Sind moderne GPU Implementierungen von Sortieralgorithmen die Lösung des Sortierproblems?

Florian Manig
Technische Hochschule Ingolstadt
Ingolstadt, Bayern
flm1761@thi.de

***Zusammenfassung*—Das Sortierproblem gilt als das meistbesprochenste und relevanteste Problem in der Informatik. Durch den Lauf der Zeit haben sich auch die Lösungen für dieses Problem weiterentwickelt. Parallelität ist das aktuelle Stichwort in der heutigen CPU und GPU Architektur. Gerade die GPU´s sind durch das Bearbeiten von tausenden Threads gleichzeitig deutlich überlegen im parallelen Arbeiten. Wie man die hohe Parallelität von GPU´s zur Lösung des Sortierproblems nutzen kann, zeigt diese Arbeit. Dabei sollen moderne GPU Implementierungen von Sortieralgorithmen vorgestellt und bewertet werden. Dazu werden anfangs grundlegene Prinzipien aus dem Sortieren und der Algorithmik als Basiswissen vermittelt. Anschließend werden einige Sortieralgorithmen, insbesondere wie man sie effizient auf einer GPU parallel implementieren kann, genauer betrachtet. Abschließend wird die Effizienz der vorgestellten Methoden mit den parallelen Implementierungen auf Multi-Core CPU´s verglichen und ein Resümee gezogen.**

***Index Terms*—Effizienetes Sortieren, Algorithmik, Paralleles Sortieren, GPU-Sorting**

I. Einleitung

A. Motivation

Das Sortieren ist eins der wichtigsten Probleme in der Informatik. Etwa ein Viertel der Rechenzeit weltweit wird für das Sortieren von Daten verwendet [2 ,vgl. S. 79]. Nach Definition versteht man in der Informatik unter dem Sortiervorgang die Sortierung eines Tupels nach einer Ordnung (z.B numerisch). Dafür werden verschiedene Verfahren, sogenannte Sorthieralgorithmen, verwendet. Aufgrund der allgemeinen Präsenz des Sortierproblems sollten diese möglichst effizient sein. Es gibt mehrere Gründe dafür, dass das Sortieren, das fundamentalste Problem der Informatik ist [3, vgl. S.148]. Viele Anwendungen müssen von Natur aus Informationen sortieren, z.B. müssen Banken um Kontoauszüge vorzubereiten, Schecks nach Scheck-Nr. sortieren. Außerdem nutzen Algorithmen das Sortieren oft als wichtige Subroutine, z.B. in einem Programm, bei dem grafische Objekte übereinander erzeugt werden, ist es möglich, dass diese Objekte zuerst nach einer "darüber" Relation sortiert werden, damit sie dann von unten nach oben erzeugt werden können. Bis heute wird an Algorithmen gearbeitet, um das Sortieren effizienter zu handhaben. Während das Implementieren von Sortieralgorithmen auf einem CPU relativ simpel ist und eher eine Frage davon ist, welchen Algorithmus für das vorliegende Problem verwendet werden soll, ist das Sortieren auf der GPU etwas komplizierter, da die GPU hoch parralel arbeitet. Da jedoch die GPU gerade im parallelen Berechnen von Daten leistungsstärker als der CPU ist, ist es wichtig auf der GPU effektiv sortieren zu können um diese Leistung effektiv zu nutzen. Wie hoch die Effektivität des Sortierens auf der GPU ist und was man tun kann um diese weiter zu erhöhen wird in folgender Arbeit gezeigt.

B. Forschungsfrage

Die Forschungsfrage dieser Arbeit lautet, ob moderne Implementierungen von Sortieralgorithmen die aktuelle Lösung des Sortierproblems sind.

II. Das Sortierproblem

Im folgenden Kapitel werden die Eigenschaften und Bedingungen, die ein Sortierproblem ausmachen, sowie das Ziel, dass jeder Sortieralgorithmus bei der Bearbeitung des Problems verfolgt, definiert. Bei einem Sortierproblem sind die Gegebenheiten eine Menge von Sätzen, die es gilt nach einer Ordnung z.B. numerisch, alphabetisch oder lexikografisch, zu sortieren [2, vgl. S. 79]. Welche dieser Ordnungsrelationen benutzt wird hängt von dem Schlüssel ab, den jeder Satz besitzt. In der Praxis ist der numerische Primärschlüssel am häufigsten vertreten, da oft nach spezifischen und einmaligen ID´s sortiert wird. Beispielsweise könnte man in einer Mitarbeiter Datenbank die Mitarbeiter nach einer Mitarbeiter-ID sortieren um z.B Suchen für Änderungen einfacher zu machen. Zwischen Schlüsseln gibt es eine Ordnungsrelation, das heißt man kann sie vergleichen bzw. ordnen [2, vgl. S. 79]. Ohne diese Ordnungsrelation könnten sie kein Primärschlüssel sein, da der Vergleich untereinander ohne die Ordnungsrelation nicht möglich ist. Die Sätze können neben der Schlüsselkomponente auch weitere Komponenten besitzen, diese sind jedoch für den Sortiervorgang irrelevant und werden eigentliche Informationen gennant [2, vgl. S. 79]. Die Mitarbeiter haben z.B. noch mehr Attribute wie Alter oder Gehalt, nach denen man auch sortieren könnte. Dies wären die eigentlichen Informationen, die in diesem Beispiel nicht der Primärschlüssel sind. Die Mitarbeiter entsprechen jetzt unseren Sätzen s_1 bis s_N . Und jeder dieser Sätze s_i hat einen Schlüssel k_i was der Mitarbeiter ID entspricht. Das Ziel ist es nun eine Permutation der Sätze 1 bis N zu finden, sodass die Sätze nach der Ordnungsrelation der Schlüssel entsprechend richtig angeordnet sind. Mathematisch kann das Ergebnis wie folgt dargestellt werden

$$k_{(1)} \geq k_{(2)} \geq \ldots \geq k_{(N)} \tag{1}$$

[2, vgl. S. 79]

Diese simple Problemformulierung ist zwar im Rahmen dieser Arbeit ausreichend, beschreibt jedoch nicht alle in der Realität vorkommenden Parameter, die für die Lösung des Problems eine Rolle spielen. Zum Beispiel wird nicht beschrieben in welcher Form die Sätze gegeben sind, wie die Umordnung geschehen soll oder ob die Anzahl der Sätze bekannt ist. Jedoch soll die genaue Definition aller möglichen Parameter nicht im Rahmen dieser Arbeit liegen.

III. Algorithmik

Um Algorithmen nach Effienz zu messen und sie zu vergleichen muss man einige Grundlagen der Algorithmik verinnerlicht haben. Im folgenden werden diese grundlegenden Begriffe erklärt um Grundverständnis aufzubauen.

A. Der Algorithmusbegriff

Da sich die Arbeit primär mit Algorithmen beschäftigt ist der Algorithmusbegriff selbst wichtiges Basiswissen. Ein Algorithmus ist eine Vorschrift zur Bewältigung einer Aufgabe als Folge von Aktionen, wobei die folgenden Bedingungen gelten [4, vgl 2.1]:

- Der Algorithmus lässt sich in endlicher Form beschreiben und ist aus Elementen von endlich vielen, durch die ausführende Maschine definierten Kommandobefehlen, aufgebaut [4, vgl 2.1].
- Der Algorithmus hat genau eine Startaktion und nach der Ausführung eines jeden Befehls ist die Menge der Folgeaktionen klar durch die Beschreibung und die bisher durchgeführten Aktionen definiert [4, vgl 2.1].
- Die Eingabe des Algorithmus ist eine Folge von Daten, die auch leer oder unendlich sein kann. Aber zu jedem Zeitpunkt während der Ausführung ist die bisher betrachtete Datenmenge endlich [4, vgl 2.1].

Diese allgemeine Definition wird erst konkret wenn man sich das verwendete Maschinenmodell ansieht, auf dem der Algorithmus ausgeführt wird. Das Maschinenmodell definiert die zur Verfügung stehenden Operationen und Kontrollkonstrukte [4, vgl 2.1]. Die üblichen Maschinenmodelle in diesem Zusammenhang sind:

- **Turing-Maschine:** Endlich viele Zustände eines endlichen Automaten sowie ein potentiell unendlich langes Speicherband stehen als Speicher zur Verfügung.

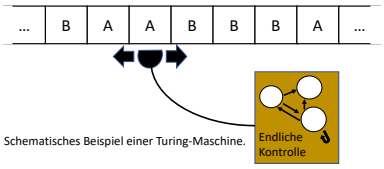

Abbildung 1. [4, vgl Kapitel 2.1 Seite 18]

Als Operationen kann der Lesekopf bewegt werden, sowie der Inhalt des Speicherbands gelesen und beschrieben werden. Die Kontrolle ist in einem endlichen Automaten abgelegt, der den implementierten Algoithmus darstellt. Eine schematische Darstellung des Maschinenmodells ist in Abbildung 1 zu sehen. Dieses Modell ist grundsätzlich genauso mächtig wie gängige Computer und als theoretisches Modell kann man es sogar mit beliebig großer Parallelität austatten [4, vgl 2.1].

- **Konkretes-Hardwarenahes-Maschinenmodell:** Konkrete Anweisungen für den Prozessor werden in einer Assemblersprache beschrieben.

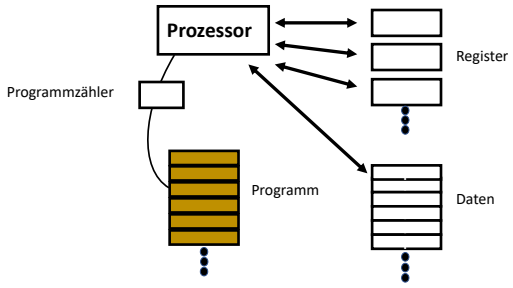

Abbildung 2. [4, vgl Kapitel 2.1 Seite 18]

Die Anzahl der verfügbaren Register ist, wie in Abbildung 2 zu sehen, eine offensichtliche Beschränkung des Modells. Dadurch ist die Beschreibung des Algorithmus jedoch in jedem Fall exakt, da kein aufwendiger Teil des Algorithmus in hochsprachlichen Anweisungen versteckt werden kann. Jedoch kann die Schreibweise oft leicht kryptisch wirken, was sich negativ auf die Lesbarkeit auswirkt und der fehlende Zwang zur Strukturierung steht im Gegensatz zu aktuellen Programmiersprachen [4, vgl 2.1].

- **Virtuelle Maschine einer Hochsprache:** Hier wird Gebrauch von einer konkreten Programmiersprache mit all ihren Konstrukten zu Ablaufkontrolle und ihren Basisbefehlen gemacht. Speicher wird in der üblichen Form als Variable deklariert und man macht sich kaum Gedanken über den Speicher, der der CPU zur Verfügung steht da die virtuelle Maschine diese Details verbirgt. Weiterführende Überlegungen der Effizienz und des Ressourcenbedarfs müssen allerdings teilweise sehr sorgfältig durchgeführt werden, was ein Nachteil der hochsprachlichen virtuellen Maschine ist [4, vgl 2.1].

Alle drei Modelle können im Hinblick auf die Anforderungen der Algorithmusdefinition als Grundlage für die Beschreibung eines Algorithmus benutzt werden.

B. Laufzeitbestimmung von Algorithmen

In der Algorithmik unterscheidet man bei der Laufzeit immer zwischen 3 Fällen, Best Case, Average Case und Worst Case. Der Best Case beschreibt den besten Fall, während der Average Case den durschnittlichen Fall und der Worst Case den schlechtesten Fall beschreibt. Zur Veranschaulichung ein Beispiel von drei Algorithmen in Abbildung 3

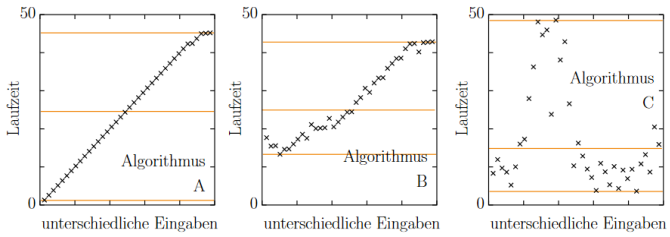

Abbildung 3. Die Laufzeit von drei Algorithmen wird für mehrere Probleminstanzen angezeigt. Die drei farbigen Linien markieren den Worst-Case (oben), Average-Case (mittig) und Best-Case (unten). Beim Vergleich der Best-Case Ergebnisse schneidet Algorithmus A am besten ab. Für den besten Average-Case wählen wir Algorithmus C. Algorithmus B wird verwendet, wenn der Worst-Case möglichst kleine Laufzeit liefern soll.[4, vgl Kapitel 2.3 Seite 27]

Die Laufzeit lässt sich als Funktion darstellen $T : \mathbb{N} \to \mathbb{R}^+$, welche abhängig von N ist. Dabei ist $n \in N$ die Problemgröße. Bei einem Sortieralgorithmus wäre n dann die Anzahl an zu sortierenden Elementen. Gerade bei Sortieralgorithmen ist für uns interessant wie stark die Laufzeit abhängig von der Problemgröße wächst [4, vgl. Kapitel 2.3]. Deswegen wurden die Landau Symbole eingeführt, Groß-O (O), Omega (Ω) und Theta (Θ). Die O-Notation beschreibt die obere Schranke der Funktion d.h. Beschränkung des Worst-Case, Omega die untere Schranke der Funktion d.h. die Beschränkung des Best-Case und Theta das Resultat aus beiden [4, vgl. Kapitel 2.3]. Zur Veranschaulichung ist in Abbildung 4 ein Vergleich der Wichtigsten und am häufigsten vorkommenden Zeitkomplexitäten. Die Abbildung lässt erahnen wie schnell welche Zeitkomplexität mit der Problemgröße wächst.

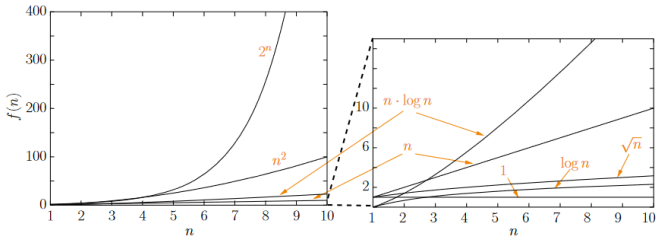

Abbildung 4. [4, vgl Kapitel 2.3 Seite 28]

C. Das "Teile und Herrsche"Prinzip

Das Prinzip des Teilen und Herrschens ist eines der wichtigsten Prinzipien für effiziente Algorithmen. Dabei wird der Fakt genutzt, dass bei der Lösung eines Problems oftmals der Problemaufwand sinkt, wenn das große Problem in mehrere kleinere Probleme geteilt wird, welche dann leichter zu "beherrschen" sind. Oftmals werden die Probleme solange geteilt bis sie sofort lösbare triviale Probleme darstellen. Dies lässt sich durch rekursive Programmierung verwirklichen. Abschließend werden die trivialen Einzellösungen der Teilprobleme wieder zu einer Gesamtlösung des Gesamtproblems vereint. Die bekanntesten Vertreter der Sortieralgorithmen die nach diesem Prinzip arbeiten sind Mergesort und Quicksort.

D. Speicherbedarf

Ein abschließender Punkt zur Algorithmik, der im Rahmen dieser Arbeit kurz erläutert werden soll, ist der Speicherbedarf. Es gibt sogenannte in-place und out-of-place Algorithmen. In-place bedeuetet, dass der Speicherbedarf des Algorithmus konstant ist und er keinen zusätzlichen Speicherbedarf benötigt. Out-of-place Algorithmen hingegen benötigen je nach Problemgröße beliebig viel zusätzlichen Speicher. Bei großen modernen Systemen mit viel Speicher ist diese Unterscheidung nicht relevant. Jedoch ist das Nutzerspektrum in Bezug auf die Systeme bei Sortieralgorithmen sehr groß, was bedeutet, dass z.b. bei Embedded Devices bei denen der Arbeitsspeicher sehr begrenzt ist doch darauf geachtet werden muss, welchen Algorithmus man benutzt.

IV. Paralleles Sortieren

Die bisherige Arbeit befasste sich mit der Algorithmik und dem allgemeinen Sortiervorgang bei dem von einer sequentiellen Abarbeitung der Befehle ausgegangen wird. Da sich Computer Platformen weiterentwickeln, ist es nötig zu den Neuerungen in der Rechnerarchitektur dementsprechende effiziente Sortiertechniken zu finden bzw. vorhande Techniken anzupassen. Einer der offensichtlichen Trends der letzen Jahre ist die vermehrte Parralelität auf Chip-Basis. Multicore CPU´s gehören heute zum Standard und es gibt jeden Grund zur Annahme, dass dieser Trend in Richtung mehr Parallelität nicht abreißen wird. An der Spitze dieses Trends sind die GPU´s. Moderne NVIDIA GPU´s haben bis zu 128 Prozessor Elemente pro Chip, die sich mit CUDA direkt in C programmieren lassen [5, vgl Seite 1].

A. CUDA

CUDA ist eine, im Jahre 2007, von Nvidia entwickelte Programmier-Technik mit welcher Programmteile durch den Grafikprozessor abgearbeitet werden können. Dies ist gerade bei hochgradig parallelisierbaren Programmabläufen hilfreich, da die GPU dort deutlich schneller arbeitet als die CPU. CUDA wird normalerweise in C geschrieben. In einem CUDA Programmier Modell unterscheidet man zwischen zwei verschiedenen Teilen. Der sequentielle Teil für den Host und der parallele Teil, der auf einem externen parallel arbeitendem Gerät, meistens einer GPU, ausgeführt wird [5, vgl Seite 2]. Ein CUDA Programm wird von dem NVCC(Nvidia CUDA Compiler) kompiliert, indem es die CUDA Schlüsselwörter benutzt um den Host Code vom Kernel Code zu unterscheiden. Der Host Code ist normaler C Code, der mit dem vorhandenen Standard C/C++ Compiler kompiliert wird und als klassischer CPU Prozess ausgeführt wird. Der Code für das externe Gerät ist mit CUDA Schlüsselwörtern markiert um parallele Funktionen zu markieren, welche Kernels genannt werden. Dieser wird weiter von einer Laufzeit Komponente von NVCC kompiliert und anschließend von der GPU als tausende von Threads ausgeführt. In Abbildung 5 ist das Kompilieren eines CUDA Programms noch einmal bildlich dargestellt.

Das Prinzip von CUDA ist also die Parallelität der Datenverarbeitung einer GPU für andere Zwecke zu nutzen als die

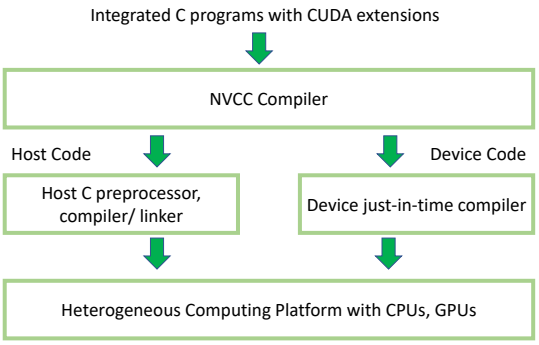

Abbildung 5. [6, vgl Kapitel 3.2]

Berechnung grafischer Umgebungen. Zum Beispiel auch für die Ausführung von Sortieralgorithmen.

B. Sortieralgorithmen

In folgendem Abschnitt werden einige wichtige Sortieralgorithmen vorgestellt und zusätzlich erläutert, wie diese parallelisert werden können. Da es durch die Vielzahl an Algorithmen, im Rahmen dieser Arbeit, nicht möglich ist jeden einzeln zu betrachten, werden nur 3 Beispielalgorithmen besprochen. Sollte im folgenden Teil dieser Arbeit ein nicht besprochener Sortieralgorithmus namentlich genannt werden, wird auf eine Arbeit verwiesen, in der dieser besprochen wird.

1) Radix-Sort: Der Radix-Sort Algorithmus ist einer der ältesten Algorithmen und wird zu den effektivsten gezählt, wenn es darum geht kleine Tupel zu sortieren. Radix-Sort Algorithmen lösen Sortierprobleme der Komplexität n. Damit dies effizient passieren kann müssen die Schlüsselelemente die gleiche Länge l besitzen. Durch eine zusätzliche Abfrage kann Radix Sort auch Schlüsselelmente mit unterschiedlicher Länge verarbeiten, dies verschlechtert allerdings die Laufzeitkomplexität. Bei n l stelligen Elementen ergibt sich eine Laufzeit von $\mathcal{O}(d\,n)$. Eine weitere Einschränkung des Radix-Sort ist, dass die Schlüsselelemente einem endlichen Alphabet der Länge m angehören müssen. Somit gilt z.B. für die Dezimalzahlen $m = 10$ oder für das Alphabet $m = 26$. Der Algorithmus selbst besteht aus d Zügen die jeweils die i´te Nummer des Schlüsselelements betrachten und die Eingabesequenz dementsprechend ordnen. Dabei wird beim least significant bit des Schlüssels gestartet, um sich dann zum most significant bit vorzuarbeiten. Man unterscheidet bei dem Sortiervorgang der Züge zwischen Counting sort und Bucket sort [5, vgl Kapitel 4]. In einem Zug kann jeder Schlüssel in einen von r Eimern bzw. buckets platziert werden. Um zu berechnen an welchem Index des Outputs das Element stehen muss, was man auch als Rang des Elements bezeichnet, müssen die Anzahl der Elemente in niedrigeren Eimern gezählt und mit der Anzahl der Elemente die sich schon im aktuellen Eimer befinden addiert werden. Nachdem der Rang jedes Elements bestimmt wurde, können die Elemente dementsprechend geordnet im Output Array verteilt werden [5, vgl Kapitel 4].

Zur Veranschaulichung ist in Abbildung 6 ein simples Radix-Sort Beispiel dargestellt.

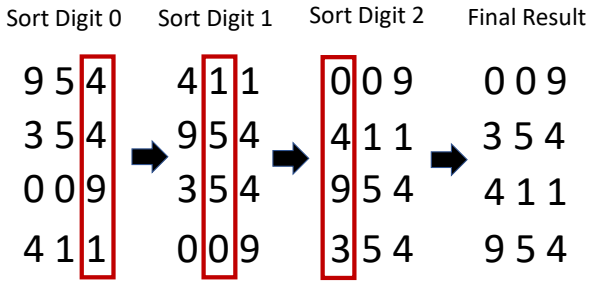

Abbildung 6. [7, vgl abstract]

Es gibt großes Potential für Parallelität im Counting-Sort Teil jedes Zuges des Radix-Sort Algorithmus. Im einfachsten Fall wird 1bit jedes Schlüssels pro Zug betrachtet. Die Berechnung des Rangs jedes Elements kann mit einer einzigen parallelen prefix Summenbildung oder Scan Operation erfolgen. Diese Herangehensweise den Radix Sort Algorithmus zu implementieren ist sehr simpel, jedoch nicht gerade effektiv wenn sich die Arrays in einem externen Speicher befinden. Für jedes Bit, Länge der Schlüssel wird eine Operation zum globalen Speicher verteilt, welche die gesamte Sequenz ordnet und sortiert. Da der Datentransfer zu bzw. von externem Speicher auf modernen Prozessorn relativ teuer ist, vermeidet man diese Menge an Datenverkehr soweit möglich [5, vgl Kapitel 4.1]. Man kann jedoch die Menge an verteilten Operationen vermindern, indem man anstatt einem Bit pro Zug, b Bits pro Zug betrachtet. Somit verringert sich die Anzahl der nötigen Operationen. Vorraussetzung ist ein Bucket-Sort mit 2^b Buckets pro Phase. Um den Bucket Sort zu parallelisieren teilt man die Input Sequenz in Blöcke, welche dann separat sortiert werden. Die Ergebnisse werden so gespeichert, dass man mit einer einzigen Scan Operation den Offset der Buckets in jedem Block bestimmen kann [5, vgl Kapitel 4.1]. Die Implementierung eines Radix-Sort Algorithmus auf der GPU wird in [5, Kapitel 4.2] beschrieben. Diese verfolgt die gleiche Idee wie gerade beschrieben. Dabei sind jedoch die wichtigsten Punkte die Menge an Operationen, die auf den globalen Speicher zugreifen müssen zu minimieren, damit man die Speicherbandbreite effizient nutzt [5, vgl Kapitel 4.2]. Jeder Schritt des Radix-Sort wird mit 4 seperaten CUDA Kernels implementiert.

1) Jeden Block auf dem On-Chip Speicher nach der i´ten Nummer sortieren
2) Die Offsets für jeden r Bucket berechnen und sie global speichern.
3) Die Präfix Summe der entstandenen Tabelle bilden
4) Die finale Position jedes Elements in der sortierten Ausgabeliste mit Hilfe der Präfix Summe berechnen und die Elemente zu ihren vorgesehenen Positionen verteilen

Der gesamte Algorithmus im Detail ist in [5, Kapitel 4.2] beschrieben.

2) Merge-Sort: Ein weiterer Beispielalgorithmus ist der Merge-Sort Algorithmus, der nach dem "Teile und Herrsche" Prinzip arbeitet. Der Algorithmus teilt den zu sortierenden Tupel in mehrere Blöcke, sortiert diese und fügt sie wieder zu einem sortierten Array zusammen. Dabei werden die Blöcke oft so lange weiter geteilt bis sie triviale Teilprobleme darstellen. Neben dem Sortieren ist die Prozedur des Zusammenfügens allein schon sehr hilfreich. Beispielsweise wenn neue Elemente in eine bereits sortierte Liste eingefügt werden soll, ohne diese Sortierung fehlerhaft zu machen [5, vgl Kapitel 5]. Das Sortieren der einzelnen unabhängigen Blöcke kann natürlich parallelisiert werden und mit mehreren Sortiertechniken wie Radix-Sort oder Quick-Sort [13, Kapitel 2] erzielt werden. Die Zeitkomplexität von Merge-Sort beträgt $\mathcal{O}(n \log n)$.

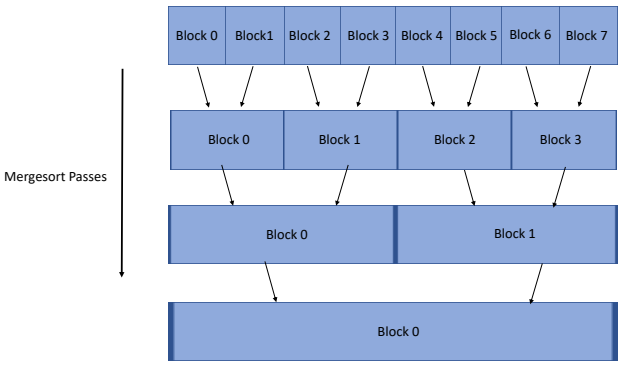

Abbildung 7. [5, vgl. Kapitel 5.2]

Die Prozedur des Zusammenfügens von unabhängigen Blöcke, welche in Abbildung 7 zu sehen ist, weißt von Natur aus schon eine Paralellität auf. Während jedoch die Größe der Blöcke bei jedem Schritt steigt, sinkt gleichzeitig auch die Anzahl der übrigen Blöcke. Daraus entsteht eine sehr grobe Parallelität, die nicht die mögliche Effizienz einer parallel arbeitenden Architektur vollstens nutzt. Ziel ist es also den Merge-Sort so zu gestalten, dass im paarweisen zusammenfügen der sortierten Blöcke, möglichst fein, die volle Paralelität ausgenutzt wird. Eine effiziente Methode dafür wird in [9] vorgestellt, indem man die zwei Sequenzen, die anschließend wieder zusammengefügt werden nach ihren Elemente Rängen als Teilwerte aufteilt. Der Rang eines Array Elements ist die Anzahl der Elementen, die kleiner oder gleich groß sind wie das gegebene Element. Zwei Sequenzen, die wieder vereint werden, werden nach den gleichen Zerteilwerten partitioniert, somit hat jeder entstehende Unterblock einen dazugehörigen Unterblock in der anderen Sequenz. Idealerweise sollte die Zeitkomplexität des Zusammenfügens aber von der Gesamtanzahl der Blöcke abhängen und nicht von der Art und Weise wie sie geteilt wurden, da so ein konstanter Speedup durch den kompletten Vorgang des Zusammenfügens entstehen kann[5, vgl. Kapitel 5.2]. Ein weiterer effizienter Algorithmus, wie man sortierte Blöcke parallel mit Hilfe der Binären Suche zusammenfügt, wird in [8] vorgestellt [5, vgl. Kapitel 5.1]. In [5] wird ein paralleler Merge-Sort Algorithmus für GPU-Sorting vorgestellt. Dabei werden, wie bereits beschrieben, die Daten in k Blöcke geteilt, wobei beachtet wird, dass die Blöcke klein genug sind um in den GPU eigenen on-chip Speicher zu passen, parallel sortiert und anschließend in $logk$ Schritten parallel zusammengefügt werden. Für das Sortieren wird ein Bitonic-Sort Algorithmus benutzt. Mehr zu diesem Algorithmus in [11]. Bei dem Merge-Verfahren wird man immer in die Situation kommen, b sortierte Blöcke zusammenfügen zu müssen, die zusammen alle n input Elemente besitzen. Nun ist das Vorgehen, dass man ein zufälliges Element e aus zwei Blöcken nimmt, seinen Rang ermittelt und den Block an diesem Element in zwei Teile teilt. Selbiges tut man auch im anderen Block. Nun hat man 4 Teile A,B,C und D wobei gilt $i \in [A, C] < e$ und $i \in [B, D] > e$ während i für jedes beliebige Input-Element steht. Nun kann man jeweils A mit C, und B mit D zusammenfügen und die zwei Ergebnisse konkatenieren um den finalen zusammengefügten Block zu erhalten [5, vgl. Kapitel 5.2]. Eine Visualisierung dieses Vorgehens ist in Abbildung 8 abgebildet.

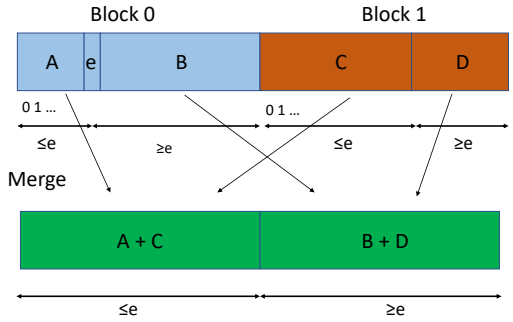

Abbildung 8. [5, vgl. Kapitel 5.2]

Dieses Verfahren verdoppelt die mögliche Parallelisierbarkeit. In der Praxis werden mehrere Werte e und deren Rang ermittelt, somit können mehrere Wertemengen ermittelt werden, die anschließend mit dem entsprechenden Teil des anderen Blocks konkateniert werden können. Dadurch nutzt man die Paralellität aus. Zusätzlich wird jedes 256. Elemente als Wert e benutzt, damit man sicher stellt, dass die entstehenden Unterblöcke auf den schnellen GPU On-Chip Speicher passen [5, vgl. Kapitel 5.2]. Der gesamte Algorithmus wird in [5, Kapitel 5.2] genauestens erklärt.

3) Bubble-Sort: Als letzten Beispiel Algorithmus wird der Bubble-Sort Algorithmus betrachtet. Beim Bubble-Sort wird ein Tupel von links nach rechts durchlaufen. Bei jedem Schritt wird dabei das aktuelle Element mit dem darauf folgenden Element verglichen. Stimmt die Reihenfolge der Elemente mit der gewünschten Sortierung überein, wird in den nächsten Schritt übergegangen und das vorherige Folge-Element wird wiederum mit seinem Folge-Element verglichen. Stimmt die Reihenfolge nicht, werden die Elemente getauscht und anschließend wird das nächste Element betrachtet. Dieses

Verfahren wird so lange wiederholt bis die Sortierung fertig ist. Bei jedem Schritt muss das zuletzt betrachtete Element des vorherigen Schritts nichtmehr betrachtet werden, da bei diesem Verfahren je nach gwünschter Sortierrichtung der größte bzw. kleinste Wert immer nach ganz hinten geschoben wird. Die Zeitkomplexität von Bubble-Sort beträgt $\mathcal{O}(n^2)$. Es folgt der Pseudo Code einer möglichen Implementierung des Bubble-Sort Algorithmus aus [10, Kapitel 4].

```
FOR passnum BETWEEN LengthOf(array) AND 0
  FOR i BETWEEN 0 AND passnum
    IF(Array[i] > array[i+1])
      SWAP(array[i] WITH array[i+1])
    ENDIF
  ENDFOR
ENDFOR
```

Zum Vergleich folgt hier der Pseudo Code einer möglichen parallelen Implementierung des Bubble-Sort Algorithmus aus [10, Kapitel 4].

```
idx = thread_id, N = length_of(array)-1
FOR i = idx BETWEEN 0 AND N
  FOR j BETWEEN 0 AND N-1-i
    IF(array[j] > array[j+1])
      SWAP(array[j] WITH array[j+1])
    ENDIF
  ENDFOR
ENDFOR
```

In der ersten Zeile wird die Variable idx der dem Wert der Thread-ID zugewiesen. Das beduetet, dass bei jedem Durchlauf des Kernels die Thread-ID einzigartig ist. Diese Zeile ist für die korrekte Verarbeitung und Paralleliserung des Codes sehr wichtig, da der Code sonst sequentiell durchgeführt werden würde. Die Variable N entspricht der Länge des Input Arrays. In der zweiten Zeile folgt eine Schleife, bei der jede Thread-ID der Schleifenvariable zugewiesen wird. Anschließend beginnt eine Schleife für die Bestimmung des aktuellen Array Indexes, worauf eine Vergleichsabfrage der zwei Indizes folgt, wobei anschließend bei Bedarf die Werte der Indizes getauscht werden [10, vgl. Kapitel 4].

V. Effizienz

Im folgenden Kapitel wird die Frage geklärt, wie effizient parallele GPU Implementierungen von Sortieralgorithmen im Vergleich zu den parallelen CPU Implementierungen sind. In Abbildung 9 kann man sehen wie sich die in dieser Arbeit beschriebenen Methoden auf die Effizienz auswirken. Verglichen wird die Sortier-Rate in Abhängigkeit von der zu sortierenden Sequenzgröße. Dabei werden die zwei in Kapitel 4 vorgestellten Implementierungen des Merge-Sort und des Radix-Sort Algorithmus mit parallelen CPU Implementierungen auf einem 8Kern Prozessor verglichen. Der in dieser Arbeit vorgestellte Radix-Sort Algorithmus ist, wie in Abbildung 9 zu sehen, mit Abstand am effizientesten. Auch der in dieser Arbeit vorgestellte Merge-Sort Algorithmus kann mit den CPU-Implementierungen mithalten und ist sogar an vielen Stellen effizienter. Wobei der Radix-Sort ab einer Sequenzlänge von 10.000 Elementen der mit Abstand effzienteste Algorithmus ist. Dies ist dadurch möglich, so viel Arbeit wie möglich auf dem schnellen on-chip Speicher der Grafikkarte zu konzentrieren und feinen Parallelismus zu nutzen um die volle Effizienz aus den tausenden möglichen parallelen Threads dieser Architektur zu schöpfen [5, vgl Kapitel 7].

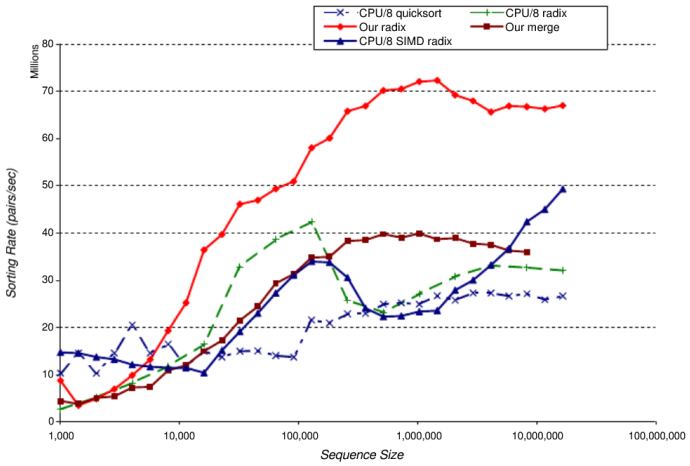

Abbildung 9. [5, vgl. Kapitel 7]

Es ist selbstverständlich, dass parallele Implementierungen von Sortieralgorithmen auf einer hoch parallel arbeitenden Architektur wie einer GPU, weitaus effizienter sind als die sequentielle Ausführung auf einer CPU. Als Verdeutlichung wie viel schneller parallele GPU Implementierungen arbeiten ist in Abbildung 10 der Speed-Up von einer sequentiellen Implementierung zu einer parallel arbeitenden GPU Implementierung von verschiedenen Sortieralgorithmen unter verschiedenen Ausgangssituationen dargestellt.

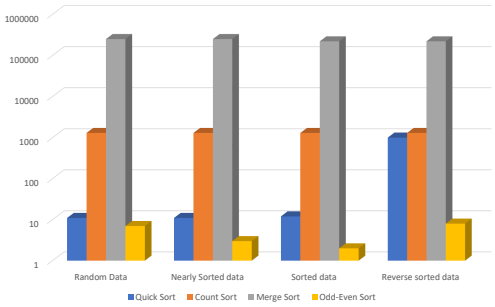

Abbildung 10. [12, vgl. Kapitel 3]

Der Speed-Up sagt aus wie viel schneller der Sortiervorgang durch die Parallelisierung erfolgt ist. Wie man sehen kann profitiert nicht jeder Algorithmus gleich viel von der Paralellität. Gerade beim Merge-Sort erreicht man jedoch imense Speed-Up Zahlen, da der Merging-Schritt sehr aufwendig ist und diesen effizient zu parallelisern der Effizienz enorm hilft. Mehr Informationen zum hier präsentierten Quick-Sort Algorithmus und Odd-Even-Sort Algorithmus findet man in [13, Kapitel 2] und [14, Kapitel 2].

VI. Conclusion

Diese Arbeit hat gezeigt, dass moderne GPU Implementierungen von Sortieralgorithmen gegenwärtig die effektivste Möglichkeit des Sortierens sind. Durch die hohe Parallelität der Grafikkartenarchitektur kann man mit einer Implementierung, die diese Architektur vollstens nutzt, hohe Effizienz beim Sortiervorgang erreichen. Diese Effizienz übersteigt das aktuell mögliche auf CPU Implementierungen, da die GPU hoch parallel arbeitet. Um auf die grundlegende Frage der Arbeit zurückzukommen, ob moderne GPU Implementierungen von Sortieralgorithmen die Lösung des Sortierproblems sind, muss man differenzieren, was man unter einer Lösung des Sortierproblems versteht. Es sollte keine große Überraschung sein, dass von einer finalen Lösung des Sortierproblems nicht die Rede sein kann. Vielmehr ist die Lösung des Sortierproblems als gegenwärtig effizienteste Lösung zu betrachten. Und moderne GPU Implementierungen von Sortieralgorithmen sind aktuell genau das. Durch die technische Entwicklung und die Weiterentwicklung der Rechnerarchitekturen geht jedoch auch die Forschung im Hinblick auf das Sortierproblem immer weiter. Die in dieser Arbeit vorgestellten Methoden sind auf Grundlage der Frage, wie man aktuelle hoch parallele Architekturen effizient zum Sortieren nutzen kann bzw. wie man die Sortieralgorithmen auf hoch parallele Architekturen wie GPU´s effizient anpassen kann. In diesem Hinblick ist der aktuelle Stand sehr gelungen, da aktuelle GPU Implementierungen, wie in Kapitel 4 gezeigt, effizienter sind als CPU Implementierungen wie in Kapitel 5 gezeigt. Somit kann man die Themafrage mit ”Ja” beantworten und im selben Zuge einen Ausblick auf die Zukunft geben, da mit der technischen Weiterentwicklung und dem aktuellen Trend der immer größeren Parallelität in Hardware, auch immer neue Lösungen zur Verbesserung von Sortieralgorithmen entworfen werden.

Literatur

[1] N. Satish, M. Harris and M. Garland, ”Designing efficient sorting algorithms for manycore GPUs,”2009 IEEE International Symposium on Parallel and Distributed Processing, 2009, pp. 1-10, doi: 10.1109/IPDPS.2009.5161005.

[2] Thomas Ottman und Peter Widmayer. Algorithmen und Datenstrukturen. 6. Aufl. Springer Verlag, 2017

[3] . H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein,Introductionto Algorithms, Second ed. MIT Press, Sep. 2001

[4] Karsten Weicker und Nicole Weicker. Algorithmen und Datenstrukturen Springer Verlag, 2013

[5] Satish, Nadathur and Harris, Mark and Garland, Michael. (2009). Designing efficient sorting algorithms for manycore GPUs. Parallel and Distributed Processing, 2009 IPDPS 2009 IEEE International Symposium on: 2009. 23. 1-10. 10.1109/IPDPS.2009.5161005.

[6] D. B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors: A Hands-On Approach (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010

[7] Tristram, Dale and Bradshaw, Karen. (2014). Identifying attributes of GPU programs for difficulty evaluation. South African Computer Journal. 53. 10.18489/sacj.v53i0.195.

[8] D. Z. Chen. Efficient parallel binary search on sorted ar- rays, with applications. IEEE Transactions on Parallel and Distributed Systems, 6(4):440–445, 1995.

[9] T. Hagerup and C. Rub. Optimal merging and sorting on the EREW PRAM. Information Processing Letters, 33:181– 185, 1989.

[10] Yazici, Ali and Gokahmetoglu, Hakan. (2016). Implementation of Sorting Algorithms with CUDA: An Empirical Study. International Journal of Applied Mathematics, Electronics and Computers. 4. 74. 10.18100/ijamec.53457.

[11] M. F. Ionescu and K. E. Schauser, Öptimizing parallel bitonic sort,”Proceedings 11th International Parallel Processing Symposium, 1997, pp. 303-309, doi: 10.1109/IPPS.1997.580914.

[12] N. Faujdar and S. Saraswat, Ä roadmap of parallel sorting algorithms using GPU computing,”2017 International Conference on Computing, Communication and Automation (ICCCA), 2017, pp. 736-741, doi: 10.1109/CCAA.2017.8229919.

[13] H. Tang, S. Geng, X. Peng, S. Yan, Y. Zhang and Z. Wang, Ä Design of ID Sorting Module Based on Quick Sorting Algorithm,”2020 IEEE 5th International Conference on Integrated Circuits and Microsystems (ICICM), 2020, pp. 228-232, doi: 10.1109/ICICM50929.2020.9292141

[14] P. Tarasiuk and M. Yatsymirskyy, Öptimized Concise Implementation of Batcher’s Odd-Even Sorting,”2018 IEEE Second International Conference on Data Stream Mining and Processing (DSMP), 2018, pp. 449-452, doi: 10.1109/DSMP.2018.8478515.