Leibniz Universität Hannover

Institut für Theoretische Informatik

Implementierungen von parallelen Algorithmen auf GPGPUs

Masterarbeit

Stefan Dulle
Matrikel-Nr. 2841990

29. September 2016

Erstprüfer: Prof. Dr. Heribert Vollmer
Zweitprüfer: Dr. Arne Meier
Betreuer: M. Sc. Martin Lück

Zusammenfassung

Parallele Architekturen werden in Zeiten stagnierender sequentieller Prozessorleistungen immer wichtiger. In dieser Arbeit wird daher die Grafikkarte als Plattform für die Implementierung paralleler Algorithmen untersucht. Die theoretische Grundlage dafür bildet das PRAM-Modell, für das eine alternative Form vorgestellt wird. Mit dieser wird dann eine allgemeine Implementierungsstrategie von PRAM-Algorithmen auf der Grafikkarten-Architektur entwickelt, sowie einige Optimierungsmöglichkeiten diskutiert. Konkret wird diese Strategie anhand mehrerer Algorithmen durchgeführt. Als Implementierungsplattform wurde OpenCL verwendet.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der immer weiter steigende Bedarf an Rechenleistung stößt bei der bisherigen sequentiellen Prozessorarchitektur langsam an die physikalisch machbaren Grenzen. Die harten Beschränkungen bezüglich Signallaufzeiten und Transistorgrößen stellen ein Problem für ein weiterhin exponentielles Wachstum der Prozessorleistung dar, wie es ja in dem bekannten Moore'schen Gesetz prognostiziert wird. Eine völlig neuen Rechnerarchitektur (z.B. Quantencomputer) ausgenommen, sind für die nahe Zukunft parallele Architekturen die erfolgversprechendste Lösung, um den steigenden Leistungsanforderungen gerecht zu werden.

Ein Parallelrechner besitzt dazu die Fähigkeit, mehrere Prozessoren zu verwenden um Daten parallel zu verarbeiten. Somit kann, wenn die Arbeit bestmöglich auf die Prozessoren verteilt wird, die Gesamtlaufzeit linear zu der Prozessoranzahl verringert werden (im Vergleich zur sequentiellen Laufzeit). Eine Steigerung der Rechenleistung des Systems kann somit, in Grenzen, durch das Hinzufügen neuer Prozessoren erreicht werden. Das ist im Vergleich zur Verbesserung der Leistung eines einzelnen Prozessors häufig eine leichtere Aufgabe.

In der Forschung und in großen Datenverarbeitungscenters sind parallele Algorithmen schon lange unabdingbar. Gerade im *Big-Data*-Bereich können die anfallenden Datenmengen häufig nur durch parallele Architekturen verarbeitet werden. Wofür bisher jedoch spezielle Rechencluster oder Supercomputer benötigt wurden, ist in den letzten Jahre eine kostengünstige und leistungsstarke Alternative aufgetaucht: Die Grafikkarte, im Englischen: *Graphical Processing Unit (GPU)*. Eine in fast jedem modernen Computer vorhandene Hardwareeinheit, deren ursprüngliche Aufgabe die Darstellung von 3D-Grafiken und das Verwalten des Videospeichers ist. Da viele Aufgaben der GPU, wie zum Beispiel die Transformation von Koordinaten oder die Berechnung von Pixelwerten, unabhängig voneinander ausgeführt werden können, spezialisierte sie sich auf eine massiv parallele Architektur. Mit der Zeit entwickelten sich GPUs auch zunehmend weg von der Beschränkung auf den Grafikanwendungsbereich und hin zu einer allgemeinen Rechnerstruktur, welche

es erlaubt, GPUs heutzutage für eine Vielzahl von Aufgaben zu benutzten. Diese Verwendung der GPU für allgemeine Berechnungen wird auch als *General Purpose Computation on Graphical Processing Units (GPGPU)* bezeichnet.

Der Entwurf solcher GPGPU-Algorithmen ist ein intensiver Forschungsbereich, bei dem sich die Anwendungen von der Linearen Algebra mit Matrix- und Vektormultiplikation [YWO15] über Finanzsimulationen mittels Monte-Carlo [BSS10] bis hin zur Bildbearbeitung [ZCW10], Hashumkehrung [WLT11] und Sortierung [Ye+10] erstrecken. Da jedoch in der theoretischen Informatik parallele Algorithmen bereits schon lange untersucht worden sind und es daher eine Vielzahl an parallelen Modellen und Algorithmen gibt, stellt sich die Frage inwieweit diese mit der GPU-Architektur kompatibel sind.

In dieser Arbeit wird diese Frage aufgegriffen. Dazu wird gezielt das PRAM-Modell untersucht, das in Kapitel 2 beschrieben wird. Zusätzlich wird hierfür eine alternative Form vorgestellt, die sich später als einfacher für die Implementierung herausstellt. Kapitel 3 gibt dann einen kurzen Einblick in die GPU-Architektur und stellt OpenCL als verwendete GPGPU-Plattform vor. In Kapitel 4 werden dann diese beiden Bereiche verbunden, indem eine generelle Implementierungsstrategie für PRAM-Algorithmen mittels OpenCL entwickelt wird. Konkret werden diese Techniken dann in Kapitel 5 angewendet. Hier werden für bekannte sowie auch einige in dieser Arbeit entworfene, parallele Algorithmen GPU-Implementierungen vorgestellt. Umgesetzt wird dies in einer Softwareanwendung, die es dem Anwender über eine grafische Oberfläche ermöglicht, die Algorithmen auszuführen. Abschließend wird in Kapitel 6 ein Fazit gezogen, inwieweit GPUs für die Implementierung von PRAM-Algorithmen geeignet sind und welche Hürden dabei auftreten.

Kapitel 2

Das PRAM-Modell

2.1 Definition

Eine *Parallel Random Access Machine (PRAM)* ist ein hypothetischer Parallelrechner, der zur Beschreibung und Analyse von parallelen Algorithmen verwendet wird. Sie besteht aus n Prozessoren, die die Instruktionen eines Algorithmus parallel abarbeiten. Die Prozessoren arbeiten dabei im Gleichschritt, sodass nach jeder Anweisung eine Synchronisierung aller Prozessoren stattfindet. Mit einer Anweisung wird daher nur begonnen, wenn jeder Prozessor die vorherige Anweisung vollendet hat.

Innerhalb des Algorithmus können Variablen verwendet werden. In dieser Arbeit beschränken sich die erlaubten Datentypen auf rationale Zahlen und Felder. Als Konvention werden für die Variablennamen von Zahlen Kleinbuchstaben, und für die von Feldern Großbuchstaben vergeben. Eine *Speicherzelle* bezeichnet entweder eine Zahlenvariable oder ein Element eines Feldes und stellt die kleinste Speichereinheit einer PRAM dar. Variablen lassen sich in *globale* und *private* Variablen unterteilen. Eine globale Variable ist für alle Prozessoren sichtbar und hat einen einzigen globalen Wert. Private Variablen sind dagegen nur für den einzelnen Prozessor sichtbar und können je nach Prozessor unterschiedliche Werte haben.

Die in einem Algorithmus verwendeten Variablen werden am Anfang inklusive ihrer Sichtbarkeit deklariert. Für Felder wird zusätzlich die Größe (Anzahl an Elementen) festgelegt. Zusätzlich existiert noch die spezielle globale Variable n und die private Variable p, die in jedem Algorithmus vorhanden sind und nicht explizit deklariert werden müssen. Der Wert von n ist gleich der Gesamtanzahl von Prozessoren in der PRAM, wobei p die Prozessornummer (auch Prozessor-ID genannt) des jeweiligen Prozessors ist.

Algorithmus 1 [Mei15, Alg. 2.1] zeigt einen simplen Algorithmus, der die Summe eines Feldes A[1] berechnet und in $A[0]$ abspeichert.

Algorithmus 1 Summieren eines Feldes mit einer PRAM

```
1: global A[2n]
2: private k
3: k ← 2n
4: while k > 1 do
5:     if p < k then
6:         A[p] ← A[2p] + A[2p + 1]
7:     k ← k/2
```

Weiterhin werden mit $\alpha(p)$ und $\beta(p)$ beliebige, von der Prozessor-ID abhängige, Turing-berechenbare Ausdrücke beschrieben. Ein solcher Ausdruck berechnet einen numerischen Wert und hat keine Seiteneffekte (d.h. es finden keine Variablenzuweisungen statt). Die Menge der in einem Ausdruck verwendeten Variablen wird durch die Funktion Ψ beschrieben und wird *Abhängigkeiten* des Ausdrucks genannt. Für den Ausdruck $\alpha(p) = A[2p] + A[s]$ gilt beispielsweise $\Psi(\alpha) = \{A, s\}$ (p, n sind nicht Teil der Abhängigkeiten).

Meta-Programm

Um das Arbeits- und Synchronisationsverhalten einer PRAM besser definieren zu können, wird in dieser Arbeit eine Transformation von PRAM-Algorithmen in ein äquivalentes *Meta-Programm* durchgeführt. Grundlage des Meta-Programms ist die Klassifikation der Prozessoren zu jedem Programmpunkt als *aktiv* oder *inaktiv*. Ein Prozessor ist zu einem Programmpunkt aktiv, falls dieser Prozessor alle notwendigen Bedingungen von Verzweigungen und Schleifen erfüllt, damit der Programmpunkt von dem Prozessor erreicht werden kann.

Wie auch in Abbildung 2.1 zu sehen, findet eine Änderungen der Menge von aktiven Prozessoren nur bei einer Änderung der Programmtiefe statt (d.h. beim Eintritt oder Verlassen von Schleifen und Verzweigungen). Da eine einfache Zuweisung keine Auswirkung auf den Programmfluss hat, können Prozessoren auch nicht ihren Aktivitätsstand ändern. Bei Schleifen gibt es die Ausnahme, dass bei gleicher Programmtiefe nach jedem Schleifendurchlauf die Menge der aktiven Prozessoren kleiner werden kann, wenn Prozessoren die Schleifenbedingung nicht mehr erfüllen. Der Fall, dass ein Prozessor einen inaktiven Prozessor wieder aktiviert, indem er dessen Schleifenbedingung wieder erfüllt, wird dabei nicht erlaubt, sodass dieser weiterhin inaktiv bleibt.

[1] Die Länge des Feldes muss einer Potenz von 2 entsprechen.

```
1: {Alle Prozessoren aktiv}
2: if p < 3 then
3:     {Prozessoren 0, 1, 2 aktiv}
4:     if p = 1 then
5:         {Nur Prozessor 1 aktiv}
6:     {Prozessoren 0, 1, 2 aktiv}
7:     for i ← 0 to p do
8:         {Bei i = 0: Prozessoren 0, 1, 2 aktiv}
9:         {Bei i = 1: Prozessoren 1, 2 aktiv}
10:        {Bei i = 2: Prozessoren 2 aktiv}
11:        {Bei i ≥ 3: Kein Prozessor aktiv}
```

Abbildung 2.1: Aktive Prozessoren in einem PRAM-Algorithmus ($n \geq 3$)

Es gilt somit, dass die Prozessoraktivität hierarchisch bezüglich der Blockstruktur des Programms ist. Die in einer Programmtiefe aktiven Prozessoren sind auch in der darüberliegenden Programmtiefe aktiv. Somit folgt auch die wichtige Erkenntnis, dass inaktive Prozessoren durch Eintritt in eine tiefere Programmebene nicht wieder aktiv werden können.

Insgesamt erlauben diese Eigenschaften nun die Betrachtung der aktiven Prozessoren als Stack von Prozessormengen: $T_0 T_1 \dots T_{d-1}$. Beim Eintritt in einen Programmblock werden die dort aktiven Prozessoren auf den Stack gelegt und beim Verlassen wird die oberste Menge wieder entfernt. Die derzeit aktiven Prozessoren sind daher stets auf der obersten Menge des Stacks zu finden. Da zu Beginn eines Algorithmus alle Prozessoren aktiv sind, ergibt sich die unterste Menge T_0 als $\{0, \dots, n-1\}$. Die maximale Größe des Stacks d ist gleich der maximalen Programmtiefe des Algorithmus und kann daher als statische Konstante gesehen werden.

Das Meta-Programm verwaltet nun diesen Stack und führt die Variablenzuweisungen aller aktiven Prozessoren T_k gleichzeitig aus. Im Algorithmus wird dies durch den Allquantor $\forall$ ausgedrückt. Zur Verwaltung des Stacks wird in k die aktuelle Programmtiefe gespeichert. Ein entscheidender Unterschied zum PRAM-Programm ist, dass die Anweisungen des Meta-Programms nicht mehr parallel, sondern sequentiell ausgeführt werden. Die Laufzeit bleibt jedoch identisch zu der einer PRAM, wenn man erlaubt, dass die Laufzeiten der Zuweisungen durch $\forall$-Quantoren unabhängig von der Größe der Menge ist, auf die sie angewendet werden.

Das Meta-Programm für Algorithmus 1 ist in Algorithmus 2 zu finden. Eine Beobachtung dabei ist, dass private Variablen nun als n-dimensionale Vektoren gesehen werden, bei dem x_i der Wert der Variable x für Prozessor i ist.

Algorithmus 2 Summieren eines Feldes als Meta-Programm

```
1:  global A[2n]
2:  private k
3:  T_0 ← {0, ..., n − 1}
4:  k_p ← 2n ∀p ∈ T_0
5:  T_1 ← {p ∈ T_0 | k_p > 1}
6:  while T_1 ≠ ∅ do
7:      T_2 ← {p ∈ T_1 | p < k_p}
8:      A[p] ← A[2p] + A[2p + 1] ∀p ∈ T_2
9:      k_p ← k_p/2 ∀p ∈ T_1
10:     T_1 ← {p ∈ T_1 | k_p > 1}
```

Im nächsten Abschnitt werden die in einem PRAM-Algorithmus erlaubten Anweisungen beschrieben und erläutert, wie sie sich ins Meta-Programm übersetzten lassen.

2.2 Meta-Programm-Transformation

Zuweisungen

Die Zuweisung einer numerischen Variable enthält auf der linken Seite die Variable und auf der rechten Seite einen Ausdruck $\alpha(p)$, der den neuen Wert der Variable berechnet. Das äquivalente Meta-Programm führt die Zuweisung für alle derzeit aktiven Prozessoren aus. Ein inaktiver Prozessor überspringt daher jegliche Zuweisungen.

$x \leftarrow \alpha(p)$	$x_p \leftarrow \alpha(p)\ \forall p \in T_k$

Die Zuweisung einer Feld-Variable ist relativ ähnlich. Hier werden jedoch zwei Ausdrücke verwendet: $\alpha(p)$ für den Index des Feldes und $\beta(p)$ für den Wert der geschrieben wird.

$A[\alpha(p)] \leftarrow \beta(p)$	$A[\alpha(p)] \leftarrow \beta(p)\ \forall p \in T_k$

Generell gilt für jede Zuweisung, dass zunächst jeder Prozessor die Ausdrücke $\alpha(p)$ und $\beta(p)$ auswertet und *danach* die Variable beschreibt. So werden Speicherkonflikte, wie zum Beispiel bei $A[p] \leftarrow A[2p]$ vermieden. Hier würden daher alle Prozessoren zunächst den Wert von $A[2p]$ bestimmen und danach diesen in $A[p]$ eintragen. Wäre dies nicht der Fall, so könnte es sein, dass ein Prozessor den bereits geschriebenen Wert eines anderen Prozessors verwendet.

Verzweigungen

Ein Prozessor führt die Anweisungen innerhalb eines **if**-Blocks nur aus, wenn er den Ausdruck $\alpha(p)$ zu wahr auswertet. Prozessoren, die die Bedingung nicht erfüllen, sind daher für die Dauer des Blocks inaktiv.

Das zugehörige Meta-Programm (unten zu sehen) erhöht zunächst die Programmtiefe k, die nach dem Verlassen des Blocks wieder verringert wird. Dadurch wird das Hinzufügen und Entfernen der Prozessormenge auf dem Stack realisiert. Die innerhalb der Verzweigung aktiven Prozessoren T_k ergeben sich dann aus den aktiven Prozessoren der tieferen Ebene T_{k-1}, die $\alpha(p)$ erfüllen. Da k erhöht wurde, ist für alle Anweisungen innerhalb des Blocks dies die neue Menge an aktiven Prozessoren.

if $\alpha(p)$ **then**	$k \leftarrow k + 1$
... Anweisungen ...	$T_k \leftarrow \{p \in T_{k-1} \mid \alpha(p) = true\}$
	... Anweisungen ...
	$k \leftarrow k - 1$

Schleifen

Ein Prozessor führt die Anweisungen innerhalb eines **while**-Blocks wiederholt solange aus, bis er $\alpha(p)$ zu falsch auswertet. Prozessoren, die die Schleife frühzeitig beenden, bleiben dann inaktiv, bis mit der nächsten Anweisung begonnen wird.

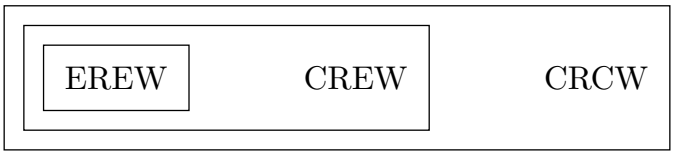
Das Meta-Programm ist zunächst relativ ähnlich zu dem einer Verzweigung, indem die Programmtiefe beim Eintritt erhöht und nach Verlassen wieder verringert wird. Auch bilden alle aktiven Prozessoren, die die erste Schleifenbedingung erfüllen, die neue Menge aktiver Prozessoren. Nach jedem Schleifendurchlauf werden aus dieser die dann nicht mehr aktiven Prozessoren entfernt (welche die Schleifenbedingung nicht mehr erfüllen).

Offen ist jedoch die Frage, wie oft im Meta-Programm die Schleife ausgeführt werden soll. Theoretisch kann jeder Prozessor eine unterschiedliche (auch unendliche) Anzahl von Schleifendurchläufen haben. Praktisch gesehen macht es aber nur Sinn Schleifen auszuführen, in denen mindestens 1 Prozessor aktiv ist. Daher wird im Meta-Programm die Schleife ausgeführt, bis T_k leer ist.

while $\alpha(p)$ **do**	$k \leftarrow k + 1$
... Anweisungen ...	$T_k \leftarrow \{p \in T_{k-1} \mid \alpha(p) = true\}$
	while $T_k \neq \emptyset$ **do**
	... Anweisungen ...
	$T_k \leftarrow \{p \in T_k \mid \alpha(p) = true\}$
	$k \leftarrow k - 1$

2.3 Speicherkonflikte

Ein *Speicherkonflikt* tritt auf, wenn zwei Prozessoren gleichzeitig auf eine Speicherzelle zugreifen wollen. Ein Zugriff kann dabei entweder lesend oder schreibend sein. Um zu definieren, inwieweit diese Kombinationen erlaubt sind, bzw. wie sie behandelt werden, existieren verschiedene Betriebsmodi, durch die PRAMs in eine Stufenhierarchie eingeordnet werden können:



Exclusive-Read/Exclusive-Write (EREW)

Die strengste Form erlaubt weder das gleichzeitige Lesen noch Schreiben einer Speicherzelle. Algorithmus 1 liegt in dieser Klasse.

Concurrent-Read/Exclusive-Write (CREW)

Diese Stufe erlaubt das gleichzeitige Lesen einer Speicherzelle, aber verbietet den gleichzeitigen Zugriff, wenn mindestens ein Prozessor schreibend zugreift.

Concurrent-Read/Concurrent-Write (CRCW)

In der letzten Stufe werden nun auch gleichzeitige schreibende Zugriffe erlaubt. Um zu definieren, welchen Wert eine Speicherzelle hat, wenn sie von mehreren Prozessoren beschrieben wird, existieren wiederum mehrere Strategien:

- *common*: Ein Spezialfall, in dem das gleichzeitige Schreiben nur erlaubt ist, wenn jeder Prozessor den gleichen Wert schreiben möchte. Die Speicherzelle enthält danach diesen Wert.

- *arbitrary*: Die Speicherzelle enthält den Wert eines zufälligen beschreibenden Prozessors.
- *priority*: Eine Priorisierung der Prozessoren wird definiert (bspw. nach der Prozessor-ID). Die Speicherzelle enthält danach den Wert des Prozessors mit der höchsten Priorität.

Im Weiteren werden jedoch hauptsächlich Algorithmen aus CREW betrachtet, weswegen auf diese Strategien nicht weiter eingegangen wird. Als Speicherkonflikt wird somit auch nur noch der Fall gesehen, wenn von den zwei Prozessoren mindestens einer schreibend zugreift.

2.4 Praktische Realisierbarkeit

Zwar ist das PRAM-Modell ein einfaches Mittel zur Diskussion und Entwurf von parallelen Algorithmen, jedoch existieren auch einige Kritikpunkte, was die Implementierung von PRAM-Algorithmen in der Praxis angeht.

Fortnow [For05], Casanova, Legrand und Robert [CLR08, Kapitel 1.5], sowie Kessler und Keller [KK07] erläutern, dass einige Eigenschaften des PRAM-Modells in der Praxis kaum erreichbar sind. Zum einen kritisieren sie, dass eine von der Eingabelänge abhängige Anzahl von Prozessoren auf realen Rechner nicht machbar ist. Moderne parallele Architekturen erreichen zwar hohe Parallelisierungsgrade, jedoch existiert immer ein endliches Limit an tatsächlich verfügbaren Recheneinheiten.

Ein weiterer Kritikpunkt richtet sich gegen die globale Synchronisierung der Prozessoren nach jeder Anweisung. Je nach Anwendungsgebiet (z. B. bei über das Internet verbundenen Rechnernetzen) kann diese zu einer extrem erhöhten Laufzeit führen, die die eigentliche Laufzeitkomplexität des Algorithmus in den Hintergrund stellt. Heutige parallele Architekturen arbeiten daher eher asynchron und versuchen globale Synchronisierungen zu minimieren.

Zusätzlich ist auch die Existenz eines globalen, parallelen und in konstanter Zeit zugreifbaren Speichers nicht selbstverständlich. Die Zusatzkosten, die durch das Sichern der Speicherkonsistenz entstehen, sind in einer realen Implementierung nicht zu unterschätzen.

Infolgedessen ist es derzeit kaum möglich, einen unmodifizierten PRAM-Algorithmus in der vom PRAM-Modell beschriebenen Laufzeit auszuführen. Dies soll aber nicht bedeuten, dass das PRAM-Modell keine Anwendung mehr hat. Auf Grund der Mächtigkeit und einfachen Definition sind PRAMs für das *Prototyping*[2] von parallelen Algorithmen nützlich. Stellt es sich als schwer heraus, einen

[2]Entwicklung eines Prototypen während eines Designprozesses

effizienten PRAM-Algorithmus zu finden, so ist es unwahrscheinlich, dass eine effiziente Implementierung auf einer realistischeren (und damit auch restriktiveren) Plattform gelingt. Im nächsten Kapitel wird eine solche Plattform vorgestellt: Die GPU.

Kapitel 3

Einblick in die GPGPU-Programmierung

3.1 GPU-Architektur

Die Architektur moderner GPUs orientiert sich an dem *Stream Processing*-Modell. Anders als in einer CPU, in der der Prozessor auf die Daten über einen frei adressierbaren Speicher zugreift, sieht das Stream Processing-Modell die Daten als Strom, der auf mehrere Prozessoren aufgeteilt wird. Pro Datenelement führt ein Prozessor eine Funktion (*Kernel*) aus und kann dabei neue Daten erzeugen, die wiederum einen neuen Datenstrom bilden.

Es ist leicht zu sehen, wie dieses Modell in dem ursprünglichen Grafikanwendungsbereich verwendet wurde: Die darzustellenden Polygondaten bilden den Ursprungsdatenstrom. In einem ersten Kernel werden die Koordinaten transformiert und wieder ausgegeben. Auf diesen neuen Daten erzeugt der Rastarisierer für jedes vom Polygon verdeckte Pixel ein Fragment, welches die interpolierten Vertexattribute beinhaltet. In einem letzten Schritt wird aus einem Fragment ein Farbwert berechnet, bei dem beispielsweise Texturzugriffe und Lichtberechnungen stattfinden. Auf Grund dieses Datenflusses von einer Station zur nächsten wird auch oft von einer *Pipeline*-Architektur gesprochen.

Dieses Modell hat einige entscheidende Vorteile gegenüber der einer CPU, wenn es um die Parallelisierung geht. Da die Anwendung eines Kernels auf ein Datenelement unabhängig von anderen Datenelementen geschehen kann, lässt sich die Arbeit leicht parallel ausführen, indem der Datenstrom von einem *Scheduler* disjunkt auf die zur Verfügung stehenden Prozessoren aufgeteilt wird. Durch diese explizite Zuordnung der Daten auf die Prozessoren, kann auch auf eine aufwendige Cache-Hierarchie verzichtet werden, da ein Prozessor nur die ihm zugeteilten Daten benötigt und nicht selbst auf den Speicher zugreift.

GPUs realisieren nun dieses Modell, indem sie mehrere tausend[1] Stream-Prozessoren besitzen, die parallel die Kernels ausführen. Um die Effizienz zu steigern, bearbeitet dabei ein Stream-Prozessor nicht nur ein Datenelement, sondern führt die Funktion direkt auf mehreren Datenelementen aus. Anders gesehen führt ein Stream-Prozessor somit mehrere Einzelprozessoren parallel aus. Diese Gruppe von, in der Regel 32 oder 64, Einzelprozessoren wird als *Warp* oder auch *Wavefront* bezeichnet. Erreicht wird diese Parallelisierung durch eine dem *SIMD*[2]-Modell ähnliche, datenparallele Instruktionsausführung. Dabei haben die Register eines Stream-Prozessors für jeden Einzelprozessor einen getrennten Wert. Die Instruktionen werden dann von der Hardware parallel auf allen Einzelwerten eines Registers ausgeführt. Für Prozessoren innerhalb eines Warps gilt somit, dass sie komplett synchron arbeiten, da sie sich ein Program-Counter teilen. Ein anderer Name dieser Art von Instruktionsausführung ist daher auch *SIMT* (Single Instruction, Multiple Threads).

Ein Problem tritt jedoch auf, wenn eine Verzweigung (z.B. eine `if`-Anweisung) dazu führt, dass einige Prozessoren einen anderen Pfad nehmen als der Rest des Warps. Da das SIMT-Modell nur eine gleichzeitige Instruktion erlaubt, ist es nicht möglich beide Pfade gleichzeitig auszuführen, da diese unterschiedliche Anweisungen beinhalten können. Gelöst wird dieses Problem dadurch, dass die beiden Code-Pfade sequentiell nacheinander ausgeführt werden, wobei die Prozessoren, die den Pfad nicht gewählt haben, deaktiviert werden (auch *Masking* genannt). Inaktive Prozessoren bedeuten jedoch eine Verringerung der Effizienz, da eine Instruktion bei der nur ein Prozessor aktiv ist, die gleichen Laufzeitkosten hat, als wären alle Prozessoren aktiv. Ein genereller Leitfaden bei der Optimierung von Kernels ist daher, die Verzweigung innerhalb einer Warps zu vermeiden.

Insgesamt ergibt sich dann die Gesamtanzahl an möglichen parallel laufenden Prozessoren aus dem Produkt der Anzahl der Stream-Prozessoren und der Größe eines Warps. Zwar ist die Leistung eines Einzelprozessors einer GPU verglichen zu der einer CPU um mehrere Größenordnungen schlechter, allerdings wird dies durch die schiere Anzahl an Prozessoren wieder ausgeglichen. Rein rechnerisch besitzen moderne GPUs daher eine deutlich höhere Maximalleistung als CPUs, wie in auch Abbildung 3.1 deutlich wird.

[1] Bsp: 2816 bei der *AMD R9 390*

[2] Singe Instruction, Multiple Data

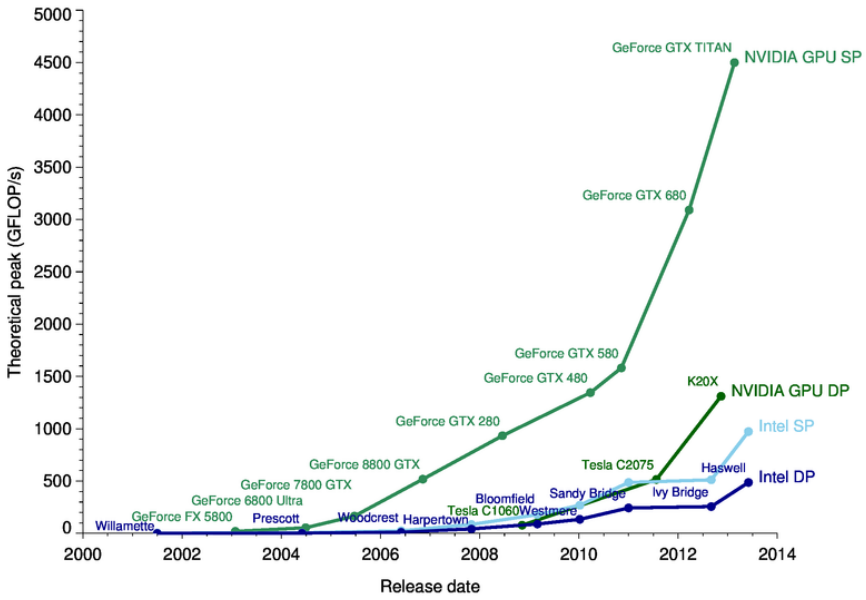


Abbildung 3.1: Performance Trends von GPUs und CPUs in *Single Precision* (SP) und *Double Precision* (DP) nach [Gal13]

3.2 OpenCL

OpenCL (Open Computing Language) ist eine von der Khronos Group entwickelte Spezifikation, welche eine Plattform für die Implementierung paralleler Algorithmen darstellt. Umgesetzt wurde diese unter anderem von Nvidia, AMD und Intel für ihre jeweiligen GPUs. Dies macht OpenCL zu der zurzeit einzigen plattformunabhängigen Möglichkeit der GPGPU-Programmierung. Für den Anwender ist OpenCL als C-API verfügbar und wird in der Regel von dem GPU-Hersteller veröffentlicht. Ein OpenCL-Programm lässt sich generell in das *Host-* und *Device*-Programm unterteilen.

Host-Programm

Das Host-Programm ist ein gewöhnliches, sequentielles CPU-Programm, welches die OpenCL-API benutzt. Aufgaben des Host-Programms sind unter anderem:

- Das Laden und Kompilieren des Device-Programms
- Die Allokation und Initialisierung von Speicherobjekten (Buffers)

- Das Starten von Kernels und Speichertransfers zwischen Host- und Device-Speicher

Befehl	**Beschreibung**
`clEnqueueWriteBuffer`	Speichertransfer von Host zu Device
`clEnqueueReadBuffer`	Speichertransfer von Device zu Host
`clEnqueueNDRangeKernel`	Startet einen Kernel
`clSetKernelArg`	Setzt ein Argumente eines Kernel

Tabelle 3.1: Überblick wichtiger OpenCL-API-Funktionen

Command Queue

Die *Command Queue* ist ein im Host-Programm erstelltes Objekt, das einen Kommunikationskanal vom Host zu einem oder mehreren OpenCL fähigen Hardwarekomponenten (Devices) darstellt. Im Host-Programm können über die `clEnqueue<>`-Funktionen Befehle in die Command Queue eingetragen werden, welche dann asynchron von den verbundenen Devices ausgeführt werden.

Abbildung 3.2 zeigt einen typischen Programmablauf über die Command Queue.

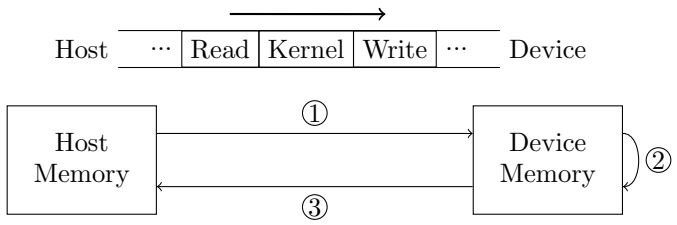


Abbildung 3.2: Command Queue mit typischer Befehlsabfolge

1. Mit einem Write-Befehl werden die Daten vom Host- zum Device-Speicher kopiert.
2. Ein oder mehrere Kernels manipulieren den Device-Speicher.
3. Der Device-Speicher wird wieder zum Host zurückgelesen.

Ein Kopieren der Daten zwischen Host- und Device-Speicher ist notwendig, da diese physikalisch voneinander getrennte Speicherbereiche darstellen und Kernels nur auf den Device-Speicher zugreifen können. Idealerweise sollten Daten daher möglichst lange im Device-Speicher bleiben, sodass mehrere Kernels auf ihnen arbeiten können, ohne sie unnötig kopieren zu müssen.

Worksize, Workitems und Workgroups

Beim Starten eines Kernels über eine Command Queue muss neben den Argumenten auch ein n-dimensionaler Integerbereich (*Worksize*) angegeben werden. Dieser ist zur zur Zeit aber durch OpenCL auf maximal 3 Dimensionen beschränkt. Die Worksize gibt an, wie viele einzelne Kernel-Instanzen (*Workitems*) ausgeführt werden sollen. Jedes Element in der Worksize stellt die Prozessor-ID eines Workitems dar, welche innerhalb des Kernels über die Funktion `get_global_id` abgefragt werden kann. Übertragen auf das Stream Processing-Modell wird hier ein Datenstrom von n-dimensionalen ID-Vektoren generiert, auf den der Kernel angewendet wird.

Zusätzlich werden Workitems auch noch zu Arbeitsgruppen (*Workgroups*) gruppiert. Beim Starten eines Kernels muss die Anzahl der Workitems pro Workgroup angegeben werden. Die Gesamtanzahl an Workgroups und die Zuteilung der Workitems zu Workgroups wird dann von OpenCL bestimmt. Die maximale Größe einer Workgroup ist von der Komplexität des Kernels und der unterliegenden Hardware abhängig. Üblich sind aber Größen von maximal 128 bis 256 Workitems pro Workgroup auf GPUs. Wird im Folgenden ein Workitem innerhalb einer Workgroup betrachtet, so wird dafür das Adjektiv *lokal* benutzt. *Global* wird dagegen für den Bezug zur gesamten Worksize verwendet.

Abbildung 3.3 zeigt diese Unterteilung beispielhaft anhand einer zweidimensionalen Worksize.

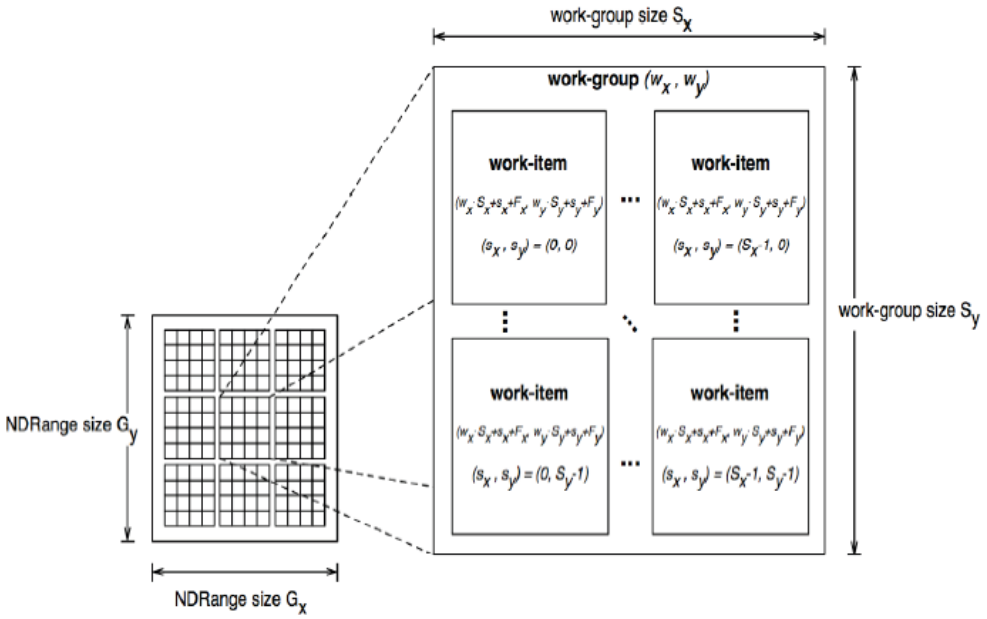


Abbildung 3.3: Einteilung einer zweidimensionalen Worksize (hier NDRange) in Workgroups und Workitems. [Gro15]

Device-Programm

Das Device-Programm ist eine in *OpenCL C* verfasste Menge von Kernelfunktionen und wird, ähnlich wie OpenGL-Shader, zur Laufzeit kompiliert. Innerhalb eines Kernels können auf die vom Host-Programm gesetzten Kernel-Argumente zugegriffen werden. Auch gibt es eine Reihe von eingebauten Kernelfunktionen, die verwendet werden können. Diese ermöglichen unter anderem:

- Den Zugriff auf spezielle Hardwarekomponenten der GPU, wie z.B. das Sampling und Interpolieren von Texturen.
- Die Abfrage von Informationen über die aktuelle Kernel-Instanz, etwa der Workitem-ID.
- Spezielle Workgroup-bezogene Funktionen, wie eine lokale Synchronisierung oder asynchrone Speichertransfers.

Befehl	**Beschreibung**
`get_global_id`	Globale ID des Workitems
`get_global_size`	Anzahl an Workitems
`get_local_id`	Lokale ID des Workitems
`get_local_size`	Anzahl an Workitems in der Workgroup
`barrier`	Synchronisiert alle Workitems einer Workgroup

Tabelle 3.2: Überblick wichtiger OpenCL-Kernelfunktionen

Synchronisationsverhalten

Generell laufen die Workitems eines Kernels komplett asynchron voneinander ab und es gibt auch keine offizielle Möglichkeit, eine globale Synchronisation aller Workitems innerhalb eines Kernels zu erreichen. Grund dafür ist, dass nicht immer alle Workitems echt parallel ausgeführt werden können, wenn zum Beispiel nicht genügend Prozessoren zur Verfügung stehen. In diesem Fall muss ein Prozessor mehrere Workitems sequentiell ausführen. Eine Synchronisation zwischen diesen sequentiell ausgeführten Workitems ist nun nicht mehr möglich, da das erste Workitem bereits vollständig ausgeführt werden musste, bevor mit dem zweiten begonnen werden konnte. Die einzige Art eine globale Synchronisierung zu erreichen, ist nachdem ein Kernel abgeschlossen wurde. Die Command Queue hat dazu die Eigenschaft, dass mit einem Kernel erst begonnen wird, wenn alle Workitems des vorherigen Kernels vollendet wurden.

Anstatt einer globalen Synchronisation, ist jedoch eine lokale Synchronisation innerhalb Kernels im Rahmen einer Workgroup möglich. Dazu stellt OpenCL die

Workgroup-Funktion `barrier` bereit, aus der erst zurückgesprungen wird, wenn alle Workitems der Workgroup diese ausgeführt haben. Somit kann sichergestellt werden, dass die Anweisungen vor der `barrier`-Anweisung von allen Workitems der Workgroup ausgeführt wurden, bevor mit den nächsten Anweisungen fortgesetzt wird.

Device-Speicher

OpenCL teilt den, in den Kernels zugreifbaren Device-Speicher, in mehrere voneinander getrennte Speicherbereiche ein, welche die hierarchische Struktur der Workgroups und Workitems widerspiegeln.

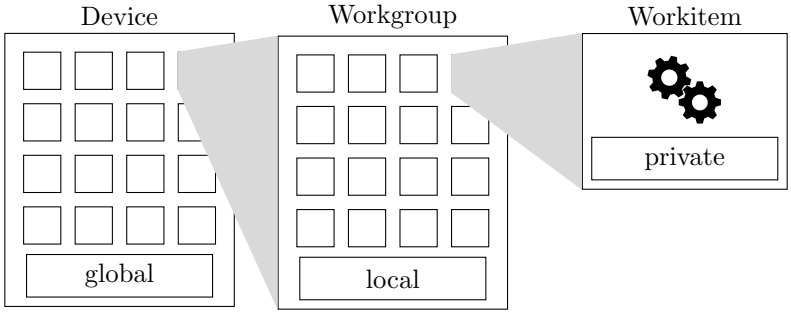


Abbildung 3.4: Speicherhierarchie in OpenCL

Auf der obersten Device-Ebene existiert der globale Speicher. Dieser ist der einzige vom Host-Programm zugreifbare Speicher und bleibt über den gesamten Programmablauf konsistent, sodass die Daten auch in mehreren Kernels benutzt werden können. Aus diesem Grund findet die Ein- und Ausgabe der Kernels hierüber statt. Der Nachteil des globalen Speichers ist jedoch, dass er im Vergleich zu den anderen Speicherbereichen, die größten Zugriffskosten hat.

Auf der nächsten Ebene existiert innerhalb einer Workgroup der lokale Speicher. Dieser ist nur für die Workitems einer Workgroup sichtbar und auch nur während eines Kernels gültig. Nachdem ein Kernel beendet wurde, kann nicht mehr sicher auf den lokalen Speicher zugegriffen werden. Er kann daher nicht verwendet werden, um Daten zwischen zwei Kernels auszutauschen. Der Vorteil dieses Speichers ist jedoch, dass er bis zu 100x geringere Zugriffskosten hat als der globale Speicher, da dieser direkt im Cache eines Stream-Prozessors liegt [NVI09].

Der private Speicher stellt dann den schnellsten, aber auch eingeschränktesten Speicherbereich da. Er ist nur für ein einzelnes Workitem sichtbar und liegt in der Regel direkt in einem Register des Stream-Prozessors.

Alle Speicherbereiche erlauben es, parallel lesend auf sie zuzugreifen. Jedoch macht OpenCL keine klare Aussage, wie der Fall, wenn parallel mehrere Worki-

tems schreibend auf eine Speicherzelle zugreifen, behandelt wird. Es ist somit vom schlimmsten Fall auszugehen, sodass die Speicherzelle in diesem Fall einen korrupten Wert beinhaltet. Insgesamt ist der Device-Speicher in OpenCL somit ähnlich zu der einer CREW-PRAM zu sehen.

3.3 Modell für GPGPU-Algorithmen

Da im weiteren Verlauf der Arbeit auf die Implementierung durch GPUs eingegangen wird, ist ein Modell für diese Art von Algorithmen erforderlich. Der direkte OpenCL-Code wird nicht aufgelistet, um die aufwendige OpenCL-Initialisierung zu vermeiden und um die Lesbarkeit der Algorithmen zu verbessern.

Wie auch in OpenCL findet in den GPGPU-Algorithmen eine Trennung zwischen Host- und Device-Programm statt. In der Notation werden diese Teile durch eine horizontale Linie kenntlich gemacht. Im oberen Teil wird das Host- und im unteren Teil das Device-Programm aufgelistet. Das Host-Programm wird in klassischem sequentiellem Pseudocode verfasst. Hier kann mit der neuen Anweisung

$$\textbf{enqueue } \textsc{MyKernel}(args) \textbf{ on } \{0, \ldots, n\}/m$$

die Ausführung des MyKernel-Kernel mit den Argumenten $args$ auf einer Worksize von $\{0, \ldots, n\}$ in Workgroups der Größe m gestartet werden. Spielt die Workgroups keine Rolle, so kann $/m$ auch weggelassen werden. Für die angegebene Worksize muss wie in OpenCL gelten, dass sie einen zusammenhängenden Integerbereich darstellt.

Was im Gegensatz zu OpenCL jedoch vernachlässigt wird, ist die Deklaration der Command Queue und die Initialisierung und Transfers von Speicherobjekten. Für die Command Queue wird angenommen, dass bereits eine gültige Verbindung zum Device hergestellt wurde. Das Management der Speicherobjekte geschieht automatisch. Das bedeutet, dass für jedes Speicherobjekt, das als Argument in einem Kernel verwendet wird, wird, wenn nötig, ein entsprechender Speichertransfer ausgeführt. Genauso wird bei späteren Zugriffen auf ein Objekt im Host-Programm der Speicher wieder zurückgelesen.

Im Device-Programm kann mit dem Programmblock **kernel** eine neue Kernelfunktion definiert werden, welche innerhalb des Host-Programm aufgerufen werden kann. Für die Argumente des Kernels muss mit **global**, **local** oder **private** der Speicherbereich angegeben werden, in dem sie liegen. Mehrere aufeinanderfolgende Variablen können in dem gleichen Speicherbereich definiert werden, indem die Speicherbereiche der folgenden Variablen weggelassen wird. Zusätzlich können innerhalb des Device-Programms die von OpenCL definierten Kernelfunktionen (siehe beispielsweise Tabelle 3.2) verwendet werden.

Ein simples Beispiel GPGPU-Programm, welches parallel paarweise die Elemente zweier Felder A, B sortiert, ist in Algorithmus 3 zu sehen.

Algorithmus 3 Simples GPGPU-Programm

```
1: A[n], B[n]
2: enqueue PairwiseSort(A, B) on {0, ..., n}
3: kernel PairwiseSort(global A[n], B[n])
4:     p ← get_global_id(0)
5:     if A[p] > B[p] then
6:         tmp ← A[p]
7:         A[p] ← B[p]
8:         B[p] ← tmp
```

Kapitel 4

PRAMs auf GPGPUs

Nachdem nun sowohl mit der PRAM ein theoretisches Modell, als auch mit der GPU und OpenCL eine reale Plattform vorgestellt wurde, stellt sich die Frage, inwieweit beide miteinander kompatibel sind und ob es möglich ist, PRAM-Algorithmen direkt auf der GPU-Architektur auszuführen.

Eine solche Implementierung von PRAM-Algorithmen wurde auch bereits von Brenner, Keller und Kessler [BKK12] untersucht, bei der eine Transformation der PRAM-Progammiersprache *Fork* auf die OpenCL ähnliche *CUDA*-Plattform entwickelt wurde. Auch existieren einige Versuche wie das *XMT*-Projekt [Vis15], in dem ein Computerchip entwickelt wurde, der direkt PRAM-Algorithmen ausführen kann, jedoch konnten diese sich in einem praktischen Anwendungsbereich nicht durchsetzten.

4.1 Direkte Implementierung

Auf den ersten Blick scheint OpenCL eine ideale Plattform für die Implementierung von PRAM-Algorithmen zu sein. Es bietet einen global, parallel les- und schreibaren Speicher an, bei dem die Anweisungen des PRAM-Algorithmus innerhalb eines Kernels implementiert werden können. Dieser kann dann auf einer dynamisch große Eingabegrößen gestartet werden und innerhalb eines Workitems kann auf die Prozessor-ID p über `get_global_id(0)` zugegriffen werden.

Wie jedoch aber bereits zu vermuten ist, liegt das Problem in dem Synchronisierungsverhalten der beiden Architekturen. Die globale Synchronisierung, die von einer PRAM nach jeder Anweisung garantiert wird, kann innerhalb einer Kernel-Ausführung nicht erreicht werden (siehe Abschnitt 3.2). Dadurch treten bei einer naiven Implementierung Fehler auf, wenn ein Workitem auf eine Speicherzelle zugreift, die von einem anderen Workitem innerhalb des gleichen Kernels beschrieben wird. Es kann in diesem Fall keine Aussage getroffen werden, in welcher Reihenfol-

ge die beiden Workitems auf die Speicherzelle zugreifen, da sie komplett asynchron voneinander laufen. Die Definition eines Speicherkonfliktes muss somit für Kernels verschärft werden, indem dieser bereits auftritt, wenn über den gesamten Algorithmus zwei Workitems auf die gleiche Speicherzelle (auch an unterschiedlichen Programmstellen) lesend und schreibend zugreifen. Da somit eine Kommunikation der Prozessoren über den globalen Speicher nicht mehr möglich ist, lassen sich die meisten nicht trivialen PRAM-Algorithmen nicht umsetzten.

PRAM in einer Workgroup

Möchte man trotzdem ein PRAM-Algorithmus direkt durch einen Kernel implementieren, so können die von OpenCL bereitgestellten Synchronisationsmöglichkeiten innerhalb Workgroup verwendet werden. Da dies jedoch erfordert, dass alle Workitems in der selben Workgroup sind, und Workgroups nur eine maximale Größe haben können, wird die Möglichkeit verloren, beliebige Eingabegrößen zu bearbeiten. Dies mag zunächst ein großer Verlust sein, jedoch wird sich auch diese Art von PRAM-Implementierung in Kapitel 5 noch als nützlich herausstellen.

Erfüllt man diese Einschränkung, so kann man, indem nach jedem Speicherzugriff mit `barrier()` eine Synchronisation erzwungen wird, den PRAM-Algorithmus direkt innerhalb eines Kernels implementieren. Möchte man diese explizite Synchronisation vermeiden, so besteht auch die Möglichkeit, die in Abschnitt 3.1 beschrieben Warps zu nutzten. Innerhalb eines Warps gelten die gleichen Synchronisationseigenschaften wie in einer PRAM. Somit kann in diesem kleinen Rahmen ein PRAM-Algorithmus unverändert implementiert werden. Da Warps jedoch in der OpenCL-Spezifikation nicht enthalten sind, stellt sich die Verwendung dieser Eigenschaften als sehr hardwareabhängig heraus, da diese nicht auf jeder Architektur vorhanden sein müssen. Aus diesem Grund wird in dieser Arbeit auf diese Technik nicht weiter eingegangen. Andere Arbeiten ([Ye+10]) verwenden diese Technik jedoch, um sehr performante Algorithmen zu ermöglichen.

Synchronisation durch Atomics

Eine letzte Möglichkeit, Synchronisierungen auch über die Workgroupgrenzen hinaus zu erreichen, wurde von Xiao und Feng [XF10] untersucht. Diese Technik verwendet *Atomics*, um über ein *Spin-Lock* die Workitems synchron zu halten. Ein Atomic ist eine spezielle globale Datenstruktur, die sich sicher auch von mehreren Prozessoren parallel beschreiben lässt. Ein Spin-Lock erreicht über eine leere Schleife, dass der Programmfluss angehalten wird, bis eine Bedingung erfüllt wird. Nun kann über ein Atomic C, das mit 0 initialisiert wird, eine globale Synchronisation erreicht werden:

Vor der Synchronisierung
atomic($C \leftarrow C + 1$)
while atomic($C < n$) **do** —
Nach der Synchronisierung

Jeder Prozessor inkrementiert C und wartet dann, bis $C = n$ ist. So kann garantiert werden, dass alle n Prozessoren diesen Punkt erreicht haben. Jedoch treten hier auch Probleme auf, wenn beispielsweise die Anzahl an Workitems zu groß ist um echt parallel ausgeführt zu werden. In diesem Fall wird das Spin-Lock zur Endlosschleife, da die Prozessoren nicht dazu kommen können, die restlichen Workitems auszuführen. Aus diesem Grund ist diese Technik für eine allgemeine Implementierungsstrategie ebenfalls ungeeignet.

4.2 Meta-Programm-Implementierung

Nachdem sich die direkte Implementierung eines PRAM-Algorithmus als schwierig herausgestellt hat, wird nun eine Implementierung des in Kapitel 2 vorgestellten Meta-Programm entwickelt. Da das Meta-Programm jedoch nur eine andere Schreibweise für einen PRAM-Algorithmus ist, kann dadurch auch eine generelle PRAM Implementierung erzielt werden. Dazu wird das Meta-Programm durch das Host-Programm ausgeführt und die auf Mengen bezogenen, parallelen Zuweisungen werden durch Kernels implementiert.

Die Zuweisung $x_p \leftarrow \alpha(p)\ \forall p \in T_k$ kann folgendermaßen umgesetzt werden werden:

enqueue $\textsc{Assign}_\alpha(x, \Psi(\alpha))$ **on** T_k

kernel $\textsc{Assign}_\alpha$(**global** $x[n], \Psi(\alpha)$)
    $p \leftarrow$ `get_global_id(0)`
    $x[p] \leftarrow \alpha(p)$

Es wird ein neuer Kernel $\textsc{Assign}_\alpha$ definiert, der den Ausdruck α auswertet und das Ergebnis in x speichert. Die Variable x, die im Meta-Programm ein n-dimensionaler Vektor ist, wird nun als Feld der Größe n dargestellt. Der Kernel benötigt neben diesem Feld auch die Abhängigkeiten des Ausdrucks α. Im Host-Programm wird dieser Kernel dann für alle derzeit aktiven Prozessoren T_k gestartet.

Eine Feld-Zuweisung $A[\alpha(p)] \leftarrow \beta(p)\ \forall p \in T_k$ kann ähnlich implementiert werden. Allerdings muss ein *Double-Buffering* verwendet werden, um mögliche Speicherkonflikte zu vermeiden:

---

global $\Gamma[\cdot]$
enqueue FIELD-ASSIGN$(\Gamma, \Psi(\alpha), \Psi(\beta))$ **on** T_k
swap $\Gamma \leftrightarrow A$

---

kernel FIELD-ASSIGN(**global** $\Gamma[\cdot], \Psi(\alpha), \Psi(\beta)]$)
    $p \leftarrow$ `get_global_id(0)`
    $\Gamma[\alpha(p)] \leftarrow \beta(p)$

Da es möglich ist, dass A in den Abhängigkeiten von α und β vorkommt, kann in diesem Fall ein Speicherkonflikt auftreten, wenn eine Speicherzelle von unterschiedlichen Prozessoren gelesen und beschrieben wird. Um dies zu umgehen, wird ein zweites Feld Γ mit der gleichen Größe wie A verwendet, in das das Ergebnis der Zuweisung zwischengespeichert wird. Ein Speicherkonflikt kann nun nicht mehr auftreten, da A innerhalb des Kernels nicht mehr beschrieben wird. Um trotzdem zu erreichen, dass A nach der Anweisung den korrekten Wert hat, kann entweder der Inhalt von Γ nach A kopiert werden, oder effizienter im Host-Programm der von A adressierte Speicherbereich mit dem von Γ getauscht werden. Somit wird erreicht, dass jede weitere Verwendung von A in Wirklichkeit den Speicherbereich von Γ verwendet. Da Γ nun den Speicher des alten A besitzt, funktioniert dies auch innerhalb einer Schleife. Hier würden die beiden Speicherbereiche immer wieder hin und her getauscht werden.

Problem: Nicht zusammenhängende Worksizes

Beide Arten von Zuweisungen haben jedoch noch das Problem, dass sie die Menge der aktiven Prozessoren direkt als Worksize verwenden. Es kann jedoch sein, dass diese nicht immer zusammenhängend ist. Beispiel: $T_k = \{0, 3, 6, 7\}$ hat die Lücken $\{1, 2\}, \{4, 5\}$. Es wird daher für diese Fälle eine Darstellung der aktiven Prozessoren gesucht, die es ermöglicht die Kernels auf einer zusammenhängenden Worksize zu starten. Auch muss sie es erlauben, die Menge der aktiven Prozessoren zu verkleinern und zu testen, ob sie leer ist, um die Verzweigung und Schleife des Meta-Programms zu realisieren. Im Folgenden werden zwei mögliche Methoden vorgestellt.

4.2.1 Bitset-Methode

Eine naheliegende Lösung ist es, die Mengen $T_0, \ldots, T_{d-1}$ als Bitsets $B_0, \ldots, B_{d-1}$ mit jeweils n Elementen darzustellen. Es gilt dann

$$p \in T_k \Leftrightarrow B_k[p] = 1 \text{ und } p \notin T_k \Leftrightarrow B_k[p] = 0$$

Zuweisungen werden dann auf der gesamten Worksize $\{0, \ldots, n-1\}$ ausgeführt, wobei innerhalb eines Kernels geprüft wird, ob der Prozessor zur Zeit aktiv ist (d. h. $B_k[p] = 1$).

Verzweigungen

Ähnlich kann dann auch eine Verzweigung $T_i \leftarrow \{p \in T_j \mid \alpha(p) = true\}$ realisiert werden. Da sich hier die Menge der aktiven Prozessoren nicht vergrößert, kann B_{k+1} aus dem logischen Und von B_k und dem Wert des Prädikats geschehen:

---

enqueue BRANCH$_\alpha(B_i, B_j, \Psi(\alpha))$ **on** $\{0, \ldots, n\}$

---

kernel BRANCH$_\alpha$(**global** $B_i[n], B_j[n], \Psi(\alpha)$)
  $p \leftarrow$ `get_global_id(0)`
  $B_i[p] \leftarrow B_j[p] \wedge \alpha(p)$

Durch den BRANCH$_\alpha$-Kernel wird das neue Bitset B_i der neuen Programmtiefe bestimmt. Hier sind alle Prozessoren aktiv, die sowohl in der alten Ebene B_j aktiv waren, als auch das Prädikat α erfüllen.

Schleifen

Aufwendiger ist dagegen die Implementierung von Schleifen, da hier getestet werden muss, ob die Menge leer ist. In diesem Fall wäre $B_k[p] = 0$ für alle p.

Eine einfache Möglichkeit ist, im Host-Programm über das Bitset zu iterieren, um diese Eigenschaft zu überprüfen. Da dies jedoch zu einem linearen Blowup in der Laufzeit führen würde, kann versucht werden, durch eine Parallelisierung diesen Effekt zu verringern.

Offensichtlich ist die Menge nicht leer, wenn mindestens ein Prozessor aktiv ist. Somit gilt

$$T_K \neq \emptyset \Leftrightarrow B_k[0] \vee B_k[1] \vee \cdots \vee B_k[n-1].$$

Das Problem ist daher im Grunde eine Oder-*Reduktion* aller Elemente des Bitsets. Reduktion beschreibt hier den Vorgang, eine assoziative Funktion (hier das logische Oder) auf eine Menge anzuwenden, sodass das Ergebnis die Kombination

aller Elemente darstellt. Im Detail wird auf einen solchen parallelen Reduktionsalgorithmus in Abschnitt 5.1 eingegangen. Zunächst sei diese Funktionalität jedoch in der Funktion $\textsc{Reduce}_\vee$ enthalten. Sie überprüft, ob mindestens eine Zelle des aktuellen Bitsets $\neq 0$ ist. Somit lässt sich nun auch die Schleife umsetzten, indem als Schleifenbedingung $\textsc{Reduce}_\vee(B_k)$ verwendet wird.

Diskussion

Da sich nun alle 3 Arten von PRAM-Anweisungen (Zuweisung, Verzweigung, Schleifen) durch dieser Methode realisieren lassen und Kernel auf einer zusammenhängenden Worksize gestartet werden, ist theoretisch die Implementierung beliebiger PRAM-Algorithmen möglich. Die Laufzeitkomplexität steigt dabei nur bei der Auswertung von Schleifen, welche eine zu der Prozessoranzahl logarithmische Laufzeit pro Durchlauf benötigen.

Ein weiterer Nachteil ist, dass alle Kernel immer mit n Workitems gestartet werden, auch wenn in Wirklichkeit nur wenig Prozessoren aktiv sind. Die so unnötig gestarteten Workitems fallen zwar durch den Test $B_k[p] = 1$ wieder weg, jedoch stellt sich dies für die Warps als nachteilig heraus, denn die nicht relevanten Workitems blockieren immer noch eine Stelle im Warp, an welcher ein aktiver Prozessor hätte ausgeführt werden können. Somit fällt an Stellen mit nur wenig aktiven Prozessoren die relative Gesamtperformance.

Dieser Effekt ist auch von der Lokalität der inaktiven Prozessoren abhängig. Unter der Annahme, dass die Einteilung in Warps sequentiell geschieht (d. h. der erste Warp der Größe k beinhaltet Prozessoren 0 bis $k-1$, der zweite k bis $2k-1$ usw.), wird eine bessere Effizienz erzielt, wenn die aktiven und inaktiven Prozessoren möglichst gruppiert auftreten. Dies ermöglicht einem Warp, der nur inaktiven Prozessoren enthält, den Kernel früh zu beenden, um eine andere Menge von Prozessoren auszuführen.

Beispiel

Algorithmus 4 zeigt das Ergebnis, wenn man die Transformationen der Bitset-Methode verwendet, um das Meta-Programm aus Algorithmus 2 zu implementieren.

Algorithmus 4 Algorithmus 2 implementiert durch die Bitset-Methode

$A[2n], k[n]$
$B_0[n], B_1[n], B_2[n]$
$\Gamma[2n]$
enqueue $\textsc{Assign}_{2n}(k, B_0)$ **on** $\{0, \ldots, n-1\}$
enqueue $\textsc{Branch}_{k>1}(B_1, B_0, k)$ **on** $\{0, \ldots, n-1\}$
while $\textsc{Reduce}_{\vee}(B_1)$ **do**
  enqueue $\textsc{Branch}_{p<k}(B_2, B_1, k)$ **on** $\{0, \ldots, n-1\}$
  enqueue $\textsc{Field-Assign}_{p,A[2p]+A[2p+1]}(\Gamma, A, B_2)$ **on** $\{0, \ldots, n-1\}$
  swap $\Gamma \leftrightarrow A$
  enqueue $\textsc{Assign}_{\frac{k}{2}}(k, B_1)$ **on** $\{0, \ldots, n-1\}$
  enqueue $\textsc{Branch}_{k>1}(B_1, B_1, k)$ **on** $\{0, \ldots, n-1\}$

kernel $\textsc{Assign}_{2n}$(**global** $k[n], B[n]$)
  $p \leftarrow$ `get_global_id(0)`
  if $B[p]$ **then**
    $k[p] \leftarrow 2n$

kernel $\textsc{Branch}_{k>1}$(**global** $B_i[n], B_j[n], k[n]$)
  $p \leftarrow$ `get_global_id(0)`
  $B_i[p] \leftarrow B_j[p] \wedge k[p] > 1$

kernel $\textsc{Branch}_{p<k}$(**global** $B_i[n], B_j[n], k[n]$)
  $p \leftarrow$ `get_global_id(0)`
  $B_i[p] \leftarrow B_j[p] \wedge p < k[p]$

kernel $\textsc{Field-Assign}_{p,A[2p]+A[2p+1]}$(**global** $\Gamma[2n], A[2n], B[n]$)
  $p \leftarrow$ `get_global_id(0)`
  if $B[p]$ **then**
    $\Gamma[p] \leftarrow A[p] + A[2p+1]$

kernel $\textsc{Assign}_{\frac{k}{2}}(k[n], B[n])$
  $p \leftarrow$ `get_global_id(0)`
  if $B[p]$ **then**
    $k[p] \leftarrow \frac{k[p]}{2}$

4.2.2 Listen-Methode

In der zweiten Methode werden dagegen die Mengen der aktiven Prozessoren $T_0, \ldots, T_{d-1}$ direkt als Listen $L_0, \ldots, L_{d-1}$ gespeichert. Da diese Listen eine maximale Länge von n haben, werden sie als Felder mit einer Kapazität von n Elementen realisiert, wobei die eigentliche Länge in der gesonderten Variable n_i gespeichert wird. Zuweisungen können nun direkt die Länge der aktuellen Liste n_k für ihre Worksize verwenden. Innerhalb eines Kernels wird die Prozessor-ID nicht mehr über die OpenCL-Funktion `get_global_id(0)` bestimmt, sondern aus der aktuellen Liste L_k gelesen:

$$p \leftarrow L_k[\texttt{get\_global\_id(0)}]$$

Verzweigungen

Schwieriger gestaltet sich nun jedoch die Auswertung von Verzweigungen. Da sich hier die Menge der aktiven Prozessoren verkleinern kann, würde bei einem naiven Löschen dieser Prozessoren aus der Liste Lücken entstehen, welche dann zu Problemen führen, wenn ein Prozessor seine Prozessor-ID bei einer Zuweisung bestimmen möchte. Es ist daher notwendig die Listen *kompakt* zu halten, sodass keine Lücken zwischen aktiven Prozessoren sind. Auch muss die neue Länge dieser Liste bestimmt werden, damit in folgenden Zuweisung die richtige Worksize verwendet werden kann.

Erneut kann dieses Problem leicht sequentiell im Host-Programm gelöst werden, jedoch würde dies wieder einen linearer Komplexitätszuwachs pro Verzweigung bedeuten. Allerdings besteht auch hier die Möglichkeit durch Parallelisierung eine bessere Lösung zu finden. Die Idee dabei ist, dass jeder Prozessor parallel die Anzahl, der von ihm aus linken (d. h. mit kleinerer ID) aktiven Prozessoren bestimmt und sich damit an die korrekte Stelle in der Ergebnisliste kopiert. Dieses Summieren aller linken Prozessoren kann auch als *Präfixsumme* aufgefasst werden, auf welche in Abschnitt 5.2 inklusive eines parallelen Algorithmus eingegangen wird. Schematisch findet dann die Verkleinerung einer Liste wie folgt statt:

1. Für die aktiven Prozessoren (1) wird das Prädikat ausgewertet und in (2) zwischengespeichert.
2. Auf (2) wird die Präfixsumme ohne den eigenen Wert berechnet und in (3) gespeichert.
3. Prozessoren (1), die das Prädikat erfüllen (2), werden in die neue Liste (4) an den Index übernommen, der in (3) steht.
4. Die Länge der neuen Liste ergibt sich aus der Summe der letzten beiden Zellen von (2) und (3).

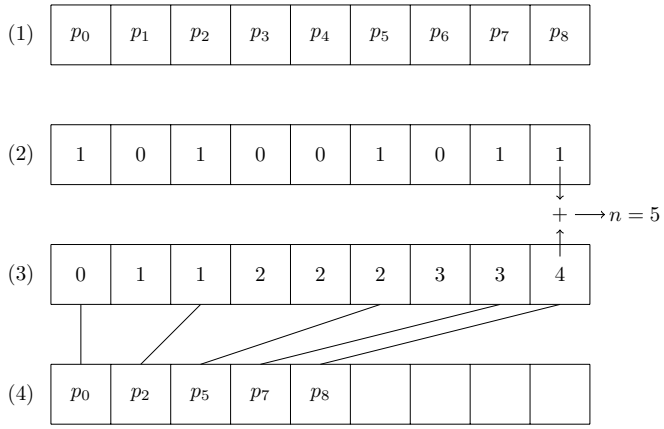


Abbildung 4.1: Beispiel der Anwendung einer Verzweigung

Dieser Algorithmus wird durch die Funktion $\textsc{Filter}_\alpha(L_i, L_j, n_j, \Psi(\alpha)) \rightarrow n_i$ beschrieben. Sie führt obige Schritte auf den ersten n_j Elementen der Eingabeliste L_j aus und kopiert die n_i Elemente, die das Prädikat α erfüllen, in die Ergebnisliste L_i.

Eine Verzweigung $T_i \leftarrow \{p \in T_j \mid \alpha(p) = true\}$ lässt sich somit durch $n_i \leftarrow \textsc{Filter}_\alpha(L_i, L_j, n_j, \Psi(\alpha))$ im Host-Programm realisieren.

Schleifen

Das Auswerten der Schleifenbedingung ist dagegen, im Vergleich zur Verzweigung, trivial. Da zu jedem Zeitpunkt die Anzahl n_k der aktiven Prozessoren im Host-Programm bekannt ist, kann als Schleifenbedingung einfach $n_k > 0$ werden, welches keine weitere Berechnung erfordert.

Diskussion

Wie auch die Bitset-Methode, ermöglicht die Listen-Methode alle erforderlichen PRAM-Ausdrücke und kann somit beliebige PRAM-Algorithmen realisieren. Dabei können auch einige Probleme der Bitset-Methode gelöst werden. Durch die kompakte Listendarstellung werden die Worksizes möglichst klein gehalten und Warps führen nun nur wirklich aktive Prozessoren aus. Auch lässt sich dadurch der Test, ob noch aktive Prozessoren vorhanden sind, im Vergleich zur Bitset-Methode deutlich einfacher umzusetzen. Dafür ist jedoch die Ausführung einer Verzweigung deutlich schwieriger und bringt, wie die Schleife der Bitset-Methode, einen zu der Prozessoranzahl logarithmischen Komplexitätszuwachs mit sich. Da jedoch Verzweigungen in der Regel häufiger als Schleifen auftreten, lässt sich argumentieren, dass die Listen-Methode hier im Nachteil ist.

Beispiel

Wie auch bei der Bitset-Methode, zeigt Algorithmus 5 das Ergebnis, wenn das Meta-Programm aus Algorithmus 2 durch die Listen-Methode implementiert wird.

Algorithmus 5 Algorithmus 2 implementiert durch die Listen-Methode

$A[2n], k[n]$
$L_0[n], L_1[n], L_2[n]$
$\Gamma[2n]$
$n_0 \leftarrow n$
enqueue $\textsc{Assign}_{2n}(k, L_0)$ **on** $\{0, \ldots, n_0 - 1\}$
$n_1 \leftarrow \textsc{Filter}_{k>1}(L_1, L_0, n_0, k)$
while $n_1 > 0$ **do**
  $n_2 \leftarrow \textsc{Filter}_{p<k}(L_2, L_1, n_1, k)$
  enqueue $\textsc{Field-Assign}_{p,A[2p]+A[2p+1]}(\Gamma, A, L_2)$ **on** $\{0, \ldots, n_2 - 1\}$
  swap $\Gamma \leftrightarrow A$
  enqueue $\textsc{Assign}_{\frac{k}{2}}(k, L_1)$ **on** $\{0, \ldots, n_1 - 1\}$
  $n_1 \leftarrow \textsc{Filter}_{k>1}(L_1, L_1, n_1, k)$

kernel $\textsc{Assign}_{2n}$(**global** $k[n], L[n]$)
  $p \leftarrow L[\texttt{get\_global\_id(0)}]$
  $k[p] \leftarrow 2n$

kernel $\textsc{Field-Assign}_{p,A[2p]+A[2p+1]}$(**global** $\Gamma[2n], A[2n], L[n]$)
  $p \leftarrow L[\texttt{get\_global\_id(0)}]$
  $\Gamma[p] \leftarrow A[p] + A[2p + 1]$

kernel $\textsc{Assign}_{\frac{k}{2}}(k[n], L[n])$
  $p \leftarrow L[\texttt{get\_global\_id(0)}]$
  $k[p] \leftarrow \frac{k[p]}{2}$

4.3 Optimierung

Die bisherigen Verfahren hatten eine allgemeine Implementierung des Meta-Programms zu Ziel. Auf Grund dieser Allgemeinheit, führen diese aber oft zu unübersichtlichen und unschönen Programmen (siehe Algorithmus 4 und 5). Dieser Abschnitt stellt daher einige Optimierungen vor, die Redundanzen entfernen und das resultierende Programm übersichtlicher zu machen.

4.3.1 Zusammenfassung unabhängiger Zuweisungen

Naheliegend ist es, voneinander unabhängige Zuweisungen zusammenzufassen. Zwei Zuweisungen

$$S_1 : \textsc{Assign}_\alpha(x, \Psi(\alpha))$$
$$S_2 : \textsc{Assign}_\beta(y, \Psi(\beta))$$

können zusammengefasst werden, wenn

1. sie aneinander grenzen, d. h. es gibt keine Anweisung, die zwischen S_1 und S_2 steht.
2. die Abhängigkeiten und beschriebenen Variablen jeweils gegenseitig disjunkt sind. Das heißt, Variablen aus $\Psi(\alpha)$ werden in S_2 und Variablen aus $\Psi(\beta)$ werden in S_1 jeweils nicht beschrieben. Für Felder kann diese Anforderung jedoch leicht gelockert werden. Hier muss dies nur für jede Speicherzelle einzeln gelten.

Gelten diese Bedingungen, so können beide Zuweisungen innerhalb eines Kernels geschehen, da zwischen ihnen keine Speicherkonflikte mehr bestehen und eine globale Synchronisierung damit nicht mehr notwendig ist.

---

enqueue $\textsc{Assign}_\alpha(x, \Psi(\alpha))$ **on** T_k
enqueue $\textsc{Assign}_\beta(y, \Psi(\beta))$ **on** T_k

---

kernel $\textsc{Assign}_\alpha($**global** $x[n], \Psi(\alpha))$
  $p \leftarrow$ `get_global_id(0)`
  $x[p] \leftarrow \alpha(p)$
kernel $\textsc{Assign}_\beta($**global** $y[n], \Psi(\beta))$
  $p \leftarrow$ `get_global_id(0)`
  $y[p] \leftarrow \beta(p)$

$\Downarrow$

$$\Downarrow$$

enqueue $\textsc{Assign}_{\alpha,\beta}(x, y, \Psi(\alpha), \Psi(\beta))$ **on** T_k

kernel $\textsc{Assign}_{\alpha,\beta}$(**global** $x[n], y[n], \Psi(\alpha), \Psi(\beta)$)
    $p \leftarrow$ `get_global_id(0)`
    $x[p] \leftarrow \alpha(p)$
    $y[p] \leftarrow \beta(p)$

Diese Optimierung kann nun solange ausgeführt werden, bis es keine unabhängigen Zuweisungen mehr gibt. Für eine zusammengesetzte Zuweisung muss dabei beachtet werden, dass sie mehr als eine Variable beschreibt und als Abhängigkeiten die Zusammenfassung aller in ihr berechneten Ausdrücke hat.

4.3.2 Double-Buffering entfernen

Wie bereits in Abschnitt 4.2 erläutert, kann eine Feld-Zuweisung

$$A[\alpha(p)] \leftarrow \beta(p)$$

im generellen Fall nicht direkt ausgeführt werden, sondern muss einen Zwischenspeicher verwenden. Kann jedoch gezeigt werden, dass für alle p gilt, dass kein anderer Prozessor in α oder β auf $A[\alpha(p)]$ zugreift, so kann der Zwischenspeicher übersprungen werden und das Ergebnis direkt in A geschrieben werden. Es können dabei keine Speicherkonflikte mehr auftreten, da jede Speicherzelle die von einem Prozessor beschrieben wird, von keinem anderen Prozessor gelesen wird. Somit kann folgende Optimierung angewendet werden, bei der der Hilfsspeicher Γ entfällt.

global $\Gamma[\cdot]$
enqueue $\textsc{Field-Assign}_{\alpha,\beta}(\Gamma, \Psi(\alpha), \Psi(\beta))$ **on** T_k
swap $\Gamma \leftrightarrow A$

kernel $\textsc{Field-Assign}_{\alpha,\beta}$(**global** $\Gamma[\cdot], \Psi(\alpha), \Psi(\beta)$)
    $p \leftarrow$ `get_global_id(0)`
    $\Gamma[\alpha(p)] \leftarrow \beta(p)$

$$\Downarrow$$

enqueue $\textsc{Field-Assign}_{\alpha,\beta}(A, \Psi(\alpha), \Psi(\beta))$ **on** T_k

kernel $\textsc{Field-Assign}_{\alpha,\beta}$(**global** $A[\cdot], \Psi(\alpha), \Psi(\beta)$)
    $p \leftarrow$ `get_global_id(0)`
    $A[\alpha(p)] \leftarrow \beta(p)$

Ein einfacher Fall, bei dem dieses Kriterium gilt, ist wenn $\alpha = p$ und in jedem Vorkommen von A in β, auch nur auf den Index $A[p]$ zugegriffen wird (z. B. $A[p] \leftarrow A[p] + B[p]$). Hier verwendet ein Prozessor dann nur den jeweils eigenen Index und es können somit keine Speicherkonflikte mit anderen Prozessoren auftreten.

Es gibt jedoch auch komplexere Ausdrücke, in denen diese Bedingung erfüllt wird, wie zum Beispiel $A[2p] \leftarrow A[2p] + A[2p+1]$. Wichtig ist hier die Erkenntnis, dass $A[2p+1]$ von keinem anderen Prozessor beschrieben wird, da auf der linken Seite nur Speicherzellen mit geradem Index beschrieben werden.

Je nach Algorithmus kann es daher Sinn machen, das Speicherlayout leicht zu modifizieren, um dieses Kriterium zu erfüllen.

4.3.3 Vereinfachung von Verzweigungen

Ein häufiger Fall, gerade nachdem Zuweisungen zusammengefasst wurden, ist, dass in einer Verzweigung nur eine einzige Zuweisung geschieht. Die durch die Verzweigung entstehende Menge an aktiven Prozessoren wäre daher nur für einen Kernel relevant. Eine unter Umständen aufwendige Berechnung dieser Menge im Host-Programm macht daher keinen Sinn. Effizienter ist es dagegen, den Test ob ein Prozessor die Zuweisungen ausführt, direkt im Zuweisungskernel umzusetzen. Hierfür muss lediglich gelten, dass die in der Zuweisung beschriebenen Variablen nicht in der Bedingung der Verzweigung verwendet werden, um zu verhindern, dass ein Speicherkonflikt entsteht.

Umgesetzt werden kann dies am Beispiel der Bitset-Methode wie folgt:

---

enqueue $\textsc{Branch}_\alpha(B_{k+1}, B_k, \Psi(\alpha))$ **on** $\{0, \ldots, n-1\}$
enqueue $\textsc{Assign}_\beta(x, \Psi(\beta), B_{k+1})$ **on** $\{0, \ldots, n-1\}$
{Weiter auf Programmebene k}

---

kernel $\textsc{Branch}_\alpha$(**global** $B_i[n], B_j[n], \Psi(\alpha)$)
    $p \leftarrow$ `get_global_id(0)`
    $B_i[p] \leftarrow B_j[p] \wedge \alpha(p)$

kernel $\textsc{Assign}_\beta$(**global** $x, \Psi(\beta), B[n]$)
    $p \leftarrow$ `get_global_id(0)`
    if $B[p]$ **then**
        $x \leftarrow \beta(p)$

$\Downarrow$

$\Downarrow$

enqueue CONDITIONAL-ASSIGN$_{\alpha,\beta}(A, \Psi(\alpha), \Psi(\beta), B_k)$ **on** $\{0, \ldots, n-1\}$

kernel CONDITIONAL-ASSIGN$_{\alpha,\beta}(x, \Psi(\alpha), \Psi(\beta), B[n])$
    $p \leftarrow$ `get_global_id(0)`
    if $B[p] \wedge \alpha(p)$ **then**
        $x \leftarrow \beta(p)$

4.3.4 Meta-Variablen auslagern

Eine Variable wird als *Meta*-Variable bezeichnet, wenn eine Zuweisungen dieser Variable immer von allen Prozessoren ausgeführt wird und der beschriebene Wert unabhängig von p ist. Das bedeutet, dass der Wert dieser Variable zu jedem Zeitpunkt über alle Prozessoren identisch ist. Häufig tritt dies bei Zählvariablen in Schleifen auf.

In diesen Fällen lässt sich die Variable aus den Kernels in das Host-Programm verschieben. Demnach werden nun für Zuweisungen dieser Variable keine Kernels mehr benötigt, sondern sie finden durch eine normale Variablenzuweisung im Host-Programm statt. An Stellen, in denen die Variable in einem Ausdruck verwendet wird, übergibt das Host-Programm den aktuellen Wert der Variable als ein Kernel-Argument.

In folgendem Beispiel ist diese Optimierung zu sehen:

$\{k$ ist Meta-Variable$\}$
enqueue ASSIGN$_{2k}(k)$ **on** T_k
enqueue FIELD-ASSIGN$_{p,k\cdot p}(A, k)$ **on** T_k

kernel ASSIGN$_{2k}$(**global** k)
    $p \leftarrow$ `get_global_id(0)`
    $k[p] \leftarrow 2k[p]$

kernel FIELD-ASSIGN$_{p,k\cdot p}$(**global** $A[\cdot], k[n]$)
    $p \leftarrow$ `get_global_id(0)`
    $A[p] \leftarrow k[p] \cdot p$

$\Downarrow$

$\Downarrow$

$\{k$ ist Meta-Variable$\}$
$k \leftarrow 2 \cdot k$
enqueue FIELD-ASSIGN$_{p,k\cdot p}(A, k)$ **on** T_k

---

kernel FIELD-ASSIGN$_{p,k\cdot p}$(**global** $A[\cdot]$, **private** k)
    $p \leftarrow$ `get_global_id(0)`
    $A[p] \leftarrow k \cdot p$

Da auf die Meta-Variable innerhalb des Kernels nur lesend zugegriffen wird, kann sie direkt im privaten Speicherbereich eines Workitems liegen.

4.3.5 Vereinfachung von Schleifen

Die letzte in dieser Arbeit betrachteten Optimierungsmöglichkeit behandelt die Vereinfachung von Schleifen. Bereits in Abschnitt 2.2 wurde auf das Problem von allgemeinen Schleifen eingegangen, dass diese nur ausgeführt werden können, indem innerhalb des Host-Programms überprüft wird, ob in dem jeweils nächsten Durchlauf noch ein Prozessor aktiv ist.

Das ist jedoch aus mehreren Gründen schlecht für die Effizienz auf GPUs. Zum einen erfordert es einen Informationsaustausch von den Kernels zum Host-Programm und damit verbunden einen teuren Speichertransfer. Zum anderen ist noch schlimmer die Tatsache, dass die GPU-Pipeline durch das Abfragen der aktiven Prozessoren im Host-Programm immer wieder unterbrochen wird. Dort kann der nächste Schleifendurchlauf erst begonnen werden, wenn die GPU den letzten Durchlauf tatsächlich ausgeführt hat, um dann zu entscheiden, ob mit dem nächsten begonnen wird. In dieser Zeit hat die GPU jedoch keine Kernel mehr, die sie ausführen kann. Für eine optimale Performance sollte die Command Queue aber immer gut gefüllt sein, damit der Scheduler der GPU die Arbeit bestmöglich auf die vorhandenen Stream-Prozessoren verteilen kann. Ist es daher möglich zu umgehen, dass im Host-Programm auf die GPU gewartet werden muss, so sollte eine deutliche Performancesteigerung erzielt werden.

Schleifen ins Host-Programm verschieben

Ursprünglich mussten in Schleifen diese Tests verwendet werden, da sonst keine andere Möglichkeit bestand zu bestimmen, ob noch Prozessoren aktiv waren. Ist es aber möglich, für eine konkrete Schleife eine Aussage über die maximale Anzahl von Schleifendurchläufen aus Sicht des Host-Programms zu treffen, so kann dies umgangen werden. Eine klassische Schleifenart, die dies ermöglicht, ist die for-Schleife, welche über eine Zählvariable $\alpha(p)$ Schleifendurchläufe ausführt. Im

allgemeinen Fall kann α natürlich ein beliebiger Ausdruck sein, über dessen Wert im Host-Programm keine Aussage getroffen werden kann. Interessanter sind jedoch die Fälle, in denen α im Host-Programm eingeschränkt werden kann. Ist α beispielsweise nur von p abhängig, so kann statisch das p_{max} bestimmt werden, für das α maximal ist. Im Host-Programm kann nun einfach die Schleife mit $\alpha(p_{max})$ Durchläufen ausgeführt werden. Da sie den Wert des maximalen Prozessors verwendet, wird sichergestellt, dass kein anderer Prozessor noch mehr Durchläufe benötigt. Natürlich ist es immer noch notwendig, die aktiven Prozessoren innerhalb der Schleife zu bestimmen, um zu verhindern, dass Prozessoren zu viele Durchläufe ausführen.

Ist α dagegen nur von der Gesamtanzahl an Prozessoren n und Meta-Variablen abhängig, so kann auch dies entfallen, da α dann den gleichen Wert für alle p hat. Das Host-Programm führt eine solche Schleife dann einfach $\alpha(0)$ mal aus. Auch kann für eine solche Schleife die Bestimmung der aktiven Prozessoren entfallen, da alle Prozessoren bis zum letzten Durchlauf aktiv bleiben.

Schleifen in die Kernels verschieben

Ein anderer Ansatz versucht dagegen genau das Gegenteil zu erreichen und die Schleifen komplett in die Kernels zu verschieben. Die Idee dabei ist sehr ähnlich zu die der Vereinfachung von Verzweigungen. Wird innerhalb einer Schleife nur noch eine einzige Zuweisung ausgeführt, so kann die Schleife direkt innerhalb des Zuweisungskernels geschehen. Dazu müssen jedoch noch einige weitere Bedingungen erfüllt werden. Zum einen dürfen, wie bei der Verzweigung, die in der Schleifenbedingung verwendeten Variablen nicht innerhalb der Schleife beschrieben werden, und zum anderen müssen zwei aufeinanderfolgende Zuweisungen der Schleife nach Unterabschnitt 4.3.1 zusammenfassbar sein. Grund dafür ist, dass zwei Schleifendurchläufe nun nicht mehr durch eine globale Synchronisierung getrennt sind und zwei aufeinanderfolgende Zuweisungen damit keine Speicherkonflikte enthalten dürfen.

Kapitel 5

Implementierungen

In dem vorherigen Kapitel wurde eine allgemeine Implementierungsstrategie für PRAM-Algorithmen vorgestellt, mit der PRAMs beliebiger Größe simuliert werden können. Zwar ist es so theoretisch möglich, beliebige PRAM-Algorithmen relativ effizient zu implementieren, jedoch wird in den meisten Fällen die GPU-Architektur nicht optimal genutzt. In diesem Kapitel werden daher anhand einiger Problemstellungen, praktisch anwendbaren Implementierung gezeigt und zum Teil Anpassungen beschrieben, die eine effizientere Nutzung der GPU-Architektur erlauben.

Bemerkung:
Um die hier vorgestellten Algorithmen möglichst kurz zu halten, werden einige nebensächliche Aspekte, wie Speichertransfers zwischen dem globalen und lokalen Speicher und die Behandlung von nicht passenden Eingabegrößen vernachlässigt. In der tatsächlichen Implementierung sind diese jedoch vorhanden, sodass alle Algorithmen beliebige Eingabegrößen verarbeiten können. Auch werden die Schritte vom PRAM-Algorithmus zur GPU-Implementierung hier nicht im Detail nachvollzogen, da diese lediglich die Optimierungsregeln aus dem vorherigen Kapitel darstellen.

5.1 Reduktion

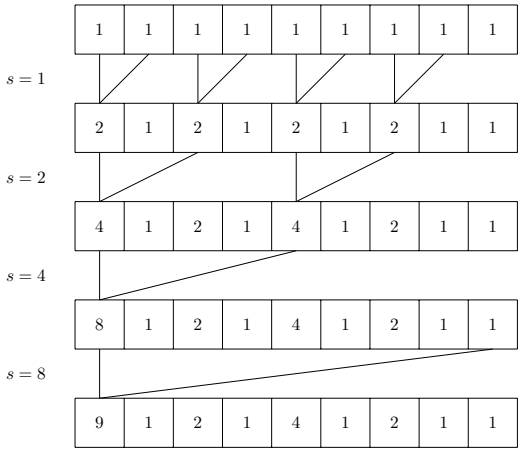
Eine der einfachsten Algorithmen auf Feldern ist die Reduktion einer Menge durch eine assoziativen Funktion $\oplus$ (wie Addition oder Multiplikation). Anwendung findet diese Funktion auch in dieser Arbeit in Unterabschnitt 4.2.1, in der sie verwendet wurde, um zu testen ob noch alle Prozessoren aktiv sind (Oder-Reduktion). Eine naheliegende parallele Brechungsstrategie für diese Art von Funktionen ist die Verwendung von Berechnungsbäumen. Die Idee ist hier, dass durch die Assoziativität der Funktion unabhängige Teilergebnisse berechnet werden können, die in

einem Baum schrittweise zusammengeführt werden, sodass in der Wurzel das Gesamtergebnis der Reduktion steht. Als Ebene im Berechnungsbaum, werden dann alle Knoten bezeichnet, die die gleiche Distanz zur Wurzel haben. Die Berechnung dieser Knoten in einer Ebene kann parallel von einer EREW-PRAM geschehen. Insgesamt müssen so $\mathcal{O}(\log n)$ Ebenen berechnet werden, falls der Baum balanciert ist. Algorithmus 6 zeigt einen naive Implementierung von Algorithmus 1, bei der Optimierung des vorherigen Kapitel bereits angewendet wurden.

Algorithmus 6 Naive GPGPU-Implementierung der Reduktion

```
1: procedure REDUCE(A[n])
2:     s ← 2
3:     while ⌈2n/s⌉ > 1 do
4:         enqueue REDUCESTEP(A, s) on 0 ... ⌈n/s⌉
5:         s ← 2s
6: kernel REDUCESTEP(global A[·], private s)
7:     a ← 2s · get_global_id(0)
8:     b ← a + s
9:     if b < n then
10:        A[a] ← A[a] ⊕ A[b]
```

Ein Unterschied zu dem ursprünglichen PRAM-Algorithmus ist jedoch, dass die Ebenen des Berechnungsbaums mit Hilfe eines Versatzes s (engl. *stride*) adressiert werden. Das bedeutet, dass zwei logisch aufeinanderfolgende Elemente durch s-1 dazwischenliegende Elemente getrennt sind. In diesem Fall ist das i-te Element auf der k-ten Berechnungsebene in $A[i \cdot 2^k]$ zu finden. Der Versatz verdoppelt sich somit in jeder Ebene. Durch diese Anpassung kann die in Unterabschnitt 4.3.2 beschriebene Optimierung angewendet werden und auf ein Double-Buffering verzichtet werden. Einfacher ist dies an einem Beispiel zu sehen:



Hier wird deutlich, dass die von einem Prozessor beschriebenen Speicherzellen auch nur von ihm gelesen werden.

Verbesserung

Betrachtet man den Algorithmus, so fällt auf, dass dieser, wie für PRAM-Algorithmen üblich, nur die globale Speicherdimension nutzt: In einem Schritt berechnet jeder Prozessor die Reduktion von zwei Elementen und schreibt das Ergebnis wieder zurück in den globalen Speicher. Das ist jedoch für die Effizienz des Algorithmus auf GPUs nicht optimal: Zum einen wird für jede Ebene des Reduktionsbaum ein neuer Kernel gestartet, was zu gewissen Zusatzkosten führt, da der Scheduler diese Ausführung planen muss. Zum anderen wird ausschließlich der globalen Speicher verwendet, der ja wie in Abschnitt 3.2 beschrieben, verhältnismäßig hohe Zugriffskosten hat.

Workgroups nutzen

Eine bessere Strategie ist es daher, die lokale Dimension der GPU-Architektur zu nutzen. Um die Workgroups mit in den Berechnungsprozess einzubeziehen, muss das Gesamtproblem in gleichartige Teilprobleme zerlegt werden, die dann von den Workgroups gelöst werden, indem sie die lokale Synchronisierungsmöglichkeiten ausnutzen. Hier können dann, wie bereits angekündigt, direkte PRAM-Implementierungen innerhalb einer Workgroup zum Einsatz kommen. Zu Beachten ist jedoch, dass auf Grund der maximalen Größe der Workgroups, die Größe dieser Teilprobleme beschränkt ist. Für den Entwurf solcher Algorithmen ist es daher einfacher die Workgroup-Größe als externen Parameter zu sehen, der durch den Algorithmus nicht modifiziert werden kann. Im Folgenden wird diese Größe mit m beschrieben.

Glücklicherweise ist eine solche Zerlegung im Fall der Reduktion leicht möglich. Anstatt in einem Knoten des Berechnungsbaums nur jeweils 2 Elemente miteinander zu reduzieren, werden nun m Elemente durch eine Workgroup reduziert. Die Reduktion der m Elemente innerhalb eine Workgroup kann wiederum durch einen internen Berechnungsbaum geschehen, der jedoch auf Grund der lokalen Synchronisierungsmöglichkeiten in einem Kernel stattfinden kann. Die genaue Implementierung dieser Workgroup-Reduktion spielt dabei aber keine Rolle für den äußeren Berechnungsbaum. So können auch beispielsweise die ab OpenCL 2.0 zur Verfügung stehenden Workgroup-Funktionen `work_group_reduce` genutzt werden, um diese Aufgabe zu lösen.

Sequentiellen Anteil hinzufügen

Diese Idee kann nun noch weiter ausgeführt werden, um auch die private Dimension miteinzubeziehen. Ziel ist es hierbei, die Anzahl an Elementen die von einer Workgroup bearbeitet werden können, zu erhöhen, damit der lokale Speicher besser ausgenutzt wird. Da jedoch die Anzahl der Prozessoren m nicht weiter erhöht werden kann, besteht eine andere Möglichkeit darin, die Anzahl der Elemente zu erhöhen, indem jeder Prozessor einen gewissen sequentiellen Berechnungsanteil durchführt. Dazu führt ein Prozessor zunächst auf c Elementen sequentiell die private Reduktion aus und benutzt dieses Ergebnis dann für die lokale Reduktion innerhalb der Workgroup. Aus Sicht der Workgroup werden nun jeweils $m \cdot c$ Elemente reduziert.

Durch diese geschachtelte Berechnung können die oben genannten Probleme gelöst werden: Der erhöhte Verzweigungsgrad im globalen Berechnungsbaum führt zu einer Verringerung der Tiefe, was demzufolge auch die Anzahl der benötigten Kernel-Ausführungen verringert. Dadurch, dass ein Großteil der Berechnungen nun auf der lokalen und privaten Ebene stattfindet, wird auch der Zugriff auf den globalen Speicher reduziert. Konkret lässt sich die Anzahl an Speicherzugriffen in einem Berechnungsbaum bei einer Eingabegröße n mit Verzweigungsgrad k durch

$$\text{MemAccess}(n, k) = \sum_{i=0}^{\log_k n} \frac{n}{k^i} + \frac{n}{k^{i+1}}$$

berechnen. In jedem Berechnungsschritt müssen alle Elemente in dieser Ebene gelesen werden (erster Summand) und alle Elemente für die nächste Ebene wieder geschrieben werden (zweiter Summand). Dies summiert sich über alle Ebenen des Berechnungsbaums. Es ist zu sehen, dass mit steigendem k die Anzahl der Speicherzugriffe sinkt.

Zusammenfassung

Insgesamt findet dann die Berechnung wie folgt statt:

1. Jede Workgroup lädt die $m \cdot c$ Elemente, für die sie zuständig ist, in ihren lokalen Speicher.

2. Jeder der m Prozessoren in einer Workgroup berechnet auf den c Elementen, für die er zuständig ist, die sequentielle Reduktion.

3. Die Ergebnisse der privaten Reduktionen der Prozessoren werden dann innerhalb der Workgroup über einen Reduktionsbaum weiter reduziert.

4. Das Gesamtergebnis der Workgroup wird wieder in den globalen Speicher zurückgeschrieben.

5. Wenn im letzten Schritt noch mehr als eine Workgroup aktiv war, dann wird der Versatz mit $m \cdot c$ multipliziert und bei dem ersten Schritt wieder begonnen.

Implementierung

Die eigentliche Implementierung ist nun sehr ähnlich wie in Algorithmus 6.

Algorithmus 7 Verbesserte GPU-Reduktion

```
1: procedure REDUCE(A[n])
2:     s ← m · c
3:     while ⌈(m·c·n)/s⌉ > 1 do
4:         enqueue REDUCESTEP(A, L[m · c], s) on {0, ..., ⌈n/s⌉}/m
5:         s ← s · m · c
6: kernel REDUCESTEP(global A[·], local L[m · c], private s)
7:     LOADCHUNK(A, L, s)
8:     LOCALREDUCE(L)
9:     SAVERESULT(A, L[0], s)
```

Durch den REDUCESTEP-Kernel werden mit zunehmenden Versatz die Ebenen des Berechnungsbaums ausgeführt. Hier wird jedoch zusätzlich noch der lokaler Speicher der Größe $m \cdot c$ übergeben und die Ausführung wird in Workgroups der Größen m gestartet. Innerhalb des Kernels wird mit der Hilfsfunktion LOADCHUNK zunächst der Bereich des globalen Speichers, den die Workgroup bearbeitet, in den lokalen Speicher geladen. OpenCL stellt dabei praktischerweise die Workgroup-Funktion `async_work_group_strided_copy` bereit, die Daten mit einem Versatz vom globalen in den lokalen Speicher kopiert. Ähnlich dazu wird am Ende des Kernels mit SAVERESULT der lokale Speicher wieder an die passende Stelle im globalen Speicher geschrieben. Da diese Funktionen jedoch nur den Speichertransfer steuern, werden sie hier nicht weiter erläutert. Der eigentlich relevante Teil liegt in der Funktion LOCALREDUCE. Sie führt die private und lokale Reduktion auf den Daten im lokalen Speicher aus.

Algorithmus 8 Lokale Reduktion (GPU)

```
1:  procedure LocalReduce(L[m · c])
2:      p ← get_local_id(0)

3:      a ← L[p · c]
4:      for i ← 1 ... c − 1 do
5:          a ← a ⊕ L[p · c + i]
6:      L[p · c] ← a
7:      barrier(CLK_LOCAL_MEM_FENCE)

8:      s ← c
9:      while ⌈(m·c)/(2s)⌉ > 1 do
10:         l ← 2s · p
11:         r ← l + s
12:         if r < m · c then
13:             A[l] ← A[l] ⊕ A[r]
14:         s ← 2s
15:         barrier(CLK_LOCAL_MEM_FENCE)
```

1. Zeile 3 - 7: Jeder Prozessor reduziert sequentiell die Elemente $L[p \cdot c]$ bis $L[p \cdot c + c - 1]$ und speichern das Ergebnis in $L[p \cdot c]$.

2. Zeile 8 - 15: Die Ergebnisse der privaten Reduktion werden innerhalb der Workgroup weiter reduziert. Dazu wird eine PRAM-Variante von Algorithmus 6 verwendet. Um zu erreichen, dass diese nur auf den Ergebnissen der privaten Reduktion arbeitet, wird der Versatz mit c initialisiert.

Ein weiterer wichtiger Teil sind die `barrier`-Anweisungen, die die lokale Synchronisation nach jedem Speicherzugriff sicherstellen. Sie müssen nach der privaten Reduktion und nach jedem Schritt der lokalen Reduktion geschehen, damit die Änderungen eines Prozessors für die anderen sichtbar sind. Das Argument `CLK_LOCAL_MEM_FENCE` signalisiert dabei, dass sich die Synchronisierung nur auf den lokalen Speicher bezieht.

Evaluation

Als Evaluation der Optimierung wurde ein simpler Laufzeitvergleich zwischen der nicht optimierten Reduktion und der optimierten Variante mit unterschiedlichen Werten für m und c aufgestellt.

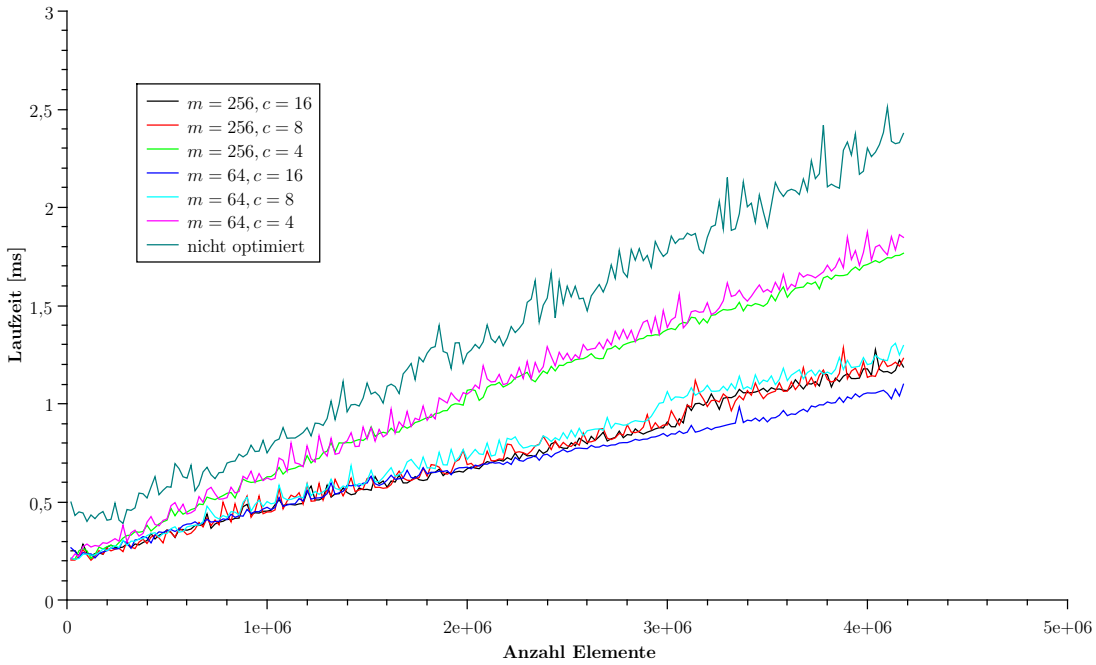


Abbildung 5.1: Vergleich der Reduktionsalgorithmen

Generell lässt sich erkennen, dass alle Algorithmen eine lineare Laufzeitkomplexität haben. Die logarithmische Komplexität, wie sie in einer idealen PRAM zu erwarten wäre, kann nicht erreicht werden. Der Grund dafür ist, dass die vom PRAM-Algorithmus geforderten n Prozessoren nicht vorhanden sind. Da es nur eine konstante Anzahl an gleichzeitig laufenden Workitems geben kann, ist auch nur eine Verbesserung der Laufzeit um einen konstanten Faktor möglich. Zwischen den Algorithmen ist zu sehen, dass die nicht optimierte Variante die höchste Laufzeit hat, wogegen die optimierte Variante mit $m = 64, c = 16$ die geringste Laufzeit hat. Die restlichen Parametervariationen führen aber auch zu ähnlichen Laufzeiten, mit der Ausnahme für $c = 4$, bei der der eine Verschlechterung erzielt wird.

Zwar sind die Ursachen für diese Laufzeitunterschiede sehr stark von der unterliegenden GPU-Architektur abhängig, jedoch kann trotzdem versucht werden, einige Vermutungen aufzustellen, warum diese auftreten. Dass für ein zu kleines c die Laufzeit schlechter ausfällt, kann die Vermutung bestätigen, dass eine bessere Auslastung des lokalen Speichers positiv für die Laufzeit ist. Dass wider Erwarten die Workgroup-Größe keinen signifikanten Einfluss auf die Laufzeit hat, kann in diesem Fall damit zusammenhängen, dass für diesen einfachen Kernel, die Erhöhung der Baumtiefe keinen signifikanten negativen Einfluss hat.

5.2 Präfixsumme

Ein eng mit der Reduktion verwandtes Problem ist die *Präfixsumme* (auch *Scan* genant). Die Präfixsumme einer Sequenz von Zahlen ist eine neue Sequenz, deren Wert an Stelle i die Reduktion aller Elemente aus der Ursprungssequenz mit Index kleiner oder kleiner-gleich i ist. Dabei wird zwischen der *inklusiven* Präfixsumme, welche den Wert an Stelle i beinhaltet, und der *exklusiven* Präfixsumme, welche nur die Werte mit Index kleiner i beinhaltet, unterschieden.

Sequenz	2	3	2	5	1	4
inklusive Präfixsumme	2	5	7	12	13	17
exklusive Präfixsumme	0	2	5	7	12	13

Es existieren mehrere PRAM-Algorithmen, die die Präfixsumme berechnen ([Ble90], [Mei15, Kapitel 2.4.2]), jedoch treten bei ihnen ähnliche Probleme wie bei der naiven Implementierung des Reduktion auf, sodass sie den lokalen Speicher nicht optimal nutzen. Am Vorbild der Optimierung der Reduktion wird daher nun ein auf die GPU-Architektur optimierter Präfixsummenalgorithmus entwickelt.

Zerteilung der Präfixsumme in Teilprobleme

Grundlage dieses Algorithmus ist erneut die Aufteilung des Gesamtproblems in konstant große Teilprobleme, die von den Workgroups lokal berechnet werden. Jedoch stellt sich dies im Fall der Präfixsumme als schwieriger heraus, da ein Element von allen linken Elementen (d. h. mit kleinerem Index), und damit auch den Workgroups, abhängig ist. Um dies zu lösen, durchläuft der Algorithmus zwei Phasen. Die erste Phase (Up-Phase) ist sehr ähnlich zu der in Berechnung der Reduktion. In ihr wird die Präfixsumme auf jeweils $m \cdot c$ Elementen in einer Workgroup durch einen Berechnungsbaum berechnet. Anders als bei der Reduktion werden aber alle Elemente wieder in den Speicher zurückgeschrieben (sie spielen in der nächsten Ebene jedoch keine Rolle mehr) und die Eingabe für die nächste Ebene ist nicht das erste, sondern das letzte Element. Erneut werden die Ebenen des Berechnungsbaums solange ausgeführt, bis nur noch eine Workgroup aktiv ist. Abbildung 5.2 zeigt die Up-Phase auf einer Beispielsequenz.

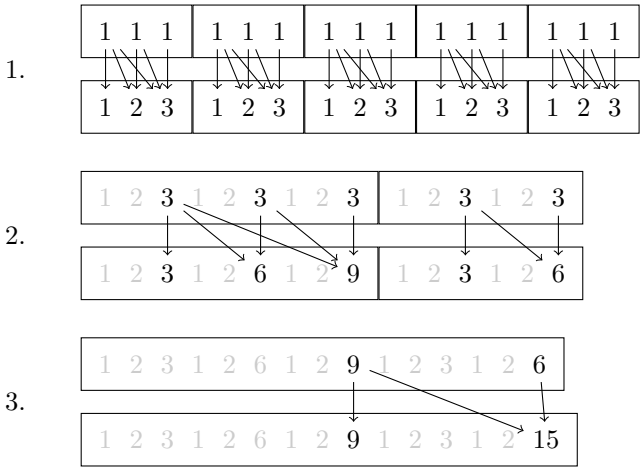


Abbildung 5.2: Beispield der Up-Phase bei $m \cdot c = 3$

Die jeweilige Ein- und Ausgabe der Berechnungsebenen sind untereinander aufgelistet. Die von einer Workgroup bearbeiteten Elemente werden von einem Rechteck umschlossen. Ausgegraute Elemente sind in der aktuellen Ebene nicht mehr aktiv. Pfeile zeigen die Reduktionen an, die von einer Workgroup ausgeführt werden.

Offensichtlich ist nach diesem Schritt die Präfixsumme noch nicht korrekt. Deswegen werden nun in der zweiten Phase (Down-Phase) diese Teilergebnisse korrigiert. Dazu wird der Berechnungsbaum, der in der ersten Phase konstruiert wurde, wieder herabgestiegen. Dabei reduziert eine Workgroup das letzte Element der von ihr aus links gesehen Workgroup auf alle, bis auf das letzte, ihrer eigenen Elemente. Die erste Workgroup überspringt diesen Schritt, da sie keine linke Workgroup hat. Abbildung 5.3 zeigt diesen Schritt auf dem Beispiel.

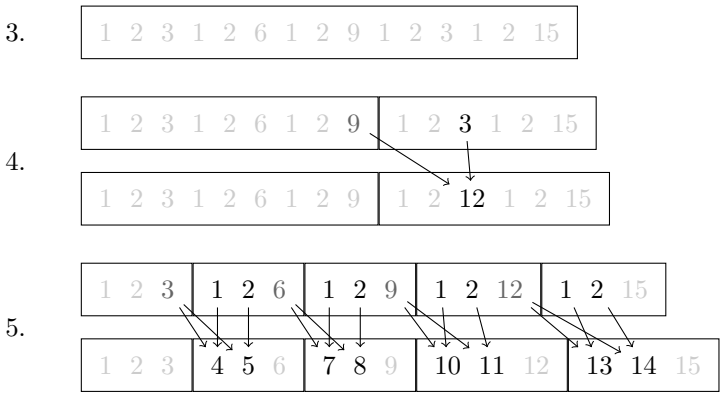


Abbildung 5.3: Beispiel der Down-Phase bei $m \cdot c = 3$

Nachdem diese Phase abgeschlossen ist (d. h. der Versatz ist wieder 1), besitzt jedes Element den korrekten globalen Präfixsummenwert. Auch gilt, dass während der Down-Phase keine Speicherkonflikte auftreten, da das jeweils letzte Elemente einer Workgroup, das von der rechten Workgroup gelesen wird, in der eigenen Workgroup nicht beschrieben wird.

Korrektheit

Um die Korrektheit des Algorithmus zu zeigen, werden nun die Begriffe der *globalen Korrektheit* und *lokalen Korrektheit* definiert. Ein Element der Eingabesequenz ist *global korrekt*, wenn es, bezogen auf die gesamte Sequenz, den korrekten Präfixsummenwert hat. Ein Element ist *lokal korrekt*, wenn es, bezogen auf die Teilsequenz der Elemente, die von den Elementen der Workgroup, in der das Element zuletzt aktiv war über eine Reduktionsbeziehung erreichbar sind, den korrekten Präfixsummenwert hat. Eine andere Betrachtung dieser Teilsequenzen ist die Menge der von diesem Knoten aus von erreichbaren Blattknoten im Berechnungsbaum.

Offensichtlich ist für die Korrektheit des Algorithmus eine globale Korrektheit aller Elemente erforderlich. Für diese wird jedoch zunächst gezeigt, dass nach der Up-Phase alle Elemente lokal korrekt sind und die Elemente, die im letzten Schritt aktiv waren, auch global korrekt sind.

Die erste Behauptung lässt sich durch eine Induktion über die Berechnungsebenen E_1 bis E_n zeigen. Elemente, die nur in E_1 aktiv waren, besitzen durch die lokale Präfixsumme den korrekten Wert bezogen auf die $m \cdot c$ Elemente ihrer Workgroup. Da dies die unterste Ebene ist, sind keine weiteren Elemente über eine Reduktionsbeziehung erreichbar und der Induktionsanfang gilt somit. Für den Induktionsschritt $k \to k+1$ ist die Kernerkenntnis, dass sich der Präfixsummenwert eines Elements auch aus bereits vorreduzierten Elementen berechnen kann. Für den Präfixsummenwert an Stelle i kann daher, anstatt alle Elemente $e_1 \dots e_i$ einzeln zu reduzieren, auch eine Reduktion von $b_1 \dots b_k$ stattfinden, solange in diesen genau die Reduktion aller Einzelelemente $e_1 \dots e_i$ enthalten ist.

Da als Eingabe in die Ebene E_{k+1} immer die letzten Elemente einer Workgroup aus der vorherigen Ebene E_k verwendet werden, haben diese, auf Grund der Eingenschaft, dass das letzte Element der Präfixsumme die Gesamtreduktion der Sequenz darstellt, die Reduktion aller $m \cdot c$ Elemente aus ihrer vorherigen Workgroup. Da in E_k die Induktionshypothese gilt, werden in der lokalen Präfixsumme in E_{k+1} wiederum lokal korrekte Ergebnisse berechnet, da diese die die vorreduzierten Ergebnisse aus E_k verwenden, welche insgesamt, alle in dem Teilbaum reduzierten Elemente beinhalten.

Die zweite Behauptung folgt dann aus der Tatsache, dass im letzten Schritt nur eine Workgroup aktiv war. Somit gilt, dass die $m \cdot c$ Elemente dieser Workgroup

die Wurzel des Berechnungsbaums darstellen, die daher gesamte Eingabesequenz erreichen können. In diesem Fall ist die oben gezeigte lokale Korrektheit gleich der globalen Korrektheit.

Dieser Sachverhalt wird auch nochmal in Abbildung 5.4 verdeutlicht.

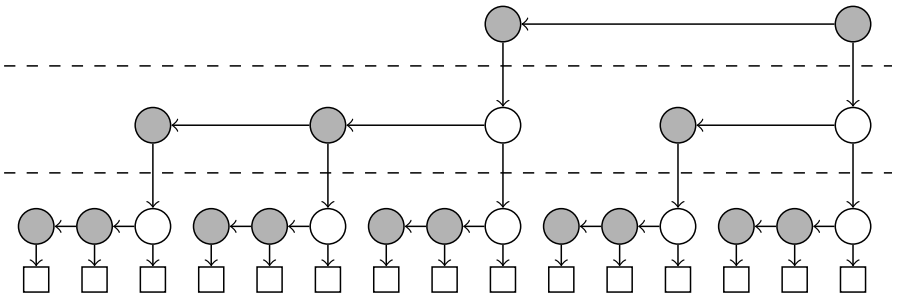


Abbildung 5.4: Berechnungsbaum nach der Up-Phase

Hier stellen die Kästchen die Werte der Eingabesequenz dar und Knoten stehen für einen zwischenzeitlichen Wert, der während einer Ebene des Berechnungsbaums erreicht wurde, die durch gestrichelte Linien gekennzeichnet werden. In Grau sind die Knoten, die zuletzt in einer Workgroup aktiv waren und bezeichnen daher den Wert, der endgültig im Feld steht. Pfeile zeigen die Reduktionsbeziehungen, sodass sich der Wert eines Knoten aus allen über die Pfeile erreichbaren Kästchen ergibt. Die lokale Korrektheit zeigt sich hier dadurch, dass jeder graue Knoten genau alle Kästchen in dem von ihm aufgespannten Teilbaum erreichen kann.

In der Down-Phase werden nun die lokal korrekten Elemente jeweils global korrekt gemacht. Für die globale Korrektheit fehlt ihnen lediglich die Gesamtreduktion aller der vom Teilbaum aus links gesehenen Elemente. Da die letzten Elemente der Up-Phase bereits global korrekt sind, können sie verwendet werden um die Elemente der nächsttieferen Ebene zu korrigieren. Da so auch alle in dieser Ebene aktiven Elemente global korrekt werden, kann mit gleicher Begründung Ebene für Ebene angepasst werden. Deutlich wird dieser Prozess in Abbildung 5.5.

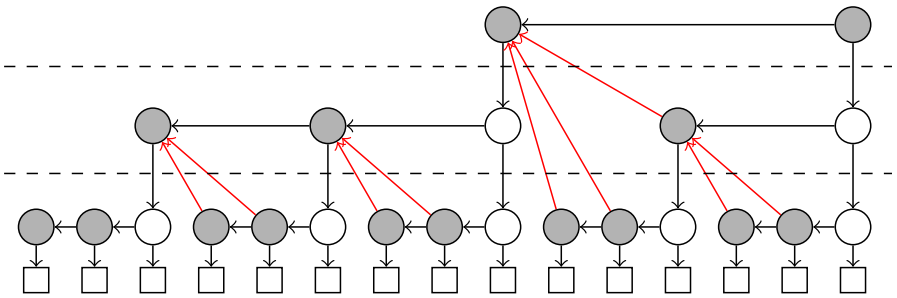


Abbildung 5.5: Berechnungsbaum nach der Down-Phase

Hier sind in Rot die durch die Down-Phase hinzugekommenen Reduktionsbeziehungen eingetragen. Deutlich wird nun, dass jeder graue Knoten genau alle von ihm aus linken Kästen erreichen kann und somit der Wert auch die Reduktion aller dieser Elemente darstellt. Somit sind alle Elemente global korrekt.

Implementierung

Die Implementierung dieses Algorithmus verwendet im wesentlichen zwei Kernels, welche einen Schritt der Up- bzw. Down-Phase ausführen. Das Host-Programm ruft diese lediglich nacheinander mit dem korrektem Versatz auf.

Algorithmus 9 GPU-Präfixsumme

```
1:  procedure PREFIXSUM(A[n])
2:      s ← 1
3:      while s < n/m do
4:          enqueue UPSTEP(A, L[m · c], s) on {0, ..., ⌈n/s⌉}/m
5:          s ← s · m · c
6:      while s ≥ 1 do
7:          enqueue DOWNSTEP(A, L[m · c], s) on {0, ..., ⌈n/s⌉}/m
8:          s ← s/(m·c)
9:  kernel UPSTEP(global A[n], local L[m · c], private s)
10:     LOADCHUNK(A, L, s)
11:     LOCALPREFIXSUM(L)
12:     SAVECHUNK(A, L, s)
13: kernel DOWNSTEP(global A[n], local L[m · c], private s)
14:     LOADCHUNK(A, L, s)
15:     x ←GETLEFTRESULT(A)
16:     p ← get_local_id(0)
17:     for i ← 0 ... c − 1 do
18:         if p · c + i < m · c − 1 then
19:             L[p · c + i] ← L[p · c + i] ⊕ x
20:     SAVECHUNK(A, L, s)
```

Der Kernel der Up-Phase hat große Ähnlichkeit zu dem Kernel des optimierten Reduktionsalgorithmus. Mit LOADCHUNK und SAVECHUNK werden die $m \cdot c$ Elemente einer Workgroup zwischen dem globalen und lokalen Speicher kopiert. In LOCALPREFIXSUM wird dann die Präfixsumme auf den lokalen Daten berechnet. Erneut kann hier die Implementierung frei gewählt werden und ähnlich wie bei der Reduktion bietet OpenCL hierfür aber auch die Workgroupfunktionen

`work_group_scan_inclusive` an. In dieser Implementierung wurde eine simple Adaption des PRAM-Algorithmus aus „Efficient Algorithms“, Kapitel 2.4.2 [Mei15] verwendet. Die Berechnung der lokalen Präfixsumme findet dann, wie in Algorithmus 10 zu sehen, in 3 Schritten ab:

Algorithmus 10 Berechnung der lokalen Präfixsumme

```
1:  function LocalPrefixsum(L[m · c])
2:      p ← get_local_id(0)

3:      a ← L[p · c]
4:      for i ← 1 ... c − 1 do
5:          a ← a ⊕ L[p · c + i]
6:          L[p · c + i] ← a
7:      barrier(CLK_LOCAL_MEM_FENCE)

8:      s ← 1
9:      while s < m do
10:         if p ≥ s then
11:             x ← L[(p + 1) · c − 1] ⊕ L[(p − s + 1) · c − 1]
12:         barrier(CLK_LOCAL_MEM_FENCE)
13:         if p ≥ s then
14:             L[(p + 1) · c − 1] ← x
15:         barrier(CLK_LOCAL_MEM_FENCE)
16:         s ← 2s

17:     if p ≠ 0 then
18:         for i ← 0 ... c − 1 do
19:             L[p · c + i] ← L[p · c + i] ⊕ L[p · c − 1]
```

1. Zeile 3 - 7: Jeder Prozessoren berechnet von $L[p{\cdot}c]$ bis $L[p{\cdot}c{+}c{-}1]$ sequentiell die Präfixsumme.

2. Zeile 8 - 16: Mit der Reduktion der Elemente eines Prozessors wird innerhalb der Workgroup die lokale Präfixsumme berechnet. Hierbei muss darauf geachtet werden, dass nur auf das letzte der c Elemente eines Prozessors zugegriffen wird.

3. Zeile 17 - 19: Die ersten $c - 1$ Elemente eines Prozessors wird mit Hilfe des letzten Wertes des vorherigen Prozessor auf den korrekten Präfixsummenwert korrigiert. Der erste Prozessor überspringt dies.

Der Kernel des Down-Schritts ist dagegen verhältnismäßig einfach. Erneut wird der entsprechende Bereich der Workgroup in den lokalen Speicher geladen. Zusätzlich wird nun aber auch mit GetLeftResult das Resultat der linken Workgroup des vorherigen Schrittes bestimmt. Dieses Ergebnis wird dann von jedem Prozessor auf die c Elemente des Prozessor reduziert. Hier ist darauf zu achten, dass das letzte Element $L[m \cdot c - 1]$ nicht beschrieben wird.

Evaluation

Wie auch bei dem Reduktionsalgorithmus wurde für die Optimierung der Präfixsumme ein Laufzeitvergleich zwischen den unterschiedlichen Algorithmen und Parameterkombinationen aufgestellt. Für den Vergleich zu einer nicht optimierten Variante, wurde der gleiche Algorithmus, der in Algorithmus 10 zur Berechnung der lokalen Präfixsumme innerhalb der Workgroup verwendet wurde, genommen und angepasst, um auf der Gesamtsequenz im globalen Speicher zu funktionieren.

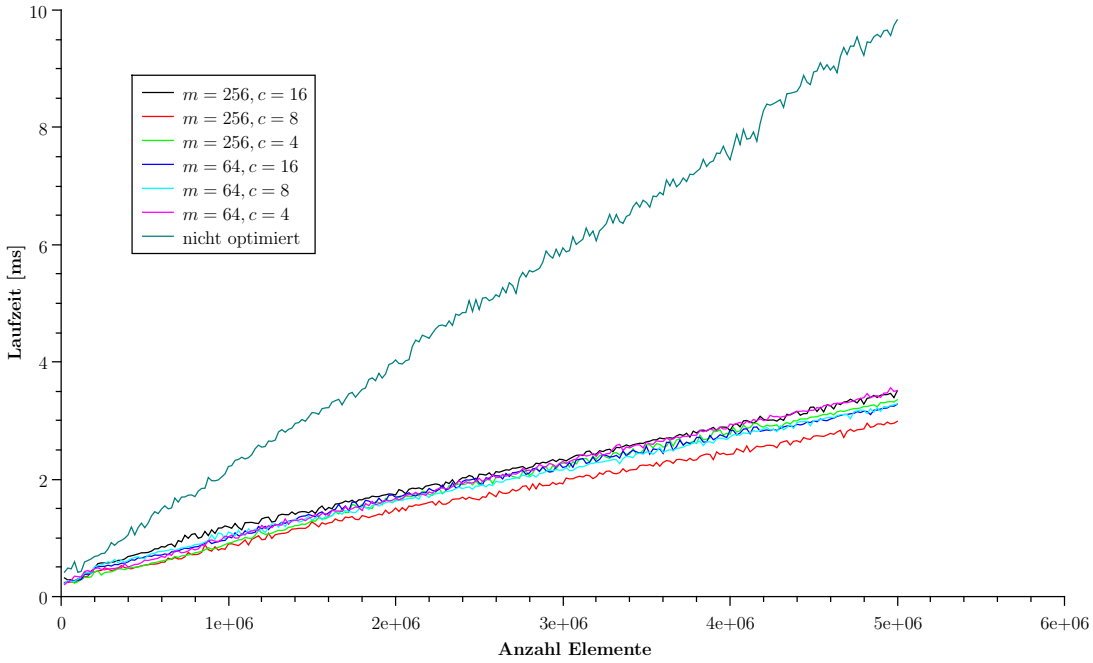


Abbildung 5.6: Vergleich der Prefixsummenalgorithmen

Erneut ist festzustellen, dass alle Varianten des optimierten Algorithmus ein besseres Laufzeitverhalten haben als die des nicht optimierten Algorithmus. Dieser Unterschied ist hier sogar noch deutlicher zu sehen als es bei der Reduktion der Fall

war. Grund dafür kann die Tatsache sein, dass die Präfixsumme aufwendiger zu berechnen ist und dass eine Optimierung somit deutlicher zum Ausdruck kommt.

Zwischen den einzelnen Parametervariationen der optimierten Variante sind erneut nur geringe Unterschiede zu sehen. Interessant ist jedoch, dass die beste Laufzeit für $m = 256$ und $c = 8$ erreicht wird. Diese Beobachtung passt zu der Vermutung, dass ein zu geringer Wert für c schlecht ist, da so der lokale Speicher nicht optimal ausgenutzt wird, ein zu hoher Wert aber auch nachteilig ist, da so die Parallelität der GPU nicht genutzt werden kann.

5.3 Sortierung

Das Sortieren einer Menge von Zahlen ist ein bekanntes Problem, welches eine sequentielle Laufzeitkomplexität von $\mathcal{O}(n \log n)$ hat. Um das Sortieren zu parallelisieren, werden häufig sogenannte Sortiernetzwerke verwendet, die eine feste Reihe von Vergleichen beschreiben die teils parallel ausgeführt werden können.

In dieser Arbeit wird dagegen eine parallele Variante des bekannten Merge-Sort Algorithmus implementiert, die in „Efficient Algorithms“, Kapitel 2.5 [Mei15] beschrieben wurde. Grundlage des Merge-Sort ist, dass jeweils zwei sortierte Teillisten zu einer wiederum sortierten Liste zusammengefügt werden (*merge*). Parallel lässt sich dieses Mergen zweiter Listen der Größe m und n mit $m + n$ Prozessoren in einer Laufzeit von $\mathcal{O}(\log \max(m, n))$ durchführen: Jeder Prozessor hat die Aufgabe, ein Element der beiden Listen an die korrekte Position der Ergebnisliste zu kopieren. Um diesen Index zu berechnen, muss er die Anzahl an Elementen finden die kleiner sind als der eigene Wert. Innerhalb der eigenen Liste ist dies leicht, da auf Grund der Sortierung genau $i - 1$ Elemente kleiner sind, wenn i der Index des Prozessors in der Liste ist. In der anderen Liste kann dank der Sortierung mittels der RANK-Funktion eine binäre Suche in $\mathcal{O}(\log n)$ durchgeführt werden. Als Eingabe erhält diese eine obere und untere Grenze und ein Vergleichselement x und berechnet damit die Anzahl an Elementen in der Teilsequenz, die kleiner als x sind.

```
procedure RANK(A[·], l, h, x, b)
    l_0 ← l
    while l ≤ h do
        i ← ⌊(h+l)/2⌋
        if compare(A[i], x) + b ≤ 0 then l ← i + 1
        else h ← i − 1
    return l_0 − l
```

Nun kann der Gesamtindex berechnet werden, indem i mit dem Ergebnis der

Rank-Funktion addiert wird. Ein Problem tritt jedoch auf, wenn in den Listen Elementen mehrfach vorkommen. In folgendem Beispiel berechnen zwei Elemente den gleichen Index und überschreiben sich dadurch.

Eingabe	1	3	3	2	3	6
Index	$0+0$	$1+1$	$2+1$	$0+1$	$1+1$	$2+3$
Ausgabe	1	2	3	3	?	6

Um dieses Problem zu umgehen, hat die RANK-Funktion einen weiteren Parameter b. Für $b = 0$ wird die Anzahl an Elementen echt kleiner, für $b = 1$ dagegen die Anzahl kleiner gleich x gesucht. Die Elemente der ersten Liste verwenden nun $b = 0$ und die der zweiten Liste $b = 1$. Somit wird sichergestellt, dass keine zwei Elemente der beiden Liste den gleichen Index berechnen.

Implementierung

Nun kann eine Gesamtberechnungsstrategie formuliert werden. Die Teillisten werden direkt hintereinander im Eingabefeld dargestellt. In einem ersten Schritt muss zunächst eine erste Sortierung der Teillisten stattfinden, damit der Merge-Schritt angewendet werden kann. Hierfür kann ein einfaches Sortiernetzwerk wie das Even-Odd-Sort, oder das effizientere Bitonic-Sort verwendet werden. Nach diesem Schritt sind die Teillisten $A[0] \dots A[m-1], A[m] \dots A[2m-1], \dots$ jeweils sortiert. Nun werden jeweils zwei nebeneinanderliegende Teillisten nach der oben beschriebenen Merge-Methode zusammengefügt, bis nur noch eine Teilliste vorhanden ist, welche die sortierte Gesamtsequenz darstellt. Alternativ kann auch der initiale Sortierschritt übersprungen werden und direkt mit Teillisten der Größe 1 angefangen werden. Jedoch müssen in diesem Fall mehr Merge-Schritte ausgeführt werden, was sich als schlechter für die Performance herausgestellt hat.

Algorithmus 11 Merge-Sort GPU

```
1: procedure SORT(A[n])
2:     enqueue INIT(A, L[m]) on {0 ... n − 1}/m
3:     k ← m
4:     while ⌈n/k⌉ > 1 do
5:         enqueue MERGESTEP(A, B, k) on {0 ... n − 1}
6:         swap A ↔ B
7:         k ← 2k
8: kernel MERGESTEP(global A[n], B[n], k)
9:     p ← get_global_id(0)
10:    y_0 ← 2k · ⌊p/2k⌋
11:    y_1 ← y_0 + k
12:    y_2 ← y_1 + k
13:    if p < y_1 then
14:        x ← p + RANK(A, y_1, y_2 − 1, A[p], 0)
15:    else
16:        x ← p − k + RANK(A, y_0, y_1 − 1, A[p], 1)
17:    B[x] ← A[p]
```

In der Implementierung wird zunächst mit dem INIT-Kernel die Vorsortierung berechnet. Danach werden mit dem MERGESTEP-Kernel jeweils zwei aneinanderliegende Teillisten zusammengefügt. Die Größe der Teillisten wird dabei in k übergeben, welche sich nach jedem Schritt verdoppelt. Da die Ergebnisse eines Merge direkt wieder in das Feld zurückgeschrieben werden und dabei andere Elemente überschreiben würden, kann das Double-Buffering nicht entfernt werden. Innerhalb eines Merge-Schrittes berechnen die Prozessoren mit y_0, y_1, y_2 Anfang, Mitte und Ende der nebeneinanderliegenden Teillisten. Die linke Liste läuft dann von $A[y_0]$ bis $A[y_1-1]$ und die rechte von $A[y_1]$ bis $A[y_2-1]$. Je nachdem, in welchen Bereich der Prozessor fällt, wird der Index in der Ergebnisliste x unterschiedlich berechnet und der aktuelle Wert an diese Stelle kopiert.

Graphen

Neben den einfachen linearen Feldern werden im Folgenden nun auch Graphenstrukturen betrachtet. Ein Graph wird dabei als eine Knotenmenge V und Kantenmenge $E \subseteq V \times V$ definiert. Die Kanten sind dabei ungerichtet und ungewichtet.

5.4 Distanzen in Graphen

Bei der Betrachtung von Graphen ist die Bestimmung von Knotendistanzen ein häufig auftretendes Problem. Die Distanz $\delta(u, v)$ zweier Knoten u, v ist definiert als die minimale Länge k einer Kantenfolge $(u, w_1), (w_1, w_2), \ldots (w_{k-1}, v)$, die die beiden Knoten verbindet.

In diesem Abschnitt wird eine parallele Variante des bekannten Floyd-Warschall-Algorithmus entwickelt, der die Distanzen aller Knotenpaare mit $|V|^2$ Prozessoren in einer Laufzeit von $\mathcal{O}(|V|)$ berechnet. Der Algorithmus überprüft dabei, ob für alle Knotenkombinationen i, j, k die Distanz von i über k nach j kleiner ist als die direkte Distanz von i nach j. In diesem Fall ergibt sich die neue Kontendistanz $\delta(i, j)$ zu $\delta(i, k) + \delta(k, j)$. Nun lässt sich ein simpler CREW-PRAM Algorithmus konstruieren, bei dem jeder Prozessor für ein Knotenpaar (i, j) über alle k iteriert und diesen Test ausführt.

Algorithmus 12 Floyd-Warschall-Algorithmus (PRAM)

global $D[n]$, **private** i, j
$i \leftarrow \left\lfloor \frac{p}{\sqrt{n}} \right\rfloor$
$j \leftarrow p \mod \sqrt{n}$
for $k \leftarrow 0 \ldots \sqrt{n} - 1$ **do**
    $D[p] \leftarrow \min(D[p], D[i \cdot \sqrt{n} + k] + D[k \cdot \sqrt{n} + j])$

Die Distanzen werden dabei in einer zweidimensionale Matrix eingetragen, welche in dem Feld D Zeile für Zeile gespeichert wird. Da die Matrix quadratisch ist, kann auf einen Eintrag (i, j) über den Index $i \cdot \sqrt{n} + j$ zugegriffen werden. Initialisiert wird dieses Feld von außerhalb wie folgt:

$$D[i \cdot \sqrt{n} + j] = \begin{cases} 1 & \text{Wenn } (i, j) \in E \\ 0 & \text{Wenn } i = j \\ \infty & \text{Sonst} \end{cases}$$

Dieser PRAM-Algorithmus lässt sich nun sehr einfach als GPGPU-Programm realisieren. Die for-Schleife lässt sich inklusive ihrer Meta-Variable k komplett ins

Host-Programm verschieben. Auch ist es möglich, durch die Nutzung einer zweidimensionalen Worksize die Bestimmung von i und j zu vereinfachen. Nun bleibt lediglich die Zuweisung über. Hier zeigen sich jedoch potentielle Speicherkonflikte, falls $i = k$ oder $j = k$ ist. Jedoch kann gezeigt werden, dass an diesen Stellen der Wert in dem Schritt nicht verändert wird. Da per Definition $\delta(x, x) = 0$ ist, gilt $\delta(i, k) + \delta(k, j) = \delta(i, i) + \delta(i, j) = \delta(i, j)$. Somit wird nie eine kleinere Distanz gefunden. Aus diesem Grund ist es möglich, den Schritt für diese Prozessoren zu überspringen, ohne die Korrektheit des Algorithmus zu verlieren. Für die Implementierung bedeutet dies, dass ein Prozessor die Zuweisung nur ausführt, wenn $i \neq k$ und $j \neq k$ ist. Nun wird sichergestellt, dass die Indizes mit k nicht beschrieben werden und der Kernel so ohne Speicherkonflikte ausgeführt werden.

Algorithmus 13 Floyd-Warschall-Algorithmus (GPU)

```
1: procedure ShortestPaths(D[n^2])
2:     for k ← 0 ... n − 1 do
3:         enqueue ShortestPathStep(D, k) on {(0, 0), ..., (n − 1, n − 1)}
4: kernel ShortestPathStep(global D[n^2], private k)
5:     i ← get_global_id(0)
6:     j ← get_global_id(1)
7:     if k ≠ i ∧ k ≠ j then
8:         D[i + j · n] ← min(D[i + j · n], D[i + k · n] + D[k + j · n])
```

5.5 Maximale Matchings

Ein weiteres interessantes Graphenproblem ist die Bestimmung von *maximalen Matchings*. Ein Matching ist eine Kantenmenge, sodass jeder Knoten in höchstens einer Kante verwendet wird. Für ein maximales Matching gilt weiter, dass kein anderes Matching existiert, welches mehr Kanten verwendet. In diesem Abschnitt wird speziell für *bipartite* Graphen ein paralleler Algorithmus entwickelt, der ein solches maximales Matching berechnet. Ein bipartiter Graph zeichnet sich dadurch aus, dass sich die Knoten in zwei Knotenklassen U, V partitionieren lassen, sodass keine Kante zwei Knoten der gleichen Klasse verbindet. Der dazu verwendete Algorithmus basiert auf einem von Khosla [Kho13] entwickeltem sequentiellen Algorithmus.

Bälle und Körbe

Die dort betrachtete Problemstellung ist die bestmögliche Aufteilung von einer Menge von Bällen auf eine Menge von Körbe zu finden, wenn jeder Ball nur in eine begrenzte Untermenge von möglichen Körben gelegt werden kann (Im Folgenden als *Kandidaten* des Balls bezeichnet). Diese Beziehungen lassen sich auch leicht als bipartiten Graph darstellen: Die Bälle und Körbe stellen die jeweils disjunkten Knotenklassen dar, wobei Kanten zwischen einem Ball und seinen Kandidaten existieren. Offensichtlich ist dieser Graph bipartit, da ein Ball nur in einen Korb gelegt werden kann und somit keine Kanten zwischen Bällen existierten. Gleiches gilt für die Körbe.

Der Algorithmus beschreibt nun eine Strategie, wie die Bälle auf Körbe aufgeteilt werden, sodass möglichst wenig Bälle am Ende in keinem Korb liegen. Bezogen auf einen Graphen ist dies äquivalent zu einem maximalen Matching:

- Die Aufteilung beschreibt ein Matching, da pro Korb nur ein Ball erlaubt ist und ein Ball auch gleichzeitig nur in einem Korb liegen kann.
- Das Matching ist maximal, da sonst ein anderes Matching existiert, in dem weniger nicht zugeordnete Bälle vorhanden sind.

Algorithmus 14 zeigt die sequentielle Variante des Algorithmus. Für jeden Ball wird die AssignBall-Funktion aufgerufen, welche versucht, den Ball einem seiner Kandidaten zuzuteilen. Sind alle Kandidaten bereits durch andere Bälle belegt, so wird einer dieser Bälle entfernt und der Ball dort hineingelegt. Für den entfernten Ball wird die Funktion wieder rekursiv aufgerufen.

Algorithmus 14 Bestimmung eines maximalen Matchings auf einem bipartiten Graphen [Kho13]

```
1:  procedure FindMatching(U, V)
2:      L(v) ← 0 ∀v ∈ V
3:      T(v) ← φ ∀v ∈ V
4:      for x ∈ U do
5:          AssignBall(x, L, T)
6:      return T

7:  procedure AssignBall(x, L, T)
8:      Wähle den Korb v der Kandidaten des Balls x mit dem kleinsten Label
        L(v).
9:      if L(v) > n − 1 {Ball kann nicht zugeordnet werden} then
10:         return
11:     else
12:         L(v) ← min_{w|w∈x\v} L(v) + 1
13:         if T(v) ≠ φ {Ersetzte vorhandenen Ball} then
14:             y ← T(v)
15:             T(v) ← x
16:             AssignBall(yL, T)
17:         else
18:             T(v) ← x
```

Gesteuert wird dieser Prozess durch ein Label L der Körbe V. Bei der Zuordnung eines Balles wird dieses erhöht (Zeile 12) und ein Ball versucht sich immer in den Korb mit dem geringsten Label zu legen (Zeile 8). Dadurch wird erzielt, dass Bälle in stark umkämpfen Körben weniger oft ersetzt werden, da diese ein großes Label besitzen. Die Zuordnung eines Balls wird abgebrochen, falls alle Labels der Kandidaten $\geq n$ sind. In diesem Fall lässt sich kein Korb für diesen Ball finden.

Parallelisierung

Der Algorithmus ist nun in sofern für die Parallelisierung attraktiv, da die Zuordnung eines Balles keinen globalen Zustand benötigt. Es werden lediglich die lokalen Eigenschaften des Balls und seiner Kandidaten verwendet (Khosla nennt diese Methode daher auch *lokale Suche*). Dies ermöglicht die parallele Zuordnung von Bällen, solange diese sich keine gemeinsamen Kandidaten teilen. In diesem Fall treten natürlich Konflikte auf, wenn beide Bälle sich in den gleichen Korb legen wollen. Der in dieser Arbeite entwickelte parallele Algorithmus umgeht dieses Problem jedoch, indem er zunächst eine solche Mehrfachzuordnung erlaubt und diese

in einem weiteren Schritt wieder korrigiert. Dadurch werden aufwendige Synchronisierungen vermieden, welches eine einfache GPU-Implementierung erlaubt.

Der parallele Algorithmus läuft nun in jeweils zwei Schritten ab:

1. Alle noch nicht zugeordneten Bälle iterieren über ihre Kandidaten und ordnen sich dem Korb mit dem geringsten Label zu. Dabei setzten sie ihr eigenes Label auf das zweit-geringste Label ihrer Kandidaten (oder ∞, wenn der Knoten sonst keinen möglichen Korb hat).

2. Alle Körbe iterieren über die Bälle, für die sie Kandidaten sind und bestimmen den Ball mit dem größten Label der sich ihnen zugeordnet hat. Das Matching zu diesem Ball wird akzeptiert und alle anderen Bälle, die sich diesem Korb zugeordnet haben, werden wieder zurückgesetzt. Auch ergibt sich das neue Korb-Label aus dem inkrementierten Label des ausgewählten Balls.

Diese Schritte werden nacheinander solange ausgeführt, bis alle Bälle zugeordnet sind, oder insgesamt n Durchläufe geschehen sind.

Korrektheit

Es ist leicht zu sehen, dass nach jedem zweiten Schritt ein korrektes Matching erreicht wird. Jeder Ball ordnet sich im ersten Schritt nur einem Korb zu und ein Korb wählt im zweiten Schritt auch nur einen Ball aus. Dass dieses Matching allerdings, wie im sequentiellen Algorithmus, mit der Zeit zu einem maximalen Matching wird, ist nicht direkt ersichtlich. In diesem Teil wird zwar kein kompletter Beweis für diese Behauptung gegeben, es werden jedoch Argumente gebracht warum das Grundprinzip des parallelen Algorithmus ähnlich zu dem des sequentiellen Algorithmus ist.

Der Grundgedanke ist, dass die Reihenfolge, in der die Bälle im sequentiellen Algorithmus zugeordnet werden, keine Rolle für die Korrektheit des Algorithmus spielt. Für die Hauptschleife, in der für alle Bälle die AssignBall-Funktion aufgerufen wird, ist dies leicht zu sehen. Dort wird lediglich ohne feste Reihenfolge über alle Bälle iteriert. Gleiches sollte jedoch auch für die Zuordnung von ersetzten Bällen gelten, welche ja direkt nach dem Entfernen wieder rekursiv zugeordnet werden. Da ein Ball aber keinen Zustand besitzt, welcher diese direkte Zuordnung erfordert, spricht nichts dagegen, zunächst einen anderen noch nicht zugeordneten Ball zuzuordnen. Aus Sicht des Algorithmus unterscheiden sich diese, bis auf ihre Kandidaten, nicht.

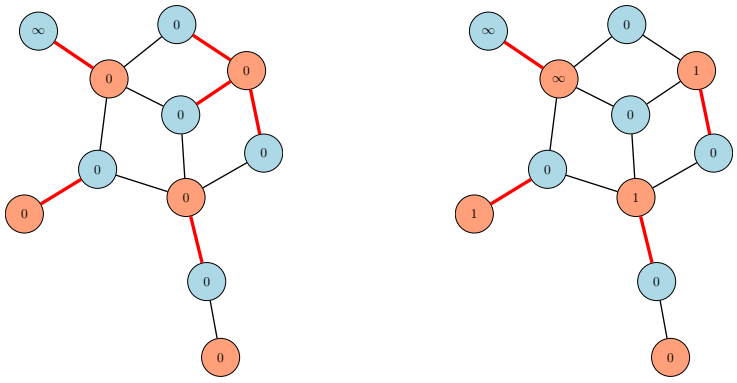
Akzeptiert man dies, so könnte ein immer noch korrekter Algorithmus arbeiten, indem er alle noch nicht zugeordneten Bälle in einer Warteschlange sammelt und

ersetzte Bälle ans Ende dieser anhängt. Es würde dann immer das erste Element der Warteschlange zugeordnet werden, bis diese leer ist.

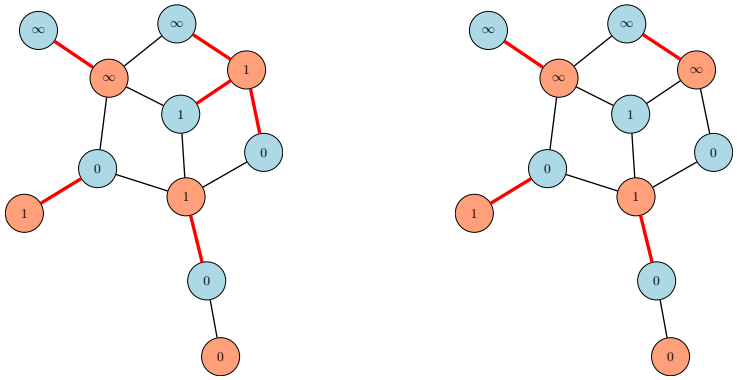
Der parallele Algorithmus kann nun als ein solcher Warteschlangenalgorithmus interpretiert werden. Anstatt jedoch immer nur einen Ball aus der Warteschlange zuzuordnen, werden im ersten Schritt parallel alle derzeit vorhanden Bälle zugeordnet. Die bei diesen Zuordnungen wieder frei werdenden Bälle werden im zweiten Schritt bestimmt, wenn die Körbe ihre Mehrfachzuordnungen wieder zurücksetzten.

Beispiel

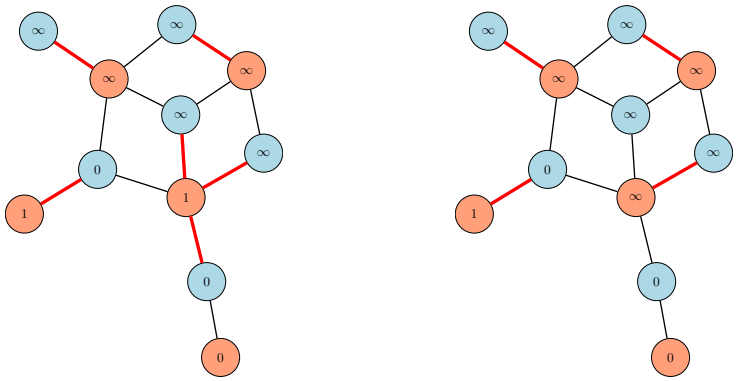
Um den Ablauf des Algorithmus zu veranschaulichen, wird dieser Anhand eines Beispielgraphen durchgeführt. Hier werden die Bälle in Blau und die Körbe in Orange dargestellt. Kanten die aktuell im Matching enthalten sind werden Rot markiert.



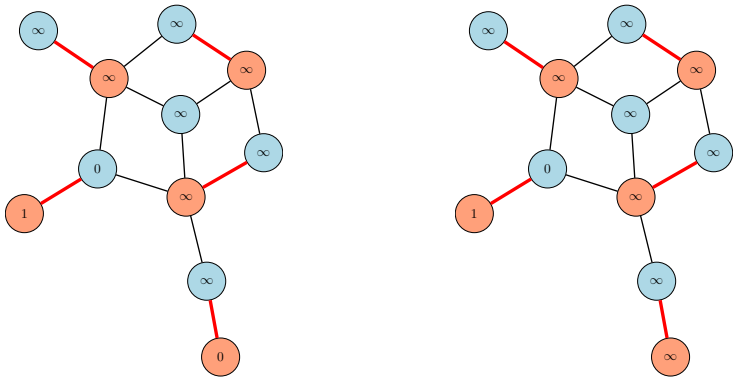
Die ersten beiden Schritt laufen hauptsächlich zufällig ab, da noch alle Labels gleich sind. Bälle wählen einen beliebigen Korb aus ihren Kandidaten aus und die Körbe wählen ihrerseits einen beliebigen zugeordneten Ball aus.



Die nächsten beiden Schritte sind dagegen interessanter. Bälle die nach dem zweiten Schritt wieder zurückgesetzt wurden, versuchen erneut einen Korb zu wählen. Hier hat beispielsweise der obere Ball einen Vorteil, da er nun als Label ∞ hat, weil sein einzig anderer Korb bereits endgültig besetzt ist. Der mittlere Korb hat jedoch noch einen anderen möglichen Kandidaten und hat daher nur ein Label von 1.



Nach einigen Schritten zeigt sich, dass sich das ∞-Label unter den Knoten verbreitet. Das ist eine natürliche Folge, wenn ein Ball keine anderen Kandidaten mehr hat, dem er sich zuordnen kann. Ein unendliches Label in einem Korb bedeutet daher, dass die Zuordnung zum aktuell gemachten Ball endgültig ist.



Nach den letzten Schritten wird nun das maximale Matching mit 5 Kanten erreicht. Der einzige nicht enthaltene Ball in der Mitte erkennt, dass er keine Chance mehr hat sich zuzuordnen, da alle Kandidaten ein unendliches Label haben.

Darstellung des Graphen

Um den Algorithmus zu implementieren, wird eine Darstellung des Graphen im Speicher benötigt, die einen möglichst effizienten Zugriff auf die benötigten Gra-

pheigenschaften erlaubt. Da im Algorithmus hauptsächlich auf die Nachfolger eines Knotens zugegriffen wird, stellt sich die Darstellung als Adjazenzmatrix als nachteilig heraus, da diese Operation dort eine Laufzeit von $\mathcal{O}(|V|)$ besitzt (Es muss für alle Knoten geprüft werden, ob eine Kante existiert). Eine bessere Möglichkeit ist dagegen die Verwendung von Adjazenzlisten. Hier kann diese Anfrage in einer Laufzeit von $\mathcal{O}(\deg(v))$ beantwortet werden, da die Nachfolger eines Knoten direkt gespeichert werden. Um jedoch nicht $|V|$ einzelne Adjazenlisten zu verwalten, wurde eine von Dinneen, Khosravani und Probert [DKP] inspirierte kompakte Darstellung der Adjazenlisten verwendet.

In dieser Darstellung wird ein Graph durch zwei Felder I, S (Index und Successors) beschrieben. S hat $2 \cdot |E|$ Einträge und beinhaltet hintereinander die Adjazenzlisten der Knoten. I besitzt $|V| + 1$ Einträge und ist ein Index auf den Start der Adjazenzliste eines Knoten in S. Die Nachfolger eines Knoten v sind somit in $S[I[v]]$ bis $S[I[v+1]-1]$ zu finden. Da das Ende einer Adjazenzliste immer durch den Start der Liste des nächsten Knoten $(v + 1)$ bestimmt wird, muss für den letzten Knoten ein weiteres Element in I hinzugefügt werden, das an die virtuelle nächste Stelle nach dem Endes des S-Feldes zeigt. Für den Fall dass ein Knoten keine Nachfolger besitzt, so zeigt dessen V-Eintrag an den Start der Adjazenzliste des nächsten Knoten, wie auch in Abbildung 5.7 bei v_2 zu sehen ist.

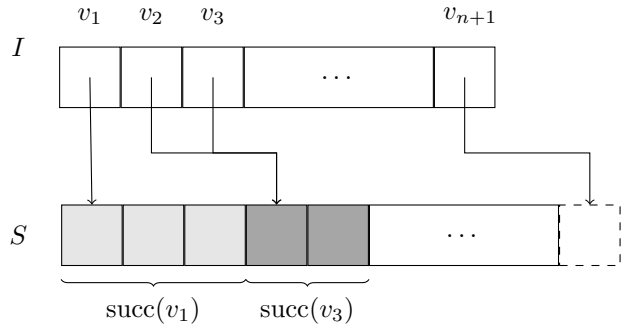


Abbildung 5.7: Kompakte Adjazenlistendarstellung eines Graphen

Um diese Darstellung nun für bipartite Graphen zu verwenden, werden die beiden Knotenmengen wie getrennte Graphen betrachtet. Die Felder I_U, S_U stellen die U-Knoten und I_V, S_V die V-Knoten dar. Da auf Grund der Bipartität des Graphen V-Knoten nur mit U-Knoten verbunden sind, werden die Einträge in I_V als Indizes für S_U und Einträge in I_U als Indizes für S_V gesehen.

Neben dem Graphen müssen auch die Matching-Beziehungen und Knotenlabels gespeichert werden. Hierfür werden jeweils die Felder M_U, M_V und L_U, L_V verwendet. Zwei Knoten u, v sind gematched, wenn $M_U[u] = v$ bzw. $M_V[v] = u$

ist. Für nicht zugeordnete Bälle wird ein Wert von -1 vergeben und -2 für die Bälle, die endgültig nicht zugeordnet werden können.

Implementierung

Im Host-Programm wird der Eingabegraph zunächst in die beschriebene Adjazenlistendarstellung transformiert und die restlichen Felder mit passenden Werten initialisiert. Danach werden die beiden Schritte n-mal ausgeführt.

Algorithmus 15 Host Programm des bipartiten Matchings

```
1: procedure BipartitMatching(G)
2:     I_U[n+1], S_U[·], M_U[n], L_U[n], I_V[m+1], S_V[·], M_V[m], L_V[m]
3:     Berechne I_U, S_U, I_V, S_V aus G
4:     Initialisiere M_U, M_V mit −1 und L_U, L_V mit 0
5:     for i ← 0 ... n − 1 do
6:         enqueue ChooseBin(I_U, S_U, M_U, L_U, L_V) on {0, ..., n − 1}
7:         enqueue ChooseBall(I_V, S_V, M_V, L_V, M_U, L_U) on {0, ..., n − 1}
8:     return U_m, V_m
```

Eine klare Verbesserungsmöglichkeit wäre es hier, die Schleife schon vorzeitig zu beenden, wenn bereits alle Bälle zugeordnet sind. Da sich aber, wie auch in Kapitel 4 beschrieben, eine solche datengetriebene Programmsteuerung als schwierig herausstellt, wurde sie hier nicht verwendet.

In dem ChooseBin-Kernel wird nun der erste Schritt ausgeführt, bei dem sich die Bälle einem Korb zuordnen.

Algorithmus 16 Bälle ordnen sich einem Korb zu

```
1: kernel ChooseBin(global I_U[n], S_U[·], M_U[n], L_U[n], L_V[m])
2:     u ← get_global_id(0)
3:     if M_U[u] = −1 then {Ball noch nicht zugeteilt}
4:         x ← x' sodass L_V[x'] = min_{y∈{I_U[u],...,I_U[u+1]−1}} L_V[S_U[y]]
5:         w ← min_{y∈{I_U[u]...I_U[u+1]−1}∧S_U[y]≠x} L_V[S_U[y]]
6:         if x exists ∧ L_V[x] ≠ ∞ then {Korb gefunden}
7:             M_U[u] ← x
8:             L_U[u] ← w
9:         else {Ball kann nicht zugeteilt werden}
10:            M_U[u] ← −2
```

Hier wird zunächst geprüft, ob dieser Ball noch nicht zugeordnet ist (Zeile 3). Ist dies der Fall, so wird mit x der Korb mit dem geringsten Label unter den

möglichen Kandidaten (Zeile 4) und mit w das zweitgeringste Label der Kandidaten (Zeile 5) bestimmt. Hierzu wird der Start- und End-Index in der Adjazenzliste bestimmt und auf allen diesen Elementen das minimale Label gesucht. Hat der Knoten mindestens einen Nachfolger und hat dieser Korb ein endliches Label, so wird sich diesem Korb zugeordnet. In dem Fall, dass x ein unendliches Label hat, macht eine Zuordnung kein Sinn, da dieser Korb bereits endgültig vergeben ist.

Mit dem ChooseBall-Kernel wird dann dementsprechend der zweite Schritt ausgeführt.

Algorithmus 17 Körbe suchen sich einen Ball aus

1: **kernel** ChooseBall(**global** $I_V[m]$, $S_V[\cdot], M_V[m], L_V[m], M_U[n], L_U[n]$)
2: $v \leftarrow$ `get_global_id(0)`
3: $x \leftarrow x'$ sodass $L_U[x'] = \max_{y \in \{I_V[v] \dots I_V[v+1]\} \wedge M_U[S_V[y]]=v} L_U[S_V[y]]$
4: **if** x exists **then** {Mindestens 1 Ball wurde zugeteilt}
5: $M_V[v] \leftarrow x$
6: $L_V[v] \leftarrow L_U[x] + 1$
7: $M_U[u] \leftarrow -1 \; \forall u \mid M_U[u] = v \wedge u \neq x$

Jeder Korb bestimmt mit x den Ball mit dem maximalen Gewicht, für den er ein Kandidat ist und welcher sich ihm zugeordnet hat (Zeile 3). Die Zuordnung dieses Balls wird akzeptiert (Zeile 5), indem das Korb-Label auf das inkrementierte Ball-Label gesetzt wird (Zeile 6) und die Zuordnung aller anderen Bälle wieder zurückgesetzt wird (Zeile 7).

5.6 Zusammenhangskomponenten

Ein weiteres Graphenproblem ist die Bestimmung von Zusammenhangskomponenten (kurz ZHK). Die ZHK eines Knoten ist definiert als der Teilgraph aller von diesem Knoten aus erreichbaren Knoten. Die Berechnung aller ZHKs ist somit eine Partitionierung der Knoten, sodass alle Knoten in einer Partition in der gleiche ZHK liegen. Eine mögliche Darstellung der ZHKs ist durch eine Label-Funktion L, sodass $L(v) = L(u)$ genau dann, wenn u, v in der gleichen ZHK liegen.

Ein PRAM-Algorithmus, der dies berechnen kann, wurde von Borůvka beschrieben und auch in „Efficient Algorithms“, Alg. 2.10 [Mei15] vorgestellt. Ursprünglich war dieser für die Berechnung des Minimalen Spannbaums vorhergesehen, jedoch kann dieser leicht modifiziert auch die ZHKs bestimmen.

Die Grundidee des Algorithmus ist, dass pro ZHK ein Repräsentantenknoten gewählt wird, der dann das Label darstellt. Die Knoten einer ZHK bestimmen

diesen Repräsentanten indem sie sukzessive per *Pointer-Jumping*[1] die Kanten zu dem jeweils kleinsten Knoten, bezogen auf die Knoten-ID, umbiegen.

Dieses Finden der Repräsentantenknoten kann parallel von den Knoten ausgeführt werden. Algorithmus 18 zeigt den dafür notwendigen PRAM-Algorithmus.

Algorithmus 18 PRAM-Algorithmus zur Bestimmung der ZHKs [Mei15]

1: $C[p] \leftarrow p$
2: **for** $i \leftarrow 1 \dots \lceil \log n \rceil$ **do**

3: $M \leftarrow \{C[j] | A[p \cdot n + j] = 1 \wedge C[j] \neq C[p]\}$
4: **if** $M = \emptyset$ **then** $T[p] \leftarrow C[p]$
5: **else** $T[p] \leftarrow \min(M)$

6: $M \leftarrow \{T[j] | C[j] = p \wedge T[j] \neq p\}$
7: **if** $M = \emptyset$ **then** $T[p] \leftarrow C[p]$
8: **else** $T[p] \leftarrow \min(M)$

9: **if** $C[p] = p$ **then** $C[p] \leftarrow T[p]$
10: **if** $C[C[p]] = p$ **then** $C[p] \leftarrow \min(C[p], c)$

11: **for** $k \leftarrow 1 \dots \lceil \log n \rceil$ **do** $C[p] \leftarrow C[C[p]]$

Grundsätzlich lässt sich der Algorithmus in 4 Schritte unterteilen, die $\log n$-mal ausgeführt werden (im Algorithmus durch Leerzeilen kenntlich gemacht). Da in jedem Schritt lesend auf die Felder, die im vorherigen Schritt beschrieben wurden, zugegriffen wird, ist eine globale Synchronisierung und damit eine Trennung in unterschiedlichen Kernels unabdingbar.

Zunächst werden hier einige Hilfs-Kernels aufgelistet, die die Initialisierung, das Kopieren und das Pointer-Jumping ermöglichen und an mehreren Stellen zum Einsatz kommen.

[1] Eine Technik bei der eine Indirektionsstufe überbrückt wird. Bspw: $A[p] \leftarrow A[A[p]]$

```
1: kernel INIT(global C[n])
2:     p ← get_global_id(0)
3:     C[p] ← p
4: kernel COPY(global A[n], B[n])
5:     p ← get_global_id(0)
6:     A[p] ← B[p]
7: kernel POINTERJUMP(global A[n], B[n])
8:     p ← get_global_id(0)
9:     A[p] ← B[B[p]]
```

Nun kann mit der Implementierung des ersten Schrittes begonnen werden. Da OpenCL innerhalb eines Kernels keine Mengen-Datenstrukturen kennt und eine dynamische Allokation von Speicher nicht möglich ist, wurde die Definition der Menge M übersprungen und x direkt als Minimum über C bestimmt, für alle möglichen j, die die Mengenbedingung erfüllen (Zeile 3). Der Rest des Schrittes ist soweit unverändert möglich: Lesend wird auf A und C zugegriffen, beschrieben wird dagegen nur T.

```
1: kernel STEP1_cc(global A[n^2], C[n], T[n])
2:     p ← get_global_id(0)
3:     x ← min_{j∈{0...n−1}∧A[p·n+j]=1∧C[j]≠C[p]} C[j]
4:     if x exists then T[p] = x
5:     else T[p] = C[p]
```

Der zweite Schritt ist sehr ähnlich zu dem ersten. Erneut wird mit M eine Menge definiert und aus ihr das Minimum gesucht. Neben den veränderten Mengenbedingungen kann jedoch der Zwischenspeicher Γ nicht umgangen werden, da auf T auch lesend in der Mengenbedingung zugegriffen wird. Nach dem Kernel wird daher der Hilfskernel COPY verwendet um das Ergebnis von Γ nach T zu kopieren.

```
1: kernel STEP2_cc(global C[·], T[·], Γ[·])
2:     p ← get_global_id(0)
3:     x ← min_{j∈{0...n−1}∧C[j]=p∧T[j]≠p} T[j]
4:     if x exists then Γ[p] = x
5:     else Γ[p] = C[p]
```

Beim dritten Schritt treten weitere Probleme auf. Da in Algorithmus 18 in

Zeile 10 auf den Wert von C zugegriffen wird und dieser in der vorherigen Zeile beschrieben wurde, können diese Zuweisungen nicht zusammengefasst werden. Auch macht die Zuweisung von $C[C[p]]$ Probleme, da hier das beschriebene Feld für den Index verwendet wird. Somit entsteht ein Speicherkonflikt eines Prozessors p zum Prozessor $C[p]$. Dies wird umgangen, indem der Wert von $C[C[p]]$ getrennt mit dem PointerJump-Kernel zwischen Step3a und Step3b berechnet und in Γ gespeichert wird.

```
1: kernel Step3a(global C[·], T[·])
2:     p ← get_global_id(0)
3:     if C[p] = p then C[p] ← T[p]
4: kernel Step3b(global C[·], Γ[·])
5:     p ← get_global_id(0)
6:     if Γ[p] = p then C[p] ← min(C[p], p)
```

Für den vierten Schritt müssen keine neuen Kernel verwendet werden, da hier lediglich ein Pointer-Jumping stattfindet und dafür der PointerJump-Kernel verwendet werden kann. Allerdings muss auch hier Γ als Zwischenspeicher verwendet werden, um Speicherkonflikte zu umgehen.

Insgesamt führen alle diese Kernels zu folgendem Host-Programm:

```
1: enqueue Init(C) on {0, ... n − 1}
2: for i ← 1 ... ⌈log2(n)⌉ do
3:     enqueue Step1cc(A, C, T) on {0, ... n − 1}
4:     enqueue Step2cc(C, T, Γ) on {0, ... n − 1}
5:     enqueue Copy(T, Γ) on {0, ... n − 1}
6:     enqueue Step3a(C, T) on {0, ... n − 1}
7:     enqueue PointerJump(Γ, C) on {0, ... n − 1}
8:     enqueue Step3b(C, Γ) on {0, ... n − 1}
9:     for j ← 1 ... ⌈log2(n)⌉ do
10:        enqueue PointerJump(Γ, C) on {0, ... n − 1}
11:        enqueue Copy(C, Γ) on {0, ... n − 1}
```

5.7 Minimaler Spannbaum

Wie bereits erwähnt ermöglicht der Algorithmus zur Berechnung der Zusammenhangskomponenten auch leicht die Berechnung des minimalen Spannbaums. Zur Definition des minimalen Spannbaums wird das bestehende Graphmodell um eine Kantengewichtsfunktion $w : E \to \mathbb{R}$ erweitert.

Die Kantengewichte werden dabei, ähnlich wie die Adjazenz- und Distanzmatrix, als Gewichtsmatrix W dargestellt. Der minimale Spannbaum eines gewichteten Graphen ist nun ein zusammenhängender Teilgraph, der alle ursprünglichen Knoten umfasst, aber keine Zyklen beinhaltet und das Gesamtkantengewicht minimiert. Eine Einschränkung des verwendeten PRAM-Algorithmus ist, dass der Minimale Spannbaum eindeutig sein muss.

In Algorithmus 19 ist die modifizierte Version des Algorithmus gezeigt.

Algorithmus 19 PRAM-Algorithmus zur Bestimmung des Minimalen Spannbaums [Mei15]

$C[p] \leftarrow p$
for $i \leftarrow 1 \dots \lceil \log n \rceil$ **do**

$M \leftarrow \{j | C[j] \neq C[p]\}$
$w \leftarrow \min_{j \in M}(W[p \cdot n + j])$
if $w = \infty$ **then** $T[p] \leftarrow C[p]$
else $T[p] \leftarrow j$ sodass $W[p \cdot n + j] = w$

$M \leftarrow \{j | C[j] = p \wedge C[T[j]] \neq p\}$
$w \leftarrow \min_{j \in M}(W[j \cdot n + T[j])$
if $w = \infty \vee C[p] \neq p$ **then** $T[p] \leftarrow C[p]$
else
    $j \leftarrow j'$ sodass $W[j' \cdot n + T[j']] = w$
    $T[p] \leftarrow C[T[j]]$
    Markiere Kante $(j, T[j])$

if $C[p] = p]$ **then** $C[p] \leftarrow T[p]$
if $C[C[p]] = p$ **then** $C[p] \leftarrow \min(C[p], c)$

for $k \leftarrow 1 \dots \lceil \log n \rceil$ **do** $C[p] \leftarrow C[C[p]]$

Es ist zu sehen, dass bis auf die ersten beiden Schritte der Algorithmus identisch zu dem aus Abschnitt 5.6 ist. Somit reicht es aus, nur für diese Schritte eine neue Implementierung zu entwickeln. Die Grundidee des modifizierten Algorithmus

ist, dass für jeden Knoten v eine Kante (v, u) existieren muss, die im minimalen Spannbaum liegt, da dieser sonst nicht mehr zusammenhängend ist (v kann nicht erreicht werden). Parallel bestimmen nun alle Knoten diese Kante und erweitern den entstehenden Graphen sukzessive, bis dieser zusammenhängend ist und damit den minimale Spannbaum bildet.

Im ersten Schritt wird nun, anstatt der Knoten mit der kleinsten ID, der Knoten, zu dem die Kante mit dem kleinsten Gewicht existiert, gesucht.

```
1: kernel STEP1_mst(global W[·], C[·], T[·])
2:     p ← get_global_id(0)
3:     w ← min_{j∈{0...n−1}∧C[j]≠C[p]} W[pn + j]
4:     if w ≠ ∞ then
5:         T[p] ← j sodass W[p · n + j] = w
6:     else T[p] ← C[p]
```

Gleiches gilt dann auch für den zweiten Schritt, sodass hier der Knoten mit dem minimalen Kantengewicht bestimmt wird. Zusätzlich findet auch die Konstruktion des Minimalen Spannbaums statt, indem die Kanten in einer Adjazenzmatrix markiert werden.

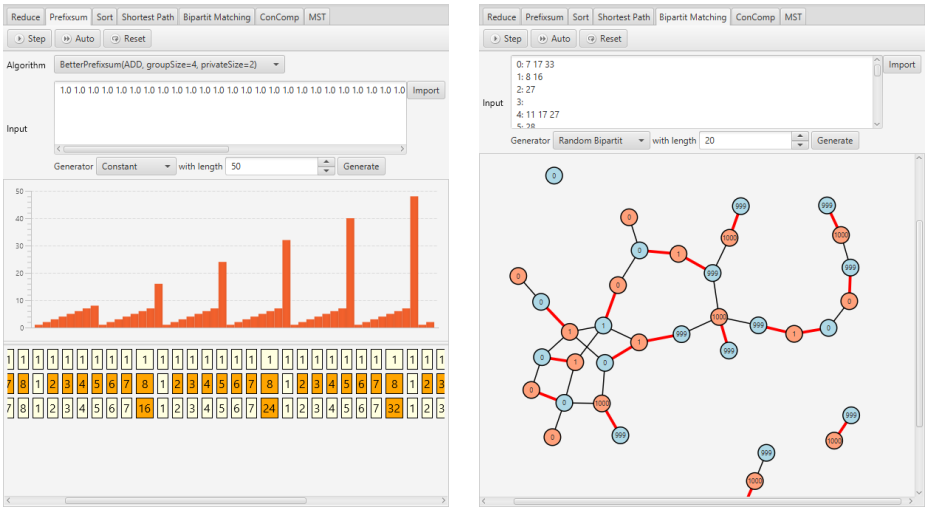
```
kernel STEP2_mst(global W[·], C[·], T[·], A[·], Γ[·])
    p ← get_global_id(0)
    w ← min_{j∈{0...n−1}∧C[j]=p∧C[T[j]]≠p} W[j · n + T[j]]
    if w ≠ ∞ ∧ C[p] = p then
        j ← j′ sodass W[j′ · n + T[j′]] = w
        A[j · n + T[j]] ← 1
        Γ[p] ← C[T[j]]
    else Γ[p] ← C[p]
```

Das Host-Programm bleibt mit der Ausnahme der ausgetauschten Kernel identisch.

Softwareanwendung

Alle der in diesem Kapitel beschrieben (und auch einige nicht erwähnte) Algorithmen wurden in einer, mit Java und OpenCL-Bindings von JogAmp[2] verfassten, Softwareanwendung implementiert.

Die Anwendung ermöglicht es dem Nutzer, über eine grafische Oberfläche die Algorithmen auf selbstgewählten oder automatisch generierten Eingaben auszuführen. Der Ablauf eines Algorithmus kann dabei Schritt für Schritt nachvollzogen werden, um einen besseren Eindruck der Arbeitsweise zu erhalten. Der aktuelle Ergebniszustand wird dabei von der Anwendung visualisiert. Für Felder wird dazu der aktuelle Inhalt in einem Balkendiagramm dargestellt und eine Historie vorheriger Zustände angezeigt. Graphen werden mit einem dynamischen Force-Layout angeordnet, welches auch eine interaktive Modifikation des Graphen erlaubt. Je nach Algorithmus können Knoten und Kanten farblich hervorgehoben werden oder mit Labels annotiert werden.



Eine detaillierte Bedienungsanleitung ist im Anhang zu finden.

[2] https://jogamp.org/jocl/www/

Kapitel 6

Fazit & Ausblick

In dieser Arbeit wurde gezeigt, dass eine Implementierung von PRAM-Algorithmen auf modernen GPU-Architekturen generell möglich ist, wenn auch nicht immer ohne einen gewissen Laufzeitblowup mit sich zu bringen. Dieser folgt aus der Tatsache, dass zum Erreichen der globalen Synchronisierung einer PRAM ein zusätzlicher Verwaltungsaufwand der aktiven Prozessoren entsteht. Jedoch zeigt sich, dass viele PRAM-Algorithmen bereits so gestaltet sind, dass durch Optimierungen diese Techniken nicht zur Anwendung kommen müssen.

Für eine Implementierung haben sich als optimal solche Algorithmen herausgestellt, die nur wenig globale Synchronisierungspunkte benötigen und deren Ablauf möglichst datenunabhängig ist. Letzteres ermöglicht in vielen Fällen die Verschiebung des gesamten Programmflusses in das Host-Programm, was ein Management der aktiven Prozessoren hinfällig macht. Dies ermöglicht auch eine optimale Verwendung der Command Queue, da keine Daten von der GPU zur CPU zurückfließen müssen.

Für wirklich performante Implementierungen sind PRAM-Algorithmen jedoch nicht am besten geeignet. Diese können die für eine optimale Laufzeit wichtigen Konzepte, wie die lokale Dimension (Workgroups), nicht nutzen, da sie im klassischen PRAM-Modell nicht enthalten sind. Auch stellt es sich als schwer heraus, diese im Nachhinein bei einer Implementierung zu verwenden, da der Algorithmus in den meisten Fällen nicht dafür geeignet ist, die Berechnung auf mehrere Ebenen zu verteilen. Ein optimaler GPU-Algorithmus sollte aber versuchen, die zu verrichtende Arbeit auf alle Hierarchiestufen (global, lokal, privat) zu verteilen, um die jeweils geringer werdenden Speicherzugriffskosten auszunutzen.

Für einige simple PRAM-Algorithmen, wie die Reduktion und Präfixsumme, wurde in dieser Arbeit eine solche Anpassung entwickelt. Für komplexere Algorithmen stellte sich eine solche Anpassung jedoch als zunehmend schwieriger heraus. In komplexeren Daten, wie zum Beispiel Graphen, ist eine Partitionierung in konstant große Bereiche häufig nicht trivial möglich. Auch ist die Kombination von

Teillösungen zu einer Gesamtlösung nicht immer klar und erfordert einen kreativen Aufwand.

Diese Arbeit kommt somit zum Schluss, dass sich viele der komplexeren PRAM-Algorithmen nicht optimal auf GPUs implementieren lassen, da die beiden Architekturen an diesen Stellen zu weit auseinandergehen. Um realitätsnäher zu bleiben, wäre daher ein restriktiveres PRAM-Modell notwendig, was die Synchronisation der Prozessoren einschränkt und dabei ein Workgroup-Konzept einführt. Wo dagegen PRAM-Algorithmen gut zum Einsatz kommen können, ist, wenn im Rahmen eine Workgroup ein Teilproblem berechnet werden muss. In diesem Bereich lassen sich PRAM-Algorithmen mit Hilfe der lokalen Synchronisierungsmöglichkeiten innerhalb eines Kernels leichter realisieren.

Betrachtet man den praktischen Programmierprozess von GPU-Programmen, so ist dieser noch deutlich nicht so komfortabel wie der eines CPU-Programms. Das hängt damit zusammen, dass die GPGPU-Programmierung noch relativ neu ist und es somit noch keine einheitlichen Prozessorarchitekturen oder Befehlssätze gibt. Dies hemmt die Entwicklung von höheren Programmiersprachen und Entwicklungstools und macht die GPU-Programmierung hochgradig abhängig von den jeweiligen Hardwareherstellern.

Inwieweit sich dies in der Zukunft verbessert, bleibt offen. Bei einem optimistischer Ausblick sollten GPUs jedoch als alternative Recheneinheit neben der CPU weiter an Bedeutung gewinnen. Demzufolge ist auch eine genauere Auseinandersetzung mit dem theoretischen Aspekt dieser Architektur sinnvoll, um parallele Algorithmen entwerfen zu können, die sich auch effizient in der Praxis realisieren lassen.

Anhang A

Software-Bedinungsanleitung

In diesem Anhang wird ein Bedienungsanleitung für die im Rahmen dieser Arbeit entwickelte Software gegeben. Strukturell ist der Code in 2 Module unterteilt: `paralg-core` und `paralg-gui`.

In dem Core-Modul sind alle Kernfunktionalitäten wie die Implementierung der parallelen Algorithmen inklusive Testfälle, sowie ein OpenCL-Wrapper und ein kleines Graphen-Framework enthalten. Das Gui-Modul ist von dem Core-Modul abhängig und stellt eine grafische Benutzeroberfläche bereit, um die Algorithmen zu starten. Diese Zweiteilung ermöglicht es in Zukunft, die Algorithmen in anderen Projekten zu verwenden, ohne dass die GUI-Klassen benötigt werden.

A.1 Build

Zum Kompilieren der Anwendung wird ein *Java8 SDK* und *Maven*[1] benötigt. Beide Module lassen sich nun über eine Kommandozeile mit

```
cd impl/paralg
mvn install
```

kompilieren. Ein getrenntes Kompilieren ist auch möglich, indem in die jeweiligen Unterordnet `impl/paralg/paralg-core` und `impl/paralg/paralg-gui` gewechselt wird und dort `mvn install` ausgeführt wird. Wenn auf einem Rechner dies erstmalig geschieht, wird eine Internetverbindung benötigt, da Maven automatisch die vom Core-Modul benötigten OpenCL Abhängigkeiten herunterlädt[2]. Während des Build-Vorgangs startet Maven auch automatisch die Testfälle des Core-Moduls. Ist dies nicht erwünscht, so können mit der Option `-Dmaven.test.skip=true` die Testfälle übersprungen werden. Nach dem Build sind die erstellen JARs im `target` Ordner der beiden Module zu finden.

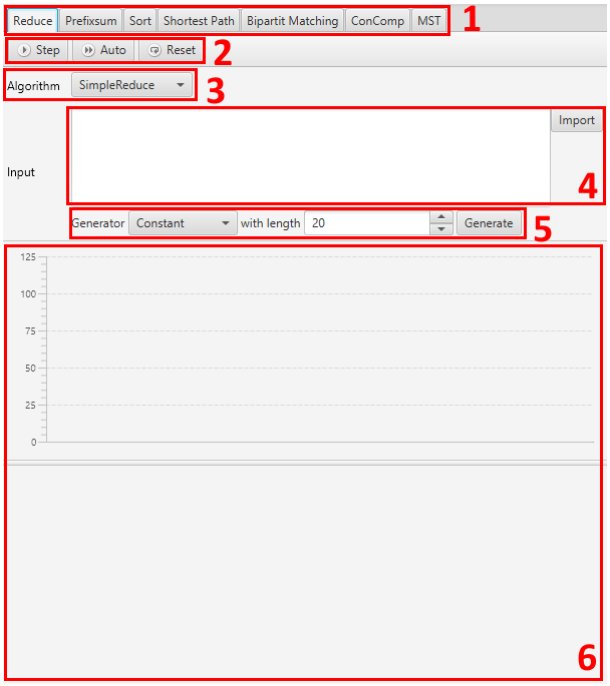
[1] `https://maven.apache.org/`

[2] Diese werden unter `~/.m2/repository` gespeichert

A.2 Anwendung

Die Anwendung lässt sich durch die vom Gui-Modul erzeugten JAR starten. Für die Ausführung ist notwendig, dass sich der Ordner `lib` immer an der gleichen Stelle wie die JAR-Datei befindet, da dieser die externen Abhängigkeiten enthält.

Die Anwendung sollte dann folgendes Fenster öffnen:



1. Über den Hauptreiter kann die Problemstellung ausgewählt werden. Die Reihenfolge ist dabei identisch zu der in dieser Arbeit: Reduktion, Präfixsumme, Sortierung, Kürzeste Wege, Bipartites Matching, Zusammenhangskomponenten und Minimale Spannbäume.

2. Die Steuerung des aktuellen Algorithmus.

   - Step: Führt einen Schritt des Algorithmus aus. In der Regel ist dies eine Kernel-Ausführung im Host-Programm.
   - Auto: Ruft wiederholt (mit kurzer Verzögerung) die Step-Funktion auf.
   - Reset: Setzt den Algorithmus wieder auf den Ausgangszustand zurück.

   Wenn der Algorithmus durchgelaufen ist, werden die Step- und Auto-Buttons ausgegraut und es muss ein Reset stattfinden, damit er erneut ausgeführt werden kann.

3. Je nach Problemstellung kann es mehrere Algorithmen geben, die dieses lösen. Über das Dropdown-Menü kann ein gewünschter Algorithmus ausgewählt werden. Bei einer Änderung wird automatisch ein Reset für alle Algorithmen ausgeführt.

4. Die Eingabe des Algorithmus kann hier textuell eingegeben werden. Das Eingabeformat unterscheidet sich dabei zwischen Algorithmen die auf Feldern, oder Graphen arbeiten. Anstatt den Inhalt in das Textfeld manuell zu schreiben, kann auch direkt der Inhalt einer Datei über den Import-Button verwendet werden. Bei einer Änderung der Eingabe wird automatisch ein Reset für alle Algorithmen ausgeführt.

5. Alternativ zur Eingabe über das Eingabefeld, können auch Generatoren benutzt werden um die Eingabe automatisch zu erzeugen. Dazu kann über das Dropdown-Menü ein Generator ausgewählt werden und über das Textfeld die Größe der generierten Ausgabe eingestellt werden. Durch Klicken des Generate-Buttons wird dann eine Eingabe erzeugt und in das Eingabefeld geschrieben.

6. Hier wird der aktuelle Zustand des Algorithmus visualisiert. Erneut wird hier zwischen Feldern und Graphen unterschieden.

Eingabeformat: Felder

Felder werden als eine durch Whitespace getrennte Sequenz von Zahlen deklariert. Beispiel: 0.1 2 .23 -23 4e-2 erzeugt das Feld $[0.1, 2.0, 0.23, -23.0, 0.04]$

Eingabeformat: Graphen

Graphen werden über Adjazenzlisten deklariert. Dazu kann pro Zeile die Adjazenzliste eines Knoten angegeben werden, indem der Knoten gefolgt von ":" und einer Liste von durch Whitespace getrennten Nachfolgerknoten geschrieben wird. Ein Knoten wird als Integer angegeben, wobei die Menge aller Knoten keinen Zusammenhängenden Integerbereich darstellen muss. Die Gesamtmenge an Knoten wird aus allen in den Adjazenzlisten vorkommenden Knoten bestimmt. Beispiel:

$$\begin{array}{ll} 0: & 1 \quad 2 \\ 1: & 2 \quad 3 \\ 2: & 3 \end{array}$$

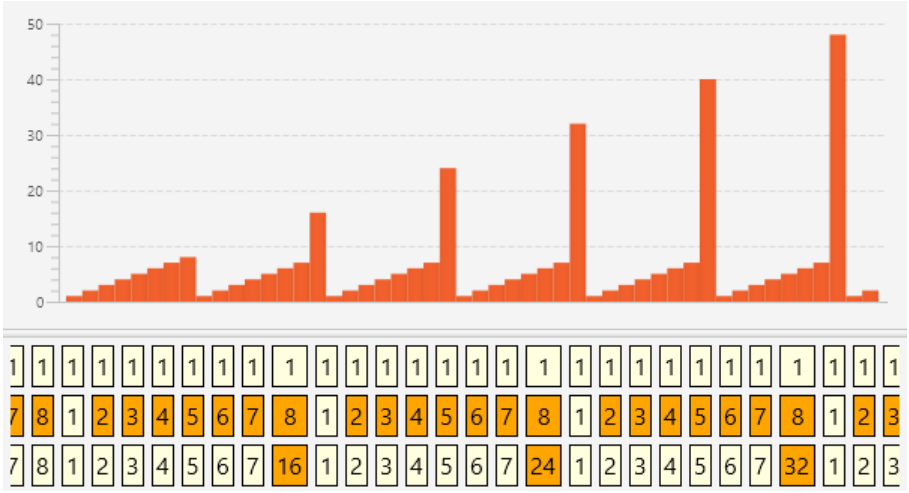
erzeugt den Graphen $V = \{0, 1, 2, 3\}$, $E = \{(0, 1), (0, 2), (1, 2), (1, 3), (2, 3)\}$

Generatoren: Felder

Für die automatische Generierung von Eingaben für Felder stehen folgende Generatoren zur Verfügung:

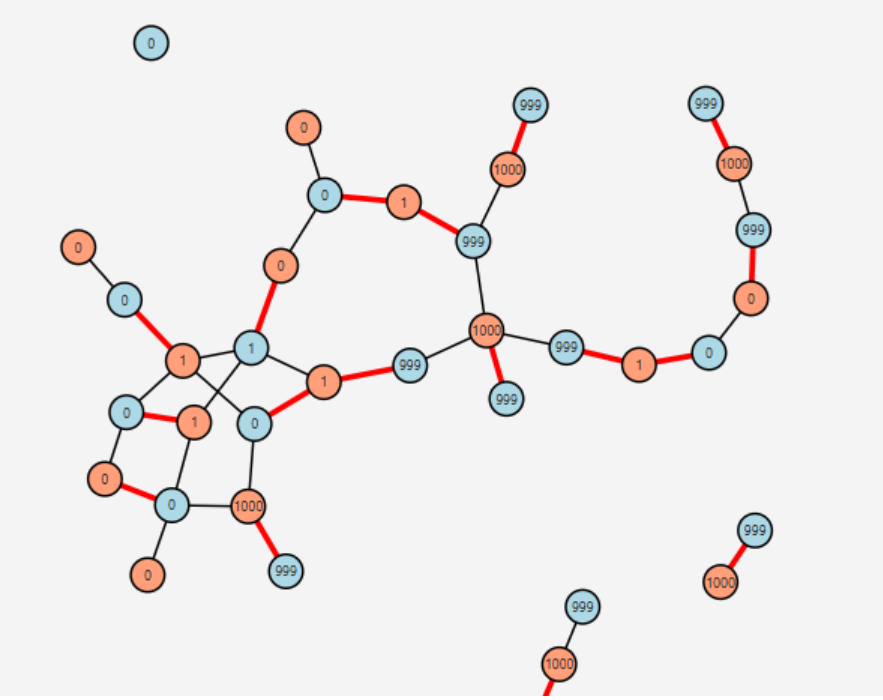
- Constant: Jedes Element hat den Wert 1.
- Random: Die Elemente haben einen Zufallswert zwischen 0 und 10.
- Shuffle: Die Elemente stellen eine zufällige Permutation von 0 bis n dar.
- Ascending: Die Elemente stellen die steigende Sequenz von 0 bis n dar.
- Descending: Die Elemente stellen die fallende Sequenz von n bis 0 dar.

Visualisierung: Felder



Die Visualisierung von Felder hat 2 Komponenten. Im oberen Teil wird der aktuelle Inhalt des Feldes als Balkendiagramm dargestellt. Die Skalierung findet dabei automatisch statt, sodass der gesamte Wertebereich dargestellt werden kann. Im unteren Bereich ist der Verlauf der Werte aus den vorherigen Schritten zu sehen. Die unterste Reihe ist dabei stets der aktuelle Stand. Farblich hervorgehoben sind die Elemente, die sich im Vergleich zum vorherigen Schritt geändert haben.

Visualisierung: Graphen



Graphen werden mit einem interaktiven Force-Layout dargestellt. Knoten können über per Drag-and-Drop mit der linken Maustaste verschoben werden. Zusätzlich ist hier eine Modifikation des Graphen möglich. Dazu können mit der rechten Maustaste neue Knoten erzeugt werden, indem in einen leeren Bereich geklickt wird. Durch das Klicken und gedrückt halten der rechten Maustaste auf einen Knoten kann eine neue Kante aufgespannt werden. Abgeschlossen wird dieser Vorgang, indem die Taste auf einem Zielknoten wieder losgelassen wird. Das Entfernen von Knoten und Kanten ist durch Auswahl des jeweiligen Elements per Linksklick und dann über die Tastatur mit der Entfernen-Taste möglich. Jegliche Änderungen im Graphen spiegeln sich automatisch auch in der Eingabefeld wieder. Das gleiche gilt für die Rückrichtung. Eine Skalierung des Graphen kann bei gedrückter Steuerungstaste über das Mausrad geschehen. Weiterhin ist es möglich, den Graphen einzufrieren, indem die Leertaste gedrückt wird (dazu muss ein Element des Graphen ausgewählt sein). Auch kann das Bild des Graphen als eine LaTeX-Datei exportiert werden, indem die T-Taste gedrückt wird. Die generierte Datei verwendet das Tikz-Paket.

Die Eigentliche Visualisierung hängt von der Art des Algorithmus ab:

- Die Distanzen im Graphen werden für den ausgewählten Knoten sowohl textuell, als auch Farbgradient angezeigt.

- Beim den Matchings in bipartiten Graphen wird die Knotenklasse farblich gekennzeichnet. Die Labels der Knoten werden textuell angezeigt. Kanten, die im aktuellen Matching enthalten sind, werden rot dargestellt.
- Für die Zusammenhangskomponenten wird pro Knoten über ein Label die aktuelle Zusammenhangskomponente angezeigt. Auch wird das Label als Knotenfarbe verdeutlicht.
- Beim minimalen Spannbaum wird an den Kanten das Kantengewicht angezeigt und der minimale Spannbaum in Rot hervorgehoben.

Literatur

[BKK12] Jurgen Brenner, Jörg Keller und Christoph W. Kessler. „Executing PRAM Programs on GPUs“. In: *Proceedings of the International Conference on Computational Science, ICCS 2012, Omaha, Nebraska, USA, 4-6 June, 2012*. 2012, S. 1799–1806. DOI: 10.1016/j.procs.2012.04.198.

[Ble90] Guy E. Blelloch. *Prefix Sums and Their Applications*. Techn. Ber. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990.

[BSS10] A. Bernemann, R. Schreyer und K. Spanderen. „Pricing structured equity products on GPUs“. In: *High Performance Computational Finance (WHPCF), 2010 IEEE Workshop on*. 2010, S. 1–7. DOI: 10.1109/WHPCF.2010.5671821.

[CLR08] Henri Casanova, Arnaud Legrand und Yves Robert. *Parallel Algorithms*. 1st. Chapman & Hall/CRC, 2008. ISBN: 9781584889458.

[DKP] Michael J. Dinneen, Masoud Khosravani und Andrew Probert. *Using OpenCL for Implementing Simple Parallel Graph Algorithms*.

[For05] Lance Fortnow. *What happened to the PRAM?* 2005. URL: http://blog.computationalcomplexity.org/2005/04/what-happened-to-pram.html.

[Gal13] Michael Galloy. *CPU vs GPU performance*. 2013. URL: http://michaelgalloy.com/2013/06/11/cpu-vs-gpu-performance.html.

[Gro15] Khronos Group. *The OpenCL Specification Version 2.0*. 2015.

[Kho13] Megha Khosla. „Balls into Bins Made Faster“. In: *Algorithms – ESA 2013: 21st Annual European Symposium, Sophia Antipolis, France, September 2-4, 2013. Proceedings*. Hrsg. von Hans L. Bodlaender und Giuseppe F. Italiano. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 601–612. DOI: 10.1007/978-3-642-40450-4_51.

[KK07] Christoph Kessler und Jörg Keller. „Models for parallel computing: Review and perspectives“. In: *PARS Mitteilungen* 24.0177-0454 (2007), S. 13–29.

[Mei15] Arne Meier. „Efficient Algorithms“. Lecture Notes. Juli 2015.

[NVI09] NVIDIA. *NVIDIA OpenCL Best Practices Guide*. Techn. Ber. NVIDIA, 2009. URL: http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/nvidia_opencl_bestpracticesguide.pdf.

[Vis15] Uzi Vishkin. *Explicit Multi-Threading (XMT): A PRAM-On-Chip Vision*. 2015. URL: http://www.umiacs.umd.edu/users/vishkin/XMT/index.shtml.

[WLT11] Hongwei Wu, Xiangnan Liu und Weibin Tang. „A fast GPU-based implementation for MD5 hash reverse“. In: *2011 IEEE International Conference on Anti-Counterfeiting, Security and Identification*. 2011, S. 13–16. DOI: 10.1109/ASID.2011.5967405.

[XF10] Shucai Xiao und Wu-chun Feng. „Inter-block GPU communication via fast barrier synchronization“. In: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. 2010, S. 1–12. DOI: 10.1109/IPDPS.2010.5470477.

[Ye+10] Xiaochun Ye u. a. „High performance comparison-based sorting algorithm on many-core GPUs“. In: *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE. 2010, S. 1–10.

[YWO15] C. Yang, Y. Wang und J. D. Owens. „Fast Sparse Matrix and Sparse Vector Multiplication Algorithm on the GPU“. In: *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. 2015, S. 841–847. DOI: 10.1109/IPDPSW.2015.77.

[ZCW10] Nan Zhang, Yun shan Chen und Jian li Wang. „Image parallel processing based on GPU“. In: *Advanced Computer Control (ICACC), 2010 2nd International Conference on*. Bd. 3. 2010, S. 367–370. DOI: 10.1109/ICACC.2010.5486836.

Erklärung

Hiermit versichere ich, Stefan Dulle (Matrikelnummer: 2841990), dass ich diese Masterarbeit mit dem Thema

Implementierungen von parallelen Algorithmen auf GPGPUs

selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt, sowie Zitate kenntlich gemacht habe.

Stefan Dulle
Hannover, den 29. September 2016