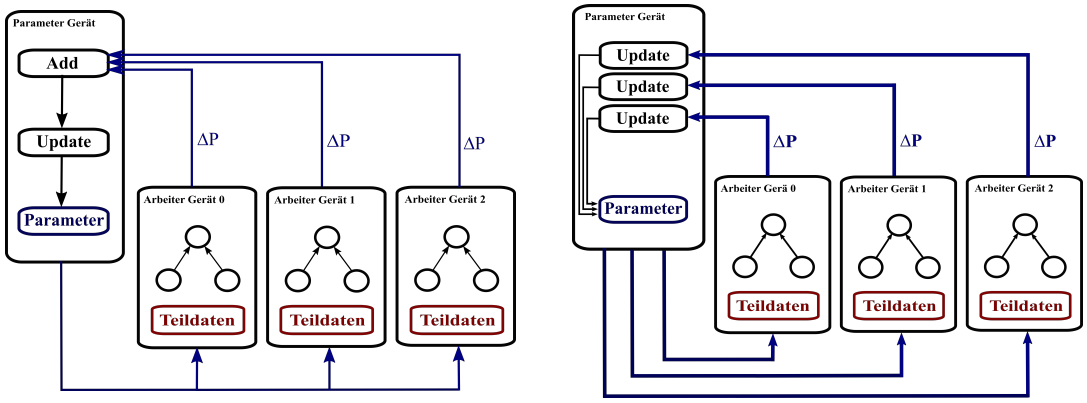
2.1.3 Verteiltes TensorFlow

Für die Parallele Verarbeitung der Graphen gibt es standardmäßig zwei Ansätze. Der eine Ansatz ist die Datenparallelität, bei welcher man das gesamte Modell mehrmals parallel mit verschiedenen Daten ausführt. Der andere Ansatz ist die Modellparallelität, bei welcher man ein Modell aufteilt und dieses auf verschiedene Geräte verteilt. Im Folgenden werden diese beiden Modelle im Bezug auf TensorFlow genauer erklärt werden. Danach werden weitere Konzepte von TensorFlow für die Verteilung der Graphen beschrieben. Dies sind als erstes Server und Cluster und als zweites die Geräteplatzierung.

Datenparallelität

Wie der Name schon sagt, wird bei der Datenparallelität nicht der Graph parallel verarbeitet, sondern die Daten. Dazu wird das gesamte Modell repliziert und auf verschiedenen Geräten platziert. Jeder dieser Modelle arbeitet mit einem unterschiedlichen Datensatz. Zum Beispiel kann man einen Datensatz von 1000 Bildern in 10 Datensätze von 100 Bildern aufteilen und diese dann auf 10 Geräte verteilen.

Damit das von jedem Gerät gelernte Wissen an die anderen weitergegeben werden kann, gibt es zwei Möglichkeiten zur Aktualisierung der Parameter. Einen Synchronen und einen Asynchronen Ansatz.



(a) Synchronus

(b) Asynchronus (Quelle)

Abbildung 2.5: Asynchronus und Synchronus Datenparallelität. Abgeändert von [ABC+16]

- Beim Synchronen Ansatz, wartet jedes Gerät nach der kompletten Ausführung des Graphen auf die anderen Geräte. Die Ergebnisse aller Geräte werden dann aufsummiert und der Parameter wird dann mit dem Ergebnis aller Geräte gleichzeitig aktualisiert.
- Beim Asynchronen Ansatz warten die Geräte nicht aufeinander. Das bedeutet, sobald ein Gerät fertig ist mit der aktuellen Ausführung des Graphen, werden die Parameter aktualisiert und es startet mit der nächsten Ausführung. Der Vorteil davon ist, dass die Daten schneller abgearbeitet werden können, da der Synchronisationsschritt entfällt. Der Nachteil davon ist, dass die laufenden Modelle nicht immer mit den aktuellsten Parametern arbeiten, was die Konvergenz Geschwindigkeit verringert.

Ein Beispiel der beiden Ansätze bietet Abb. 2.5a.

Zur Umsetzung der Datenparallelität wird normalerweise auf eine Parameter- und Arbeiter-Server Architektur zurückgegriffen. Bei dieser befinden sich die Parameter auf einem oder mehreren Parameterservern, dessen einzige Aufgabe die Speicherung und Verwaltung der Parameter ist. Die Verarbeitung des Graphen findet auf den Arbeiter-Server statt, welche mit den Parameter Servern kommunizieren.

Der Vorteil dieser Architektur ist die Reduzierung der Kommunikation. Würde jeder Arbeiter die gesamten Parameter verwalten, müsste er nach jeder seiner Berechnungen mit jedem anderen Arbeiter kommunizieren um die Parameter zu synchronisieren. Mit der Parameter- und Arbeiter-Server Architektur, muss jeder Arbeiter nur mit den Parameter Servern kommunizieren.

Modellparallelität

Im Gegensatz zur Datenparallelität wird bei der Modellparallelität das Modell auf verschiedene Geräte verteilt. Dies geschieht meist durch einen Experten, welcher durch verschiedene Heuristiken, die Operationen platziert. Heuristiken können dabei sein, dass bei einem rekurrenten neuronalen Netz, die einzelnen Schichten auf verschiedene Geräte platziert werden. Dadurch müssen nur die von der jeweiligen Schicht benötigten Parameter auf die jeweiligen Geräte gelegt werden, was den

benötigten Speicheraufwand und die benötigte Kommunikation reduziert.
Die automatische Platzierung der Knoten ist noch ein offenes Problem und wird in dieser Arbeit untersucht.

Cluster und Server

Damit die Nutzung von mehreren Rechnern möglich ist, bietet TensorFlow die Möglichkeit an Server und Cluster zu definieren. Ein Cluster ist eine Menge von Servern, welche über ein Wörterbuch initialisiert wird. Ein Beispiel für eine Definition eines Clusters ist:

```
cluster = tf.train.ClusterSpec({"Tasks1": ["192.168.0.1:2222", "192.168.0.2:2222"],
                                "Tasks2": ["192.168.0.3:2222", "192.168.0.4:2222"])
```

Der Befehlt erstellt ein Cluster mit zwei Jobs, Tasks1 und Tasks2, welche jeweils zwei Tasks haben. Die einzelnen Tasks sind über den Namen ihres übergeordneten Jobs und über ihre ID anzusprechen. In dem obigen Beispiel wären die Namen der vier Tasks also:

```
/job:Tasks1/task:0
/job:Tasks1/task:1
/job:Tasks2/task:0
/job:Tasks2/task:1
```

Ein Beispiel für eine Definition eines Servers ist:

```
server = tf.train.Server(cluster, job_name="Tasks1", task_index=0)
```

Der Befehl erstellt einen Server, welcher durch die im übergebene Clusterdefinition alle anderen Server im Cluster kennt. Der so erstellte Server ist für den Task 0 von jobs "Tasks1"verantwortlich.

Geräteplatzierung

Für die Platzierung der Knoten auf die Geräte, bietet TensorFlow mehrere Möglichkeiten an. Die erste Möglichkeit ist die explizite Platzierung, welche über die *tf.device* Funktion möglich ist. Eine Beispiel dafür ist:

```
with tf.Device{"/device:GPU:0"}:
  variable = tf.Variable([1.0])
```

Mit „with“ wird ein Kontext geöffnet in dem alle in ihm definierten Operationen auf das entsprechende Gerät platziert werden. Es ist auch möglich nur den Typen des Gerätes anzugeben, anstatt das exakte Geräte. Ein Beispiel dafür ist:

```
with tf.Device{"/device:GPU"}:
  variable = tf.Variable([1.0])
```

Dies würde nur festlegen, dass die in diesem Kontext definierten Operationen auf eine GPU platziert werden müssen. Falls der Nutzer keine Platzierung für eine Operation angibt, wird sie standardmäßig auf die GPU mit der niedrigsten ID oder falls keine GPU Kernel Implementierung vorhanden ist, auf die CPU platziert.

Eine weitere Möglichkeit der Platzierung ist über die Colocation von Operationen. Bei diesen gibt der Nutzer für eine Operation nicht das exakte Device an, sondern definiert Operationen, welche zusammen platziert werden müssen. Folgender Befehl würde zum Beispiel „variable0“ und „variable1“ colocieren:

```
variable0 = tf.Variable([1.0])
with graph.colocate_with(Variable1):
  variable1 = tf.Variable([2.0])
```

Wie bei dem expliziten platzieren der Operationen, wird ein Kontext geöffnet, in welchem alle in ihm definierten Operationen mit einer Operation colociert werden. Operationen die miteinander colociert werden müssen, werden in dieser Arbeit Colocationsgruppe genannt werden. Falls mehrere Operationen in einer Colocationsgruppe Restriktionen an die Platzierung haben, werden nur Geräte betrachtet, welche alle Restriktionen erfüllen. Es ist möglich, dass kein solches Gerät gefunden wird, weil sich zwei Constraints widersprechen. Das kann zum Beispiel passieren, wenn eine Operation der Colocationsgruppe auf die GPU und eine andere Operation auf die CPU platziert werden soll. Falls der Nutzer von TensorFlow ein *soft placement* erlaubt, werden nur sich nicht widersprechende Constraints betrachtet und die Colocationsgruppe wird auf ein dafür passendes Device platziert. Falls der Nutzer ein *soft placement* nicht erlaubt, kann die Colocationsgruppe nicht platziert werden und TensorFlow bricht mit einem Fehler ab.

Für die Parameter- und Arbeiter-Server Architektur, die bei der Datenparallelität zum Einsatz kommt, bietet TensorFlow, eingebaute Funktionen an. Normalerweise nutzt man bei einer solchen Architektur mehr als nur einen Parameter Server. Zur automatischen Platzierung der Variablen auf die verschiedenen Server bietet TensorFlow mehrere Möglichkeiten an. Die Verteilung der Parameter auf die Parameter Server, kann dann mit folgenden Befehl erreicht werden:

```
with tf.device(tf.train.replica_device_setter(cluster=cluster)):
  variable0 = tf.Variable([1.0])
  variable1 = tf.Variable([2.0])
  variable2 = tf.Variable([3.0])
```

Die Funktion *tf.train.replica_device_setter* bekommt als Parameter ein Cluster übergeben. Wie Cluster in TensorFlow funktionieren ist in Abschnitt 2.1.3 beschrieben. Als weitere Parameter, kann man der Funktion übergeben, welcher Teil der Server aus dem Cluster Parameter Server sind. Der Standard Wert dafür ist „ps“, was bedeutet, dass alle Server aus dem Cluster, die den Wert „ps“ zugewiesen sind, die Parameter Server sind.

Die Standard Strategie, mit welcher die Funktion die Parameter platziert ist Round Robin. Wenn wir von zwei Parameter Servern ausgehen, würde die Funktion in unserem Beispiel, *variable0* und *variable2* auf den ersten Parameter Server und *variable1* auf den zweiten Parameter Server platzieren. Die Größe der verschiedenen Parameter wird dabei nicht beachtet, wodurch keine Garantie der Lastenverteilung gegeben wird. TensorFlow erlaubt es dem Nutzer seine eigene Strategie zu implementieren, indem er der Methode ein callable als Argument übergibt.

2.2 Maschinelles Lernen

In diesem Abschnitt werden relevante Themen des maschinellen Lernens vorgestellt werden, welche zum Verständnis dieser Arbeit benötigt werden. Dazu wird als Erstes eine kurze Übersicht über das maschinelle Lernen gegeben. Danach werden drei relevante Themengebiete des maschinellen Lernens beschrieben. Als Erstes die Klassifizierung, welche versucht Datensätzen eine Klasse zuzuweisen. Als zweites, dass Clustering mit welchem es möglich ist Datensätze in verschiedene Gruppen, welche Cluster genannt werden, zusammenzufassen. Als letztes wird auf das Deep Learning eingegangen werden, welches das für TensorFlow am häufig genutzte Modell ist. Dies sind bei weitem nicht alle Arten des maschinellen Lernens, aber die für diese Ausarbeitung wichtigsten.

2.2.1 Übersicht

Die Grundidee des maschinellen Lernens ist, dass ein Computer aus Daten etwas lernt. Dies unterscheidet sich von herkömmlichen Algorithmen insofern, dass ein Programm, dass zur Lösung eines Problems geschrieben wurde, entweder funktioniert oder nicht. Das bedeutet, dass wenn ein Programm für eine Eingabe eine falsche Lösung berechnet, dann wird es bei gleichen Ausgangsbedingungen, wie z. B. den Seed des Zufallszahlengenerators, auch bei weiteren Ausführungen auf ein falsches Ergebnis kommen[GBC16].

Ein lernender Algorithmus soll aus seinen Fehlern und auch Erfolgen eine Erkenntnis gewinnen und erkennen, dass er diese Erkenntnis auch auf andere ähnliche Probleme anwenden kann. In unserem vorherigen Beispiel würde das bedeuten, dass es möglich ist, dass ein Algorithmus für eine Eingabe zu einem Problem zuerst auf ein falsches Ergebnis kommt. Dann aber durch das Lernen auf anderen Datensätzen, sein Modell verbessert und so bei erneutem Ausführen des Problems zu einem richtigen Ergebnis kommt. Somit hat der Algorithmus sich verbessert ohne, dass ein Programmierer etwas ändern musste.

Im Allgemeinen lässt sich das maschinelle Lernen in drei Phasen aufteilen, wobei nicht jeder Algorithmus alle Phasen hat.

Training In der Trainingsphase arbeitet der Algorithmus mit dem Trainingsdatensatz. Je nach Algorithmus sieht dieser anders aus. Bei der Klassifizierung ist dies ein schon im Voraus ausgewerteter Datensatz, welcher den die einzelnen Daten schon die richtigen Klassenlabel zuweist. Im Gegensatz dazu braucht das Clustering nur einen einfachen Datensatz, welcher nicht im Voraus ausgewertet werden muss.

Aus diesen Trainingsdaten versucht der Algorithmus ein Modell zu finden, welches die Daten am besten beschreibt. Die Qualität und Menge der Daten sind dabei von einer großen Wichtigkeit. Befinden sich z. B. nur Daten aus einer Klasse in den Testdaten, kann der Algorithmus natürlich nichts über die anderen Klassen lernen.

Ein weiteres Problem in der Trainingsphase ist das Overfitting. Bei diesen hat das gelernte Modell auf dem Trainingsdatensatz eine sehr gute Performanz, aber eine nur schlechte Performanz für den Validierungsdatensatz. Das liegt daran, dass sich das Modell zu stark an den Trainingsdatensatz angepasst und dadurch von Annahmen ausgeht die nur für diesen gelten aber nicht in der Allgemeinheit.

Alter	Gehalt	Beruf	Kreditwürdig
20	60k	Manager	ja
29	35k	Manager	ja
18	35k	Ingenieur	nein
50	70k	Sekretär	ja
41	55k	Manager	nein

Tabelle 2.1: Kleiner Beispiel Datensatz, welcher sich mit den Entscheidungsbaum aus Abb. 2.6 klassifizieren lässt

Um dem Overfitting entgegenzuwirken gibt es je nach Algorithmus verschiedene Ansätze. Bei Entscheidungsbäumen ist eine Standardmethode den beim Lernen gefunden Entscheidungsbaum zu stutzen [BKK+98]. Durch das richtige stutzen ist es möglich den Entscheidungsbaum zu generalisieren. Und bei neuronalen Netzen gibt es die Möglichkeit des Dropouts [SHK+14], bei diesem werden zufällig Einheiten abgeschaltet und auf einem so kleineren Netz gelernt. Dadurch sollen Abhängigkeiten zwischen den Einheiten reduziert werden und dadurch das Overfittig reduziert werden.

Validation In der Validierungsphase wird das in der Trainingsphase gefundene Modell evaluiert. Die Validierung des Modells kann auf verschiedene Weisen passieren. Eine Möglichkeit ist die Validierung von einem Experten, welcher sich mit dem Problem auskennt und somit das korrekte Verhalten kennt. Eine andere Möglichkeit ist die Anwendung des Modells auf einen Validierungsdatensatz, welcher wie der Trainingsdatensatz ausgewertet sein muss.

Eine häufig genutzte Methode zur Validierung des Modells ist die cross-validation [RTL09]. Eine Art der cross-validation ist die k-fold cross-validation bei der die Trainingsdaten in k Partitionen aufgeteilt wird. Jeder dieser k Partitionen wird einmal als Validierungsdatensatz genutzt, während der Algorithmus die anderen $k - 1$ Partitionen als Trainingsdatensatz benutzt. Somit wird der Algorithmus für einen Datensatz k-mal ausgeführt.

Application Die dritte und letzte Phase ist die Anwendung des gelernten Modells auf neue Daten. Bei der Klassifizierung würde der Algorithmus für neue Daten, bei welchem die korrekte Klassen noch nicht bekannt sind, eine Klasse zuweisen. Beim Clustering würde der Algorithmus den neuen Daten die passenden Cluster zuweisen.

2.2.2 Klassifizierung

Ein großes Themengebiet des maschinellen Lernens ist die Klassifizierung. Bei der Klassifizierung wird versucht einem Datensatz eine passende Klasse zuzuweisen. Dabei können das binäre Klassen sein, wie z. B. die Einteilung von Email in Spam und Nichtspam oder aber auch n-äre Klassen, wie z. B. eine Einschätzung des medizinischen Zustandes eines Patienten [Qui86].

Für die Klassifizierung gibt es viele verschiedene Algorithmen. Einer der davon bekanntesten Algorithmen ist der Entscheidungsbaum. Ein Beispiel für einen simplen Entscheidungsbaum bietet Abb 2.6 und den passenden Datensatz dazu Tab. 2.1.

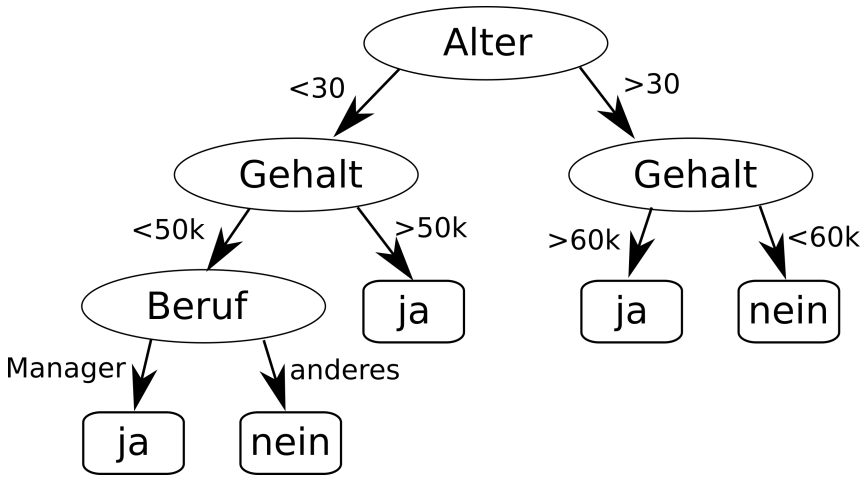


Abbildung 2.6: Ein Beispielhafter Entscheidungsbaum. Er bestimmt ob eine Person Kreditwürdig ist nach Alter, Gehalt und Beruf.

Die Nutzung eines Entscheidungsbaumes ist simpel. Man startet von der Wurzel und läuft immer die Kante entlang, welche dem Wert der zu klassifizierenden Daten entspricht. Wenn man an einen Blattknoten ankommt hat man die entsprechende Klasse gefunden. Zur Erzeugung eines Entscheidungsbaumes müssen erst einmal die Feature bekannt sein. Ein Feature kann z.B. das Alter eines Menschen oder die Farbe eines Objektes sein. Es ist eine Eigenschaft die in den Daten vorhanden ist und die einen Einfluss auf die Klasse haben könnte.

Nachdem die Features bekannt sind versucht man Feature zu finden, welche die Daten bestmöglich in verschiedene Klassen aufteilen. Dafür sucht man sich das Feature aus, welches einen Unreinheitswert(impurity value) minimiert. Dieser Unreinheitswert gibt an, wie schlecht bzw. wie gut die Aufteilung der Daten ist. Ein Beispiel für einen Uhreinheitswert ist z. B. der GINI-Index [Atk70].

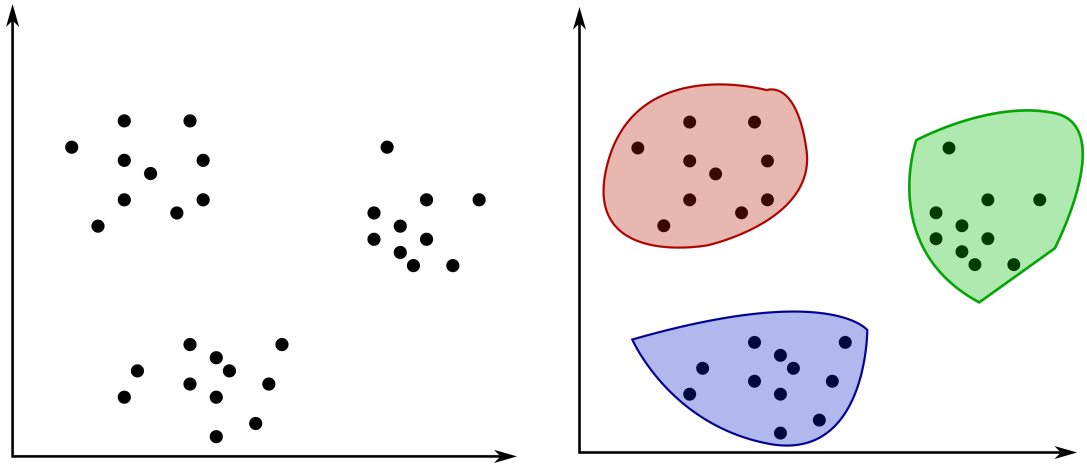
Nachdem man das Feature gefunden hat, welches den niedrigsten GINI-Koeffizienten für die Daten hat, sucht man für die daraus entstehenden Unterdatenmengen jeweils wieder ein Feature, welches den GINI-Index minimiert. Dies wird solange wiederholt, bis einer von den folgenden Fällen auftritt.

- Die Daten sind eindeutig in Klassen aufgeteilt.
- Es gibt keine weiteren Features mehr, an denen die Daten gesplittet werden können.
- Ein vorher festgelegter Wert der gewünschten Genauigkeit wurde erreicht.

Nach der Erzeugung eines Entscheidungsbaumes kann es, wie in Kapitel Unterabschnitt 2.2.1 beschrieben, zum Overfitting kommen. Um diesen entgegenzuwirken, kann der Entscheidungsbaum z. B. gestutzt werden, was bedeutet das die Aufteilung nach den letzten Features entfällt [BKK+98].

2.2.3 Clustering

Das Clustering versucht für einen Datensatz Cluster zu finden, welche möglichst viele Gemeinsamkeiten innerhalb der Cluster und möglichst wenige Gemeinsamkeiten mit Mitgliedern von anderen Clustern haben. Im Gegensatz zur Klassifizierung ist das Clustering ein unbeaufsichtig-



(a) Daten Punkte **(b)** Clustering

Abbildung 2.7: Beispiel eines Clustering

tes(unsupervised) Verfahren. Das Bedeutet, dass es zum Lernen keinen Experten benötigt, welcher ihm ausgewertete Testdaten bereitstellt. Ein Beispiel für ein Clustering auf einem 2-Dimensionalen Datensatz zeigt Abb. 2.7.

Wie schon bei der Klassifizierung ist ein wichtiger erster Schritt die Identifizierung der Features für den Datensatz. Nach dem man die Features bekannt sind, gibt es viele verschiedene Ansätze. Ein bewährter und bekannter Algorithmus ist der k-means Algorithmus.

Der k-means Algorithmus besteht aus vier folgenden Schritten [JMF99]:

1. Wähle k zufällige Datensätze aus, welche die Zentren der Cluster werden.
2. Weise jedem Knoten das Cluster zu, welchem es am nächsten ist. Die Nähe eines Knotens zu einem Cluster kann sich z. B. über den least squared Error berechnen lassen. Es sind aber auch andere Heuristiken möglich.
3. Berechne die neuen Zentren der Cluster. Hier kann der Durchschnittswert aller im Cluster enthaltenen Datensätze genommen werden.
4. Falls der Algorithmus noch nicht konvergiert ist, fange wieder von Schritt 2 an. Konvergenz wäre z. B. keine oder eine nur sehr kleine Änderungen der Clusterzuweisung im 3. Schritt.

2.2.4 Deep Learning

Die Wahl der richtigen Features ist für einen lernenden Algorithmus sehr wichtig [GBC16]. Ein Beispiel wäre eine Algorithmus, welcher Klassifiziert, ob eine Person Kreditwürdig ist oder nicht. Wenn er nur auf Features, wie „Alter“ und „Gehalt“ trainiert wurde, kann er mit anderen Features nichts anfangen. Wenn er neue Daten bekommen würde die zusätzliche Aussagen, über „Beruf“, „Bildungsstand“ machen, wäre es Sinnvoll diese Daten miteinzubeziehen, aber der Algorithmus könnte das nicht, da sie nicht zu den festgelegten Features gehören.

Ein weiteres Problem sind redundante und irrelevante Features. Ein redundantes Features hat keinen neuen Informationsgehalt, welcher nicht schon von anderen Features abgedeckt wird. Ein irrelevantes Feature hat keine wichtigen Informationen und somit keine Auswirkung auf das Ergebnis. Dadurch hat die Wahl der richtigen Features auch Einfluss auf den Speicheraufwand und die Geschwindigkeit des Algorithmus [GE03].

Dabei kann die Wahl der richtigen Features sehr kompliziert sein. Ein Beispiel hierfür ist die Bilderkennung. Wenn wir ein Programm schreiben wollen, welches versucht zu erkennen, ob ein Bild einen Menschen zeigt oder nicht, wäre es sinnvoll ein Feature zu haben, welches aussagt, dass ein Gesicht auf dem Bild ist. Aber so etwas in Pixel-werten auszudrücken ist sehr kompliziert [GBC16]. Je nach Tageszeit, Position des Menschen und Umgebung können die Pixel-Werte sehr unterschiedlich ausfallen.

Für solche Probleme ist, dass Deep Learning gut geeignet. Der große Vorteil von ihm ist, dass es die Repräsentation der Features lernt und das es die Möglichkeit hat kompliziertere Features mit einfacheren Features auszudrücken[GBC16]. Ein Beispiel dafür bietet Abb. 2.8. Hier sieht man den typischen Schichtenaufbau des Deep Learnings. Die Schichten zwischen Input und Output welche die Features repräsentieren, werden Hidden Layer genannt. Die erste Schicht sind die Inputdaten, bei welcher jeder Pixel einem Knoten entspricht. In der zweiten Schicht extrahiert das Netzwerk Kanten aus den Informationen der Pixel. In der dritten Schicht extrahiert das Netzwerk kompliziertere Features, wie z. B. Augen oder ein Mund, aus den Kanten der vorherigen Schicht. In der vierten Schicht können diese Features dann genutzt werden, um z. B. Gesichter zu erkennen.

Wichtig ist hier zu wissen, dass die Features nicht von einem Experten vorgegeben wurden, sondern dass sie vom Netzwerk selbständig gelernt wurden. Um zu verstehen, wie ein tiefes neuronales Netz lernt müssen die Hidden Layer kurz genauer betrachtet werden. Jeder Hidden Layer besteht aus einer vorher von einem Experten festgelegten Anzahl an Knoten. Jeder Knoten ist mit Knoten der vorherigen Schicht und mit Knoten der darauffolgenden Schicht verbunden. Die Verbindungen sind Gewichte, welche Aussagen wie wichtig der verbundene Input für dieses Feature ist. Der Output eines Knotens wird über eine nicht-lineare Funktion berechnet. Eine beliebte Standardfunktion dafür ist die ReLu Funktion $f(x) = max[0, x]$, welche erstmals von Hahnloser et al. [HSM+00] eingeführt wurde. Die Variable x repräsentiert die gewichtete Summe über alle Inputs des Knotens.

Der Lernvorgang eines tiefen neuronalen Netzes ist das Lernen der Gewichte. Dies basiert auf einen sogenannten Forward und Backwardpass [LBH15]. In dem Forwardpass arbeitet der Algorithmus mit gelabelten Daten und durchläuft das Netzwerk vom Anfang zum Ende. Es startet mit der Inputschicht, durchläuft die Hidden Layer und kommt zu einem Ergebnis in der Outputschicht. Danach aktualisiert er mit einem Backwardpass alle Gewichte im Netzwerk. Dazu vergleicht der Algorithmus den Ist-Wert mit dem Soll-Wert und berechnet von der letzten Schicht ausgehend einen Fehler-gradienten für jeden Knoten und läuft diesen einen Schritt entlang. Für die genaue Umsetzung des Backwardpasses, wie z. B. die Art der Berechnung des Gradienten, gibt es viele verschiedene Ansätze.

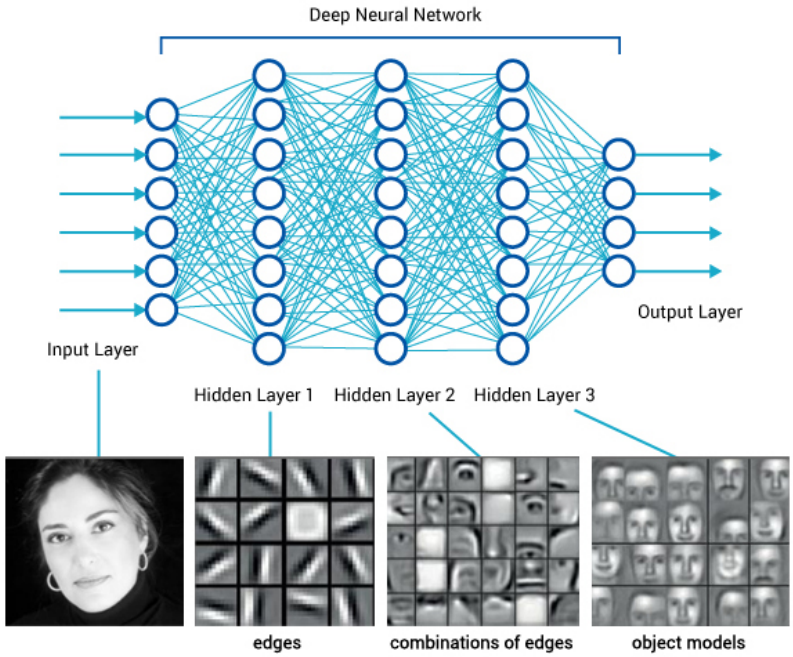


Abbildung 2.8: Die Feature Extraktion des Deep Learnings. Quelle: [Col]

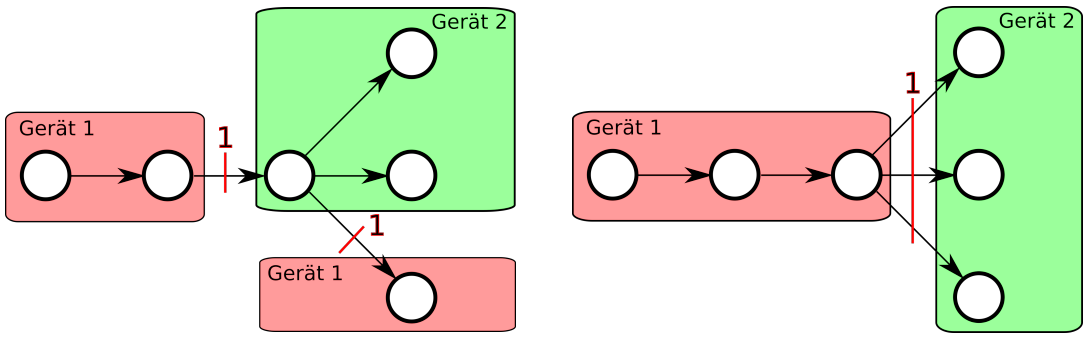
3 Problemstellung

Gegeben ist ein gerichteter Graph $G = \{V, E\}$, wobei $V = \{v_1, \ldots, v_n\}$ eine Menge von Operationen und $E = \{e_1, \ldots, e_m\}$ eine Menge von Kanten $E : V \times V$.
Weiter ist eine Menge von Geräten $D = \{d_0, \ldots, d_k\}$ gegeben.
Eine Platzierung ist eine Funktion $P : V \rightarrow D$, welche jeder Operation ein Gerät zuweist.
Operationen haben eine Ausführungszeit $t_e(V)$, welche angibt wie lange ein Gerät benötigt, um sie zu verarbeiten.
Tensoren E haben eine Kommunikationszeit $t_c(E)$, welche angibt, wie lange eine Kommunikation des Tensors benötigt, falls zwei Operationen auf verschiedenen Geräten platziert wurden.
Eine Operation v_i kann erst ausgeführt werden, wenn alle Operationen v_j ausgeführt wurden, für welche gilt $(v_i, v_j) \in E$.
r(P) ist die Ausführungszeit des Graphen, wenn die Operationen nach P platziert wurden.
Gesucht ist eine Platzierung P, welche die Ausführungszeit r(P) minimiert.

Zusätzliche Schwierigkeiten zu diesem Problem sind, dass sich die Approximation der Ausführungszeit der Operationen sehr teuer und ungenau ist. Das bedeutet das $t_e(V)$ und $t_c(E)$ unbekannte sind. Dadurch funktionieren viele typische Graphparitionierungs Algorithmen nicht optimal. Beispiele dafür werden in den verwandten Themen in Kapitel 6 gegeben.

Ein weiteres Problem ist, dass bei der einfachen Lastenverteilung die Parallelität nicht beachtet wird. Abb. 3.1b zeigt ein Beispiel, bei dem zwar eine gute Lastenverteilung eingehalten wurde, aber die Operationen können nicht parallel ausgeführt werden, wodurch kein hoher Gewinn durch die Aufteilung entsteht.

Zusätzlich zu diesem Problem müssen die Besonderheiten von Tensorflow beachtet werden. Da ein Tensor nur einmal von einem Gerät zu einem anderen gesendet wird, ändert das den herkömmlichen MinCut. Ein Beispiel dafür zeigt Abb. 3.1. Ein Standard MinCut Verfahren mit Lastenverteilung



(a) Partitionierung mit MinCut 2 **(b)** Partitionierung mit MinCut 1

Abbildung 3.1: Beispiel das mit der Berücksichtigung von TensorFlow's Sende- und Empfang-Knoten ein besserer MinCut gefunden werden kann.

würde eine Partitionierung wie in Abb. 3.1a finden, da diese nur einen MinCut von 2 hat. Unter der Berücksichtigung, dass Tensorflow, aber nur jeden Tensor einmal versendet, lässt sich ein MinCut von 1 finden, wie Abb. 3.1b zeigt.

4 Problemlösung

In diesem Kapitel werden unsere Lösungsansätze zu dem im letzten Kapitel definierten Problems vorgestellt. Dazu wird als erstes kurz eine Übersicht über unsere Überlegungen gegeben und danach drei verschiedene Algorithmen vorgestellt.

- Den ersten Algorithmus nennen wir „Task Parallel“-Algorithmus. Bei diesem versuchen wir die Operationen so zu verteilen, dass jede GPU immer eine auszuführende Operation besitzt.
- Der zweite Algorithmus ist ein Bewertungs-Ansatz, welchen wir „Scoring“-Algorithmus nennen. Bei diesem Platzieren wir die Operationen nach einem Punktewert, welche sich aus der Lastenverteilung und dem Kommunikationsaufwand berechnet.
- Der dritte und letzte Algorithmus basiert auf der Bildung von Clustern, welchen wir „Cluster“ Algorithmus nennen. Bei diesem versuchen wir Cluster für den Datenflussgraphen zu bilden, welche die zu platzierenden Partitionen sind.

4.1 Überlegungen

Bevor unsere Lösungsansätze beschrieben werden, gehen wir auf einige Vorüberlegungen und Implementierungsdetails ein. Eine Beobachtung die wir früh gemacht haben ist, dass die Kommunikationskosten eine große Rolle spielen.

Erste Tests bei denen wir versucht haben den Graphen über ein Netzwerk auf mehreren Rechnern zu verteilen, zeigte sich als ineffizient. Die Ausführungszeit eines einfachen Graphen und auch komplexeren Graphen vervielfachte sich, da die Kosten für die Kommunikation zu hoch waren. Aus diesem Grund haben wir uns auf die Verteilung des Graphen auf verschiedene GPUs innerhalb eines Rechners beschränkt. Bei diesen konnten wir immer noch beobachten, dass die Kommunikation eine große Rolle für die Ausführungszeiten spielt. Sie war aber nicht zu teuer, als dass sich die Verteilung nicht lohnen könnte. Dadurch, dass die Kommunikation eine so große Rolle spielt, haben wir bei jedem Algorithmus eine Möglichkeit gesucht diese zu reduzieren.

Ein weiterer Faktor der immer zu beachten war, ist dass man die Operationen nicht beliebig platzieren kann. Es gibt standardmäßige Abhängigkeiten zwischen den Operationen, wodurch mehrere Operationen auf dem gleichen Gerät platziert werden müssen. Ein Beispiel dafür ist ein zustandsbehaftete Operation und sein Zustand. Um diese Faktoren nicht in jedem Schritt unserer Algorithmen zu beachten, haben wir die Colocations Funktion von TensorFlow genutzt. Dazu colocieren wir alle Knoten, welche miteinander platziert werden müssen in eine Colocations Gruppe und betrachten diese als einzelne Operationen. Den Graphen verändern wir so, dass alle Knoten, welche eine Verbindung mit einen der Colocierten Knoten hatten, danach eine Verbindung mit der Colocations Gruppe haben.

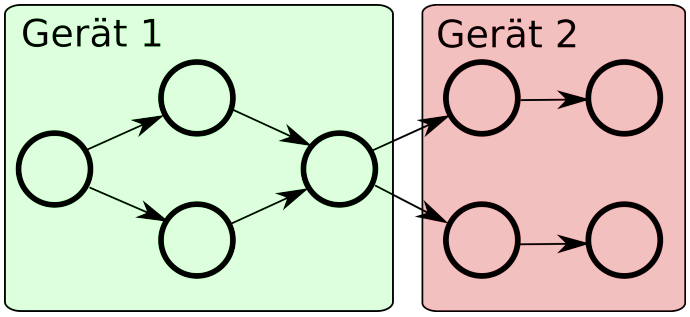
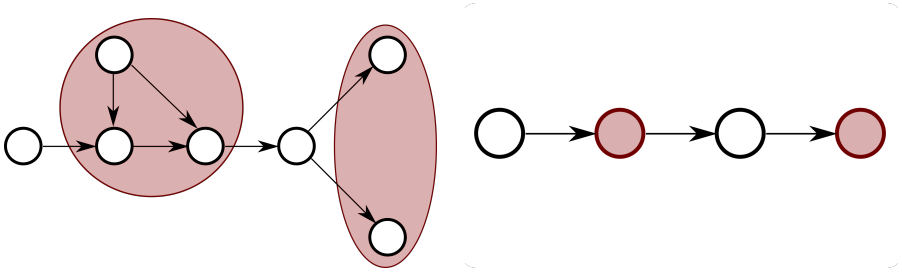


Abbildung 4.1: Ein Beispiel, bei welchem eine perfekte Lastenverteilung keine mögliche Parallelität bedeutet.



(a) Davor **(b)** Danach

Abbildung 4.2: Ein Beispielgraph bevor und nachdem die Colocationsgruppen zusammengefasst wurden. In (a) markieren die roten Kreise welche Operationen Colociert werden. In (b) sind die durch die Zusammenfassung neu enstandenen Operationen rot.

Eine weitere Überlegung ist, dass die Lastenverteilung nicht viel über die Parallelität aussagt. Ein simples Beispiel dafür bietet Abb. 4.1. Wir haben zwar eine perfekte Lastenverteilung, aber gar keine Parallelität, da die Operationen, die auf der zweiten GPU platziert wurden, warten müssen bis alle Operationen die auf der ersten GPU liegen ausgeführt werden. Dadurch hätten wir die gleiche Ausführungszeit, wie bei der Ausführung auf einer GPU, plus noch zusätzlich die Kommunikationskosten. Außer beim Clustering, haben wir versucht das Problem zu lösen, indem unsere Algorithmen den Graphen traversieren und dabei parallel auszuführende Operationen auf verschiedene Geräte platziert.

Eine weitere Beobachtung war, dass es Operationen gibt, die man zwar auf verschiedenen Geräte platzieren kann, dies sich aber nicht lohnt. Ein Beispiel dafür sind die einzelnen Operationen einer LSTM-Zelle. Jede Aufteilung der Operationen, würde durch die erhöhte Kommunikation nur eine Verlängerung der Laufzeit bedeuten. Aus diesem Grund haben wir solche Operationen auch zu einer Colocations Gruppe zusammengefasst.

Eine Überlegung von Mayer et al. [MML17] ist, dass der kritische Pfad eine große Rolle bei der Optimierung der Verteilung spielt. Dies wollen wir auch in unsere Algorithmen miteinbeziehen. Aus diesem Grund berechnen wir uns bei zwei der Algorithmen den Uprank aller Knoten des Graphen.

Der Uprank berechnet sich wie folgt [MML17]:

$$upRank(v_i) = max_{v_j \in succ(v_i}(upRank(v_j) + c_i)$$

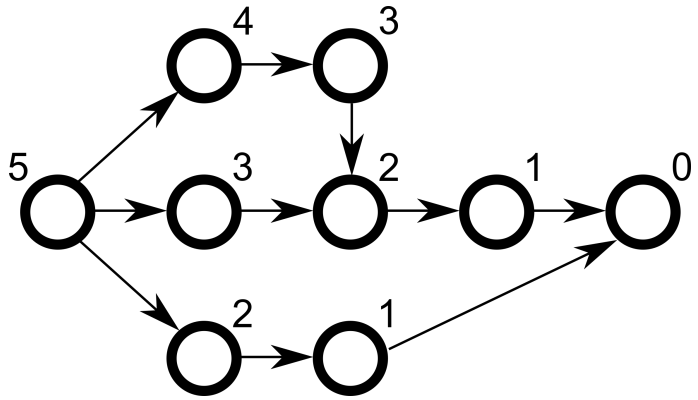


Abbildung 4.3: Der Uprank für die einzelnen Knoten für einen Beispielgraphen

Algorithmus 4.1 UpRank

```
procedure calculateUprank(Graph)
    initialisiere Queue Q
    for Knoten v in Graph do
        upRank[v] ← −1
    end for
    for Senken sink in Graph do
        upRank[v] ← 0
        füge src zu Q hinzu
    end for
    while Q ≠ ∅ do
        currentNode ← Q.pop()
        for  Eingangsknoten inNode von currentNode do
            if upRank[inNode] < upRank[currentNode]+1 then
                upRank[inNode] ← upRank[currentNode]+1
                füge inNode zu Q hinzu
            end if
        end for
    end while
end procedure
```

Dadurch ist der Uprank die Länge des längsten Pfades von einem Knoten zu einer beliebigen Senke. Abb. 4.3 zeigt die einzelnen Upranks für einen Beispielgraphen. Aus dem Uprank kann leicht der kritische Pfad berechnet werden. Man startet von dem Knoten mit dem höchsten Uprank, welcher logischerweise immer eine Quelle sein muss, und nimmt dann die Kanten die zu einem Knoten führen mit einem Uprank, welcher genau 1 niedriger ist. Falls es mehrerer solcher Kanten gibt, gibt es mehrere kritische Pfade. Damit hat man mehrere Möglichkeiten. Will man nur einen dieser kritischen Pfade berechnen, nimmt man eine zufällige dieser Kanten. Will man alle kritischen Pfade berechnen, nimmt man alle Kanten und speichert sich für jede Abzweigung einen neuen Pfad.

Der Pseudocode für die Berechnung des Upranks zeigt 4.1. Am Anfangen initialisieren wir eine Hashmap, welche Knoten ihren Uprank zuweist. Alle Knoten die keine Senke sind haben einen Defaultwert von -1 und alle Senken haben einen Wert von 0. Als nächstes initialisieren wir eine

Queue, welches die noch zu bearbeiteten Knoten enthält. Am Anfang initialisieren wir diese mit allen Senken. Solange die Queue nicht leer ist, betrachten wir den obersten Knoten in der Queue und schauen uns alle seine Eingangskanten an. Falls einer seiner Vorgängerknoten einen um genau 1 niedrigeren Uprank hat, aktualisieren wir den Uprank und fügen den Knoten zur Queue hinzu.

Ein weiteres Problem, auf das wir gestoßen sind ist, dass der externe Graph, den man in TensorFlow aufbaut zwar azyklisch ist, wir aber dennoch beobachten konnten, dass der interne Graph auf dem wir gearbeitet haben Zyklen enthalten konnte. Ein Beispiel warum das problematisch ist, zeigt der Algorithmus für die Berechnung des UpRanks. Gibt es einen Zyklus im Graphen, würde dieser nicht terminieren, da ein beliebig langer Pfad von einem Knoten des Zyklus zu einer Senke gefunden werden kann. Damit unsere Algorithmen, auch in solchen Fällen funktionieren untersuchen wir am Anfang jeden Graphen immer nach Zyklen und entfernen diese, indem wir die letzte Kante des Zyklus löschen. Dabei bedeutet letzte Kante, die Kante die der Algorithmus als letztes traversiert hat, als er den Zyklus entdeckt hat.

4.2 Task Paralleler Algorithmus

Die Idee des „Task Parallelen“-Algorithmus ist, dass wir den Graphen so aufteilen, dass es immer Operationen gibt, welche Parallel ausgeführt werden können. Alg. 4.2 zeigt den Pseudocode des Algorithmus. Am Anfang berechnen wir den Uprank wie in Abschnitt 4.1 beschrieben und intialisieren den ReadyStatus für alle Operationen mit der Anzahl ihrer Eingangskanten. Der ReadyStatus wird benötigt umzuerkennen, wann eine Operation platzierbar wird. Eine Operation ist dann ausführbar, wenn alle ihre vorherigen Operationen platziert wurden. Damit können wir den ReadyStatus für eine Operation immer um 1 senken, wenn eine Eingangsoperation platziert wurde und wenn der ReadyStatus auf 0 fällt, wissen wir, dass die Operation platzierbar wurde.

Als nächstes initalisiert der Algorithmus eine PriorityQueue, welche zunächst mit allen Quelloperationen initialisiert wird. Die PriorityQueue ist nach dem Uprank der enthaltenen Knoten sortiert. Das bedeutet die Knoten mit dem höchsten Rank sind vorne in der Queue. Der Sinn dahinter ist, dass die obersten Operationen von dieser Queue auf verschiedene Geräte platziert werden. Dadurch werden Operationen mit dem gleichen UpRank auf verschiedene Geräte platziert. Das bedeutet, dass verschiedene kritische Pfade auf verschiedenen Geräten platziert werden und somit parallel verarbeitet werden, was einen Geschwindigkeitsvorteil bringen soll. Dass wir dadurch einen kritischen Pfad auf verschiedene Geräte platzieren, da alle Operationen eines kritischen Pfades einen hohen UpRank haben, kann nicht passieren. Dafür müssten beide Operationen gleichzeitig in der ReadyQueue sein, was nicht passieren kann, da nur Operationen in der ReadyQueue sind, für welche alle vorherigen Operationen schon platziert wurden.

Nachdem der Algorithmus alles benötigten Datenstrukturen intialisiert hat, betrachtet er die obersten i Operationen von der Queue. i ist hierbei die Zahl der vorhandenen Geräte. Für diese Operationen versucht der Algorithmus den potentiellen Kommunikationsaufwand, für jede mögliche Platzierung auf die vorhandenen Geräte, zu berechnen, . Dazu betrachtet der Algorithmus die Platzierung der Eingangsoperationen für die Operationen aus der frindge. Für jedes Gerät ist der berechnete Wert, die Anzahl der Eingangsoperationen die auf dem Gerät liegen. Mit diesem Wert baut der Algorithmus sich eine HashMap auf, welches ein Operationen und Geräte Paar auf einen Wert mapt. Umso höher der Wert ist, umso besser ist die Platzierung von der Operation auf das Gerät, da schon viele Eingangsoperationen auf dem Gerät liegen und somit weniger Kommunikation entsteht.

Algorithmus 4.2 Task Parallel

```
procedure placementTaskParallel(Graph, Devices)
    calculateUprank(G)
    // ReadyStatus gibt an wie viele noch nicht verarbeitete Eingangsknoten eine Operation hat
    for Knoten v in Graph do
        ReadyStatus[v] ← Anzahl von Eingangsknoten von v
    end for
    //Ready Queus ist eine Priority Queue die nach dem Uprank sortiert ist
    initalisiere ReadyNodes mit Quellen von Graph
    while ReadyNodes ≠ ∅ do
        frindge ← Die obersten Devices.size() Knoten von ReadyNodes
        // Finde für jedes Knoten und Geräte Paar, die Kommunikationskosten herraus
        for Knoten node von frindge do
            for Gerät device von Devices do
                commToDevice [Node][Device] ← #Eingangsknoten von node auf device
            end for
        end for
        // Platziere Knoten aus der frindge auf verschiedenen Geräten,
        // sodass die aufkommende Kommunikation minimal ist.
        for int i = 0; i <DeviceSet.size();i + + do
            Finde Knoten node und Gerät device mit dem höchsten Wert in commToDevice
            Platierze node auf device
            for Folgeknoten outNode von node do
                ReadyStatus[outNode] −=1
                if ReadyStatus[outNode] == 0 then
                    Füge outNode in ReadyNodes hinzu
                end if
            end for
            Entferne node und device von commToDevice
        end for
    end while
end procedure
```

Mit der HashMap berechnet der Algorithmus die Platzierung, indem er für jedes Gerät eine Operation sucht. Als Erstes nimmt er die Operationen und Geräte Kombination mit dem höchsten Wert und platziert die Operation auf das Gerät. Als Nächstes sucht er den nächsten höchsten Wert, wobei er schon platzierte Operationen und schon genutzte Geräte nicht beachtet.

Jedes Mal, wenn er eine Operation platziert aktualisiert er den ReadyStatus von allen Ausgangsoperationen der platzierten Operation. Wenn ein ReadyStatus auf 0 fällt, wird die Ausgangsoperation in die ReadyQueue aufgenommen, da all seine Eingangsoperationen platziert wurden. Die Platzierung von den einzelnen frindges wiederholt der Algorithmus solange bis die ReadyQueue leer ist, womit alle Operationen platziert sind.

Algorithmus 4.3 Scoring Algorithmus

```
procedure placementScoring(Graph G, DeviceSet deviceSet)
    calcUprank(G)
    ReadyQueue ← Quellknoten von G
    operationenAufGerät ← HashMap initaliziert mit allen Geräten -> 0
    while ReadyQueue ≠ ∅ do
        frindge ← Oberste DeviceSet.size() Knoten von ReadyQueue
        maxComm ← Maximale Anzahl von Eingangsknoten für die frindge
        for jeden Knoten v in der frindge do
            bestScore ← 0
            bestDevice ← StandardDevice //(CPU oder GPU:0 möglich)
            // Berechne für jede mögliche Platzierung eines Knotens einen Punktewert
            for jedes Gerät device im DeviceSet do
                CommScore ← Anzahl von Eingangsknoten, welche auf device liegen
                score ← calculateScore(commScore, maxComm, operationenAufGerät)
                //Aktualisiere die beste Platzierung wenn ein besserer Punktewert gefunden wurde
                if bestScore < score then
                    bestScore ← score
                    bestDevice ← device
                end if
            end for
            Platziere Knoten auf bestDevice
            Update der ReadyQueue
        end for
    end while
end procedure
```

4.3 Scoring

Bei dem Task Parallelen Ansatz, hatten wir zwar erreicht, dass es immer Operationen gibt die parallel auszuführen sind, jedoch hat die Evaluation dieses Ansatzes gezeigt, dass die Kommunikation zu teuer ist, wodurch der Algorithmus schlechtere Ergebnisse erzeugt hatte, als die Ausführung auf nur einer GPU. Die gesamte Evaluation des Algorithmus findet sich im nächsten Kapitel 5.3.

Dadurch ist eine Idee um die Ausführungszeiten zu verkürzen die Kommunikation zwischen den GPUs zu minimieren. Ein simpler Algorithmus, wessen einziges Ziel die Minimierung der Kommunikation ist, würde aber nicht funktionieren. Die triviale Lösung dafür, wäre die Platzierung aller Operationen auf eine GPU, wodurch es zu keiner Kommunikation kommen würde. Dadurch ist eine sinnvolle zweite Anforderung an den Algorithmus, dass er versucht die Lastenverteilung zu maximieren.

Die Idee des „Scoring“-Algorithmus ist, dass wir für jede Platzierung einer Operation auf die GPUs einen Kommunikationswert und einen Lastenverteilungswert berechnen. Die Operation platzieren wir dann auf das Gerät, für welche die Operation den höchsten Wert hat. Den Pseudocode des Algorithmus zeigt Alg. 4.3. Wie schon beim „TaskParallel“-Algorithmus fangen wir damit an, dass wir eine Priority Queue mit den Quelloperationen initialisieren. Diese sortieren wir wieder,

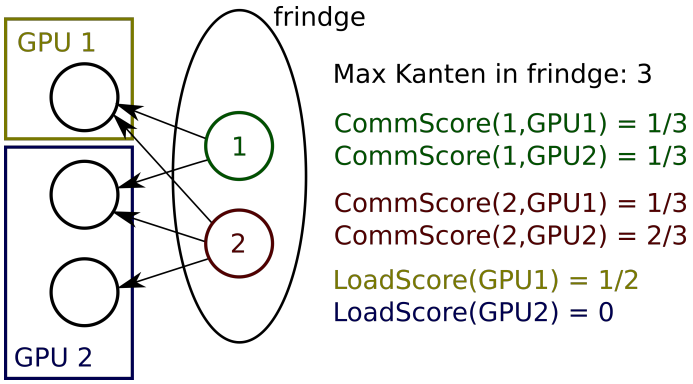


Abbildung 4.4: Ein einfaches Beispiel für die Wertungen des „Scoring“-Algorithmus.

aus dem gleichen Gründen wie beim „TaskParallel“-Algorithmus, nach dem UpRank. Neben der Queue benötigt der Algorithmus noch eine HashMap mit welcher er die Anzahl der platzierten Operationen auf jedem Gerät speichert. Diese wird am Anfang mit einem Wert von 0 für jedes Gerät im gegebenen DeviceSet initialisiert, da anfänglich noch keine Operationen platziert wurden.

Nach der Initialisierung der benötigten Datenstrukturen baut der Algorithmus sich eine frindge aus den vordersten Knoten der ReadyQueue. Im Gegensatz zum „TaskParallel“-Algorithmus, ist die Größe der fringe nicht vorgegeben. Ein guter Wert für die frindgegröße hat sich als die doppelte Anzahl der möglichen GPUs herausgestellt.

Für eine frindge sucht der Algorithmus als nächstes die maximale Anzahl an Eingangskanten für alle Operationen innerhalb der frindge. Mit diesen kann er den Kommunikationswert auf einen Wert zwischen 0 und 1 abbilden. Sie wird wie folgt für jede Kombination von Gerät d und Operation n berechnet:

$$\text{CommScore}(n, d) = \frac{|\text{InEdges of n on d}|}{|\text{Max Inedges in frindge}|}$$

Der Grund für die Berechnung der Kommunikations Score abhängig von der frindge ist, dass es in vielen Graphen viele Operationen mit einer geringen Anzahl an Kanten und wenige Operationen mit einer sehr hohen Anzahl an Kanten gibt. Würde man den relativen Kommunikationsaufwand abhängig vom gesamten Graph machen, hätten Operationen mit einer kleinen Eingangskantenanzahl fast identische kleine Kommunikationswerte und die Platzierung würde sich für diese wieder nur durch die Lastenverteilung ergeben.

Der Lastenverteilungswert wird für jedes Gerät d wie folgt berechnet:

$$\text{LoadScore}(d) = 1 - \frac{|\text{Anzahl Knoten auf d}|}{min(1, |\text{Max Anzahl Knoten auf allen devicen}|)}$$

Falls das Gerät, dass Gerät mit der größten Anzahl an Operationen ist, wertet sich die Formel zu 0 aus und falls das zu betrachtente Gerät noch keine platzirten Operationen hat, wertet sich die Formel zu 1 aus. Das Minimum wird benötigt um eine Division durch 0 zu verhindern, wenn noch gar kein Operationen platziert wurden. Abb. 4.4 bietet ein Beispiel für den Kommunikationswert und den Lastenverteilungswert für einen einfachen Fall.

Algorithmus 4.4 Clustering

```
procedure CALCULATECLUSTER(Graph G, int nrDevices)
    bestCluster ← ∅
    bestScore ← 0
    for i = 1 bis 1000 do
        testGraph ← Kopie von G
        while testGraph.numberNodes() > nrDevices do
            edge ← Zufällige Kante
            testGraph.mergeNodes(edge)
        end while
        score ← evaluate(testGraph)
        if score > bestScore then
            bestCluster ← testGraph
            bestScore ← score
        end if
    end for
end procedure
```

Um die den Gesamtwert zu berechnen, geht der Algorithmus für jede Operation jedes mögliche Gerät durch und berechnet die Werte nach den genannten Formeln. Der Kommunikationswert und der Lastenverteilungswert werden dann mit folgender Formel miteinander verrechnet:

$$\text{Score} = \alpha \cdot \text{LoadScorre} + \beta \cdot \text{CommScore}$$

Je nachdem wie man das α und das β wählt, kann man einen Fokus auf die Lastenverteilung oder auf die Kommunikation setzen. Wir haben noch andere Möglichkeiten untersucht die beiden Werte zu verrechnen, wie z. B. die Multiplikation der beiden Werte. Diese haben sich aber alle schlechter als die gewichtete Addition der beiden Werte erwiesen.

4.4 Clustering

Die Idee des Clustering Algorithmus ist, dass wir den Graphen in Cluster aufteilen, wobei die Operationen innerhalb eines Clusters stark und die Operationen zwischen den Clustern schwach miteinander verbunden sind. Dies hat den Vorteil, dass wenn wir die Cluster auf verschiedene GPUs platzieren, dass wir nur wenig Kommunikation zwischen den Clustern haben und dass Operationen die nicht so gut zu parallelisieren sind auf der gleichen GPU liegen.

Den Algorithmus zur Bildung der Cluster zeigt 4.4. Er ist eine Abänderung des Karger's Algorithmus [Kar93], welcher Standardmäßig zur Berechnung des MinCuts eines Graphen genutzt wird. Zum Anfang des Algorithmus initialisieren wir zwei Variablen. Eine Variable in der wir das bisher beste gefundene Cluster speichern und eine zweite Variable in der wir den Punktewert dieses Clusters speichern. Als Nächstes bilden wir 1000 Cluster. Der Algorithmus würde auch mit einer größeren oder kleineren Anzahl funktionieren. Eine höhere Anzahl würde den Algorithmus robuster machen, aber dafür die Laufzeit erhöhen, eine kleinere Zahl würde genau das Gegenteil bewirken. Wir empfanden, dass eine Zahl von 1000 einen guten Ausgleich bietet.

Algorithmus 4.5 mergeNodesCapped

```
procedure MERGENODESCAPPED(Edge e)
    if cluster[e.inNode].size() + cluster[e.outNode] > max then
        this.delete(e)
    else
        this.mergeNodes(edge)
    end if
end procedure
```

Die einzelnen Cluster bilden wir, indem wir solange zufällige Kanten vereinen, bis wir nur noch so viele Superknoten haben, wie wir Geräte haben. Für das so gebildete Cluster berechnen wir den Punktewert die aussagt, wie gut es ist und behalten es, falls es besser als das zuvor gefundene Cluster ist oder verwerfen es, falls es schlechter ist. Zur Berechnung des Punktewertes eines Cluster haben wir zwei verschiedene Ansätze getestet. Wie schon beim „Scoring"-Algorithmus legen wir einmal den Fokus auf die Lastenverteilung und beim anderen den Fokus auf die Kommunikationsreduktion.

Bei der Variante welche die Lastenverteilung beachtet berechnen wir den Punktewert wie folgt:

$$\text{LoadScore(Cluster)} = \frac{|\text{Kleinstes Cluster}|}{|\text{Größtes Cluster}|}$$

Der Punktewert ist also das Verhältniss vom größten Cluster zum kleinsten Cluster. Umso kleiner dieser Unterschied ist umso besser ist der Punktewert und umso besser ist auch die Lastenverteilung.

Bei der Variante welche die Kommunikation beachtet berechnen wir den Punktewert wie folgt:

$$\text{CommScore(Cluster)} = -1 \cdot \sum_{\text{v : Graph}} \sum_{\text{w : Inodes(v)}} \begin{cases} 0 & \text{, v und w sind auf dem gleichen Cluster} \\ 1 & \text{, v und w sind auf verschiedenen Clustern} \end{cases}$$

Der Kommunikationswert ist also die Anzahl von Kanten zwischen den Clustern. Den Wert haben wir mit -1 multiplzierrt, damit wir einen Punktewert haben, welcher besser ist umso höher er ist.

Bei der Kommunikationsvariante hat sich in der Evaluation gezeigt, dass der Algorithmus 1-2 große Cluster findet und die restlichen Cluster alle sehr klein sind. Dadurch hat die Variante wieder das schon beschriebene Problem, dass ale Operationen auf nur ein Gerät platziert werden, wenn nur die Kommunikation beachtet wird. Um den Entgegenzuwirken, haben wir die Kommunikationsvariante erweitert.

Wir haben eine maximale Größe für die Cluster festgelegt. Dadurch ist es nicht mehr möglich, dass sich ein 1-2 sehr große und viele kleine Cluster bilden. Um dies umzusetzen haben wir in der mergeNodes() Methode abgefragt, ob die Vereinigung der beiden Cluster größer als das Maximum wird. Falls ja, löschen wir die Kante da sie nicht mehr zu einer Vereinigung von zwei Clustern führen kann. Falls nein, vereinen wir die beiden Cluster wie vorher. Alg. 4.5 zeigt den Pseudocode dafür. Die Maximale Größe eines Cluster haben wir nach folgender Formel berechnet:

$$Max = \frac{graph.size()}{device.size()} \cdot 1.5$$

Somit kann ein Cluster um 50% größer sein, als eine Clustergröße bei perfekter Lastenverteilung.

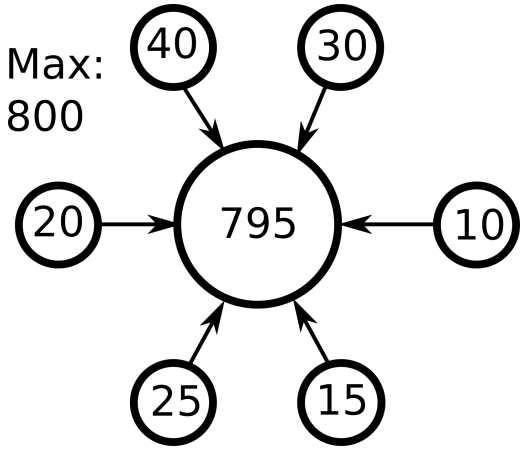


Abbildung 4.5: Ein Beispiel, bei welchem keine Cluster mehr vereint werden können. Die vielen kleinen Cluster sind nur noch mit dem großen Cluster verbunden, womit kein Cluster gebildet werden kann, welches unter dem Maximum liegt. Die Zahlen geben die Größe der Cluster an.

Die Erweiterung hat ein weiteres Problem ergeben. Es konnte passieren, dass viele kleine Cluster nur noch mit einem großen Cluster verbunden sind. Ein Beispiel dafür zeigt Abb. 4.5. Dadurch konnte es passieren, dass der Algorithmus nicht mehr terminiert, da er keine Cluster mehr vereinen konnte, es aber noch mehr als die Anzahl an Geräten gab.

Um das Problem zu lösen, haben wir die Terminationsbedingung des Algorithmus angepasst. Wir haben das Zusammenfügen der Operationen solange gemacht, bis die gewünschte Anzahl an Clustern erreicht wurde oder bis der Graph keine Kanten mehr hatte. Eines der beiden Fälle muss immer eintreten, da wir alle Kanten löschen, welche nicht mehr vereint werden können. Danach haben wir, falls noch zu viele Cluster existieren, immer die beiden kleinsten Cluster vereint, bis die gewünschte Anzahl an Clustern erreicht wurde.

5 Evaluation

In diesem Kapitel werden die drei vorgestellten Lösungen evaluiert. Dazu werden als erstes kurz die genutzten Modelle beschrieben. Dies ist erstens die neuronale maschinelle Übersetzung(NMT), welche zur automatischen Übersetzung von Texten genutzt wird. Und zweites ein rekurrentes neuronales Netz(RNN), welches zum Beispiel zur Spracherkennung genutzt werden kann. Danach wird kurz der Aufbau der Experimente beschrieben. Als letztes werden die Ergebnisse der Experimente evaluiert. Dafür wird als erstes auf jeden Algorithmus einzeln Eingegangen und danach werden alle Algorithmen miteinander verglichen.

5.1 Modelle

Wir haben uns zwei verbreitete Modelle ausgesucht, auf welchen wir unsere Algorithmen evaluieren. Dies sind ein rekurrentes neuronales Netz(RNN) und die neuronale maschinelles Übersetzung(NMT).

5.1.1 Neuronale maschinelle Übersetzung

Wenn ein Mensch einen Satz von einer Sprache in die andere Sprache übersetzt macht er das nicht Wort für Wort. Er betrachtet den gesamten Satz, überlegt sich die Bedeutung des Satzes und versucht dann die Bedeutung in der anderen Sprache auszudrücken. Auf dieser Idee basiert die NMT. Die NMT besteht aus zwei Bausteinen. Einem Encoder, welcher einen Satz auf einen Bedeutungsvektor abbildet. Und einem Decoder, welcher aus dem Bedeutungsvektor eine Übersetzung in der Zielsprache sucht. Eine Abbildung dieser Architektur bietet Abb. 5.1.

Für unser Experiment haben wir dass NMT-Modell von Thang Luong et al. an unsere Bedürfnisse angepasst [LBZ17]. Wir haben jeweils 2-Schichten für den Encoder und Decoder genutzt, welche jeweils eine Größe von 1024 haben. Die maximale Länge der Sätze haben wir auf 50 festgelegt und die Batchgröße ist 64. Wir haben alle Operationen einer LSTM-Zelle colociert.

5.1.2 Rekurrentes Neuronales Netz

Im Gegensatz zu herkömmlichen neuronalen Netzen erlaubt ein rekurrentes neuronales Netz (RNN) Verbindungen von einer Schicht in eine vorherige Schicht. Dadurch ist ein RNN ein neuronales Netz mit Schleifen. Um diese Netzwerke zu trainieren ist es üblich sie für eine bestimmte Anzahl an Schritten auszurollen. In Abb. 5.2 sieht man ein RNN, welches für vier Schritte ausgerollt wurde. Ein Vorteil von RNN im Gegensatz zu herkömmlichen neuronalen Netzen ist, dass es dadurch

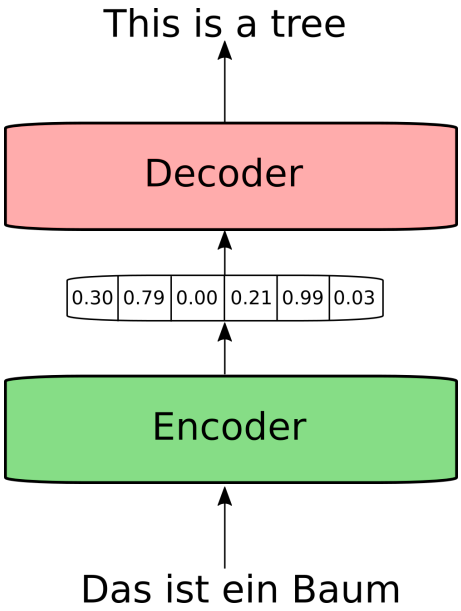


Abbildung 5.1: Encoder und Decoder Architekture des NMT

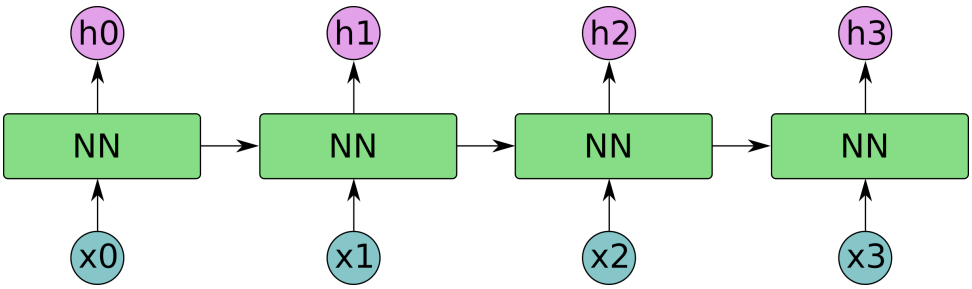


Abbildung 5.2: Ein für vier Schritte ausgerolltes RNN

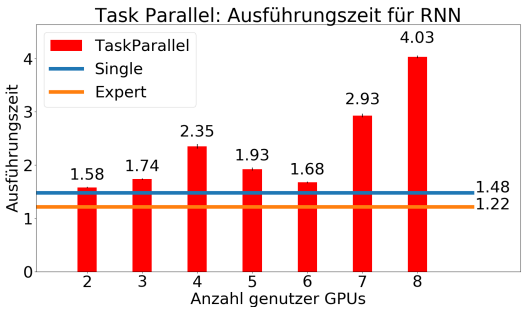
vorherige Informationen mitnutzen kann [Ola15]. Die größten Erfolge von RNNs konnte erreicht werden, wenn als das zu wiederholende Netz eine Long-Term-Short-Term Netz (LSTM) genutzt wurde[Ola15].

Für unser Experiment haben wir das RNN Model aus dem Tensorflow Tutorial für unsere Bedürfnisse angepasst [18]. Dieses basiert auf den Model von Zaremba et al. [ZSV14].

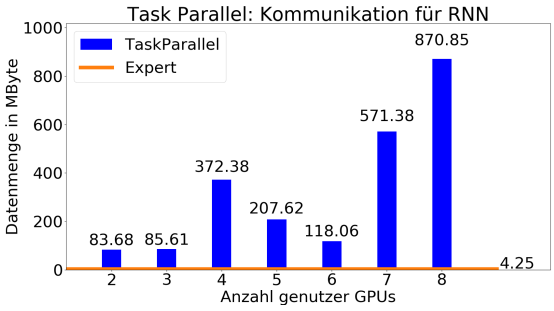
Für unser Experiment haben wir das RNN Model aus dem Tensorflow Tutorial für unsere Bedürfnisse angepasst. Dieses basiert auf dem Model von Zaremba et al. [ZSV14]. Unser genutztes Modell besteht aus zwei Schichten. Jede Schicht hat eine Größevon 1500 Einheiten und wir haben das Modell für 40 Schritte ausgerollt. Die Batchgröße Betrug 20.

5.2 Experiment Aufbau

In diesem Abschnitt beschreiben wir den Aufbau der Experimente. Sie unterteilt sich in eine Übersicht über die genutzte Hardware und eine Übersicht über den Ablauf der Experimente.



(a) Ausführungszeit TaskParallel: RNN



(b) Kommunikationsaufwand TaskParallel: RNN

Abbildung 5.3: Evaluation des „TaskParallelen"-Algorithmus für das RNN. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente.

5.2.1 Hardware Aufbau

Zur Ausführung der Experimente haben wir einen Rechner mit 8 GTX1080 genutzt. Die genutzten CPUs waren zwei Intel Xeon Gold 6148 mit 2.4GHz und 20 Kernen. Zur Verfügung standen 768GB DDR3 RAM.

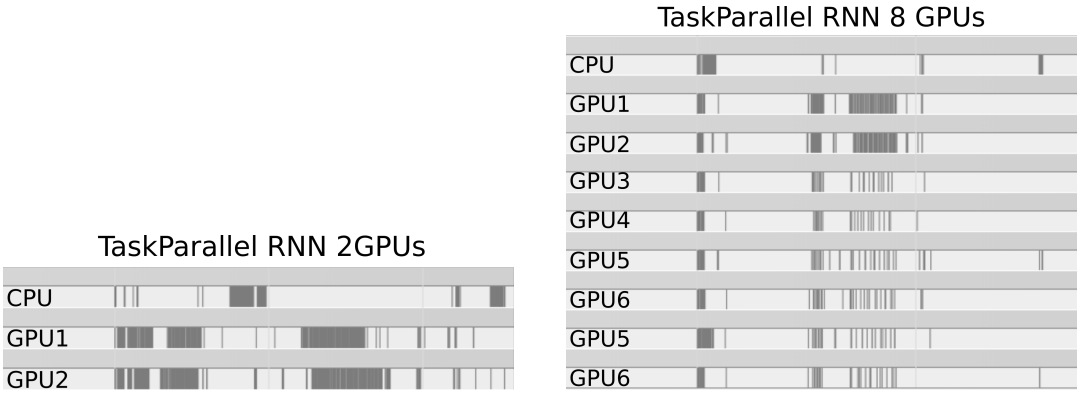
5.2.2 Ablauf der Experimente

Für die Experimente haben wir die genutzte Zahl von GPUs zwischen 1 und 8 variiert. Dazu haben wir den Befehl CUDA_VISIBLE_DEVICES genutzt mit welchem man TensorFlow mitteilen kann, welche GPUs es nutzen darf. Für jede Ausführung der Platzierung haben wir den Durchschnitt von 10 Trainingsschritten genommen. Ein Trainingsschritt ist dabei die Dauer der Ausführung eines Schrittes einer TensorFlow Session. Da die ersten Ausführungen bei TensorFlow einen Mehraufwand haben, haben wir die ersten 10 Schritte verworfen und nur ab dem 11. Schritt die Ausführungszeit gemessen.

Als Benchmarks haben wir für jedes Modell eine Expertenverteilung implementiert. Diese sind aus der Literatur bekannte Arten die beiden Modelle zu verteilen. Zudem haben wir noch eine Ausführung mit nur einer einzelnen GPU getestet um eine Vergleich zu einer nicht verteilten Ausführung zu bekommen.

5.3 Evaluation TaskParallel

Die Idee des „TaskParallel"-Algorithmus war, dass die Operationen so platziert werden, dass immer eine Operation ausführbar ist. Dazu geht der Algorithmus den Graphen von vorne nach hinten durch und platziert ausführbare Operationen auf verschiedene Geräte. Im Folgenden wird der Algorithmus für das RNN und die NMT evaluiert werden. Danach wird eine gesamt Evaluation des Algorithmus gegeben werden.



(a) 2GPUs **(b)** 8GPUs

Abbildung 5.4: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „TaskParallel“-Algorithmus für das RNN. Links zeigt die die Auslastung für 2GPUs und rechts für 8GPUs.

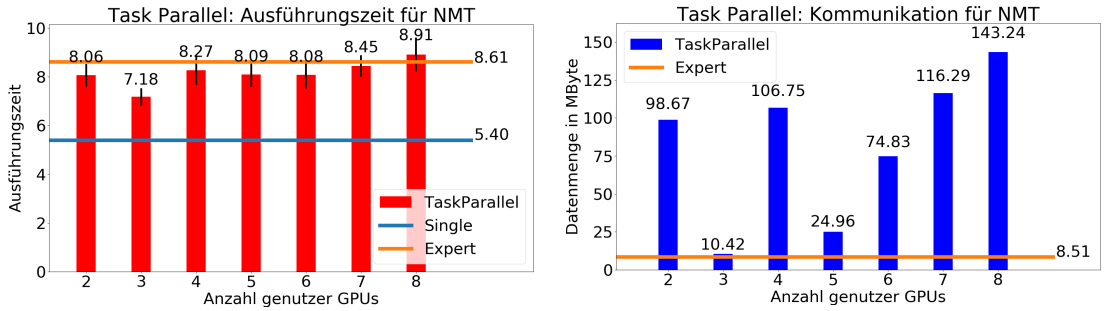
5.3.1 TaskParallel: RNN

Abb. 5.3a zeigt die Ergebnisse für die Ausführung des „TaskParallelen“-Algorithmus für das RNN im Vergleich zu einer Ausführung mit Expertenplatzierung und einer Ausführung auf einer einzelnen GPU. Das beste Ergebnis des „TaskParallelen“-Algorithmus konnte mit einer Ausführung auf 2 GPUs erzielt werden und es ist eine Tendenz zu erkennen, dass eine höhere Anzahl der verwendeten GPUs generell einen negativen Effekt auf das Ergebnis hat. Ausnahmen für diese Tendenz sind Ausführungen mit 5 und 6 GPUs, bei denen ein besseres Ergebnis als mit 4 GPUs erreicht wurde.

Betrachten wir 5.3b können wir den Grund dafür sehen. Der Graph zeigt die bei der Ausführung aufgekommene Kommunikation. Er zeigt, dass der „TaskParallele“-Algorithmus einen sehr hohen Kommunikationsaufwand hat. Bei der Ausführung mit 2GPUs hat er fast den 20 fachen Kommunikationsaufwand im Vergleich zum Experten. Eine weitere Beobachtung ist, dass die Ausführungszeiten mit den Kommunikationskosten stark korrelieren.

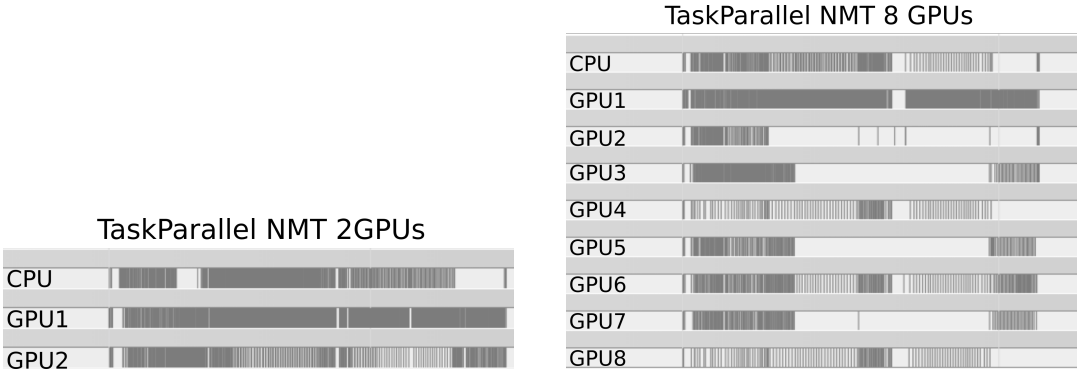
Ein Grund für die im Vergleich zu 4GPUs niedrigen Kommunikationskosten von 5 und 6GPUs könnte sein, dass das RNN eine bessere Struktur für 5 und 6 GPUs hat. Ein einfaches Beispiel für einen solchen Fall, wäre ein Graph, welcher aus fünf parallelen Pfaden besteht. Würden wir den „TaskParallelen“-Algorithmus nutzen, um den Graphen auf 4 GPUs zu platzieren, wäre er dazu gezwungen einzelne Operationen von den Pfaden auf verschiedene GPUs zu platzieren, was zu einem hohen Kommunikationsaufwand führt.

Als letztes betrachten wir die Geräteauslastung einer Ausführung mit 2 und 8 GPUs. Für 2GPUs in Abb. 5.4a kann man sehen, dass beide GPUs gleichmäßig ausgelastet sind, aber viel Idle-Zeit entsteht, da eine GPU auf das Ergebnis der anderen warten muss. Betrachten man das die Auslastung für 8 GPUs in Abb. 5.4b bietet sich ein ähnliches Bild. Alle GPUs haben zwar eine ähnliche Auslastung, es kommt aber durch die Verteilung zu sehr hohen Idle-Zeiten. Dies erklärt auch nochmals die schlechte Performanz der Ausführung für 8GPUs.



(a) Ausführungszeit TaskParallel: NMT **(b)** Kommunikationsaufwand TaskParallel: NMT

Abbildung 5.5: Evaluation des „TaskParallelen"-Algorithmus für die NMT. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente.



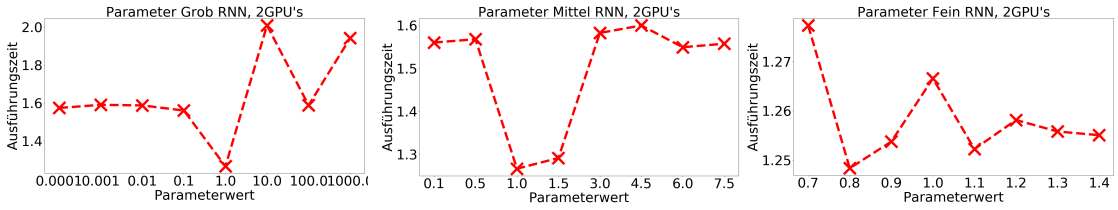
(a) 2GPUs **(b)** 8GPUs

Abbildung 5.6: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „TaskParallel"-Algorithmus für das RNN. Links zeigt die Auslastung für 2GPUs und rechts für 8GPUs.

5.3.2 TaskParallel: NMT

Abb. 5.5a zeigt die Ausführungszeiten für die NMT mit dem „TaskParallelen"-Algorithmus. Wir können sehen, dass die Erhöhung der GPUs keine so großen Auswirkungen hat wie beim RNN, aber es lässt sich wieder eine Tendenz erkennen, dass weniger GPUs bessere Ergebnisse erzielen. Betrachten man den Kommunikationsaufwand in Abb. 5.5 sieht man, dass auch hier die Ausführungszeiten mit diesen korrelieren. Interessante Fälle sind die Kommunikationskosten für 3 und 5 GPUs. Die Vermutung, wie diese zustande kommen, ist die gleiche wie beim RNN. Da der Algorithmus nicht versucht die Kommunikations zu reduzieren, kann es sein, dass die Struktur des Graphens für 3 und 5 GPUs besser geeignet ist.

Betrachtet man die Auslastung für 2GPUs in Abb. 5.6a, sieht man nichts wirklich Auffälliges. Die GPUs sind gleichmäßig stark ausgelastet, sie arbeiten viel Parallel und es gibt nur wenig Idle-Zeit. Für die Ausführung mit 8GPUs sieht das anders aus, wie Abb. 5.6b zeigt. Nur eine der GPUs ist wirklich stark ausgelastet und die anderen GPUs haben sehr viel Idle-Zeit. Das könnte eine Erklärung sein, warum die Nutzung von mehr GPUs keinen Mehrwert bringt. Sie werden kaum ausgelastet und führen nur zu einer Erhöhung der Kommunikation.



(a) Log 0.001-10000 **(b)** 1.0-100.0 **(c)** 7.0-15.0

Abbildung 5.7: Die Bestimmung des Parameterwertes für das RNN Modell. Ganz links ist eine grobe Suche, bei der logarithmisch der Größenwert des Parameters bestimmt wird. In der Mitte ist eine feinere Suche bei der die Werte um den gefundenen Wert evaluiert werden. Ganz Rechts ist eine feine Suche, bei der versucht wird den Parameterwert auf 0.1 genau zu bestimmen.

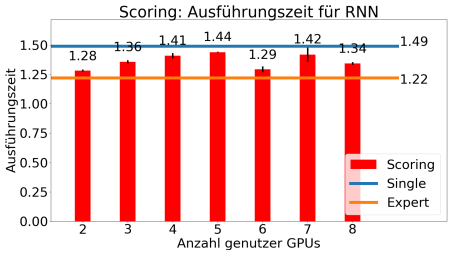
5.3.3 TaskParallel: Ergebniss

Der „TaskParallel“-Algorithmus zeigt sich als wenig Erfolgreich. Im NMT konnte der Experte zwar geschlagen werden, aber er skaliert nur sehr schlecht. Die Erhöhung der GPUs hat eher einen negativen Einfluss auf das Ergebnis als einen positiven. Die Erklärung für die schlechte Leistung kommt vermutlich aus zwei Gründen zustande. Der erste Grund sind die sehr hohen Kommunikationskosten, welche mit der Anzahl der GPUs zunehmen. Der zweite Grund ist die schlechte GPU-Auslastung. Bei der Nutzung von 8 GPUs waren diese Größtenteils Idle. Damit konnte er den großen Vorteil der 8-fachen Rechenpower nicht nutzen.

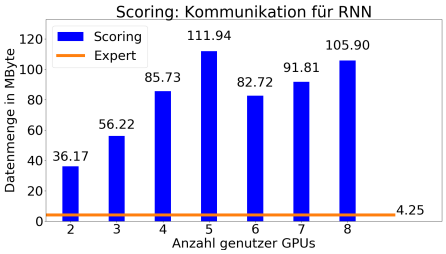
5.4 Evaluation Scoring

Die Idee des „Scoring“-Algorithmus ist, dass wir eine Score in Abhängigkeit des Load Balancing und des Kommunikationsaufwandes berechnen. Eine Frage die bei diesem Algorithmus offen ist, ist wie man die beiden Werte miteinander verrechnet. Um dies herauszufinden, haben wir den Algorithmus für verschiedene Parameter ausgeführt und den Wert mit der besten Ausführungszeit bestimmt. Nur das Verhältnis der beiden Werte hat einen Einfluss auf das Ergebnis. Das bedeutet ein Parameterwert von 10 für die Lastenverteilung und einen Parameterwert von 20 für die Kommunikation, ist das gleiche wie ein Parameterwert von 1 und 2. Deswegen haben wir den Kommunikationswert auf 1 festgesetzt und den Lastenverteilungswert variiert. Ein Parameterwert von 10.0 bedeutet, dass wir den Lastenverteilungswert um das 10-Fache höher gewichten als den Kommunikationswert. Ein Parameterwert von 0.1 bedeutet, dass wir den Kommunikationswert um das 10-Fache höher gewichten als den Lastenverteilungswert.

Nachdem wir den besten Parameterwert für das Modell bestimmt haben, haben wir diesen wieder im Vergleich zum Experten und zur Ausführung mit einer einzelnen GPU evaluiert.



(a) Ausführungszeit Scoring für RNN



(b) Kommunikationsaufwand Scoring für RNN

Abbildung 5.8: Evaluation des „Scoring"-Algorithmus für das RNN. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente. Als Paramter wurde der Wert 1.1 gewählt. Dies bedeutet, dass die Lastenverteilung um 10% stärker gewichtet wurde als die Kommunikation.

5.4.1 Parameterbestimmung für RNN

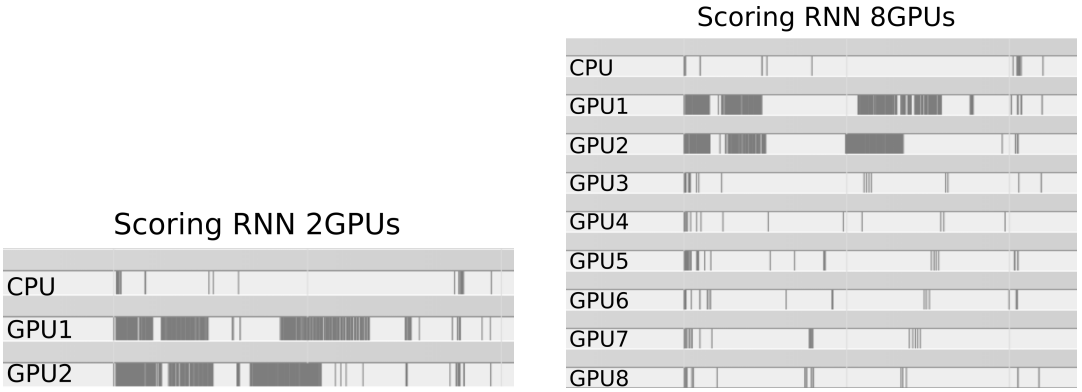
Um einen guten Parameter für den Algorithmus zu bekommen, haben wir eine Reihe von Experimenten durchgeführt. Als Erstes haben wir versucht die Größenordnung des Parameters herauszufinden. Dazu haben wir das Experiment mehrmals ausgeführt, wobei sich der Parameter zwischen jeder Ausführung um eine 10er Potenz gesteigert hatte. Das Ergebnis der Experimente zeigt 5.7a.

Betrachten man alle Werte kleiner 1.0, welche eine höhere Gewichtung der Kommunikation bedeutet, kann man sehen, dass sie alle eine ähnliche Ausführungszeit haben. Eine Erklärung für dieses Verhalten könnte sein, dass ab einer 10 Mal so starken Gewichtung der Kommunikation, der Algorithmus nur noch auf Gunsten der Kommunikation platziert. Damit hätte eine noch stärkere Gewichtung der Kommunikation keine Auswirkung auf die gefundene Platzierung. Betrachten wir im Gegensatz die Werte über 1.0, welche eine höhere Gewichtung der Lastenverteilung bedeuten, springen diese Werte sehr. Dadurch gibt es kein eindeutiges Minimum auf welche die Werte zulaufen, wodurch es möglich ist, dass es einen besseren Parameterwert gibt als den gefundenen. Der beste gefunden Wert für die grobe Suche ist eine ausgeglichene Gewichtung der beiden Bewertungen mit 1.0.

Um den Parameter genauer zu bestimmen haben wir in Abb. 5.7b, Parameterwerte im Bereich von 0.1–7.5 betrachtet. Bei diesen sieht man ein eindeutiges Minimum zwischen den Werten 1.0 und 1.5, womit die Lastenverteilung minimal besser gewichtet wird.

Um den Parameterwert auf 0.1 genau zu bestimmen haben wir eine weitere Reihe an Experimenten durchgeführt, bei welcher wir den Parameterwert von 0.7-1.5 variiert haben. Abb. 5.7c zeigt das Ergebnis dafür. Man sieht, dass die Wahl des Parameters in diesem Bereich nicht sehr bedeutsam ist. Der beste Wert unterscheidet sich vom schlechtesten Wert nur um einen Wert von 0.025s. Dadurch könnten die Unterschiede zwischen den Parameterwerten durch die Varianz zustande kommen.

Die Experimente haben ergeben, dass die beste Wahl des Parameters für das RNN eine ausgeglichene Gewichtung zwischen der Lastenverteilung und der Kommunikation ist, wobei eine kleine Tendenz zur stärkeren Gewichtung der Lastenverteilung zu erkennen ist. Für die Evaluation des Algorithmus haben wir uns deswegen auf einen Parameterwert von 1.1 festgelegt.



(a) 2GPUs **(b)** 8GPUs

Abbildung 5.9: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „Scoring"-Algorithmus für das RNN. Links zeigt die die Auslastung für 2GPUs und rechts für 8GPUs. Als Parameter wurde der Wert 1.1 gewählt

5.4.2 Scoring: RNN

Abb. 5.8a zeigt die Ausführungszeiten für den „Scoring"-Algorithmus. Wie schon beim „TaskParallelen"-Algorithmus, konnte das beste Ergebnis mit 2GPUs erreicht werden. Im Gegensatz zu diesen skaliert der „Scoring"-Algorithmus aber etwas besser. Die Skalierung des Algorithmus ist aber immer noch nicht gut. Trotz erhöhter Rechenpower durch die Nutzung von mehr GPUs verbessert sich die Performanz nicht.

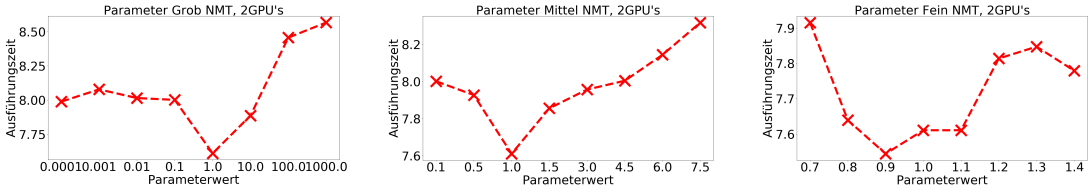
Betrachtet man die Kommunikationskosten aus Abb. 5.11b sieht man, dass sie keine Korrelation zur Anzahl der GPUs oder zur Ausführungszeit haben. Das einzige Auffällige ist, dass der Kommunikationsaufwand um ein Vielfaches höher ist, als der Experte.

Betrachten man die Geräteauslastung für 2GPUs in Abb. 5.9a, sieht man auch hier eine gleichmäßige Auslastung der GPUs. Ansonsten bietet sie keine weiteren Auffälligkeiten. Betrachtet man die Ausführungszeiten für 8GPUs in Abb. 5.9b sieht man, dass die GPUs sehr unterschiedlich ausgelastet sind. Es gibt zwei GPUs die vergleichsweise stark ausgelastet sind und die restlichen GPUs sind nur sehr wenig ausgelastet. Dadurch bringt die Erhöhung der GPUs wieder nicht viel, da die mehr Rechenpower nicht gut genutzt wird.

5.4.3 Parameterbestimmung für NMT

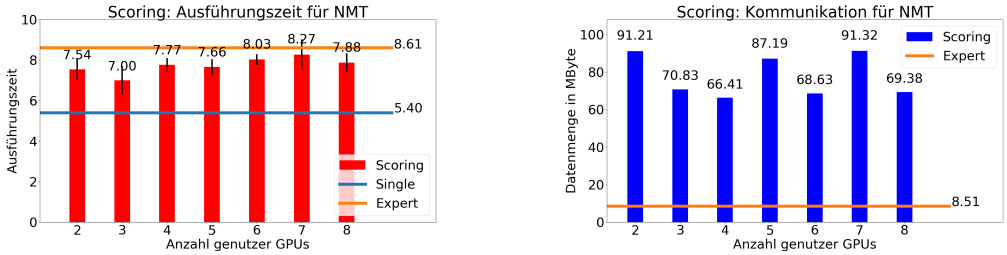
Wie für das RNN haben wir auch für die NMT versucht einen guten Parameter zu finden. Dazu sind wir wie beim RNN vorgegangen. Wir haben als erstes versucht die Größenordnung des Parameters herauszufinden und haben eine Experimentenreihe mit einer Steigung des Parameters in 10er Potenzen durchgeführt.

Das Ergebnis der Experimente zeigt 5.10a. Wie beim RNN, sieht man einen konstanten Wert für die Parameterwerte, welche die Kommunikation stärker bewerten. Der Grund dafür ist wahrscheinlich der gleiche. Bei einer 10fachen Gewichtung der Kommunikation werden die meisten Entscheidungen schon auf Grund der Kommunikationskosten getroffen. Dadurch hat die weitere Erhöhung des



(a) Log 0.001-10000 **(b)** 1.0-100.0 **(c)** 7.0-15.0

Abbildung 5.10: Die Bestimmung des Parameterwertes für die NMT Modell. Ganz links ist eine grobe Suche, bei der logarithmisch der Größenwert des Parameters bestimmt wird. In der Mitte ist eine feinere Suche bei der die Werte um den gefundenen Wert evaluiert werden. Ganz Rechts ist eine feine Suche, bei der versucht wird den Parameterwert auf 0.1 genau zu bestimmen.



(a) Ausführungszeit Scoring für NMT **(b)** Kommunikationsaufwand Scoring für NMT

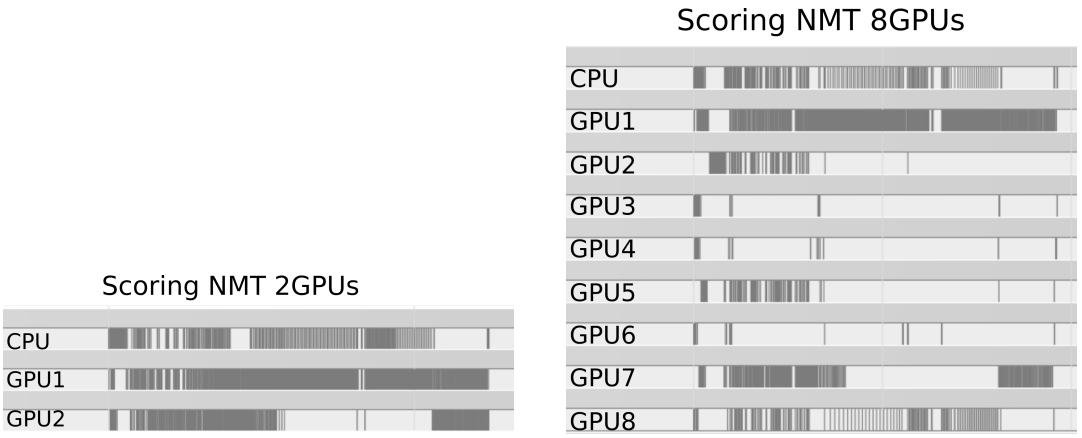
Abbildung 5.11: Evaluation des „Scoring“-Algorithmus für die NMT. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente. Als Paramter wurde der Wert 0.9 gewählt. Das bedeutet, dass die Kommunikation um 11% stärker gewichtet wurde als die Lastenverteilung.

Kommunikationsgewichtes keinen große Auswirkung auf die Platzierung. Bei der Lastenverteilung sieht, dass anders aus. Umso höher das Gewicht der Lastenverteilung ist umso schlechter ist die Performanz des Algorithmus. Hier ist die Vermutung, dass durch die stärkere Gewichtung der Lastenverteilung, die Kommunikationskosten vernachlässigt werden. Dadurch entsteht immer mehr Kommunikation, welche die Ausführungszeiten verschlechtern.

Eine feinere Unterteilung zwischen den Werten 0.1 und 7.5 bietet die Abb. 5.10b. Hier hat wieder eine gleich starke Gewichtung der beiden Werte das beste Ergebnis. Ein höherer Wert oder ein niedriger Wert als dieser, erhöht die Ausführungszeit.

Eine noch feinere Unterteilung zwischen 0.7 und 1.4 bietet Abb. 5.10c. Bei dieser sieht man die beste Performanz bei einem Parameterwert von 0.9. Eine Auffälligkeit im Gegensatz zum RNN ist, dass die Werte hier viel stärker variieren. Vergleichen man die Ausführungszeiten von 0.7 und 0.9 sieht man einen Unterschied der Ausführungszeiten von 0.5 Sekunden, obwohl der Parameter nur um 0.2 geändert wurde.

Wie beim RNN zeigt sich, dass die beste Wahl des Parameters eine ausgeglichene Gewichtung zwischen der Lastenauswertung und der Kommunikation ist, wobei eine kleine Tendenz zur stärkeren Gewichtung der Kommunikation zu erkennen ist. Die Evaluation des Algorithmus haben wir deswegen mit einem Wert von 0.9 durchgeführt.



(a) 2GPUs **(b)** 8GPUs

Abbildung 5.12: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „Scoring"-Algorithmus für die NMT. Links zeigt die Auslastung für 2GPUs und rechts für 8GPUs. Als Parameter wurde der Wert 0.9 gewählt.

5.4.4 Scoring: NMT

Abb. 5.11a zeigt die Ausführungszeiten für den „Scoring"-Algorithmus für die NMT. Als Parameterwert wurde 0.9 gewählt, was bedeutet, dass die Lastenverteilung im Verhältnis zur Kommunikation von 9:10 steht. Man sieht, dass eine Anzahl von 3GPUs die schnellse Ausführungszeit hat. Alle anderen Werte sind recht konstant und halten sich bei um die 8 Sekunden auf. Dies ist die gleiche Skallierbarkeit, wie beim RNN. Durch die Erhöhung der GPUs, verschlechtert sich die Performanz zwar nicht, aber es gibt keinen Mehrgewinn.

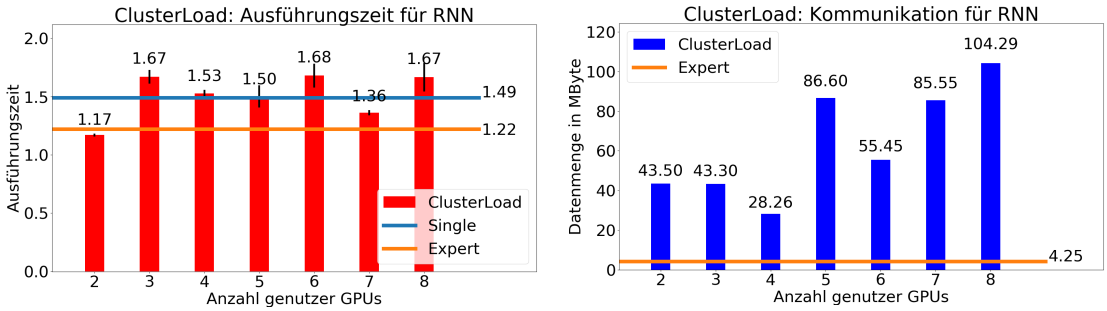
Aufällig sind die hohen Varianzen des Algorithmus. Der Algorithmus nutzt keinen Zufall, weswegen für einen Graphen und einen Parameterwert, immer die gleiche Platzierung gefunden werden sollte. Da auch die anderen Algorithmen bei der NMT eine hohe Varianz haben, ist es möglich, dass diese vom Modell und nicht von dem Algorithmus kommt.

Betrachtet man die Kommunikationskosten aus Abb. 5.11b sieht man, dass sie keine Korrelation zur Anzahl von GPUs haben. Ansonsten gibt es bietet sie keine weiteren Auffälligkeiten.

Betrachten man die Geräteauslastung für 2GPUs in Abb. 5.9a, sieht man, dass eine der beiden GPUs viel stärker ausgelastet ist als die andere. Aber ansonsten gibt es keine Auffälligkeiten. Betrachtet man die Ausführungszeiten für 8GPUs in Abb. 5.9b sieht man das gleiche Problem, wie beim RNN. Alle außer eine GPU sind nur sehr wenig ausgelastet. Dadurch kann die mehr Rechenpower wieder nicht genutzt werden.

5.5 Ergebniss Scoring

Wie schon beim „TaskParallelen"-Algorithmus sind die Ergebnisse nicht optimal. Beim RNN Modell konnten zwar nahe zu die Ergebnisse des Experten erreicht werden und beim NMT konnte dieser sogar geschlagen werden, aber der Algorithmus scheint nicht zu skalieren. Eine niedrige Anzahl von



(a) Ausführungszeit ClusterLoad: RNN

(b) Kommunikationsaufwand ClusterLoad: RNN

Abbildung 5.13: Evaluation des „ClusterLoad“-Algorithmus für das RNN. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente.

GPUs bringt bessere Ergebnisse, da bei einer hohen Anzahl die zur verfügung stehenden GPUs nicht vollständig ausgenutzt werden. Obwohl die Ausführungszeiten des Experten geschlagen werden konnten, ist ein großer Nachteil des „Scoring“-Algorithmus, dass dessen Kommunikationskosten um ein Vielfaches höher sind.

Das Fazit was sich für den „Scoring“-Algorithmus sehen lässt ist, dass man so wenig GPUs wie nötig nutzen sollte. Die Erhöhung von zwei auf drei GPUs brachte zwar einen Geschwindigkeitsgewinn von 7.1%, dafür wird aber eine GPU mehr genutzt, was die Nutzung von 50% mehr Ressourcen bedeutet.

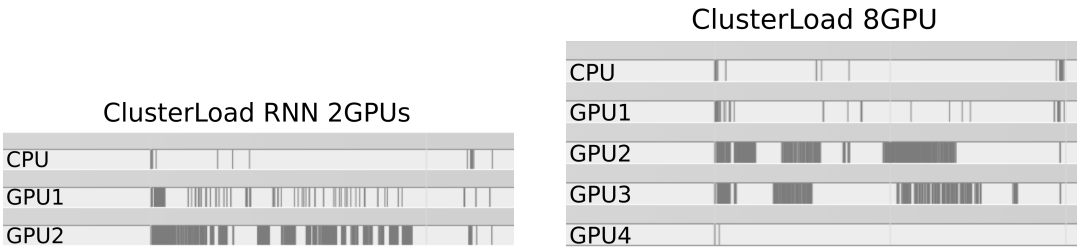
5.6 Evaluation Clustering

Beim Clustering wollen wir nutzen, dass Cluster innerhalb eines Clusters eine hohe Verbindungsdichte haben und eine niedrige Verbindungsdichte zu anderen Clustern. Dafür haben wir mit Hilfe von Karger’s Algorithmus ein Cluster Verfahren entwickelt. Für dieses haben wir drei verschiedene Versionen, welche wir evaluieren werden.

- Eine Version bei, welcher das beste Cluster über die Lastenverteilung zwischen den Clustern bestimmt wird.
- Eine zweite Version, bei welcher das beste Cluster über den Kommunikationsaufwand zwischen den Clustern bestimmt wird.
- Eine dritte Version, welche eine Abänderung der zweiten ist. Um eine bessere Lastenverteilung zu bekommen, haben wir bei dieser für die Cluster eine maximal Größe festgelegt.

5.6.1 Lastenverteilendes Clustering

Bei unserem ersten Ansatz haben wir das Cluster nach der Lastenverteilung ausgewählt. Im Verlauf der Evaluation werden wir den Algorithmus deswegen „ClusterLoad“ nennen. Die Lastenverteilung der Cluster bestimmen wir, indem wir das Verhältnis vom kleinsten zum größten Cluster betrachten.



(a) 2GPUs **(b)** 8GPUs

Abbildung 5.14: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „ClusterLoad“-Algorithmus für das RNN. Links zeigt die Auslastung für 2GPUs und rechts für 8GPUs.

RNN

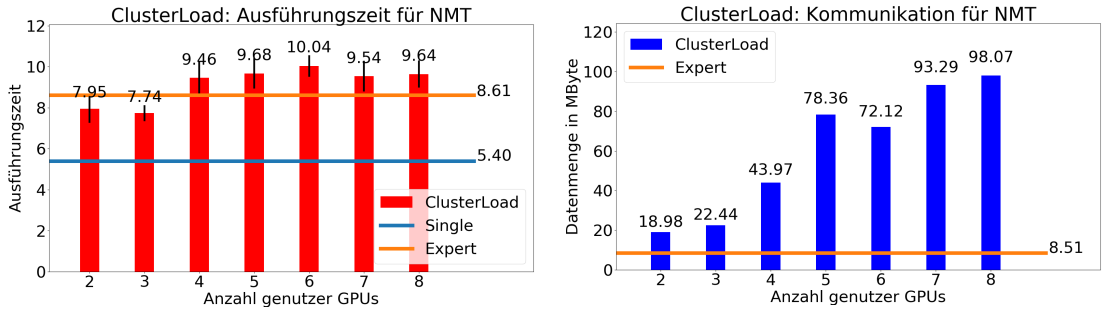
Abb. 5.13a zeigt die Ausführungszeiten des „ClusterLoad“-Algorithmus für das RNN Modell. Man kann bei 2GPUs die beste Ausführungszeit sehen. Bei dieser schafft es der Algorithmus auch den Experten zu schlagen. Dafür skaliert der Algorithmus sehr schlecht. Nur die Ausführung mit zwei GPUs hat zu einem guten Ergebnis geführt. Die Ausführungszeiten alle anderen GPU Anzahlen lassen kein wirkliches Muster oder eine Tendenz erkennen. Eine Beobachtung ist, dass der Algorithmus für die Ausführung mit zwei GPUs eine sehr niedrig Varianz hat. Alle anderen Ausführungen haben eine viel höhere Varianz.

Betrachten wir 5.13b sehen wir, dass in diesem Fall keine größere Korrelation von dem Kommunikationsaufwand zur Laufzeit existiert. Die einzige Tendenz die wieder zu erkennen ist, ist die dass eine höhere Anzahl an GPUs einen höheren Kommunikationsaufwand bedeutet.

Betrachtet man die Geräteauslastung für 2 und 8GPUs kann man eine Erklärung für die Ergebnisse finden. Für die Ausführung mit 2GPUs kann man nichts Besonderes erkennen. Es zeigt sich ein ähnliches Bild, wie zu den anderen Algorithmen. Betrachtet man aber die Geräteauslastung für die Ausführung mit 8GPUs, sieht man, dass nur vier GPUs überhaupt genutzt werden und von diesen sind nur zwei wirklich ausgelastet. Für alle anderen Anzahlen von GPUs bietet sich ein ähnliches Bild. Zwei der GPUs sind stark ausgelastet und die anderen GPUs sind kaum bis gar nicht ausgelastet. Daraus kann man erkennen, dass es in dem Graphen zwei Cluster gibt, welche der Algorithmus immer findet. Bei einer Ausführung mit zwei GPUs führt die Platzierung der beiden Cluster auf verschiedene GPUs zu einer sehr guten Laufzeit. Bei der Ausführung mit einer anderen Anzahl an GPUs findet der Algorithmus auch die beiden Cluster. Da aber noch Operationen auf die anderen GPUs platziert werden, kommt es zu einem erhöhten Kommunikationsaufwand ohne wirklichen Gewinn an Rechenpower, da die GPU kaum ausgelastet wird.

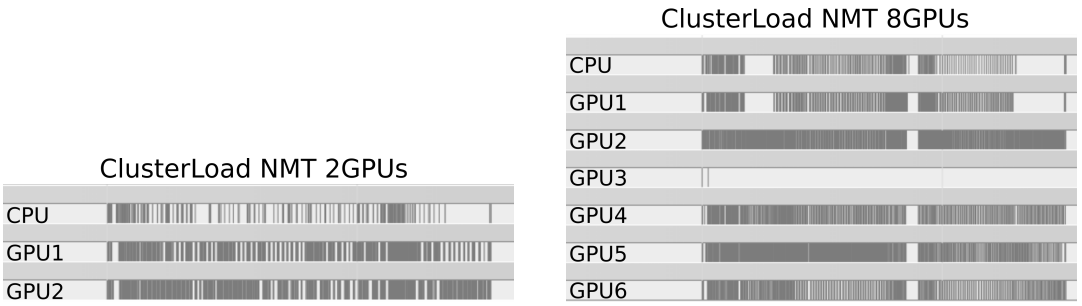
NMT

Abb. 5.15a zeigt die Ausführungszeiten für den „ClusterLoad“ -Algorithmus für die NMT. Bei diesen sieht man wieder die Tendenz, dass die Ausführungszeiten mit einer höheren Anzahl an GPUs stark zunimmt. Die Zunahme der Ausführungszeiten, lässt sich über den Kommunikationsaufwand



(a) Ausführungszeit ClusterLoad: NMT **(b)** Kommunikationsaufwand ClusterLoad: NMT

Abbildung 5.15: Evaluation des „ClusterLoad“-Algorithmus für die NMT. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente.



(a) 2GPUs **(b)** 8GPUs

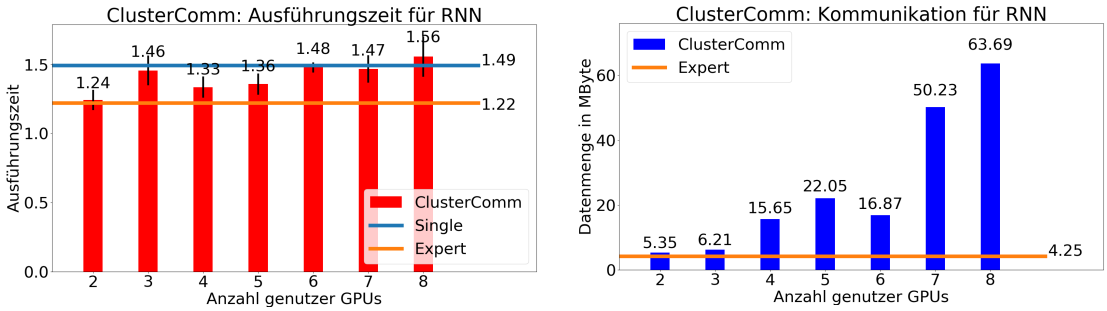
Abbildung 5.16: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „ClusterLoad“-Algorithmus für die NMT. Links zeigt die Auslastung für 2GPUs und rechts für 8GPUs.

erklären. Betrachtet man diesen in Abb. 5.15b, lässt sich eine Korrelation zwischen den beiden erkennen. Wie bei den anderen Algorithmen, ist der Kommunikationsaufwand um einiges höher als beim Experten.

Betrachtet man die Geräteauslastung für die Ausführung mit 2GPUs in Abb. 5.16a, sieht man wieder, dass die beiden GPUs gleichmäßig stark ausgelastet sind. Man sieht aber eine starke Fragmentierung der Auslastung, was bedeutet, dass die GPUs häufig Idle sind, da sie auf ein anderes Ergebnis warten. Ein interessanteres Bild zeigt die Ausführung mit 8GPUs in Abb. 5.16b. Bei diesem sieht man, dass überhaupt nur fünf der acht GPUs genutzt werden. Zwei der GPUs haben sogar eine so niedrige Auslastung, dass ihre Auslastung garnicht regestriert wurde.

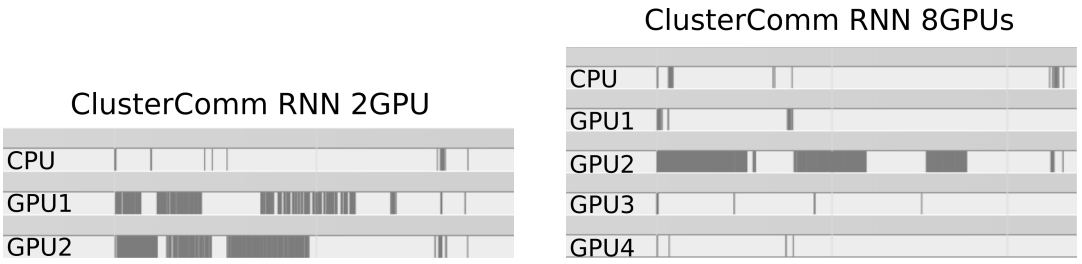
5.6.2 Kommunikationsvermeidentes Clustering

Ein anderer Ansatz für das Clustering war, das Cluster auszuwählen, welches den geringsten Kommunikationsaufwand hatte. Zur einfachen Benennung des Clusters nennen wir diese Variante des Algorithmus „ClusterComm“. Da so wiederum ein Clustering, bei welchem alle Operationen in einem Cluster sind, den geringsten Kommunikationsaufwand hätte, haben wir noch eine mindeste Lastenverteilung angegeben, welches das Clustering erfüllen muss. Wir haben festgelegt, dass



(a) Ausführungszeit ClusterComm: RNN **(b)** Kommunikationsaufwand ClusterComm: RNN

Abbildung 5.17: Evaluation des „ClusterComm“-Algorithmus für das RNN. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente.



(a) ClusterComm 2GPU's **(b)** ClusterComm 8GPU's

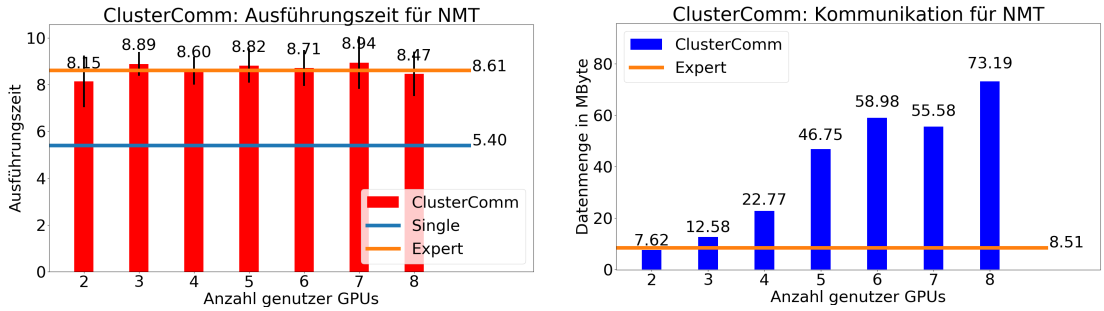
Abbildung 5.18: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „ClusterComm“-Algorithmus für das RNN. Links zeigt die Auslastung für 2GPUs und rechts für 8GPUs.

das Verhältnis vom kleinsten zum größten Cluster maximal $\frac{1}{10}$ sein darf. Dies hat aber nicht so funktioniert, wie wir es wollten. Da unser Algorithmus die Cluster zufällig erzeugt, gab es keine Garantie, das ein solches Clustering überhaupt gefunden wird. Und in der Praxis konnten wir, auch feststellen, dass der Algorithmus für eine höhere Anzahl von GPUs keine Cluster findet, bei welchem das Verhältnis vom größten zum kleinsten größer als $\frac{1}{10}$ ist.

Aus diesem Grund haben wir eine kleine Änderung an dem Algorithmus vorgenommen. Wir stoppen die Kantenkontraktion nicht mehr erst, wenn nur noch #GPU Knoten vorhanden sind, sondern der Algorithmus stoppt, wenn noch 100 Knoten im Graphen sind. Danach vereint er automatisch die kleinsten Knoten, bis nur noch #GPU Knoten vorhanden sind. Diese so erzeugten Knoten sind dann die gefundenen Cluster für den Durchlauf des Algorithmus. Von diesen Clustern, wählt der Algorithmus, dann das Cluster aus, welches den niedrigsten Kommunikationsaufwand hat.

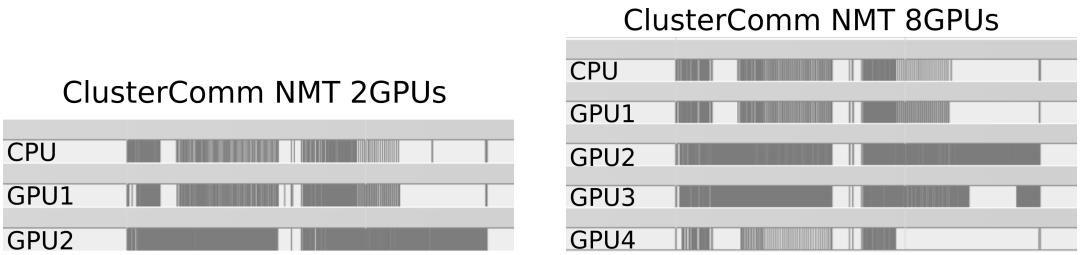
RNN

Abb. 5.17a zeigt die Ausführungszeiten für unseren „ClusterComm“-Algorithmus. Wie schon bei den anderen Algorithmen konnte mit einer Anzahl von 2 GPUs das beste Ergebnis erreicht werden. Das Interessante an diesem Graphen ist die Varianz für die Ausführung mit 2GPUs. Der „ClusterLoad“-Algorithmus hatte hier eine sehr niedrige Varianz, da er immer die gleichen zwei



(a) Ausführungszeit ClusterComm: NMT **(b)** Kommunikationsaufwand ClusterComm: NMT

Abbildung 5.19: Evaluation des „ClusterComm“-Algorithmus für die NMT. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente.



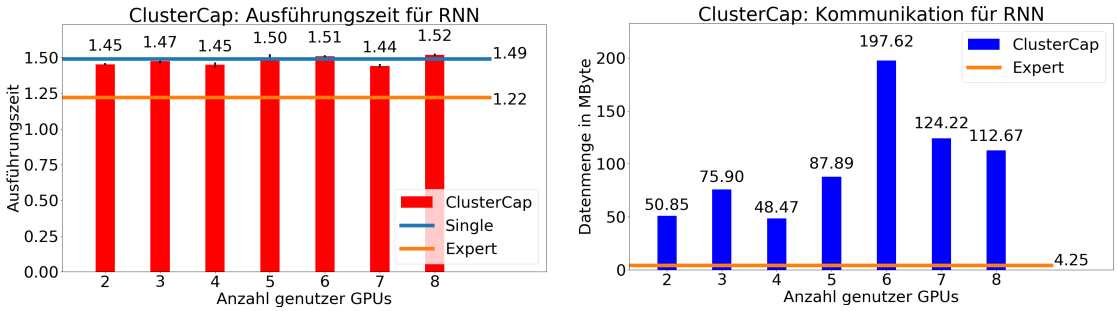
(a) ClusterComm 2GPU's **(b)** ClusterComm 8GPU's

Abbildung 5.20: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „ClusterComm“-Algorithmus für die NMT. Links zeigt die Auslastung für 2GPUs und rechts für 8GPUs.

Cluster für das RNN gefunden hat. Dadurch das der „ClusterComm“-Algorithmus, diese nicht immer findet, ist es wahrscheinlich, dass die beiden Cluster nicht den optimalen Kommunikationsaufwand haben.

In Abb. 5.17b kann man sehen, dass die Kommunikationskosten für eine niedrige Anzahl von GPUs im Vergleich zu unseren anderen vorgestellten Algorithmen sehr gering ist. Die niedrigen Kommunikationskosten des Experten konnten nicht erreicht werden, was eine Erklärung für die minimal höhere Ausfuhrzeit des Algorithmus im Vergleich zum Experten sein könnte. Da die Ausführungszeit und die Kommunikationskosten sehr ähnlich zum Experten sind, ist eine Vermutung, dass der Algorithmus, eine sehr ähnliche Verteilung zum Experten findet.

Betrachtet man die Ausführung für 2GPUs in Abb. 5.18a, sieht man nichts Besonderes. Es hat die gleichen Probleme, wie die anderen Algorithmen. Die GPUs sind zwar gleichmäßig ausgelastet, aber es gibt viel Idle-Zeit. Ein interessanteres Bild zeigt die Geräteauslatung für 8GPU's in Abb.5.18b. Sie zeigt das vorher angesprochene Problem, dass die meisten Operationen auf eine GPU gemappt werden um die Komunikations zu minimeren. Die anderen GPU's haben nur sehr wenige Operationen, womit die Nutzung von mehreren GPUs nichts bringt und sogar eher negativ ausfällt, da zusätzliche Kommunikationskosten entstehen. Unsere Anpassung des Algorithmus hat also nicht die gewünschte Wirkung gehabt, dass nicht alle Operationen auf einer GPU platziert werden.



(a) Ausführungszeit Capped Cluster: RNN **(b)** Kommunikationsaufwand Capped Cluster: RNN

Abbildung 5.21: Evaluation des „ClusterCap“-Algorithmus für das RNN. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente.

NMT

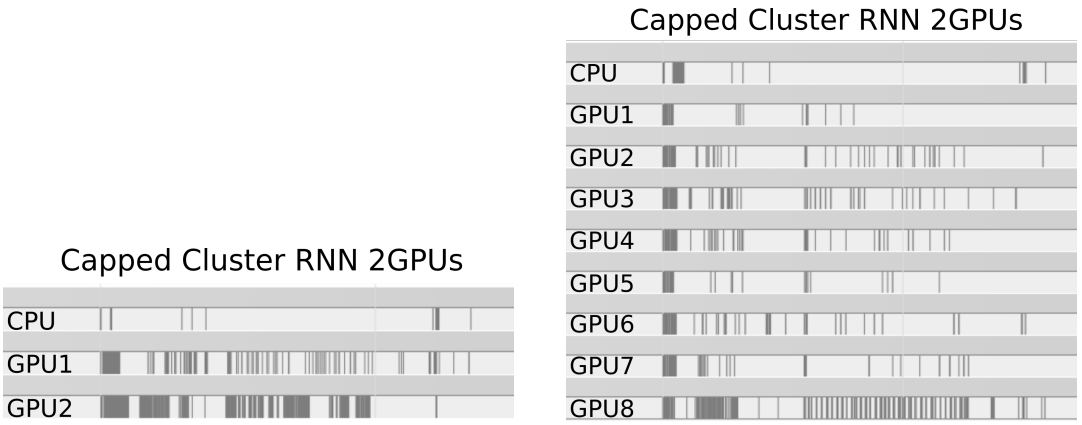
Abb. 5.19a zeigt die Ausführungszeiten, des „ClusterComm“-Algorithmus für die NMT. Es zeigt sich wieder, dass die Ausführung mit 2GPUs am besten ist. Man sieht auch, dass im Vergleich zu den anderen Algorithmen, der „ClusterComm“-Algorithmus eine vergleichsweise konstante Ausführungszeit, unabhängig von den GPUs hat.

Betrachtet man Abb.5.19b kann man sehen, dass der Kommunikationsaufwand für eine Ausführung mit 2GPUs geringer als die Expertenplatzierung ist. Auch bei der NMT hat der „ClusterComm“-Algorithmus eine ähnliche Ausführungszeit und einen ähnlichen Kommunikationsaufwand, wie der Experte. Dadurch lässt sich auch hier vermuten, dass er eine ähnliche Platzierung zum Experten findet.

In Abb. 5.20 sieht man die Geräteauslastung für die Ausführung mit 2 und 8GPUs. Bei diesen ist die Ausführung mit 8GPUs interessanter. Hier sehen wir wieder das Problem, dass der Algorithmus versucht die Kommunikation zu minimieren, ohne die Lastenverteilung zu beachten. Es werden nur die Hälfte der GPUs überhaupt genutzt und von denen sind nur zwei wirklich ausgelastet. Daran sieht man wieder, dass unsere Änderung, welche das verhindern sollte, nicht viel gebracht hat.

5.6.3 Begrenztes Clustering

Die Evaluation des „ClusterComm“-Algorithmus hat gezeigt, dass es dazu führt das möglichst viele Operationen auf möglichst wenig GPUs platziert werden. Um dies zu verbessern haben wir den Algorithmus angepasst, indem wir eine maximale Clustergröße eingeführt haben. Für die Evaluation des Algorithmus haben wir den Clustern erlaubt maximal 50% größer zu sein, als bei einem perfekt ausgewogenem Clustering. Den Algorithmus haben wir in der folgenden Evaluation „ClusterCap“ genannt.



(a) ClusterCap 2GPUs **(b)** ClusterCap 8GPUs

Abbildung 5.22: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „ClusterCap“-Algorithmus für das RNN. Links zeigt die Auslastung für 2GPUs und rechts für 8GPUs.

RNN

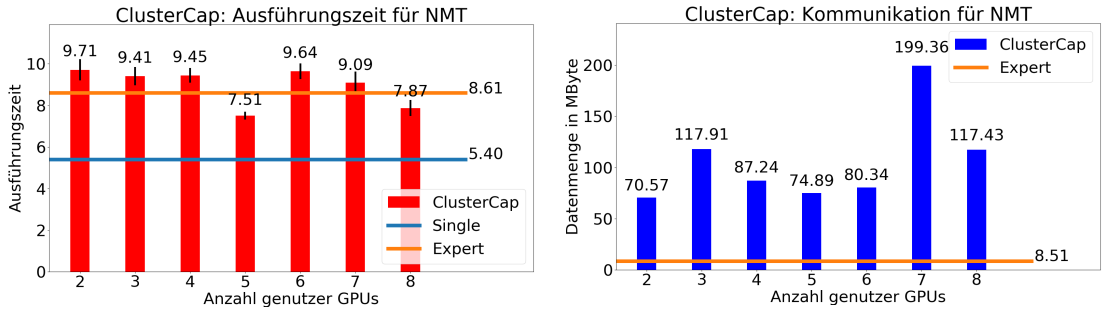
Betrachtet man die Ausführungszeiten des „ClusterCap“-Algorithmus in Abb. 5.21a sieht man, dass er unabhängig von der Anzahl der GPUs eine Ausführungszeit ähnlich zur Ausführung auf einer einzelnen GPU hat. Dies ist eine interessante Beobachtung. Bei den anderen Cluster Algorithmus konnte für 2GPUs eine Performanz ähnlich dem Experten gefunden werden. Das bedeutet, dass die Einschränkung der maximalen Größe der Cluster, es diesem Algorithmus verhindert oder stark erschwert, die gleiche Partitionierung zu finden.

Betrachten man die Kommunikationskosten in Abb. 5.21a, kann man vor allem für größeren Anzahlen von GPUs, sehr hohe Kommunikationskosten sehen. Der Grund dafür ist, dass der Algorithmus Cluster aufteilen muss, welche größer als die maximale Größe sind. Dadurch bilden sich mehr kleine Cluster, welche auch untereinander mehr vernetzt sind, was den Kommunikationsaufwand erhöht.

Betrachten man die Geräteauslastung für 2GPUs in Abb. 5.22a, kann man den Grund für die schlechte Performanz sehen. Die beiden GPUs sind zwar ähnlich stark ausgelastet, aber die Auslastung ist viel fragmentierter als bei den anderen Clusterverfahren. Daraus kann man schließen, dass die beiden GPUs häufig aufeinander warten mussten. Betrachten wir die Geräteauslastung für 8GPUs in Abb. 5.22b, sieht man ein ähnliches Problem. Die Verbesserung hat zwar ihren Zweck erfüllt, indem sie die einzelnen Operationen gleichmäßiger auf die verschiedenen GPUs verteilt hat, aber sie die GPUs sind wieder nur sehr fragmentiert ausgelastet, da sie viel auf die anderen GPUs warten müssen.

NMT

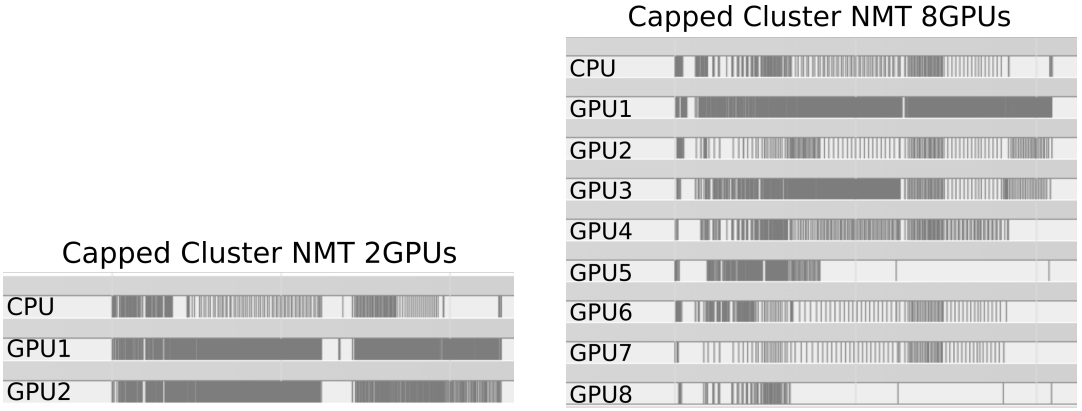
Betrachten wir die Ausführungszeiten des Algorithmus in Abb. 5.23a, sehen wir ähnliche Ergebnisse wie beim RNN Modell. Es gibt keine wirkliche Abhängigkeit zwischen der Anzahl der GPUs und der Ausführungszeit und die Ergebnisse sind schlechterr als bei den anderen Clusterverfahren. Eine Auffälligkeit ist, dass eine Ausführung für 5GPUs um fast zwei Sekunden schneller ist, als die



(a) Ausführungszeit ClusterCap: NMT

(b) Kommunikationsaufwand ClusterCap: NMT

Abbildung 5.23: Evaluation des „ClusterCap“-Algorithmus für die NMT. Links zeigt die Ausführungszeiten und rechts zeigt die Kommunikationskosten für die Experimente.



(a) Capped Cluster 2GPUs

(b) Capped Cluster 8GPUs

Abbildung 5.24: Die Auslastung der genutzen GPUs und CPUs bei der Ausführung der Experimente für den „ClusterCap“-Algorithmus für das RNN. Links zeigt die Auslastung für 2GPUs und rechts für 8GPUs.

meisten anderen GPU Anzahlen. Dies lässt sich nicht durch die Varianz erklären, da diese nicht sehr hoch ist. Eine Erklärung dafür könnte sein, dass das NMT eine Tendenz zu fünf Clustern hat. Dazu würde auch die Auslastung des „ClusterLoad“-Algorithmus passen, welche man in Abb. 5.16b sieht. Bei dieser werden fünf der acht GPUs ausgelastet, was bedeutet, dass der Algorithmus fünf Cluster gefunden hat.

Den Kommunikationsaufwand kann man in Abb. 5.23b seheh. Dieser ist, wie beim RNN auch, sehr hoch. Der Grund dafür ist vermutlich der gleiche. Der Algorithmus ist durch die maximal Größe der Cluster dazu gezwungen kleinere Cluster zu bilden, was zu einem höheren Kommunikationsaufwand führt.

Bei der Geräteauslastung zeigen sich keine Auffälligkeiten. Bei der Ausführung mit 2GPUs in Abb. 5.24a zeigt sich ein ähnliches Bild, wie für die anderen Algorithmen. Bei der Ausführung mit 8GPUs in Abb. 5.24b zeigt sich auch hier, dass der Algorithmus seinen eigentlichen Zweck erfüllt hat. Es gibt zwar immer noch zwei GPUs die etwas mehr ausgelastet sind als die anderen, aber jeder der GPUs hat einen Anteil an der Arbeit.

5.6.4 Clustering Ergebnis

Die Performanz des Clustering Algorithmus ist stark von dem Modell abhängig. Beim RNN lassen die Ergebnisse vermuten, dass sich das Modell in zwei Cluster aufteilen lässt. Da die Ausführungszeiten ungefähr mit dem Experten übereinstimmen, ist die Vermutung, dass die beiden Cluster den beiden Schichten des Modells entsprechen.

Für das NMT Modell hat der Cluster Algorithmus nicht so gut funktioniert, wie für das RNN Modell. Die Vermutung ist, dass das NMT-Modell sich nicht so leicht in Cluster unterteilen lässt, obwohl es auch zwei Schichten hat.

Am besten von den drei funktioniert der „ClusterLoad“-Algorithmus. Er hat zwar sehr ähnliche Ausführungszeiten, wie der „ClusterComm“-Algorithmus, dafür sind die Operationen aber besser auf den verschiedenen GPUs verteilt. Das ist eine wichtige Eigenschaft für einen Partitionierungsalgorithmus. Wenn die Verteilung des Graphen auf verschiedene GPUs stattfindet, weil der Graph zu groß für eine GPU ist, wäre es schlecht, wenn der Algorithmus, alle Operationen auf eine GPU platziert. Der „ClusterCap“-Algorithmus ist in diesem Fall zwar nochmal besser, er zeigt aber um einiges schlechtere Ausführungszeiten als der „ClusterLoad“-Algorithmus.

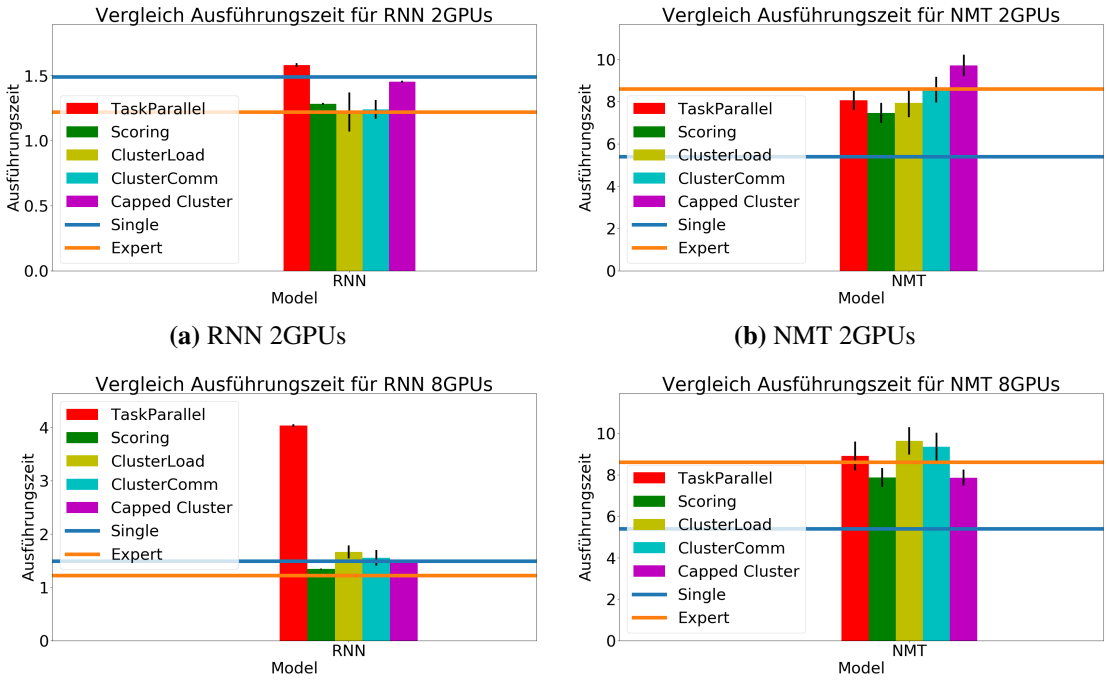
5.7 Gesamt Evaluation

In diesem Abschnitt Vergleichen wir die Ergebnisse aller vorgestellten Algorithmen. Dafür betrachten wir eine Ausführung mit 2GPUs und 8GPUs. Der Vergleich findet für 2GPUs statt, da die meisten Algorithmen für diese Anzahl die besten Ergebnisse hatten. Es gab zwar Ausnahmen, bei welchen zum Beispiel 3GPUs ein besseres Ergebnis gebracht haben, aber diese Performanzsteigerung, war nicht hoch genug, als dass sie die Nutzung von 50% mehr Rechenpower rechtfertigt. Den Vergleich für 8GPUs machen wir, um die Skalierbarkeit der Algorithmen zu betrachten. Da Modellparallelität nicht nur für die Performanz, sondern auch dann genutzt wird, wenn die Graphen zu groß sind um auf eine GPU zu passen, ist auch eine wichtige Eigenschaft der Algorithmen, dass sie gut skalieren und nicht mit zunehmender Anzahl von GPUs schlechter werden.

5.7.1 Performanz

Für die Performanz betrachten wir Abb. 5.25a und Abb. 5.25c, welche den Vergleich der Algorithmen für 2GPUs zeigen. Betrachten wir die Ergebnisse für das RNN, sehen wir, dass drei der Algorithmen ungefähr die Ausführungszeit des Experten erreichen. Der „ClusterLoad“-Algorithmus ist dabei der beste und schlägt eine Ausführung mit einer einzelnen GPU um 21% und den Experten um 4.1%. Der Nachteil des Algorithmus, ist die hohe Varianz, welche durch die nichtdeterministische Bestimmung der Cluster zustande kommt. Diese kann aber auf Kosten der Laufzeit verringert werden, indem die Anzahl der betrachteten Cluster erhöht wird.

Betrachten wir die Ergebnisse für das NMT Modell, sehen wir, dass sich eine Verteilung nicht lohnt. Der beste Algorithmus ist der „Scoring“-Algorithmus, welcher zwar um 12.4% besser als der Experte ist, aber immer noch 28.4% langsamer als eine Ausführung auf nur einer GPU. Weiter sehen wir, dass der „ClusterCap“-Algorithmus sehr schlecht ausfällt. Unsere Vermutung warum dieser so schlecht ausfällt ist, die Zusammenführung der kleinen Cluster, nachdem keine zu



(a) RNN 2GPUs

(b) NMT 2GPUs

(c) RNN 8GPUs

(d) NMT 8GPUs

Abbildung 5.25: Der Vergleich aller vorgestellten Algorithmen. Es werden die Ausführungen für 2 und 8 GPUs vergleicht. Die Ausführung für 2 GPUs ist zum Vergleich der Performanz da, während die Ausführungen von 8GPUs zum Vergleich der Skalierbarkeit genutzt wird.

vereinenden Kanten mehr gefunden werden. In diesem Fall vereinen wir immer die kleinsten Cluster, ohne auf die Kommunikation Rücksicht zu nehmen. Dadurch ist zwar die Lastenverteilung vom „ClusterCap“-Algorithmus viel besser als beim „ClusterComm“-Allgorithmus. Aber das kommt auf den Kosten der Ausführungszeit.

Insgesamt gibt es für die Performanz drei Erkenntnisse:

- 1. Falls es Cluster in dem Graphen gibt, funktioniert ein einfacher Clustering Algorithmus zur Verteilung der Operationen sehr gut.
- 2. Falls es keine Cluster gibt funktioniert der Scoring Algorithmus am besten. Da es möglich ist, den Parameter für jeden Graphen anzupassen ist dieser flexibler. Dies hat den Nachteil der extra Laufzeit für die Bestimmung der Parameter.
- 3. Weniger ist mehr. Die meisten Algorithmen hatten mit der niedrgsten Anzahl an GPUs die besten Ergebnisse. Deswegen ist häufig die Ausführung mit sowenig GPUs wie möglich am effizientesten.

5.7.2 Skalierbarkeit

Führ die Skalierbarkeit betrachten wir die Ausführungszeiten aller Algorithmen auf 8GPUs in Abb. 5.25b und 5.25d.

Betrachten wir das Ergebnis für das RNN Modell, sehen wir, dass der „TaskParallele“-Algorithmus sehr schlecht skaliert. Er braucht ca. 3 mal solange wie alle anderen Algorithmen. Am besten skaliert der „Scoring“-Algorithmus, welcher sich kaum verschlechtert im Gegensatz zur Ausführung auf 2GPUs. Dies ist aber trotzdem ein nicht so gutes Ergebnis. Der „Scoring“-Algorithmus nutzt die 8-fache Anzahl an GPUs und ist schlechter als mit nur 2 GPUs. Die drei Cluster Algorithmen skalieren noch schlechter. Sie sind bei einer Ausführung auf 8GPUs schlechter geworden als bei einer Ausführung auf einer GPU. Eine interessante Beobachtung hier ist aber, dass der „ClusterCap“-Algorithmus am besten von den dreien Skaliert. Ein Grund dafür könnte sein, dass durch die gezwungene verbesserte Verteilung der Cluster die GPUs besser ausgenutzt werden, wie man in der Evaluation des Algorithmus gesehen hat.

Betrachten wir das Ergenbnis für die NMT, sehen wir ein ähnliches Bild. Der „TaskParallele“-Algorithmus skaliert zwar besser als beim RNN, aber er hat trotzdem eine schlechtere Ausführungszeit, als der Ausführung mit 2GPUs. Der „Scoring“-Algorithmus verschlechtert sich nur gering im Gegensatz zur Ausführung mit 2GPUs. Die interessanteste Beobachtung ist wieder der „ClusterCap“-Algorithmus Dieser verbessert sich, im Vergleich zur Ausführung mit 2GPUs, sogar.

Das Ergebnis der Evaluation für die Skalierbarkeit ist, dass der „ClusterCap“-Algorithmus dafür der am besten geeignete Algorithmus ist. Der „Scoring“-Algorithmus hat zwar ein ähnliches verhalten, wenn man die Anzahl der GPUs erhöht, aber den Nachteil, dass die Operationen nicht so gut verteilt sind, wie beim „ClusterCap“-Algorithmus. Will man einen Graphen verteilen, weil er zu groß für eine GPU ist, ist es auch wichtig, dass die GPUs gleichmäßig ausgelastet sind. Ansonsten kann es leicht passieren, dass die Teilgraphen immer noch zu groß für eine einzelne GPU sind. Deswegen wäre in so einem Fall der „ClusterCap“-Algorithmus besser geeignet.

6 Verwandte Arbeiten

In diesem Kapitel werden andere Ansätze vorgestellt, welche sich mit dem Thema der Modellparallelität befassen oder mit dieser Verwand sind. Als Erstes gehen wir auf zwei Ansätze ein, welche sich direkt mit der Geräteplatzierung von TensorFlow beschäftigen.

Den ersten Ansatz den wir vorstellen ist im vorläufigen TensorFlow Paper [AAB+16] beschrieben. Bei diesem wird die Geräteplatzierung durch eine Approximation des Aufwandes der Operationen und einer Greedy-Verteilung umgesetzt.
Als Zweites beschreiben wir einen Ansatz von Mirhoseini et al. [MPL+17][MGP+18], welcher sich direkt mit der Knotenplatzierung in TensorFlow auseinander setzt. In ihrem Ansatz haben sie versucht die Platzierung der Knoten mittels Reinforcement Learning zu lernen.

Ein verwandtes Thema zu unserer Arbeit, ist die Graph Partitionierung. In diesem werden wir kurz verschiedene Algorithmen vorstellen. Als Erstes einen bewährten Algorithmus von Fiduccia und Mattheyses [FM88], dessen Idee ist, dass die Knoten die Partitionen wechseln, wenn dadurch ein Gewinn erzielt werden kann. Als Zweites werden zwei Streaming Graph Partitionierungs Algorithmen beschrieben. Einer der früheren Ansätze von Stanton und Kliot [SK12] und einen aktuelleren Ansatz von Mayer et al. [MMT+17]. Als letztes wird noch ein Multilevel Ansatz von Hendrickson und Leland [HL95] beschrieben.

6.1 TensorFlow Geräteplatzierung

In diesem Abschnitt werden zwei Ansätze vorgestellt die sich mit der Geräteplatzierung von TensorFlow beschäftigen. Der erste ist ein Greedy Algorithmus aus dem vorläufigen TensorFlow Paper, welcher nicht in den späteren Release übernommen wurde. Der zweite ist ein Ansatz bei welchem die Platzierung durch Reinforcement Learning gelernt wird.

6.1.1 Greedy Platzierung

Im vorläufigen Paper zu TensorFlow [AAB+16], wurde ein Algorithmus zur Platzierung der Knoten vorgestellt. Vor der Platzierung wird für jeden Knoten eine Ausführungsdauer approximiert oder aus vorherigen Ausführungen berechnet. Ähnlich zu unserem Task Parallel Algorithmus, startet der Algorithmus dann mit einer Queue, welche zuerst alle Quellknoten in Betracht zieht. Danach platziert er jeden Knoten aus der Queue nach und nach und fügt diejenigen Knoten in die Queue hinzu, von welchen alle Eingangsknoten schon platziert wurden.

Die Platzierung basiert dabei auf einen Greedy Ansatz. Für jede Operation, wird für jedes Gerät berechnet, wie lange es Dauern würde bis die platziere Operation ausgeführt wurde. Dabei wird die approximierte Ausführungszeit der Operation und die entstehenden Kommunikationskosten in die Berechnung miteinbezogen. Die Platzierung, welche die geringste Ausführungszeit hätte, wird gewählt und es wird mit der nächsten Operation aus der Queue fortgefahren.

Der Algorithmus ist in der veröffentlichten Version von TensorFlow nicht implementiert. Eine Begründung dafür war, dass sich in der Praxis gezeigt hatte, dass eine manuelle Platzierung sich als effizienter erwiesen hatte [Mur16]. Wir vermuten, dass dies daran liegen könnte, dass die Approximation der Kommunikation und Ausführungszeit nicht genau genug möglich ist, wie auch [MPL+17] in ihrem Paper erwähnen.

6.2 Platzierung mit Reinforcement Learning

Mirhoseini et al.[MPL+17] haben einen Ansatz entwickelt, bei welchem die Platzierung gelernt wird. In ihrem ersten Ansatz haben sie eine Seq-to-Seq model aufgebaut, welches die Platzierung von den Knoten auf die Geräte lernt. Als Reward haben sie $R(P) = \sqrt[r(P)]{2}$ genutzt, wobei P eine Platzierung und r(P) die Ausführungszeit ist. Da eine niedrigere Laufzeit ein besseres Ergebnis ist, haben sie den negativen Reward(Belohnung) genommen.

Bei der Platzierung der Knoten konnte es passieren, dass die Platzierung nicht ausführbar ist, z. B. wenn alle Knoten auf ein Gerät platziert wurde, welches nicht genug Speicher hatte. Wenn dies der Fall war, haben sie der Platzierung einen konstanten hohen Reward zugewiesen. Als policy haben sie eine Stochastische policy $\pi(P|G;\theta)$ genutzt, wessen Parameter θ mit einer Policy Gradienten Methode gelernt wurde.

Bei einer Policy Gradienten Methode, versucht man einen Gradienten zu bestimmen, mit wessen Hilfe man die policy verbessern kann. Den Gradienten bestimmt man, indem man den Algorithmus mehrmals für leicht variierte Parameter ausführt. Aus dem so gewonnen Reward und dem Delta der Parameter, kann der Gradient z. B. mittels Regression berechnet werden.

Der Ansatz von Mirhoseini et al. konnte die von ihnen gewählten Benchmarks schlagen und eine maximale Verbesserung von 23.5% für die NMT erreichen. Eine Schwäche des Ansatztes ist die hohe Lerndauer von über 20 Stunden um die Platzierung zu finden.

Azalia Mirhoseini et al. haben versucht diesen ursprünglichen Ansatz zu verbessern, indem sie die Platzierung in zwei Phasen aufgeteilt haben [MGP+18]. Die erste Phase lernt die Colocation der Knoten, während die zweite Phase lernt, wie die Colocationsgruppen zu platzieren sind. Dadurch wollten sie eine Schwäche, die sie in ihrem alten Ansatz sahen, was die benötigte vorherige Colocation von einem menschlichen Experten war, angehen.

Mit diesem Ansatz konnten sie eine maximale Verbesserung der Platzierung für ein zwei Schichtiges NMT von 60.6% erreichen. Bei einem 8-Schichtigen NMT, waren sie um 4.9% langsamer als eine Experten Platzierung. Wie schon bei ihrem ersten Ansatz ist eine Schwäche des Ansatzes die hohe Laufzeit des Algorithmus.

6.3 Graph Partitionierung

Die Verteilung eines TensorFlowgraphen auf verschiedene Geräte ist verwandt mit der Graph Partitionierung. Bei diesem versucht man einen Graphen unter Berücksichtigung von Constraints in optimale Partitionen aufzuteilen.

Im folgenden werden vier Ansätze von diesem beschrieben. Als Erstes einen bewährten Algorithmus von Fiduccia und Mattheyses [FM88]. Als Zweites werden zwei Streaming Graph Partitionierungs Algorithmen beschrieben. Als letztes wird noch ein Multilevel Ansatz von Hendrickson und Leland [HL95] beschrieben.

Fiduccia-Matteyses

Ein recht früher Ansatz der Graph Partitionierung kommt von Fiduccia und Mattheyses [FM88]. Ihre Problemstellung ist die Partitionierung von einem Graphen in zwei Teilmengen, welche minimal miteinander verbunden sind. Da diese Anforderung alleine zu einer Partitionierung führen würde, bei der eine Partition alle Knoten hat und die andere Partition keine, erlaubten sie noch eine vom Anwender definiertes Constraint z. B. eine Lastenverteilung.

Die Grundidee des Algorithmus ist das Bewegen von einem Knoten einer Partition in die andere Partition. Dabei werden die Knoten beachtet, welche den höchsten Gewinn haben. Der Gewinn ist als die mögliche Reduzierung der Kanten zwischen den Partitionen definiert.

Damit die vom Anwender definierten Constraints eingehalten werden, teilt sich der Algorithmus in zwei Phasen auf. In der ersten Phase werden zwei Partitionen erstellt, welche die Constraints erfüllen. Dafür werden nach und nach die Knoten mit dem höchsten Gewinn von der größeren Partition zur kleineren bewegt. Danach werden nur noch Knoten bewegt, wenn ihr Partitionswechsel die Constraints nicht brechen würde.

Die Schwächen dieses Ansatzes für die Geräte Platzierung von TensorFlow ist, dass es die Flussrichtung nicht beachtet. Somit könnte zwar ein Cut gefunden werden, welcher minimale Kommunikationskosten hat und die Constraints erfüllt, aber es wird nicht beachtet, dass die Graphen parallel ausführbar sein sollten. Im schlechtesten Fall hat man zwei Partitionen, welche nur sequentiell ausgeführt werden können. Eine weitere Schwäche ist, dass der Algorithmus die Kantengewichte und Knotengrößen braucht, welche nur sehr schwer abzuschätzen sind.

6.3.1 Streaming Graph Partitionierung und ADWISE

Eine bekannte Möglichkeit der Graph Partitionierung ist ein auf Streams basierter Ansatz. Ein erster Ansatz davon wurde von Stanton und Kliot [SK12] vorgestellt. Die Idee ist, dass die Knoten nach und nach in einem Stream platziert werden. Zur Berechnung der Platzierung stehen nur die Informationen der schon platzierten Knoten und die Kanten der zu platzierenden Knoten zur Verfügung.

In ihrem Paper haben sie zehn Heuristiken vorgestellt, mit welchen die Knoten platziert werden können. Drei davon haben mit einem Buffer gearbeitet, welcher es erlaubt ankommende Knoten zwischenzuspeichern und zu einem späteren Zeitpunkt zu platzieren. In ihrer Evaluation hat sich

der „Linear Deterministic Greedy“ als am besten herausgestellt. Für die Platzierung wird für jede mögliche Platzierung eines Knotens ein Wert berechnet. Dieser berechnet sich aus der Anzahl an Kanten, welche der Knoten mit dem auf dem Gerät platzierten Knoten teilt und der schon besetzten Kapazität des Gerätes. Umso mehr Knoten schon auf dem Gerät sind umso negativer fällt dieses Gewicht aus.

Der Algorithmus ist ähnlich zu unserem „Scoring“-Algorithmus. Er unterscheidet sich dahingehend, dass bei TensorFlow die Platzierung vor der Ausführung stattfinden muss. Das heißt, es ist nicht möglich die noch freie Kapazität des Gerätes zur Bewertung zu nutzen und eine solche Abschätzung wäre ungenau. Weitere Unterschiede sind, dass wir versucht haben durch den Uprank die Platzierung des kritischen Pfades zu verbessern und das wir zur Bewertung der Kommunikation nicht die Anzahl der Kanten, sondern eine aus der fringe berechnete Bewertung genutzt haben.

Ein weiterer auf den Stream Partitionierungen aufbauender Ansatz ist ADWSIE, welches von Mayer et al. [MMT+17] vorgestellt wurde. Die Idee bei ihrem Ansatz ist, dass durch die Betrachtung von mehreren Knoten gleichzeitig, eine bessere Entscheidung getroffen werden kann, wie die Knoten zu platzieren sind.

Dabei heben sie vier Grundkonzepte von ADWISE hervor.

- ADWISE nutzt keine feste Fenstergröße, sondern passt die Fenstergröße zur Laufzeit an. Falls gegebene Latenzpräferenzen gebrochen werden würden, wird die Fenstergröße reduziert, um die Berechnungszeit zu beschleunigen. Ist noch ein großer Spielraum vorhanden, bevor die Latenzpräferenzen gebrochen werden würden, wird die Fenstergröße erhöht, um die Qualität der Platzierung zu verbessern.
- Die Berechnungen der Score in einem Fenster werden nicht immer neu berechnet. Da gute Knoten höchstwahrscheinlich gut bleiben und schlechte Knoten schlecht, ist es nicht nötig alle Punktewerte zu aktualisieren, wenn ein Knoten platziert wurde und ein neuer ins Fenster kommt.
- Die Bewertung jedes Knotens berechnen sie aus drei Punktewerten. Einen Lastenverteilungswert, einen Grad-abhängigen Wert und einen Clustering Wert.
- Bei einer parallelen Ausführung von mehreren Partitionierern, kann es zu schlechten Ergebnissen kommen, wenn die Menge der gemeinsamen Partitionen groß ist. Um dem entgegenzuwirken, arbeitet jeder Partitionierer auf einer exklusiven Untermenge von Partitionen.

6.3.2 Multilevel Ansätze

Die Idee der Multilevel Ansätze ist den Graphen zu vereinfachen, indem Knoten zusammen gefasst werden und auf den so vereinfachten Graphen die Partitionierung auszuführen. Die so gefundene Partitionierung kann dann auf dem Ursprungsgraphen abgebildet werden.

Ein früher Ansatz in diesem Gebiet kommt von Hendrickson und Leland [HL95]. Ihr Algorithmus unterteilt sich in die oben genannten drei Schritte.

- **Zusammenfassen:** Als Erstes werden die Knoten des Graphen zusammengefasst. Es werden keine zufälligen Knoten genommen, sondern zuerst wird ein Maximales Matching gesucht. Dieses ist die größte Menge von Kanten, welche keine Knoten gemeinsam haben. Diese Knoten der Kanten werden dann zu einem Knoten zusammen gefasst. Dabei werden die Kanten von den alten Knoten für den neuen Knoten übernommen. Die Gewichte der Knoten summieren sich auf und die gewichte der Kanten bleiben gleich, außer ein Knoten hatte Kanten auf beide der zusammen gefassten Knoten. Dann werden die Kantengewichte für die neu entstandene Kante aufsummiert.
- **Partitionieren:** Sie sind nicht weiter auf den Partitionierungs Algorithmus eingegangen. Die einzige Anforderung an den Algorithmus ist, dass er gewichtete Kanten und Knoten verarbeiten kann. Der Grund dafür ist, dass selbst wenn der ursprüngliche Graph ungewichtet war, durch das Zusammenfassen der Knoten, diese automatisch gewichtet werden.
- **Generalisieren:** Im letzten Schritt muss die für den vereinfachten Graphen gefundene Partitionierung auf den Ursprungsgraphen angewendet werden. Dazu gehen sie die Zusammenfassung des Graphen einfach rückwärts. Das heißt sie spalten einen Knoten immer in die zwei Ursprungsknoten aus denen er entstanden ist. Um ein besseres Ergebnis zu erhalten, optimieren sie die Partitionierung nach jeder Spaltung, bevor sie wieder eine Stufe höher gehen. Dazu verwenden sie eine Abwandlung des Fiduccia-Matteyses Algorithmus [FM88].

7 Zusammenfassung und Ausblick

Durch die immer größere Präsenz der künstlichen Intelligenz und des maschinellen Lernens, werden Themen die sich mit der Optimierung dieser beschäftigen immer wichtiger. Dabei ist ein offenes Forschungsgebiet die Verteilung der Operationen auf verschiedene Geräte. Die Nutzung von bekannten Algorithmen des Graph Processing, bieten dabei häufig nicht zufrieden stellende Ergebnisse, da diese meist ein Kostenmodell des Graphen benötigen, welches nur sehr schwer und dann ungenau zu approximieren ist [MPL+17].

Um dieses Problem zu umgehen, haben wir drei Algorithmen untersucht, welche kein Kostenmodell des gesamten Graphen benötigen. Der erste davon ist der „TaskParallele“-Algorithmus, bei welchem die einzige Heuristik ist, dass die platzierten Knoten, parallel ausführbar sein sollen. Dieser lieferte aber nur schlechte Ergebnisse, da wir den Einfluss der Kommunikation auf die Performanz unterschätzt hatten. Bei einer Ausführung mit 8GPUs, war dieser fast 3 Mal so langsam wie die Ausführung auf nur einer GPU.

Aus dieser Erkenntnis haben sich zwei wünschenswerte Eigenschaften für unseren Algorithmus ergeben. Er soll erstens die Kommunikation minimieren und zweitens die Arbeitslast gleichmäßig verteilen. Um dies zu erreichen, ohne ein Kostenmodell des Graphen zu erzeugen, sind wir ähnlich zur Streaming Graph Partitionierung vorgegangen. Der Algorithmus betrachtet immer eine ausführbare frindge der Operationen und berechnet auf dieser eine Bewertung für die Kommunikation und die Lastenverteilung. Die Operationen der frindge werden, dann jeweils auf das Gerät platziert, welches für sie die beste Bewertung hatte. Um den Algorithmus flexibler für verschiedene Modelle zu machen, haben wir die Verrechnung der beiden Bewertungen über einen Parameter gesteuert. Damit ist es möglich die Lastenverteilung oder die Kommunikation höher zu gewichten. Der Algorithmus konnte für die untersuchten Modelle eine ähnliche Performanz, wie die Expertenplatzierung erreichen. Bei diesem zeigte sich, aber auch, dass er nicht gut skaliert und eine Nutzung von mehr GPUs keinen Mehrwert bringt.

Als letztes haben wir einen komplett anderen Ansatz untersucht. Bei diesem wollten wir die Eigenschaften eines guten Clustering‘s ausnutzen. Diese haben viele Verbindungen innerhalb eines Clusters und wenige Verbindungen mit Mitgliedern von anderen Clustern. Die Wahl der besten Cluster haben wir über zwei verschiedene Bewertungen gemacht. Dies waren wieder die Lastenverteilung und die Kommunikation. Bei der Bewertung mit einem Kommunikationswert hat sich das Problem ergeben, dass die meisten Operationen auf nur sehr wenige GPUs verteilt werden, da wir die Lastenverteilung nicht beachtet haben. Deswegen haben wir den Ansatz erweitert, indem wir die maximale Größe eines Cluster beschränkt haben. Die Cluster verfahren haben sich als gut und schlecht erwiesen. Auf einem Graphen auf dem wirkliche Cluster gefunden werden können, konnten sie die Performanz des Experten schlagen. Auf einem Graphen, welcher keine solche Struktur hat, haben die Clusterverfahren eine schlechtere Performanz als der „Scoring“ Ansatz gehabt.

Ein Bild was sich über alle Algorithmen gezeigt hat, ist, dass eine Nutzung von mehr GPUs keinen wirklichen Mehrwert bringt. Das Problem ist, dass bei stärkerer Gewichtung der Kommunikation die Lastenverteilung sehr schlecht war. Dadurch wurde die meiste Rechenpower der GPUs nicht genutzt. Im Gegensatz dazu war bei stärkerer Gewichtung der Lastenverteilung die Kommunikation sehr schlecht. Dadurch wurden zwar alle GPUs gleichmäßig genutzt, aber es entstand ein höherer Kommunikationsaufwand und viele Wartezeiten, welche die Laufzeit verschlechterten.

Eine weitere Beobachtung die sich aus dem Clusterverfahren ergeben hat, ist, dass es für die Graphen eine optimale Anzahl an GPUs geben könnte, ab der die Nutzung von mehr GPUs keinen Mehrwert mehr bringt. Bei dem RNN waren dies zwei GPUs und bei der NMT eine GPU. Eine mögliche Erweiterung des Clusterverfahrens könnte dazu genutzt werden die optimale Anzahl an GPUs für einen Graphen zu bestimmen.

Mit dem „Scoring“-Algorithmus, konnten wir einen Algorithmus finden, der zwar nicht skaliert, aber für eine geringe Anzahl an GPUs automatisch eine ähnliche gute Platzierung, wie eine Expertenplatzierung findet.

Literaturverzeichnis

[18] *Recurrent Neural Networks*. 2018. URL: https://www.tensorflow.org/tutorials/recurrent (zitiert auf S. 44).

[AAB+16] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin et al. „Tensorflow: Large-scale machine learning on heterogeneous distributed systems“. In: *arXiv preprint arXiv:1603.04467* (2016) (zitiert auf S. 65).

[ABC+16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al. „TensorFlow: A System for Large-Scale Machine Learning.“ In: *OSDI*. Bd. 16. 2016, S. 265–283 (zitiert auf S. 17–19, 22).

[Atk70] A. B. Atkinson. „On the measurement of inequality“. In: *Journal of economic theory* 2.3 (1970), S. 244–263 (zitiert auf S. 27).

[BKK+98] J. P. Bradford, C. Kunz, R. Kohavi, C. Brunk, C. E. Brodley. „Pruning decision trees with misclassification costs“. In: *European Conference on Machine Learning*. Springer. 1998, S. 131–136 (zitiert auf S. 26, 27).

[Bre18] A. M. BretKinsella. *SMART SPEAKER CONSUMER ADOPTION REPORT*. Markt analysis. Voicebot, 2018 (zitiert auf S. 15).

[Col] S. Collet. *Object Detectio*. URL: https://www.saagie.com/fr/blog/object-detection-part1 (zitiert auf S. 30).

[Cor17] V. S. Cornelia Gewiehs. *Maschinelles Lernen braucht Big Data*. https://www.maschinenmarkt.vogel.de/maschinelles-lernen-braucht-big-data-a-640068/. Accessed: 10.5.2018. 2017 (zitiert auf S. 15).

[DCM+12] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le et al. „Large scale distributed deep networks“. In: *Advances in neural information processing systems*. 2012, S. 1223–1231 (zitiert auf S. 17).

[dpa16] dpa. „Go-Duell Mensch vs. Software Technisches K.o.“ In: *Spiegel Online* (2016) (zitiert auf S. 15).

[FM88] C. M. Fiduccia, R. M. Mattheyses. „A linear-time heuristic for improving network partitions“. In: *Papers on Twenty-five years of electronic design automation*. ACM. 1988, S. 241–247 (zitiert auf S. 65, 67, 69).

[GBC16] I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016 (zitiert auf S. 25, 28, 29).

[GE03] I. Guyon, A. Elisseeff. „An introduction to variable and feature selection“. In: *Journal of machine learning research* 3.Mar (2003), S. 1157–1182 (zitiert auf S. 29).

[Gur14] K. Gurney. *An introduction to neural networks*. CRC press, 2014 (zitiert auf S. 15).

[HL95] B. Hendrickson, R. W. Leland. „A Multi-Level Algorithm For Partitioning Graphs.“ In: *SC* 95.28 (1995), S. 1–14 (zitiert auf S. 65, 67, 68).

[HSM+00] R. H. Hahnloser, R. Sarpeshkar, M. A. Mahowald, R. J. Douglas, H. S. Seung. „Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit“. In: *Nature* 405.6789 (2000), S. 947 (zitiert auf S. 29).

[JMF99] A. K. Jain, M. N. Murty, P. J. Flynn. „Data clustering: a review“. In: *ACM computing surveys (CSUR)* 31.3 (1999), S. 264–323 (zitiert auf S. 28).

[Kar93] D. R. Karger. „Global Min-cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm.“ In: *SODA*. Bd. 93. 1993, S. 21–30 (zitiert auf S. 40).

[LBH15] Y. LeCun, Y. Bengio, G. Hinton. „Deep learning“. In: *nature* 521.7553 (2015), S. 436 (zitiert auf S. 29).

[LBZ17] M. Luong, E. Brevdo, R. Zhao. „Neural Machine Translation (seq2seq) Tutorial“. In: *https://github.com/tensorflow/nmt* (2017) (zitiert auf S. 43).

[MGP+18] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, J. Dean. „Hierarchical Planning for Device Placement“. In: 2018 (zitiert auf S. 65, 66).

[MML17] R. Mayer, C. Mayer, L. Laich. „The tensorflow partitioning and scheduling problem: it's the critical path!“ In: *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*. ACM. 2017, S. 1–6 (zitiert auf S. 34).

[MMT+17] C. Mayer, R. Mayer, M. A. Tariq, H. Geppert, L. Laich, L. Rieger, K. Rothermel. „ADWISE: Adaptive Window-based Streaming Edge Partitioning for High-Speed Graph Processing“. In: *arXiv preprint arXiv:1712.08367* (2017) (zitiert auf S. 65, 68).

[MPL+17] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, J. Dean. „Device placement optimization with reinforcement learning“. In: *arXiv preprint arXiv:1706.04972* (2017) (zitiert auf S. 65, 66, 71).

[Mur16] D. Murray. *TensorFlow placement algorithm*. 2016 (zitiert auf S. 66).

[Ola15] C. Olah. *Understanding LSTM Networks*. 2015 (zitiert auf S. 44).

[Qui86] J. R. Quinlan. „Induction of decision trees“. In: *Machine learning* 1.1 (1986), S. 81–106 (zitiert auf S. 26).

[RTL09] P. Refaeilzadeh, L. Tang, H. Liu. „Cross-validation“. In: *Encyclopedia of database systems*. Springer, 2009, S. 532–538 (zitiert auf S. 26).

[SHK+14] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov. „Dropout: A simple way to prevent neural networks from overfitting“. In: *The Journal of Machine Learning Research* 15.1 (2014), S. 1929–1958 (zitiert auf S. 26).

[SK12] I. Stanton, G. Kliot. „Streaming graph partitioning for large distributed graphs“. In: *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2012, S. 1222–1230 (zitiert auf S. 65, 67).

[SSSG17] C. Sun, A. Shrivastava, S. Singh, A. Gupta. „Revisiting unreasonable effectiveness of data in deep learning era“. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. IEEE. 2017, S. 843–852 (zitiert auf S. 15).

[VMR] R. Vargas, A. Mosavi, L. Ruiz. „DEEP LEARNING: A REVIEW“. In: () (zitiert auf S. 15).

[ZSV14] W. Zaremba, I. Sutskever, O. Vinyals. „Recurrent neural network regularization“. In: *arXiv preprint arXiv:1409.2329* (2014) (zitiert auf S. 44).

Alle URLs wurden zuletzt am 11. 05. 2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift