

Hashing Verfahren und Hash-Join auf Multi-GPU Systemen

Vincent Gerber

Geboren am: 23. Februar 1998 in Görlitz
Matrikelnummer: 4607473
Immatrikulationsjahr: 2016

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker (Dipl.-Inf.)

Betreuer

Prof. Dr.-Ing. habil. Dirk Habich

Dipl.-Inf. Johannes Fett

Betreuer Hochschullehrer

Prof. Dr.-Ing. habil. Wolfgang Lehner

Eingereicht am: 8. Dezember 2021

Zusammenfassung

Das Hash-Join-Verfahren ist eine Methode, welche die viel genutzte Join-Funktion in Datenbanken umsetzt. Bekannte Implementierungen verwenden hierzu entweder mehrere CPUs oder CPU-GPU-Systeme, um die Operation so schnell wie möglich durchzuführen. Eine GPU allein ist bisher nur selten in Erscheinung getreten, weil der interne Speicher viel zu klein für die anfallenden Datenmengen war. Doch mit der Zeit haben sich die GPUs weiterentwickelt und der interne Speicher stieg mit jeder Generation. Aus diesem Grund befasst sich diese Arbeit mit der Implementierung und Analyse des Hash-Joins auf der GPU. Doch bevor der Hash-Join seine Anwendung findet, müssen die Tabellen-Werte gehasht werden. Für diesen Zweck werden unterschiedliche Hash-Methoden vorgestellt und auf verschiedene Güte-Kriterien untersucht. Das besondere Ziel besteht auch darin, das Verfahren speziell auf die GPU-Hardware anzupassen und somit die Hardware vollständig einzubeziehen. Das umfasst auch die Verwendung von zwei GPUs. Mit den vorgestellten Hash-Verfahren folgt die Integration in den In-Memory Hash-Join. Dieser besteht hauptsächlich aus der Radix-Partitionierung und dem Probing. Die Radix-Partitionierung und das dazugehörige Vektorisieren, werden von Grund auf implementiert. Das Probing leitet ein Schlüssekonzept, von vorhandenen Arbeiten ab. Für die Partitionierung werden zwei Varianten vorgestellt, welche die Nutzung von mehr als einer GPU ermöglichen. Das Hashing kann in der Evaluation eine Datenrate von 1.75 bis 6 Milliarden Hashes pro Sekunde erreichen. Der gesamte Hash-Join erreicht eine Tupel-Rate von bis zu 35 Millionen Tupel pro Sekunde. Aufgrund der geringen Datenrate im Hash-Join, erfolgt eine ausführliche Analyse zur Radix-Partitionierung und der Vektorbildung. Mit der Analyse wird gezeigt, dass Radix-Partitioning und das damit verbundene Vektorisieren, an den Limitierungen der Probe-Kernels scheitert.

Inhaltsverzeichnis

1	Einleitung	10
2	Grundlagen	12
2.1	Hashing	12
2.1.1	Eigenschaften	12
2.1.2	Funktion	14
2.1.3	Anwendung	14
2.1.4	Hashtabelle	16
2.2	Join	17
2.2.1	Funktion	17
2.2.2	Algorithmen	18
2.3	GPU	19
2.3.1	Aufbau	19
2.3.2	Programmierung	21
2.3.3	Speicher	22
2.3.4	Multi GPU	24
2.3.5	Historische Entwicklung	24
2.4	Zipf Verteilung	24
3	Analyse und eigene Konzepte	26
3.1	Hashing	26
3.1.1	Zielstellung	26
3.1.2	Algorithmen	27
3.1.3	Speicherverwaltung	34
3.1.4	Multi GPU	36
3.1.5	Zusammenfassung	36
3.2	Hash-Join	37
3.2.1	Zielstellung und Annahmen	37
3.2.2	R/S Datensatz	38
3.2.3	Speicherlimitierungen	39
3.2.4	Hashing	39
3.2.5	Probing	40
3.2.6	Build	41
3.2.7	Join	42
3.2.8	Vektorisierung	44
3.2.9	Multi GPU	48
3.2.10	Parameter	49

3.2.11 Zusammenfassung	49
4 Evaluation	50
4.1 Aufbau	50
4.2 Hashing	50
4.2.1 Eigenschaften	51
4.2.2 Verteilung	52
4.2.3 Vektor- und Datengröße	59
4.2.4 Kernel Parameter	62
4.2.5 Multi GPU	67
4.2.6 Zusammenfassung	69
4.3 Hash-Join	70
4.3.1 Verteilung	70
4.3.2 Probe	71
4.3.3 Vektorisierung	76
4.3.4 Zusammenfassung	86
5 Related Work	88
6 Fazit	89
6.1 Ausblick	90
7 Anhang	95

Abbildungsverzeichnis

2.1	Eine einfache Hash-Funktion	14
2.2	SHA-256 Schleife	15
2.3	Hashtabelle Separate-Chaining Implementierung	16
2.4	Inner Join	18
2.5	CPU vs GPU [20]	19
2.6	Grid, Block, Thread [20]	20
2.7	Turing Streaming Multiprozessor [21]	20
2.8	Streaming [21]	22
2.9	Speicherhierarchie	23
2.10	Multi-GPU Testsetup	23
2.11	Zipf Verteilung	25
3.1	Hashing auf mehreren GPUs	35
3.2	Tabellen Datenformat	38
3.3	GPU Hashtabelle Aufbau	40
3.4	Radix Partitionierung	44
3.5	Hash-Join auf Multi-GPU-Systemen	47
4.1	Verteilung der Testwerte für die Eigenschaftstests	51
4.2	Hash-Verteilung 32-Bit	53
4.3	Hash-Verteilung 64-Bit	54
4.4	Hash-Abstand 32-Bit	55
4.5	Hash-Abstand 64-Bit	56
4.6	Hash-Kollision 32-Bit	57
4.7	Hash-Kollision 64-Bit	58
4.8	Einfluss der Vektorgröße auf die Hash-Funktionen	59
4.9	Einfluss der Vektorgröße auf die Geschwindigkeit	60
4.10	Einfluss von Elemente/Hash, auf die Geschwindigkeit	61
4.11	Einfluss der Hash-Größe, auf die Geschwindigkeit	62
4.12	Kernel-Profile für einzelne Hash-Funktionen	63
4.13	Kernel-Profile für einzelne Thread/Block-Konfigurationen	64
4.14	Hashing mit unterschiedlichen Thread pro Block	65
4.15	Kernel-Profile für berechnete Hashes pro Thread	66
4.16	Kernel-Profile für einzelne Eingabe-Größen-Konfigurationen	67
4.17	Hashing mit unterschiedlichen GPU-Strategien	68
4.18	Hashing mit mehreren GPUs	69
4.19	Verteilungen der Testdaten	70

4.20 Tuples/Sekunde in Abhangigkeit zur Buffer-Groe	72
4.21 Einfluss der Probing Kernel-Parameter auf die Laufzeit	73
4.22 Probing mit unterschiedlichen Verteilungen	73
4.23 Probing mit mehreren Spalten	74
4.24 Tuples/Sekunde in Abhangigkeit zur Elementanzahl	74
4.25 Einfluss der Probing Kernel auf die Laufzeit	75
4.26 Tuples/Sekunde in Abhangigkeit zum Probe-Modus	75
4.27 Einfluss der Gleich- und Zipf-Verteilung, auf den Hash-Join	77
4.28 Einfluss der Verteilungen auf das Vektorisieren und Probing im Join	77
4.29 Unterschiedliche RS-Verhaltnisse im Vergleich	79
4.30 Der Einfluss der Hash-Funktion auf die Partitionstiefe	79
4.31 Der Einfluss der Partition-Buckets auf Partitionstiefe	80
4.32 Unterschiedliche Partitionenanzahl bei der Partitionierung	81
4.33 Histogram- und Swap-Kernel im Vergleich	81
4.34 Einfluss der Stream-Anzahl auf die Tupel-Rate	82
4.35 Stream-Verteilung in der Partitionierung	82
4.36 Einfluss der Vektorgroe auf die Funktionen	83
4.37 Partitionstiefen in Abhangigkeit zur Vektorgroe	83
4.38 Der Einfluss des Join-Modes auf die einzelnen Teifunktionen	84
4.39 Einfluss der GPU-Anzahl auf die Vektorisierung	85
4.40 Merging im Multi-GPU Vergleich	86
4.41 Hash-Join Tupel-Rate im Multi-GPU-System	87
7.1 Einfluss der Vektorgroe auf die Funktionen	96

Tabellenverzeichnis

3.1 Konstanten für die FNV-Hashfunktion	28
4.1 PTX Code von Load-Befehlen, für jeden Chunk-Typ	60

Algorithmenverzeichnis

1	Naive Join	18
2	FNV-1a Hash-Funktion	28
3	Addition Hash-Funktion	29
4	Shifted Addition Hash-Funktion	30
5	Multiplikation Hash-Funktion	30
6	Shifted Multiplikation Hash-Funktion	31
7	Shifted XOR Hash-Funktion	32
8	XOR Mult Hash-Funktion	33
9	HW XOR Hash-Funktion	33
10	Hashing der Tabelle	39
11	Probing in Hash-Tabelle	40
12	Hashtabelle Build-Phase	41
13	Rekursives Radix-Partitioning	45
14	Bucket-Vectorization	46

1 Einleitung

Wenn es um das Thema Datenbanken und deren Operationen geht, dann ist die Join-Funktion eine der häufig genutzten Funktionen [1]. Sie wird verwendet, um einzelnen Tabellen über eine Relation zusammenzuführen. Als Beispiel dient dabei die Zuordnung von Person und Wohnort oder Fahrzeug und Hersteller. In jedem Wohnort gibt es mehrere Personen und jeder Hersteller hat mehr als ein Fahrzeug. Aus diesem Grund liegen beide Subjekte in unterschiedlichen Tabellen, welche die Einträge aus den anderen Tabellen über eine ID referenzieren. Mit dieser ID erfolgt dann der Join, welcher die Informationen in einer Tabelle zusammenführt. Für die Art und Weise der Operation, lassen sich unterschiedliche Methoden anwenden. In dieser Arbeit erfolgt nur der Inner-Join. Der Inner-Join vergleicht alle Werte miteinander und bildet Paare aus den übereinstimmenden Spalten. Die Paare werden darauf in eine neue Tabelle übertragen, welche die Relation von beiden Subjekten beinhaltet. Um diese Aufgabe durchzuführen, können verschiedene Algorithmen und Systeme verwendet werden. Eines davon ist ein reines GPU-System, welches in dieser Arbeit genauer untersucht wird. Der Join erweist sich in der Regel als sehr ressourcenhungrige Operation, weshalb Optimierungen, speziell für Anwendung mit GPUs, erforderlich sind. Im Fall dieser Arbeit, erfolgt die Optimierung mit maximal zwei GPUs. Diese erzielen durch ihre hohe Thread-Anzahl und der hohen internen Speicherbandbreite, deutlich bessere Datenraten als manche CPU-Implementierungen [2]. Aus diesem Grund ist auch in dieser Arbeit das Ziel, den Join vollständig auf die GPU zu überführen. Die CPU übernimmt in diesen Fall nur noch die Steuerung der GPUs. Als Herausforderung gilt es, den begrenzten Global-Memory sinnvoll zu nutzen. Dieser steht jeder GPU unterschiedlich groß zur Verfügung und beträgt in dieser Arbeit, mit einer Quadro 8000 RTX, bis zu 50GB. Von den vielen verschiedenen Join-Algorithmen [3, 4], wird der Radix Hash-Join implementiert. Dieser besteht aus dem Hashing des Datensatzes, worauf die Radix-Partitionierung und darauf der Join stattfindet. Alle drei Schritte werden in den folgenden Abschnitten genauer betrachtet. Eine gesonderte Stellung bekommt das Hashing, bei dem neue Hash-Methoden gesucht und analysiert werden. Der Schwerpunkt liegt dabei darauf, diesen so effizient wie möglich auf der GPU auszuführen und beliebige GPU-Hardware-Intrinsiken zu verwenden. Die Grundlagen für dieses Gebiet werden im nächsten Abschnitt geschaffen. Darauf folgt die Vorstellung der einzelnen Hash-Funktionen, mit Blick auf die Implementierung. Als Referenz dient dabei der bekannte FNV-Algorithmus [5]. Mit der Implementierung folgt die Übertragung auf ein Multi-GPU-System, welches auf unterschiedliche Art und Weise die Hashes erstellt. Aufbauend auf den Hash-Funktionen, erfolgt die Erstellung des Hash-Joins. Dabei spielt das Probing und die Partitionierung eine wichtige Rolle. Für das Probing werden drei unterschiedliche Speichermodelle vorgestellt. Die Partitionierung teilt die Daten in kleine Partitionen auf, welche anschließen durch die Vektorisierung, in Vektoren verteilt werden. Die Vektoren stellen die Eingabe für die Build und Probe Phase dar. Zusätzlich werden Konzepte vorgestellt, um die Generierung

der Vektoren auf mehreren GPUs zu verteilen und anschließen das Probing auf den jeweiligen GPUs durchzuführen. Für jede in dieser Arbeit vorgestellten Funktion, erfolgt eine umfassende Analyse in der Evaluation. Die Hash-Funktionen werden nicht nur auf Hash-Rate, sondern auch die in den Grundlagen genannten Eigenschaften getestet. Der Hash-Join wird in das Probing und der Partitionierung unterteilt, wobei jede Funktion ihre eigenen Schwierigkeiten mit sich bringt. Ganz besonders wird dabei auf die schwächen der Partitionierung eingegangen. Zur Einordnung folgt ein Abschnitt mit verwandten Arbeiten und das Fazit aus den Ergebnissen der Evaluation.

2 Grundlagen

2.1 Hashing

In vielen Bereichen der Informatik ist das Berechnen eines Hashes ein wichtiger Bestandteil. Ein typisches Beispiel sind Key-Value Stores, welche in Programmiersprachen, auch mit der Verwendung von Hashes realisiert sind [6, 7]. In einem Key-Value-Storage liegen Daten immer im Verbund, bestehend aus einem Schlüssel (Key) und einen Gegenwert (Value), vor. Der Hash ist der Schlüssel und zeigt auf den Wert. Hier findet das Hashing als eine Art von Komprimierung ihren Einsatz. Um den Schlüssel zu bilden, wird der Ausgangswert in eine Hash-Funktion gegeben, welche den Wert in den Hash übersetzt. Die Eingabe einer Hash-Funktion, kann beliebige Werte wie Zahlen, Wörter oder andere Daten beinhalten, welche nach dem Verfahren als eine zusammenhängende Folge von Bytes repräsentiert ist. Diese kann wiederum als eine Zahl oder eine Folge von Zeichen interpretiert werden. Eine Funktion, welche diese Funktion erfüllt, ist als Message-Digest zu Bezeichnen [8]. Die Generierung eines Hashes ist jedoch keine triviale Aufgabe, weswegen die Konstruktion einer Hash-Funktion in vielen Bereichen präsent und eine Lösung speziell für den Anwendungsfall angepasst ist. Anforderungen und Teilgebiete, werden daher in den nächsten Abschnitten genauer betrachtet.

2.1.1 Eigenschaften

Bevor jedoch detailliert ein Blick auf ausgewählte Gebiete geworfen wird, müssen Regeln für eine Hash-Funktion definiert werden. Diese Regeln bestimmen die Qualität der Hash-Funktion und stellen eine Anforderungsliste, welche je nach Teilgebiet unterschiedlich gewichtet ist dar. Zusammenfassen lässt sich dies in die folgenden vier Anforderungen [9].

Kollision Eine Hash-Funktion sollte immer so wenige Kollisionen wie möglich verursachen. Eine Kollision beschreibt das Verhalten, wenn bei unterschiedlicher Eingabe der gleiche Hash-Werte ausgegeben wird. Dieses Kriterium ist eines der wichtigsten, da es die Güte der Funktion beschreibt und maßgeblich auf die Kategorie des Hashes Einfluss ausübt. Ein perfekter Hash-Algorithmus besitzt keine Kollisionen und bietet somit gewisse Vorteile, welche große Einflüsse auf die Implementierung haben kann. Des Weiteren ist, wie bereits angedeutet, eine Hash-Funktion eine Einwegfunktion, welche die Eingabe immer auf einen Wert übersetzt. In gewissen Umständen ist es eine Anforderung, dass dieser Wert nicht wieder auf die Eingabe zurückgeführt werden kann [8]. Da dieser spezielle Teil der Anforderung keine wichtige Rolle in der Arbeit spielt, wird die Thematik nur zur Gegenüberstellung im nächsten Abschnitt erwähnt.

Deterministisch Ein deterministischer Algorithmus ergibt bei gleicher Eingabe, auch immer die gleiche Ausgabe. Das ist eine verpflichtende Eigenschaft für einen Hash-Algorithmus. Wird eine nichtdeterministischer Algorithmus, ohne Kollisionen entwickelt, so ist dieser Algorithmus unbrauchbar. Da Hashes immer für Vergleichsoperationen verwendet werden, welche die Eingabe auf eine konzentrierte Form (den Hash) vergleichen. Würde eine Funktion dieser Art eine Anwendung finden, wären gleiche Eingaben nicht vergleichbar und somit das gesamte Verfahren in einen Nichtdeterminismus überführen. Daraus folgt, dass diese Eigenschaft, die wichtigste der genannten Eigenschaften sein muss. Das Verletzen der Eigenschaft, ist bei der Implementierung sehr einfach zu erkennen und daher immer gegeben, solange die Verletzung nicht gewollt ist.

Verteilung Verarbeitete Eingaben und die daraus folgenden Hash-Werte, sollten sich immer gut über den gegebenen Zahlenraum verteilen. Mit "gut" ist in diesem Fall, die gleichmäßige und vollständige Nutzung gemeint. Eine schlechte Verteilung erhöht das Risiko für Kollisionen, da der Zahlenraum nicht optimal genutzt und somit verkleinert wird. Das führt dazu, dass sich mehr Eingaben einen Hash-Wert teilen müssen. Die Grundannahme für die Verteilung ist ein gleichverteilter Datensatz. In der Realität ist jedoch mit einem Datensatz zu rechnen, wo sich die Daten in einen konzentrierten Bereich aufhalten, was auch mit dem Geburtstags-Paradoxon [9, 10] beschrieben werden kann. Mit dieser Annahme ist es deshalb auch wichtig, dass benachbarte Werte eine breite Streuung im Zahlenraum aufweisen und somit das Kollisionsrisiko verringern.

Geschwindigkeit Verarbeitungsgeschwindigkeit ist ein notwendiges Kriterium. Wie es in den folgenden Abschnitten noch gezeigt wird, kann die Qualität der Hash-Funktion, durch die Erhöhung von Iterationen und Komplexität verbessert werden [8]. Daraufhin steigt die Verarbeitungszeit [11]. Eine Steigerung könnte jedoch eine wichtige Rolle spielen, wenn zeitkritische Anwendungen (Anwendungen mit zeitlich begrenzter Laufzeit) ins Spiel kommen. Da sich diese Arbeit auch auf die Verarbeitungsgeschwindigkeit bezieht, erhält dieses Kriterium, im Verlauf der Arbeit, eine erhöhte Priorität.

Speicher Speichernutzung ist vielfältig zu betrachten. Zum einen betrifft das die Größe der Ein- und Ausgabedaten und zum anderen die Speicherauslastung des Algorithmus. Limitierungen für den Algorithmus, entstehen durch die Verarbeitungseinheiten des Prozessors und der Anbindung an die Speichermedien, wie zum Beispiel externer oder interner Speicher. Die Größe der zu verarbeitenden Daten, spielt in der Regel nur eine Rolle, wenn es speicheroptimierende Verfahren, wie Caches oder Instruktionen gibt. Die nähere Betrachtung dieser Kategorie, folgt in den nächsten Abschnitten. Zusammen mit der Geschwindigkeit, können diese die Anforderung an der Reduzierung der Kollisionen überschreiben und somit gleichwertig, zu der Hash-Funktions-Güte betrachtet werden.

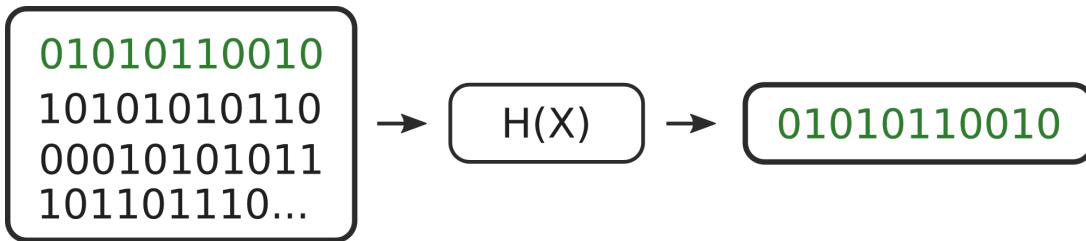


Abbildung 2.1: Eine einfache Hash-Funktion

2.1.2 Funktion

$$X = \{x \mid n \in \mathbb{N}_+ \wedge x \in \{0, 1\}^n\} \quad (2.1)$$

$$Y(n) = \{y \mid y \in \{0, 1\}^n \wedge n \geq 1\} \quad (2.2)$$

$$H : X \mapsto Y \quad (2.3)$$

Eine Hash-Funktion wurde bisher für Key-Value-Stores vorgestellt, jedoch als eine Art Blackbox, bei der eine beliebige Eingabe X verarbeitet wird. X besteht aus einer nicht leeren Menge (Folge) an Bits (Gleichung 2.1), welche zum Beispiel eine Zahl oder eine Zeichenkette (String) repräsentieren kann. Eine gewählte Hash-Funktion, auch Message-Digest $H(X)$ genannt, verarbeitet einen Wert von X und gibt einen Wert aus der Menge Y zurück (Gleichung 2.3). Zu beachten ist, dass Y wie in Gleichung 2.2 definiert ist und immer die gleiche Eingabegröße hat [12, 9]. Diese Beschränkung gilt nicht für X , was deutlich beim Hashen von Passwörtern wird [12], da diese in der Regel, eine unterschiedliche Länge an Zeichen aufweisen. Der standardmäßige Prozess, wird in Abbildung 2.1 dargestellt. Links ist die Eingabe X zu sehen, welche durch die Hash-Funktion in der Mitte zu der Ausgabe Y transformiert wird. Es ist leicht zu sehen, dass X eine größere Menge an Eingabemöglichkeiten abdecken kann, als der Ausgabewert Y . Das lässt darauf schließen, dass in $H(X)$ eine Komprimierung von X auf den Hash Y stattfindet. Diese Komprimierung kann auf unzählige Möglichkeiten geschehen. Als einfachste Variante wäre die Reduzierung von X , auf die Länge von Y , indem die überschüssigen Bits von X abgeschnitten werden. Das Verfahren würde einen validen Hash zurückgeben, mit einer sehr schlechten Güte, da nicht jeder Bit einen Einfluss auf das Endergebnis hat, weshalb eine Kollision einfach zu konstruieren ist.

2.1.3 Anwendung

Um eine höhere Hash-Qualität zu erlangen, gibt es für unterschiedliche Bereiche, spezialisierte Algorithmen. Dazu gehört die Kryptographie und eine Reihe von *SHA-** Algorithmen. SHA steht in diesem Fall, für Secure Hash Algorithmen [8] und umfasst eine Reihe von Funktionen, welche für die Erstellung von kryptographisch sicheren Hashes genutzt werden [8, 12, 13]. Ein bekannter Vertreter dieser Gruppe, ist der SHA-256 Algorithmus. Wie es der Name schon sagt, ist es ein als sicher klassifizierter Algorithmus, mit einem Hash von 256-Bit. Als sicher werden Hash-Funktionen deklariert, bei den es besonders schwer ist, eine Kollision zu finden und es quasi unmöglich ist, das Inverse von Y und somit die Eingabe X zu rekonstruieren [12]. Für die Anforderungen dieser Arbeit sind sichere Hash-Funktionen vollkommen überqualifiziert, da nur ein Interesse an der Minimierung der Kollisionen besteht. Jedoch geben kryptographische Algorithmen einen wunderbaren Einblick, wie vielfältig Hash-Verfahren aussehen können und warum sichere Hash-Verfahren nicht geeignet für einfache Vergleichsaufgaben sind. Diesen Einblick soll der SHA-256 Algorithmus liefern, wessen Teilschritt in Abbildung 2.2 dargestellt ist. Im Vergleich zur Abbildung 2.1, ist dieser deutlich komplizierter, überwindet aber die genannten Probleme sehr zuverlässig, weshalb er auch zu dem Standard für Hash-Funktionen,

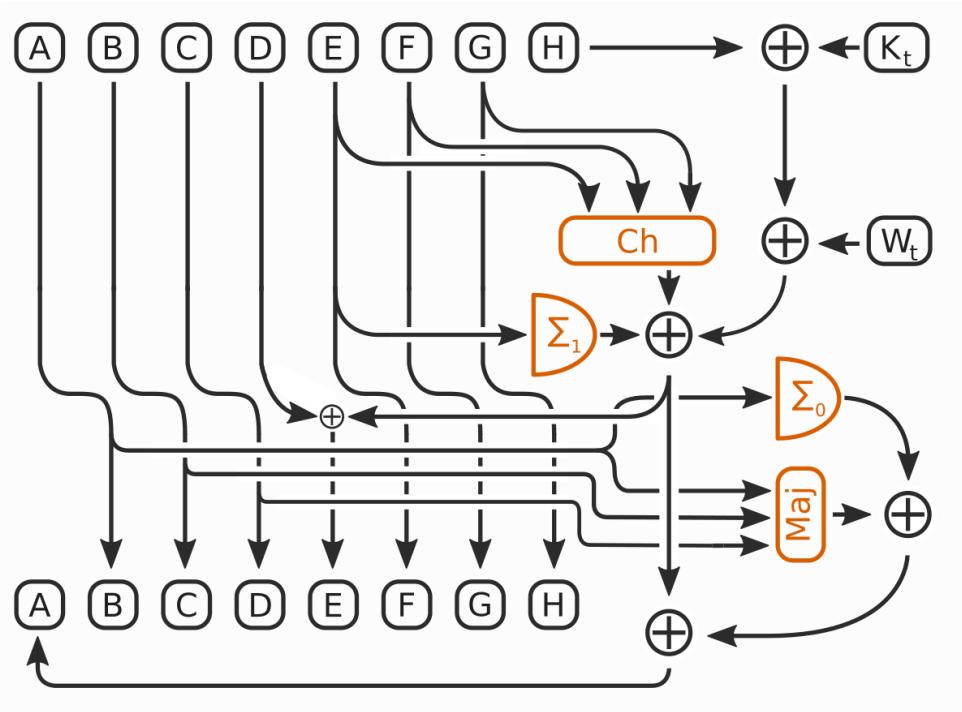


Abbildung 2.2: SHA-256 Schleife

in Sicherheitsrelevanten Bereichen gehört [8].

Um den 256Bit Hash-Wert aus X zu berechnen, müssen zuerst ein paar Vorverarbeitungsmaßnahmen getroffen werden. Dazu gehört das Padding, das die Eingabe X um einen bestimmten Wert an Bits auffüllt, sodass immer eine Eingabe, welche Teilbar durch 512 ist, entsteht. Die Fehlenden Bits werden hierzu mit Nullen aufgefüllt, wobei der Anfang der Nullen mit einer Eins als Bit-Wert gekennzeichnet ist. Die Länge der aufgefüllten Bits wird als ein 64-Bit Zahlenwert zu den auffüllenden Nullen an das Ende hinzugefügt. Dadurch entsteht ein Padding, bestehend aus dem Markierungs-Bit, den Padding-Nullen und der Länge der Padding-Nullen, mit insgesamt 64-Bit. $|X|$ ist nach diesem Schritt immer durch 512 teilbar. Ein Segment ist in 16, 32-Bit Segmente unterteilt und befüllt die ersten Behälter $W_{1\dots 16}$. Die restlichen Werte für $W_{17\dots 64}$ bestehen aus vorherigen W_t Werten, welche durch Permutation verändert wurden. In der Abbildung 2.2 sind die Behälter A bis H zu sehen, welche als zentrales Arbeitsglied der Hash-Schleife dienen. Die initialen Werte bilden sich aus der Quadratwurzeln der ersten 8 Primzahlen. Die Werte A bis D rutschen nach jeder Runde, fast unverändert einen Behälter weiter.

$$Ch(X, Y, Z) = (X \wedge Y) \oplus (\neg X \wedge Z) \quad (2.4)$$

$$Maj(X, Y, Z) = (X \wedge Y) \oplus (X \wedge Z) \oplus (Y \wedge Z) \quad (2.5)$$

Die Behälter E bis H durchlaufen einzelne logische Verknüpfungsschritte, diese sind in Gleichung 2.4 und 2.5, genauer aufgeschlüsselt. Zusätzlich beeinflusst die Eingabe, welche in W_t aufgeteilt wird das Ergebnis. In Kombination mit H, werden Konstanten mit den Werten der Quadratwurzeln der ersten 64 Primzahlen, durch K_t in die Kombinationslogik eingefügt. Die Funktionen Σ üben weitere Permutationen auf die einzelnen Werte aus. Durch die Addition (Modulo 2^{32}) verbunden, folgt die nächste Iteration mit den neuen Werten für A bis H. Dies

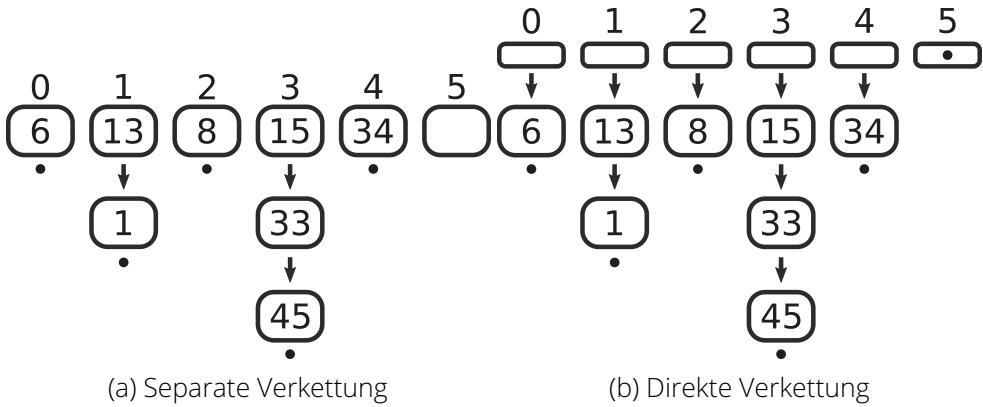


Abbildung 2.3: Hashtabelle Separate-Chaining Implementierung

wird so lange wiederholt, bis alle Teilblöcke verarbeitet wurden. Zum Schluss folgt die Konkatenation der Behälter A bis H, wodurch der Hash mit 256-Bit entsteht. Diese Erklärung ist nicht vollständig und beabsichtigt dies auch nicht. Für eine genaue Beschreibung des Verfahrens, kann die Empfehlung der "FEDERAL INFORMATION PROCESSING STANDARDS" [8] hinzugezogen werden. Für die Arbeit reicht der entstandene Eindruck und die verwendeten Instruktionen, welche im weiteren Verlauf eine Rolle spielen werden. Ein Algorithmus mit dieser Komplexität verfehlt das Ziel einer einfachen Vergleichslogik und soll daher nur als Referenz dienen.

2.1.4 Hashtabelle

In der Einleitung zu diesem Abschnitt wurde bereits von Key-Value-Stores mit Hashtabelle gesprochen. Was ein Schlüsselkonzept für diese Arbeit darstellt. Hashtabellen dienen dem Speichern von Hash-Werten, indem sie ein beliebigen Datentyp als Hash abspeichern. In Java [7] ist der Datentyp ein generisches Objekt, was immer einen Hash für den aktuellen Zustand zurückgeben können muss. Ähnlich wird es in vielen anderen Programmiersprachen gehabt [6] und verdeckt somit dem allgemeinen Anwender, die interne Logik der Hashtabelle. Die Funktion lässt sich im perfekten Fall, mit einem unendlich großen Array (Liste von Elementen) beschreiben [9]. Wenn ein Element in diesen Array eingefügt werden soll, muss ein geeigneter Platz gefunden werden. In herkömmlichen Listen ist das die erste oder letzte Stelle. Das Problem wäre gelöst wenn die Komplexität der Suche, nicht im schlimmsten Fall $O(n)$ wäre, wobei n die Anzahl der Element ist. Denn bei der Suche müsste auf jedes Element geprüft werden. Die Hashtabelle reduziert diese Komplexität auf $O(1)$, da der Hash als Index in der Liste dient. Sobald nach einem bestimmten Wert X in der Liste gesucht wird, muss der Hash Y , von X gebildet und anschließend der Wert Y als Index interpretiert werden. Dadurch ist an der Stelle Y direkt entscheidbar, ob das Element vorhanden ist oder nicht. Die Lösung bezieht sich allerdings nur auf die perfekte Hashtabelle. In realen Umständen begrenzen Speicher-Limitierung die Tabel lengröße. Als Lösung bietet Ottmann und Widmayer [9], eine Verkettung von Hashes an. Sie bezieht sich darauf, dass nach der Einschränkung des Speichers, nur noch eine endliche Menge an Hashes in der Liste platz hat. Laut Definition sind Hashes endlich, da jeder Hash immer aus einer gleichen Anzahl an Bits bestehen muss, welche durch N beschränkt ist. Mit Blick auf den SHA-256 Algorithmus, würde ein N von 256 vorliegen. Sollten alle Werte der Funktion in einer Liste gespeichert werden, müssten 2^{32} Einträge mit jeweils 256 Bits oder 32-Bytes vorliegen. Zusammengefasst würde diese Liste über 100 Gigabyte an Hashes beinhalten, was für das Speichern von 1000 Elementen, viel zu groß ist und vermutlich viele Nutzer an die Hardwarelimits führen wird. Die Verkettung der Hashes löst dieses Problem. Durch die Reduzierung der verfügbaren Einträge, folgt die Erweiterung der Speicher-Operation von Elementen,

um die Modulo Operation. Für den Hash erfolgt die Interpretation als Zahlenwert, Modulo der Listenlänge. Der Teilerrest entspricht dann dem Index in der Liste. Der neue Eintrag wird nicht immer auf eine freie Stelle in der Liste zeigen, weshalb vor dem Einfügen der Zustand an der Stelle zu überprüfen ist. Beim "Separate Chaining" (Abbildung 2.3a) findet die Konfliktlösung, durch ein neues Kettenglied statt. Als Beispiel ist der Hash mit dem Wert Sieben einzufügen. Im ersten Schritt wird der Index mit der Listenlänge bestimmt. In diesem Fall ist das die Sechs, weshalb durch die Modulo-Operation der Index Eins zurückgegeben wird. Doch an der Stelle Eins gibt es bereits ein Element mit der Zahl 13. Damit der Hash an dieser Position eingefügt werden kann, muss das Ende der Kette (Linked-List) gesucht werden, was durch die Zeiger auf die nachfolgenden Elemente erfolgt. Ist das Ende gefunden, wird ein neuer Eintrag an den letzten Eintrag mit dem Hash-Wert Sieben eingetragen. In der Abbildung 2.3 sind Zeiger immer mit Pfeilen und Zeiger ohne Ziel mit einem Punkt dargestellt. Das letzte Element an dem Index Eins, wäre daher der Hash-Wert Eins. Die Implementierung dieser Strategie bietet zwei Möglichkeiten. In der ersten Variante (Abbildung 2.3a) sind alle Einträge in der Liste bereits mit einem leerer Hash-Wert angelegt, was mit einem leeren Kästchen gekennzeichnet ist. Die zweite Variante (Abbildung 2.3b) reduziert den initialen Speicherverbrauch, indem die initialen Elemente durch Zeiger auf Elemente ersetzt werden. Dies kann den Speicherverbrauch um die Differenz der Zeigergröße und der Größe der Elemente verringern. Um einen Hash in den Listen zu finden, wird ähnlich wie bei dem Einfügen, der Index des gesuchten Hashes gebildet und anschließend in der Kette nach dem Hash gesucht. Sollte die Suche bis an das Ende der Kette ohne Ergebnis gelangen, so besitzt die Hashtabelle nicht den gesuchten Hash. Operationen, welche die vorhandenen Einträge manipulieren, spielen in dieser Arbeit keine Rolle und werden daher nicht weiter betrachtet.

$$\text{Load - Factor}(a) = \frac{n}{m} \quad (2.6)$$

Um die Länge der Ketten ein wenig kontrollieren zu können, ist mit dem Load-Faktor [14] die Länge der Tabelle veränderbar. Der Load-Faktor ergibt sich aus der Menge an Hashes, welche in die Hashtabelle einzufügen sind und der Menge an verfügbaren Plätzen (Gleichung 2.6). Die Plätze werden auch Buckets genannt. Wie es in der Gleichung zu sehen ist, bedeutet ein Load-Faktor über Eins, immer einen Überschuss an Hashes. Dies stellt eine der wichtigsten Stellschrauben, neben der Hash-Funktion, für die Hashtabelle dar.

2.2 Join

Aufbauend auf die vorgestellte Hashtabelle gibt es Varianten des Join-Verfahrens in Datenbanken, welche als Grundlage eine Hashtabelle verwenden [15]. Der Join-Operator ist eine von vielen Operationen, welche in der Welt von Datenbanken-Anfragen zu finden ist und wird für die Konkatenation von Tabellen über bestimmte Spalten angewendet [16].

2.2.1 Funktion

Für die Konkatenation der Tabellen muss mit einer Datenbankanfrage, die ausgehenden Tabellen spezifiziert werden, welche links in der Abbildung 2.4 zu sehen sind. Zwischen den beiden Tabellen ist die Operation mit dem gängigen Inner-Join Symbol gekennzeichnet. Um die resultierende Tabelle C zu extrahieren, wird für den Anfang ein naiver Join-Algorithmus, welcher auch als Pseudocode in 1 zu sehen ist durchgeführt. Die Eingabe sind die beiden gezeigten Tabellen A und B, welche jeweils eine Spalte für den Join-Vergleich angeben. Ein Join kann auch über mehrere Spalten durchgeführt werden, jedoch bietet das in diesem Beispiel keinen Mehrwert. Nachdem die Startparameter bekannt sind, ist die Iteration über jede Zeile in A und

PK1	C1
1	a
2	b
3	c
4	d



PK2	C2
1	d
2	d
3	f
4	h

PK3	PK1	PK2
1	4	1
2	4	2

Abbildung 2.4: Inner Join

Algorithm 1 Naive Join

```

Require:  $A, B, ColA, ColB$ 
 $C \leftarrow \{\}$ 
 $RowA \leftarrow Head(A)$ 
 $RowB \leftarrow Head(B)$ 
while  $RowA \neq End(A)$  do
    while  $RowB \neq End(B)$  do
        if  $value(RowA, ColA) = value(RowB, ColB)$  then
             $C \leftarrow C + (RowA, RowB)$ 
        end if
    end while
end while

```

B durchführbar. Dabei werden die genannten Spalten miteinander verglichen und bei Gleichheit beider Zeilen, an die Tabelle C angehangen. Der resultierende Datensatz C besteht nach der Berechnung nur noch aus A und B Zeilen, welche die gleichen Spaltenwerte aufweisen. In der Realität ist dies jedoch nicht immer gewünscht. Alternativ bieten Datenbanken weitere Join-Operatoren an, welche vollständig die Tabelle A oder B oder A und B im Endergebnis beinhalten. Diese werden dann Left-Outer, Right-Outer und Outer-Join genannt. Dass dieser Algorithmus nicht optimal ist, gibt der Name schon vor. Weitere Optimierungen werden daher anhand von zwei Algorithmen vorgestellt.

2.2.2 Algorithmen

Sort-Merge-Join

Um zu verhindern, dass jedes Element zu überprüfen ist, etabliert der Sort-Merge-Join [3] Algorithmus einen Sortierschritt. Dieser wird vor dem, zum naiven Join ähnlichen Vergleichs-Block, auf den zu vergleichenden Spalten ausgeführt. Der Vorteil entsteht beim begrenzen der While-Limits. Im Naiven-Join 1 werden für jede Zeile alle Zeilen der anderen Tabelle hinzugezogen und getestet. Dagegen sind beim Sort-Merge-Join die Schleifen-Durchläufe eingrenzbar, wenn die Spaltenwerte außerhalb der Werte, der gegenüberliegenden Spalte liegen [17, 18].

Hash-Join

Aufbauend auf die vorherigen Abschnitte, bietet der Hash-Join Algorithmus eine weitere Variante, die Operation effizient durchzuführen [19]. Für die Beschleunigung der finalen Tupel-Bildungsphase (auch Probing-Phase genannt), wird aus der kleinsten Tabelle R, eine Hashtabelle konstruiert. Hierfür findet im ersten Schritt die Zusammenfassung der relevanten Spalten

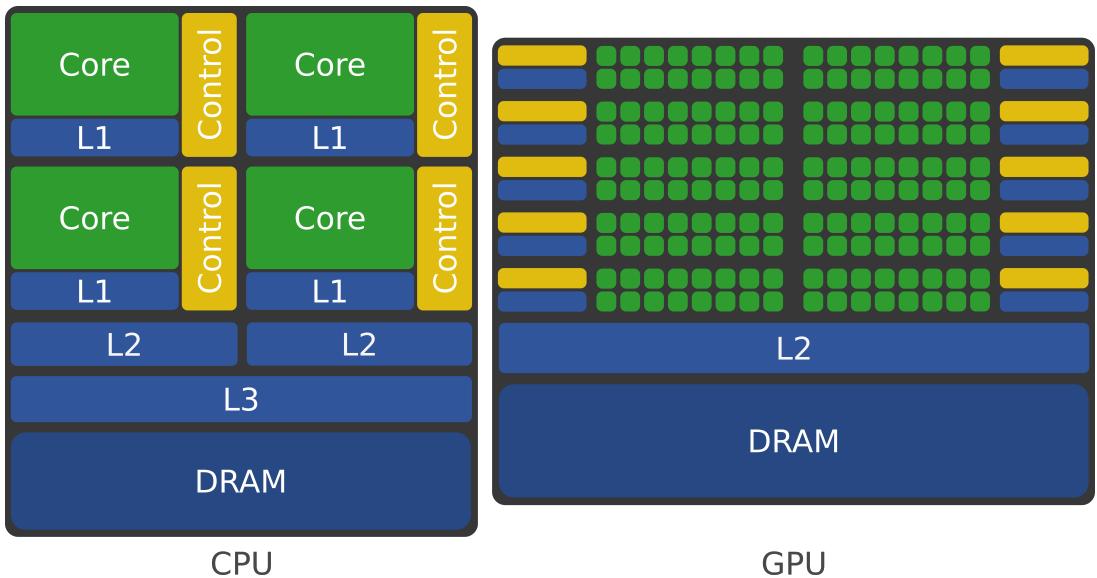


Abbildung 2.5: CPU vs GPU [20]

der Tabelle R statt, worauf anschließend die Einsortierung in eine Hashtabelle folgt. Was auch Building-Phase bezeichnet wird. Die Hashtabelle dient in der Probing-Phase, einen relevanten Kandidaten für die größere Tabelle S zu finden. Hierfür muss auch der Hash für die relevanten Spalten der S-Tabelle gebildet werden. Im Idealfall können Kandidaten bei der Iteration von S, direkt ausgeschlossen werden, wenn die Such-Komplexität von $O(1)$ berücksichtigt wird. Dies ist allerdings nur der Fall, wenn der jeweilige Eintrag in der Hashtabelle von R, nur ein oder kein Element enthält, was durch den bereits genannten Load-Faktor teilweise steuerbar ist.

2.3 GPU

Die bisherige Hash- und Join-Algorithmen wurden rein für die Central Processing Unit (CPU) entwickelt. Bei genauer Betrachtung der Hash-Operation wird deutlich, dass die Generierung eines Hash-Wertes, komplett unabhängig von den restlichen Hashes durchführbar ist. Daher ist es leicht möglich die Berechnung jedes Hashes, auf einen einzelnen Thread zu verschieben. Hierfür bietet die CPU eine limitierte Anzahl an Threads, welche gleichzeitig verarbeitbar sind. Bei einer geringen Anzahl an Hashes, ist dies vermutlich eine akzeptable Lösung, jedoch wäre eine Graphics Processing Unit (GPU) bei steigender Zahl der Elemente, eine deutlich bessere Wahl. Diese bietet eine Architektur, welche das gleichzeitige Verarbeiten von tausenden Elementen ermöglicht und somit einen größere Menge an Hashes in einem Iterations-Schritt verarbeitet.

2.3.1 Aufbau

In Abbildung 2.5 werden CPU und GPU gegenübergestellt. Die CPU wird von einer klassischen Vier-Kern-Architektur vertreten. Jeder Kern besitzt einen Kontrolleinheit und einen Cache, wodurch es möglich ist, dass jeder Kern völlig unabhängig von den restlichen Kernen agieren kann. Für eine bessere Speicheranbindung, liegen durch den L2 und L3 Cache größere, aber auch langsamere Speichermedien vor, welche sich die Kerne je nach Hierarchie teilen. All Umfassend steht der CPU der externe und größere DRAM (Dynamic Random Access Memory) zur Verfügung. Eine GPU besitzt einen Aufbau, ähnlich zur rechten Darstellung in Abbildung 2.5. Deutlich erkennbar ist die Anzahl an Kernen, welche bei Nvidia als Cuda-Cores [22] bezeichnet werden. Ein Cuda-Core stellt eine Recheneinheit in einem großen Grid an Cuda-Cores dar. Im

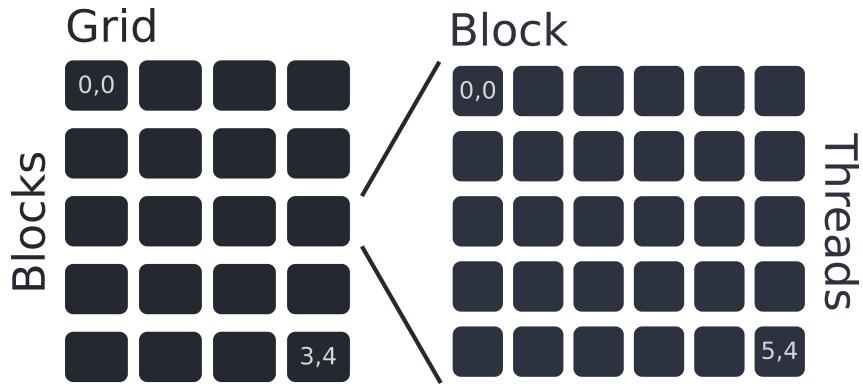


Abbildung 2.6: Grid, Block, Thread [20]

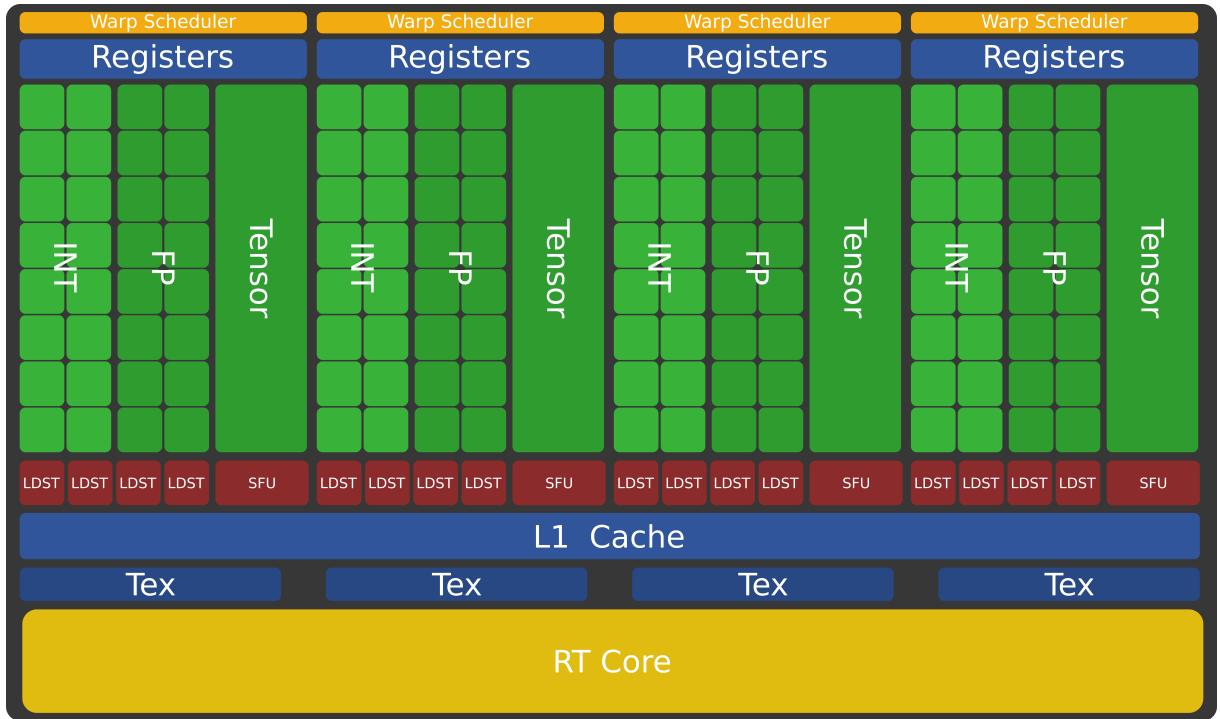


Abbildung 2.7: Turing Streaming Multiprozessor [21]

Vergleich zum Kern in der CPU, spielt ein einzelner Cuda-Core in einem Programm, eine viel kleinere Rolle. Programme für GPUs sind darauf optimiert, dass sie von so vielen Threads wie möglich gleichzeitig bearbeitbar sind, weshalb ein typischer Algorithmus für die GPU, mit mindestens einer vierstelligen Anzahl an Threads gestartet wird. Die Verteilung erfolgt auf die einzelnen Cuda-Cores. Diese Verteilung kann der Nutzer nur in der Dimensionierung bestimmen. Die Strukturen lassen sich in Blöcke unterteilen, welche auf einem Grid angeordnet sind. Ein Block ist durch die Anzahl der Threads in X-, Y- und Z-Richtung definiert. Gleichermaßen gilt für die Bestimmung der Grid-Struktur, welche die Anzahl der Blöcke beinhaltet. Die zweigeteilte Struktur wird dann auf die einzelnen Streaming Multiprozessoren (SM) verteilt. Ein einfacher Streaming Multiprozessor ist in Abbildung 2.7 schematisch dargestellt. In der Turing-Architektur [21] besitzt eine SM immer vier Teilsegmente, mit jeweils dem gleichen Aufbau. Jedes Segment besitzt eine Anzahl von insgesamt 16 Cuda-Cores, für Integer- und Floatingpoint-Operationen. Diese teilen sich eine Menge an Load/Store-Einheiten (LD/ST), für die Verarbeitung von Speicheroperationen. Zusätzlich dienen Special Function Units (SFUs) für das Verarbeiten von Instruktionen, wie Sinus oder Cosinus. Das besondere an einer SFU ist die Unabhängigkeit vom Clock-Zyklus

der Cuda-Cores, weshalb Instruktionen die Latenz der SFU überbrücken können [22]. Threads werden in einem SM, immer in einer Gruppe von 32-Threads ausgeführt und bilden einen Warp. Dieser Warp führt für alle Threads die gleiche Operation aus und setzt gegebenenfalls, die Berechnung für Threads mit alternative Rechenwege aus. Dieses Verhalten wird als Stall bezeichnet. Ein Vorteil eines Warps ist die Kommunikation zwischen Threads in einem Warp, wodurch Teilergebnisse bereits auf Warp-Ebene berechnet werden können. Zusätzlich bietet Cuda weitere Optimierungsmodelle für Speicherzugriff und Parallelisierung und ist Teil von späteren Abschnitten. Zunächst folgt ein Blick auf das generische Programmierungs-Modell geworfen.

2.3.2 Programmierung

Für die Programmierung von GPUs müssen in den ersten Schritten, nur grobe Kenntnisse über die vorherigen Architektur-Komponenten vorhanden sein. GPU Hersteller, wie NVIDIA oder AMD, bieten jeweils eine Abstraktion ihrer Hardware in einem Framework an. Nvidia bietet hierfür Cuda [20] und AMD ROCm an [23]. Da in dieser Arbeit speziell nur mit GPUs von Nvidia gearbeitet wird, ist Cuda die Grundlage für alle weiteren Abschnitte. Das Framework Cuda baut auf herkömmlichen C++/C-Code auf und ergänzt die Compiler-Tool-Chain, um den NVCC-Compiler. Bei der Erstellung eines Cuda-Programms wird der Code in Device und Host-Code unterteilt, wobei der Host-Code reines C++ repräsentiert und der Device-Code eine Mischung aus Cuda- und C++-Features ist. Die Device-Funktionen können dabei in unterschiedliche Darstellungsformen kompiliert werden. Für das direkte Ausführen auf dem Zielsystem, wird der Code direkt auf die Architektur kompiliert. Sollte der Code auch für andere Systeme zur Verfügung stehen, ist die Übersetzung in PTX-Code notwendig, welcher architekturunabhängig ist. Neben der Teilung der Funktionen, wird auch der Speicher dementsprechend geteilt. Da Funktionen auf Device-Ebene nur auf den Speicher auf der GPU zugreifen können, müssen die zu bearbeiteten Daten auf der GPU vorhanden sein. Die Allokation der Daten auf der GPU, ähnelt dem Aufruf auf der CPU. Geladene Daten auf der CPU müssen erst mit Cuda-Kopierfunktionen, auf die GPU kopiert werden. Um den manuellen Transferprozess zu umgehen, ist die externe Verwaltung der Daten, auch mithilfe des Unified-Virtual-Memories möglich. Hierbei vereint Cuda sämtlichen Host- und Device-Speicher und übernimmt die Verteilung der Ressourcen auf die Plattformen. Damit die GPU die Datenverarbeitung startet, ist das Aufrufen der Funktionen, von der Host-Seite notwendig. Der Zugriff wird über globale Funktionen gewährleistet, welche als Kernel bezeichnet werden.

Kernel

Ein Kernel ist im Grunde nichts anderes, als einen normale C++-Funktion mit den zusätzlichen Parametern der GPU. Die neuen Parameter werden gesondert, in dreifach ausgeführten spitzen Klammern, vor die Funktion geschrieben und bestehen aus der Anzahl der Threads, in einem Block und der Anzahl der Blöcke im Grid. Beide Parameter sind als ein Tupel mit jeweils X-, Y- und Z-Achse anzugeben. Alle drei Parameter sind je nach Architektur, nach oben hin beschränkt. Nach dem Thread-Parameter folgt die Größe des Shared-Memory in Bytes pro Block und wird später betrachtet. Als letztes ist der Stream anzugeben, was eine besondere Form der Aufgabenverteilung ist und deshalb auch in einem eigenen Abschnitt behandelt wird. Stream und Shared-Memory sind optionale Parameter und müssen deshalb nicht mit jedem Kernelaufruf übergeben werden. Die Aufteilung in Threads pro Block und die Anzahl der Blöcke, ist die wesentliche Besonderheit von Cuda-Kernels. Da jede Funktion als SIMD (Single Instruction Multiple Threads) ausgeführt wird, besteht ein grundlegender Teil in einem Kernel, aus der Identifizierung der einzelnen Threads. In der Implementierung wird nicht über einzelnen Threads iteriert, sondern von Cuda als vordefinierte Variablen, jeder Funktion mit

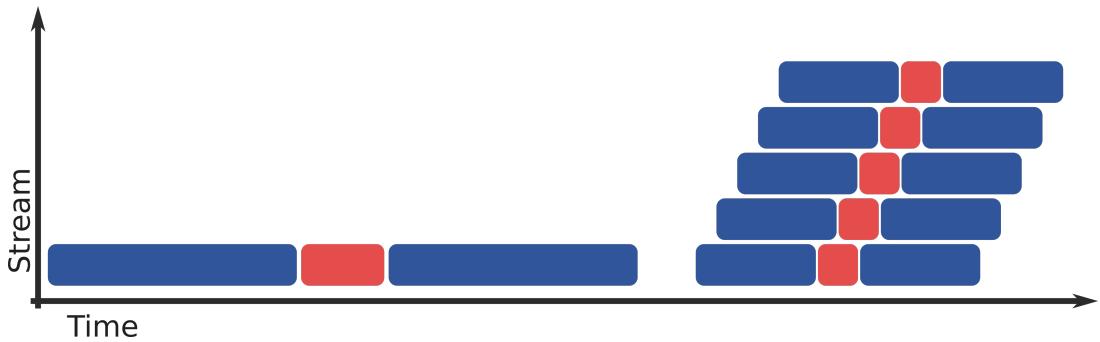


Abbildung 2.8: Streaming [21]

übergeben. Dadurch sieht jede Funktion wie eine Single-Thread-Funktion aus, jedoch wird der Datenzugriff auf die Position des Threads beschränkt. Hierfür kann die Position genau auf den Block im Grid und dem Thread im Block bestimmt werden. Ein Thread würde zum Beispiel nur einen Pixel an der Position X bearbeiten, welche anhand des Thread- und Block-Indexes im gesamten Grid berechnet wird. So ist zum Beispiel die Striding-Loop [24] eine einfache Methode, um alle Elemente in einer Liste mit einer beliebigen Anzahl von Threads zu bearbeiten. Mit Blick auf eine For-Schleife in C++, entspricht der Thread-Index im gesamten Grid, dem Startindex in der Schleife. Um Überlappung zu vermeiden, wird daher ein Inkrement von der gesamten Anzahl der Threads gewählt. Die gesamte Anzahl der Threads ergibt sich aus der Menge an Threads in Blöcken und der Menge an Blöcken im Grid.

Stream

Speicher-Operation oder Kernel-Aufrufe finden alle standardmäßig auf dem Standard-Stream (Null) der GPU statt. Ein Stream ist nichts anderes als ein Warteschlange für Aufgaben, welche auf der GPU auszuführen sind. Ist diese Warteschlange sehr lang, muss das Programm lange auf die Vollendung der Aufgabe warten, da alle Element sequentiell abgearbeitet werden. Dieses Verhalten ist nicht immer wünschenswert. Erfolgt die Übertragung von großen Datensätzen auf die GPU, so liegt eine erhöhte Wartezeit, zwischen Programmstart und Kernelstart, vor. Kleinere Datensätze würde eine geringere Latenz mit sich bringen und somit die Wartezeiten verkürzen. Abbildung 2.8 stellt beide Varianten gegenüber. Links ist der klassischen Ablauf von einem Stream zu sehen. Eine rote Operation soll einen Kernelaufruf repräsentieren und Blau, eine Speicheroperation. Der blockierende Faktor durch die Speicheroperation wird im Vergleich zum rechten Verlauf sehr gut deutlich. Dadurch dass die Speicherzugriffe auf keine Segmente aufgeteilt wurden, starten Kernel parallel und verringern somit die Idle-Zeit der SMs. Bei der Implementierung fällt zusätzlich auf, dass Aufrufe von Cuda-Funktionen mit einem eigenen Stream, nicht mehr den Programmablauf blockieren, weshalb eine zusätzliche Synchronisierung verwaltet werden muss. Cuda ermöglicht die Synchronisierung von einzelnen Streams, wodurch das Warten auf alle Funktionen nicht mehr notwendig ist.

2.3.3 Speicher

Neben der Rechenleistung ist der Speicher ein signifikanter Faktor, in der Entwicklung von Kernels. Die Speicherhierarchie einer GPU ist in Abbildung 2.9 zu sehen. Der größte Speicher auf einer GPU ist der DRAM, was auch als Global-Memory bezeichnet wird. Der Global-Memory ist von allen Threads auf einem Gerät erreichbar und ermöglicht somit die Manipulation von Threads im gesamten Grid. Zusätzlich gibt es einen Constant-Memory. Dieser wird für die Speicherung von Konstanten verwendet und ist deshalb für Device-Funktionen Read-Only. Für das Cachen des Global-Memories ist der L2-Cache verantwortlich. Er liest jeweils 32 Byte Einträge

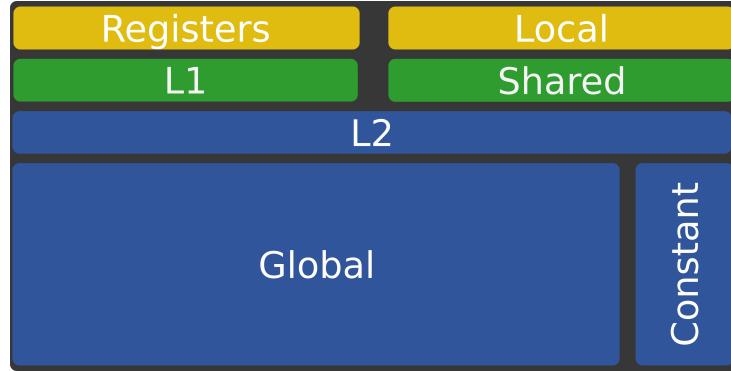


Abbildung 2.9: Speicherhierarchie

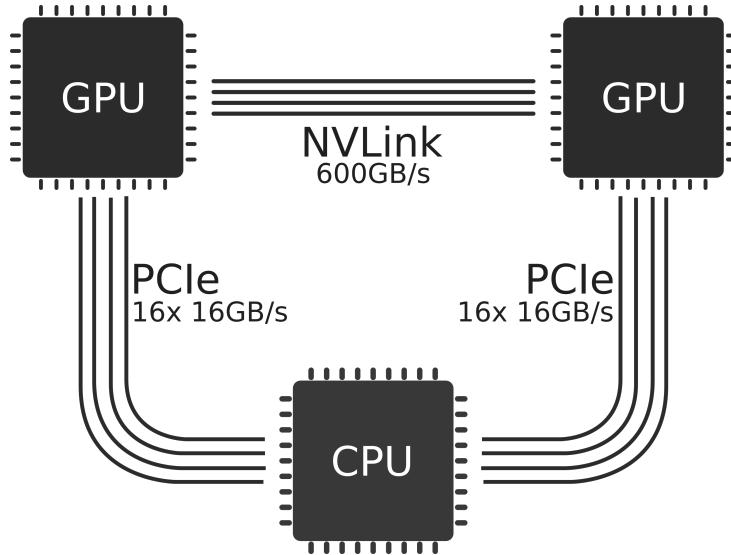


Abbildung 2.10: Multi-GPU Testsetup

aus dem Global-Memory. Jeder Block in einem Grid besitzt zusätzlich einen konfigurierbaren Cache, welcher als L1-Cache oder Shared Memory verwendbar ist. Hierfür wählt Cuda die L1-Größe, anhand der angegeben Shared-Memory-Größe im Kernel. Dieser Block ist 64KB groß und kann bis zu 48KB an den Shared-Memory übertragen. Das Verhältnis ist nachträglich anpassbar, sodass diese Grenze auch überschreitbar ist. Der Shared-Memory ist in Bänke unterteilt, welche parallel bearbeitet werden können. Greifen mehrere Threads auf die gleiche Bank zu, so wird der Zugriff serialisiert, was einem Bankkonflikt entspricht. Eine Bank entspricht 32- oder 64-Bit, was vor dem Kernelaufruf zu konfigurieren ist. Der Shared-Memory ermöglicht eine deutlich höhere Bandbreite für Daten, die häufiger in einem Algorithmus benötigt werden. Für Daten die in den L1-Cache fallen, werden immer 128 Byte Cache-Lines geladen. Auf Thread-Ebene befindet sich der Local-Memory und die Register. Der Local-Memory ist eine Art Ausweichspeicher, wenn zu viele Register notwendig sind. Er wird wie der Global-Memory, durch den L1- und L2-Cache gecached. Jeder Thread bekommt eine maximale Anzahl von 255 mal 32-Bit Register zugeordnet. Daten müssen jedoch nicht immer aufwendig in die einzelnen Speicher übertragen werden. Für Zwischenergebnisse bietet Cuda eine Palette von Warp-Funktionen an, um Werte zwischen Threads in einem Warp zu verteilen. Dies geschieht auf Register-Ebene und ist daher besonders schnell.

2.3.4 Multi GPU

Sollte der Speicher oder die Rechenleistung einer GPU nicht ausreichen, bietet Nvidia die Anbindung weitere GPUs an. Standardmäßig sind GPUs über einen PCIe-Slot, an die CPU angebunden. Eine schematische Darstellung ist in Abbildung 2.10 zu sehen. Die Anbindung der GPUs erfolgt über PCIe 3.0. mit 16 Lanes und insgesamt 16GB/s. Die PCIe Schnittstelle wird immer dann benötigt, wenn Daten von der CPU auf die GPU übertragen werden. Wenn Daten bereits auf der GPU vorhanden sind und diese von einer anderen GPU verwenden soll, dann muss nicht der langsame PCIe-Bus verwendet werden. Hierfür bietet die NVLink-Schnittstelle, eine deutlich schnellere Anbindung, von bis zu 600GB/s [25]. Der NVSwitch erweitert dieses Konzept auf bis zu 16 GPUs.

2.3.5 Historische Entwicklung

Der DRAM ist für die GPU die größte Speicheranbindung auf der Platine. Für schnelle Datenverarbeitung sollten daher alle Daten, mindestens auf dem Hauptspeicher der GPU liegen. Eine sogenannte In-Memory-Anwendung würde dieses Ziel verfolgen. Für diesen Ansatz muss eine GPU genügend Hauptspeicher aufweisen und genau an dieser Stelle kann das Vorhaben scheitern. Mit einem Blick auf die vergangenen Jahre von verschiedenen GPU-Modellen, ist ein Trend an immer größer werdenden Hauptspeicher zu erkennen. Als Beispiel sind die beiden Architekturen Fermi und Turing zu betrachten, welche acht Jahre auseinander liegen [22, 21]. Fermi ist 2010 auf den Markt erschienen und war im Besitz von 64KB an Hauptspeicher, wohingegen die Quadro-Reihe der Turing-Architektur, bis zu 48GB bieten kann. Gleichzeitig stieg auch die Anzahl der Streaming-Multiprozessoren und der Cuda-Kernen, welche einen groben Überblick, über die vorhandenen Rechenleistung geben. Fermi konnte hier mit vier SMs und insgesamt 512 Cuda-Kernen auffahren, was durch die Quadro 8000 RTX [26] mit 72 SMs und insgesamt 4608 Cuda-Kernen noch einmal deutlich gesteigert wurde. Aufgrund dieser Steigerung wird ein reine GPU-Anwendung, in dieser Arbeit implementiert.

2.4 Zipf Verteilung

$$Z_C = \frac{1}{\sum_{k=1}^K \frac{1}{k^\alpha}} \quad (2.7)$$

$$Z_D = \frac{Z_C}{k^\alpha} \quad (2.8)$$

Für das Testen der Algorithmen sind Testdatensätze notwendig. Diese können entweder vorhandene Daten sein oder spezielle generierte Verteilungen. Die einfachste Verteilung wäre die Gleichverteilung, bei der jeder mögliche Wert die gleiche Wahrscheinlichkeit besitzt. Die Zipf-Verteilung findet Anwendung, wenn sich ein Datensatz auf einen bestimmten Punkt im Parameter-Spektrum konzentrieren soll [27]. Für die Berechnung wird ein bestehende C-Implementierung verwendet [28]. Als Eingabe wird immer ein Zahlenbereich von 0 bis K erwartet. K entspricht der Anzahl der möglichen Werte in dem Datensatz. Zusätzlich ist ein Wert α notwendig, welcher auch als Skew bezeichnet wird. Er gibt die Konzentration auf einen kleinen Zahlenbereich an. Als letztes muss die Anzahl N der zu generierenden Daten angegeben werden. Für diese N Datenpunkte, wird im ersten Schritt jeweils eine Zufallszahl x gewählt, welche aus einer Gleichverteilung stammt. Des weiteren wird die sogenannte Zipf-Konstante benötigt. Diese wird wie in Gleichung 2.7 aus der Summe der Ränge k , und dem Exponenten α berechnet. Die Konstante wird im nächsten Schritt der Berechnung der eigentlichen Werte verwendet. In diesem Schritt wird jeder generierte Zufallswert, gegen den akkumulierten Zipf-Verteilungswert Z_D getestet. Z_D berechnet sich aus dem aktuellen Rang k , dem Skew-Faktor

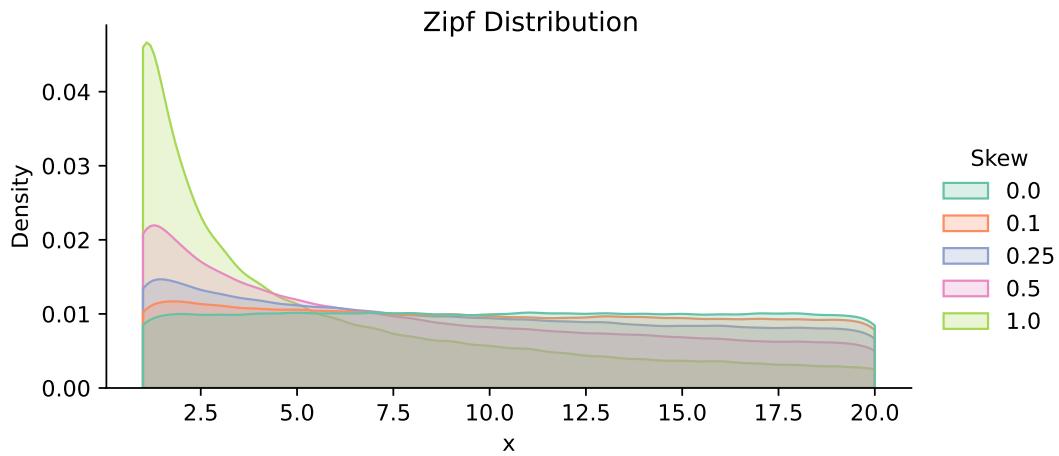


Abbildung 2.11: Zipf Verteilung

α und der berechneten Zipf-Konstante Z_C (Gleichung 2.8). Die Akkumulierung geschieht über jeden Rang-Wert, bis zum Maximum K . Sollte der Wert kleiner als der Wert x sein, so ist x an den aktuellen Wert k anzusiedeln. Das konzentrierte Verhalten ergibt sich, dadurch dass die berechnete Verteilung einen sehr steilen Anstieg bei den ersten Rängen vorweisen kann. Aus diesem Grund ist die Wahrscheinlichkeit, dass ein gleichverteilter Zufallswert in den vorderen Rängen, einen kleineren Wert als die Verteilung besitzt, viel größer. Beispielhaft ist eine Zipf-Verteilung mit unterschiedlichen Skew-Faktoren in Abbildung 2.11, über die Ränge 0 bis 20 zu sehen.

3 Analyse und eigene Konzepte

Im vorangegangen Kapitel wurden die Themengebiete Hashing, Join und GPU grundlegende betrachtet. Aufbauend auf diesen Kapitel werden nun Algorithmen vorgestellt, welche in dieser Arbeit entstanden sind und sich auf eine reine GPU-Anwendung spezialisieren. Im ersten Schritt wird auf das Thema Hashing eingegangen und anschließend im Verbund mit Hash-Joins auf einen datenbankähnlichen Verarbeitungsprozess überführt. Alle Algorithmen beziehen sich auf eine strikte Zielstellung und werden nicht von vorhandenen Systemen beeinflusst. Für die Implementierung wurde C++ mit dem Framework Cuda verwendet, weshalb Beispiele immer auf Konstrukte aus Cuda oder C++ zu finden sind. Jeder vorgestellte Algorithmus wird bezüglich seiner Eigenschaften analysiert, was die Basis für die spätere Evaluation ist.

3.1 Hashing

Hashing bedient ein breites Spektrum an Anwendungsfällen, welche bereits genauer betrachtete wurden. In dieser Arbeit soll es um besonders schnelle Hash-Algorithmen gehen, welche eine beliebige Menge an Daten in einen Hash vereinen kann. Die Verarbeitung und Generierung der Daten, findet nur auf der GPU statt. Da der Hash-Algorithmus nur ein kleines Bau teil in einem System repräsentiert, sind die Ausgangsbedingungen für den Algorithmus und Anwendungsgebiet frei wählbar. Der finale Algorithmus soll keine Daten außerhalb der GPU verwenden und die Hashes werden auch nicht außerhalb der GPU benötigt. Auch wenn das letzte Kriterium ignoriert werden kann, da jederzeit das Ergebnis von der GPU übertragbar ist, so ist diese Aussage wichtig für die spätere Evaluation, wo nur die Kernel-Zeiten betrachtet werden, da langsame Copy-Operationen keine Rolle spielen. Die CPU übernimmt die übliche Steuerung der Prozesse und wählt so zum Beispiel die richtige Hash-Funktion aus oder misst Zeiten für die spätere Evaluation.

3.1.1 Zielstellung

Ein Ziel dieser Arbeit ist das entwickeln von alternativen Hash-Funktionen, welche für die GPU zu implementieren sind. Die entstandenen Funktionen werden auf alle Eigenschaften aus dem Abschnitt Grundlagen analysiert. Für die Implementierung werden verfügbare GPU-Funktionen und herkömmliche Operatoren untersucht und miteinander kombiniert, falls die Ergebnisse auf dem ersten Blick einen validen Hash ergeben. Da ein Hash immer aus einer endlichen Anzahl an Daten besteht, muss für das Einlesen, ein optimales Verfahren entwickelt werden, was die GPU-Ressourcen optimal nutzt und die Wartezeiten minimiert. Sobald eine Hash-Funktion und ein Algorithmus für das Importieren der Daten gefunden wurde, soll das Verfahren auf

mehrere GPUs übertragen werden. Das Zielsystem umfasst zwar nur maximal zwei GPUs, jedoch ist eine generische Variante wünschenswert, damit die Skalierbarkeit vorhanden ist.

3.1.2 Algorithmen

Der SHA-256 Algorithmus (Abbildung 2.2) wurde in den vorherigen Abschnitten betrachtet und als Beispiel für eine kryptografische Anwendung vorgestellt. Dieser hatte die Aufgabe, eine geringe Kollisionsrate und eine gute Verteilung der Werte zu erzielen, was zusätzlich eine gute Verteilung von benachbarten Werten beinhaltet. Für dieses Ziel ist Laufzeit, nicht an erster Stelle, da ein schneller Algorithmus, mit zum Beispiel einer hohen Kollisionsrate, eine geringere kryptographische Sicherheit bietet. Sicherheit, müssen die im folgenden Funktionen nicht bieten können. Funktionen, wie das Vergleichen von Werten, sollen Hauptfunktionen darstellen und daher rückt Geschwindigkeit immer mehr in den Vordergrund. Ein Hash-Verfahren sollte daher das vorhandene Datum, auch wenn es in endlichen Teilen vorliegt, in so wenigen Schritten wie möglich verarbeiten können. Auf einer GPU spielt auch die Verwendung der Register, eine wichtige Rolle. Verschiedene Hilfskonstruktion belegen in der Regel, auch eine unterschiedliche Zahl an Register. Da ein Thread nur eine maximale Anzahl von 255 Register besitzen darf, da sonst der langsamere Local-Memory droht, muss diese Grenze eingehalten werden.

Auch spielt das Muster der Speicherabfrage eine wichtige Rolle. Es ist schlecht für die Cache-Nutzung, wenn jeder Thread einen zufälligen Datensatz aus dem Global-Memory lädt. Das liegt an der bereits genannten Beschränktheit der Caches. Der L1-Cache lädt immer 128Byte und der L2-Cache 32Byte als eine Cache-Line in den Speicher. Führen Threads nun einen zufälligen Zugriff auf einzelne Einträge im gesamten Datensatz durch, so führt dies zu einem erhöhten Speicherauslastung, da im Zweifel jede Cache-Ebene ein Miss meldet und an die nächste Ebene die Anfrage delegiert. Dies führt bis zum Global-Memory, was im Vergleich natürlich die langsamste Variante darstellt. Daher ist es notwendig, dass der Datensatz immer in einer geordneten Form vorliegt. Wenn der Datensatz als Liste von Elementen betrachtet wird, besitzt jedes Element einen festen Index in der Liste und kann mit diesem Index genau gefunden werden. Mit Blick auf die GPU, kann somit jeder Thread mit seinem Index, einen Index in der Liste abbilden, was dazu führt, dass benachbarte Threads auf benachbarte Element zugreifen. Eine Zugriffsform dieser Art, ermöglicht eine effizientere Nutzung der Cache-Lines. Sie wird auch als Coalesced-Access bezeichnet und ist ein Programmierparadigma, bei der Entwicklung von Algorithmen für GPUs. Neben dem Coalesced-Access, kann die Speicherauslastung auch reduziert werden, wenn die Warp-Intrinsiken von Cuda ihre Anwendung finden. Cuda bietet hierfür Funktionen an, wodurch Daten mit den benachbarten 31 Threads austauschbar sind. Dadurch müssen nicht immer Daten erst in den Global- oder Shared-Memory geschrieben werden und können bereits auf Register-Ebene akkumuliert werden. In einer Warp-Funktion können Integer-Werte übertragen werden, was einem 32-Bit oder 4Byte Wert entspricht. Soll zum Beispiel eine Datenzeile mit insgesamt 128 Byte verarbeitet werden, so wäre ein gesamter Warp mit einem Datensatz beschäftigt. In der Regel setzt dann der erste Thread, das finale Ergebnis in den Hauptspeicher. Für das Hashing wurden jedoch Warp-Intrinsiken nicht verwendet. Auch wenn sie den genannten Vorteil bieten, können benachbarte Threads auch einfach benachbarte Datenzeilen zugreifen und bieten auf den ersten Blick erkennbaren Nachteile, da es sich immer noch um einen Coalesced-Access handelt. Genauer wird dieser Aspekt im Abschnitt Speicherverwaltung betrachtet.

Die Konfiguration der Kernel bietet für jede Funktion, die gleiche Optionen. Streams spielen beim In-Memory-Hashen keine Rolle, da keine Speicher-Operationen vorliegen. Die Daten liegen bereits auf der GPU vor und der Speicher für die Hashes, kann als gegeben angesehen werden, da die Größe immer bekannt ist und bereits mit dem Laden der Daten allokiert werden kann. Deshalb sind keine weiteren Streams für das Überbrücken von Speicheroperationen

Algorithm 2 FNV-1a Hash-Funktion

Require: X , $Prime$, $Offset$

```

Hash ← Offset
for Octet in  $X$  do
    Hash ← Hash ⊕ Octet
    Hash ← Hash × Prime
end for
return Hash

```

Bits	Constant	Calculation	Hex Value
32	Offset Prime	$2^{24} + 2^8 + 0x93$	0x811C9DC5 0x01000193
64	Offset Prime	$2^{40} + 2^8 + 0xB3$	0xCBFB9CE4 84222325 0x00000100 000001B3
128	Offset Prime	$2^{88} + 2^8 + 0x3B$	0x6C62272E 07BB0142 62B82175 6295C58D 0x00000000 01000000 00000000 0000013B

Tabelle 3.1: Konstanten für die FNV-Hashfunktion

notwendig. Der Shared-Memory wird ebenfalls nicht verwendet, da jedes Element nur einmal geladen und am Ende nur einmal der Hash geschrieben wird. Es gibt Ausnahmen, bei den Elementen nicht unbedingt nur einmal geladen werden, jedoch können an diesen Stellen die Einträge vom L1- und L2-Cache profitieren. Die einzigen beiden Parameter, die übrig bleiben, sind die verpflichtenden Block- und Thread-Dimensionen. Da keine 2D- oder 3D-Strukturen notwendig sind, handelt es sich bei beiden Parameter um eindimensionale Werte. Die Thread-Anzahl muss immer durch 32 (Warp) teilbar sein und Blöcke werden abhängig von der Anzahl der Element definiert. Als zusätzlichen Parameter wird daher angegeben, wie viele Elemente ein Thread in einer Striding-Loop bearbeitet. Dementsprechend wird die Block-Anzahl reduziert, wenn ein Thread mehr als nur ein Element bearbeiten darf.

$$Blocks = \lceil \frac{N}{Threads \times Elements} \rceil \quad (3.1)$$

Die finale Anzahl an Blöcken wird mit der Formel aus Gleichung 3.1 berechnet. Die Anzahl der Elemente wird von N angegeben und die Anzahl der Threads, durch $Threads$. Wie viele Elemente ein Thread bearbeiten soll, wird durch $Elements$ festgelegt. Das Aufrunden ist nötig, damit immer ein Wert von mindestens einem Block besteht und die Konfiguration des Kernel erfolgreich ist.

FNV

Der FNV Hash-Algorithmus ist ein im Jahre 1991 entwickelter, nicht kryptographischer Hash-Algorithmus [29], dessen Name sich von den drei Entwicklern, Glenn Fowler, Landon Curt Noll und Phong Vo ableitet. Das FNV-Hash-Verfahren ist ein weit verbreitetes Verfahren, was als besonders schnell und mit einer guten Verteilung der Hashes beworben wird [30]. Daher wird diese Funktion, die Basis für die folgenden Vergleiche sein. Für die Erstellung eines Hashes, verarbeitet die Funktion immer genau einen Byte und verknüpft ihn mit dem vorherigen Hash-Wert. Genauer wird es anhand von Pseudocode in 2 aufgezeigt. Am Ende des Algorithmus wird immer ein Hash zurückgegeben. Daher muss zum Beginn des eigentlichen Verfahrens eine Variable, welche den Hash tragen wird, definiert werden. Die Initialisierung geschieht durch die Konstante, Offset. Der Offset kann eine zufällige Zahl sein oder eine vordefinierter Wert,

Algorithm 3 Addition Hash-Funktion

Require: X
 $Hash \leftarrow 0$
 for $Octet$ in X **do**
 $Hash \leftarrow Hash + Octet$
 end for
 return $Hash$

welcher von der Bit-Anzahl abhängig ist. Für Drei Konfigurationen wurden die Werte in Tabelle 3.1 aufgelistet. Wäre der Offset Null, dann ist für jeden Wert von X , der eine beliebige Menge an Oktetten besitzt und jedes Oktett den Wert Null repräsentiert, der Hash gleich. In diesem Fall ist der Wert des Hashes immer Null. Daher ist die Initialisierung mit Null nicht gestattet. Nachdem der Offset für den Hash gesetzt ist, wird über jedes Oktett in der Menge X iteriert. Wäre X eine Zeichenkette mit jeweils einem Zeichen pro Byte, so würde pro Iterationsschritt ein Zeichen in den Hash integriert werden. Innerhalb eines Iterationsschritt geschieht die eigentliche Bildung des Hashes. Hierfür wird zunächst das Oktett mit der XOR (\oplus) Operation verknüpft.

$$\begin{array}{r} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1010\ 0111 \\ \oplus\ 1000\ 0001\ 0001\ 1100\ 1001\ 1101\ 1100\ 0101 \\ \hline 1000\ 0001\ 0001\ 1100\ 1001\ 1101\ 0110\ 0010 \end{array} \quad (3.2)$$

Für das aller erste Oktett, was in diesem Fall dem Wert 167 im Dezimalsystem entsprechen soll, wird die Verknüpfung in Gleichung 3.2 demonstrativ dargestellt. Der untere Wert entspricht dem Hash und repräsentiert den Offset für einen 32-Bit Hash. Anschließend wird das Ergebnis mit der Primzahl, für die jeweilige Bit-Zahl multipliziert und bildet das erste Zwischenergebnis für den Hash. Sollte das Oktett, das letzte in der Menge X sein, so ist das Ergebnis der finale Hash. Die einzige ungeklärten Teile an dem Verfahren, sind nur noch die Konstanten. Die vordefinierten Offsets unterliegen keinem komplexen System und wurden mit dem FNV-0 Algorithmus generiert. Um zurück auf den ursprünglichen FNV-0 Hash zu gelangen, wird der FNV-1a Algorithmus auf die Version FNV-1 zurückgeführt. FNV-1 verwendet die gleichen Operationen wie FNV-1a, nur dass die Zeile mit der Multiplikation, mit der XOR-Operation vertauscht sind. Und FNV-0 entspricht FNV-1, nur dass der Offset auf Null gesetzt wird. Die Eingabe für den Hash entspricht der Zeichenkette "chongo <Landon Curt Noll> ^.^" und wurde von den Autoren aus einer echten Nachricht gewählt, was als zufällig betrachtet werden kann. Für die Primzahlen gibt es bestimmte Kriterien, welche die Verteilung der Hashes begünstigen. Diese Kriterien führen allerdings über die Theorie in dieser Arbeit hinaus und werden daher nicht näher betrachtet.

Der Kernel für die GPU, lässt sich relativ einfach aus dem Algorithmus 2 bilden. Jeder Thread bekommt ein Element x aus der Gesamtmenge X zugewiesen und berechnet den Hash, wie in Algorithmus 2. Die einzige Spezialisierung durch die GPU, ist die Zuordnung der Elemente, zu den einzelnen Threads. Wie es bereits in vorherigen Abschnitte betrachtet wurde.

Addition

Das FNV-Verfahren stützt sich auf Primzahlen, welche als Faktor dienen und für eine gute Verteilung verantwortlich sind. Im Gegensatz dazu, führt die einfache Addition nicht zu einer guten Hash-Funktion. Gegen eine gute Verteilung spricht ein einfaches Beispiel. Angenommen es wird ein Hash mit 32-Bit gebildet. Das entspricht 2^{32} Möglichkeiten, für den resultierenden Hash. Gibt es nun eine Eingabe mit Zahlen, die maximal den Wert 2^{10} repräsentieren können, so ist das größte Ergebnis 2^{11} . Dadurch würde jeder mögliche Hash in diesem Zahlenbereich liegen. Wären es mehr als eine Addition, kann der Wertebereich sich erweitern, jedoch nicht

Algorithm 4 Shifted Addition Hash-Funktion

Require: X
 $Hash \leftarrow 0$
 $Offset \leftarrow 0$
 for Octet in X **do**
 $Offset \leftarrow Offset + Octet$
 $Hash \leftarrow Hash + Shift(Octet, Offset)$
 end for
 return $Hash$

Algorithm 5 Multiplikation Hash-Funktion

Require: X
 $Hash \leftarrow 1$
 for Octet in X **do**
 $Hash \leftarrow 1 + Hash \times (Octet + 1)$
 end for
 return $Hash$

signifikant. Die Hash-Funktion bildet deshalb nur ein Drittel der Gesamtmenge an, was keiner guten Verteilung entspricht. Des weiteren sind benachbarte Werte auch nicht weit voneinander entfernt. Wenn zum Beispiel nur der letzte Summand eine Differenz von k hat, dann hat der Hash auch nur eine Differenz von k . Weitere Kollisionen entstehen durch das Kommutativgesetz der Addition, wodurch Eingaben mit gleichen Teilwerten, in unterschiedlicher Reihenfolge, trotzdem den gleichen Hash erzeugen. Der Ablauf der Hash-Bildung wird im Algorithmus 3 abgebildet. Die Initialisierung des Hashes hat nicht die Probleme, wie bei der Multiplikation, daher darf der Offset bei Null liegen. Für die Addition sind die Grenzwerte des Zahlenbereichs zu beachten, weshalb immer Modulo mit dem höchsten Wert im Zahlenbereich gerechnet wird. Um diese Schwächen zu überwinden, müssen die Additionen abhängig von der Reihenfolge der Teilwerte sein und die Ergebnisse über den gesamten Zahlenraum verteilt werden. Eine Lösung bieten die Änderungen in Algorithmus 4. Grundsätzlich wird für die Addition nur ein Offset eingefügt, damit auch kleine Werte Einfluss auf größere Werte haben können. Das Prinzip entspricht nichts anderem, als einer Multiplikation des Oktetts. Der Offset bildet sich dabei immer aus dem letzten Offset und dem Wert des aktuellen Oktetts, modulo der maximalen Bits des Hashes. Wobei die Länge zusätzlich durch die eigentliche Größe des Oktetts beschränkt wird, damit keine Bits verloren gehen und dadurch einzelne Werte, einen ungleichen Einfluss auf den finalen Hash haben. Die Shift-Funktion verschiebt anschließend den Wert des Oktetts, immer um den Offset nach links. Das verschobene Oktett und der Hash werden darauf addiert und ersetzen den bestehenden Hash. Durch die Einführung des Shifts mit dem Offset, welcher sich mit jeder Iteration aufbaut, wird eine Abhängigkeit durch die Reihenfolge geschaffen, was ein Kollisionsrisiko minimiert. Eine gute Verteilung soll durch die Shifts garantiert werden. Die Eigenschaft, dass benachbarte Werte weit voneinander entfernt, als Hash auftauchen sollen, wird nur teilweise erreicht. Geringe Unterschiede einzelner Werte führen auch nur zu kleinen Unterschieden in den Shifts, was zum bekannten Problem durch die Addition führt.

Multiplikation

In der Addition ist die resultierende Summe immer das Problem für die schlechte Verteilung, weil sich die Hashes immer nah im Wertebereich der Summanden befinden werden. Durch die Multiplikation der Faktoren, können viel größere Distanzen zwischen den Hashes und den einzelnen Faktoren entstehen, wodurch auch kleinere Zahlen einen größeren Wertebereich ab-

Algorithm 6 Shifted Multiplikation Hash-Funktion

Require: X

```
Hash  $\leftarrow 1$ 
Offset  $\leftarrow 0$ 
for Octet in  $X$  do
    Offset  $\leftarrow$  Offset + Octet
    Hash  $\leftarrow 1 + \text{Hash} \times \text{Shift}(1 + \text{Octet}, \text{Offset})$ 
end for
return Hash
```

bilden. Daher ist der nächste Schritt, die Addition aus Algorithmus 3, mit der Multiplikation zwischen Hash und Oktett zu ersetzen. Da eine Multiplikation mit Null immer den Werte Null ergeben würde, muss auch der Offset auf Eins erhöht werden. Gleiches gilt für jeden Iterationsschritt. Da alle Oktette einen gleichen Einfluss auf das Endergebnis haben sollen, darf es nicht ein Oktett geben, was dazu führt, dass ein Zwischenergebnis den Werte Null zugewiesen bekommt. Dafür muss jeweils das Oktett und der temporäre Hash einen Offset von Eins erhalten (Algorithmus 5). Da es immer Oktette geben kann, welche den Wert Null repräsentieren, würde der soeben genannte Fall eintreten. Einen weiteren Fall deckt der Offset für den temporären Hash ab. Da es wie bei der Addition, zu Überläufen in der Kombination beider Werte führen kann, besteht die Möglichkeit, dass Teilergebnisse auf Null zurückfallen. Für die Berechnung werden nur Datentypen verwendet, welche nicht vorzeichenbehaftet sind und im späteren Verlauf, für die Programmiersprache typische, "unsigend" Bezeichnung erhalten. Bei diesen Datentypen fangen die Zahlen wieder von Null an, sobald ein Overflow auftritt, was der Modulo-Operation mit $2^{Bits} - 1$ entspricht. Daher reicht es aus, immer den Wert mit Eins zu addieren und somit die Null als Hash zu verhindern. Die Multiplikation unterliegt auch, wie die Addition, dem Kommutativgesetz. Daher gelten die gleichen Folgen, wie bei der Addition und die Maßnahmen ähneln sich. Im Algorithmus 6 wird die neue Variante dargestellt. Der Shift erfüllt die gleiche Funktion, wie bei der Addition und die Berechnung des Offsets stimmt auch überein. Anders als bei der Addition, sind die Teilergebnisse größer, was für einen kleineren Offset sprechen würde. Jedoch sind die Ergebnisse, der Byte-Multiplikation noch zu gering, als dass eine sinnvolle Verteilung mit wenigen Bytes entstehen könnte. Eine Lösung wäre daher die Erhöhung des Hash-Startwerts, wie beim FNV-Algorithmus. Zusätzlich bietet die Erhöhung des Offsets den Vorteil, dass bereits beim ersten Wert ein sehr großer Shift und somit ein höherer Wert erzielt wird. Beide Anpassungen bedingen sich nicht und können auch als eigenständige Ergänzung integriert werden. Für die Erstellung eines Kernels kann die Beschreibung aus der Addition wiederverwendet werden. Lediglich die Initialisierung der Parameter und die Operationen auf dem Hash müssen auf die Multiplikation angepasst werden.

XOR

Algorithm 7 Shifted XOR Hash-Funktion

```

Require:  $X$ 
     $Hash \leftarrow 0$ 
     $Offset \leftarrow 0$ 
    for Octet in  $X$  do
         $Offset \leftarrow Offset + Octet$ 
         $Hash \leftarrow Hash \oplus Shift(Octet, Offset)$ 
    end for
    return  $Hash$ 

```

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

(3.3)

Wird der Übertrag von der Addition entfernt, so bleibt nur noch der XOR-Operator (\oplus) zurück. Als Auffrischung ist das XOR in der Wahrheitstabelle 3.3 dargestellt. Der Operator kombiniert zwei Bits und gibt nur dann Eins zurück, wenn die Bits einen unterschiedlichen Wahrheitswert besitzen. Im Gegensatz zu den anderen typischen binären Operanden, ist die Wahrscheinlichkeit der ausgegebenen Wahrheitswerte gleichverteilt. Weshalb er sich bessere für das Hashen eignet, da der Hash nicht zu bestimmten Bit-Werten neigt. Einen weiteren Vorteil kann XOR im Schwerpunkt Geschwindigkeit vorweisen. So sind XOR-Verknüpfungen durch eine sehr geringe Latenz geprägt [31]. Für die Implementierung wurde, wie bei der Addition und der Multiplikation, ein Algorithmus als Basis geschaffen, um grundlegende Eigenschaften des Hashes zu evaluieren. Hierfür kann der Algorithmus 3 betrachtet werden, welcher genau die gleiche Funktion erfüllt, nur dass die Addition durch den XOR-Operanden \oplus ausgetauscht wurde. Für einen 8Bit-Hash wäre das vollkommen ausreichend, jedoch können größere Hashes nicht bedient werden, da immer nur in dem ersten Byte gearbeitet wird. Wie auch in den anderen Verfahren genannt, erfolgt deshalb ein Shift zum Most-Signifikant-Bit. Die Größe des Shifts wird wieder vom aktuellen Oktett und dem vorherigen Offset bestimmt. Der daraus resultierende Algorithmus ist in 7 zu sehen. Auch wenn es bei unsigned Datentypen keine Rolle spielt, so ist es ein wenig einfacher mit dem XOR-Operanden zu arbeiten, da keine Overflows zu berücksichtigen sind. Für die Implementierung des Kernels, wird wieder die Beschreibung aus der Addition verwendet.

XOR Multiplikation

$$x = x^{-1} \quad (3.4)$$

Ohne die Shift-Operation besitzt der XOR-Operand eine kleine Schwäche. Für das XOR gilt, dass das inverse Element immer gleich dem ausgehenden Element ist. Was der Gleichung 3.4 entspricht. Das neutrale Element ist die Null und somit ergibt $x \oplus x^{-1}$, immer Null. Mit dem Shift wäre der Fehler bereits behoben, jedoch besteht auch die Möglichkeit, den Shift, gleich durch die Multiplikation mit einem Wert aus X , zu ersetzen. Hierfür wird alternierend multipliziert und ein binäres XOR durchgeführt. In Algorithmus 8 wird der Wechsel durch den Oktettindex, der Position des Oktetts in X gewählt. Wichtig ist wieder der initiale Hashwert, welcher auf Eins gesetzt wird. Ohne der Eins, führen Multiplikation zurück auf Null. Gleicher gilt auch für den Schritt mit der Multiplikation, weil die XOR-Operationen auch Hashes mit Null erzeugt. Das

Algorithm 8 XOR Mult Hash-Funktion

Require: X

```
Hash ← 1
OctetIndex ← 0
for Octet in  $X$  do
    if OctetIndex mod 2 then
        Hash ← Hash ⊕ Octet
    else
        Hash ← 1 + Hash × (1 + Octet)
    end if
    OctetIndex ← OctetIndex + 1
end for
return Hash
```

Algorithm 9 HW XOR Hash-Funktion

Require: X

```
Hash ← 0
Offset ← 0
for  $x_{32}$  in  $X$  do
    Hash ← Hash ⊕ __funnelshiftl( $x_{32}, x_{32}, Offset$ )
    Hash ← __brev(Hash)
    Offset ← Offset + 1
end for
return Hash
```

Reihenfolgeproblem ist durch das alternierende Berechnen des Hashes überwunden, trotzdem können zusätzliche Shifts, wie bei den vorherigen Verfahren, eingebaut werden. Um die Verteilung noch zusätzlich zu verbessern, ist es möglich mehrere Iteration der Schleife aus den jeweiligen Verfahren durchzuführen. Dadurch entsteht ein neuer Parameter, welcher die Anzahl der Schleifendurchläufe angibt. Wenn dieser Zusatz angewendet wird, dann werden die Namen mit einem N als Präfix ergänzt. So würde aus "xor_mult_hash", "n_xor_mult_hash" werden. Die Namen stehen für die Funktionen, in der C++-Implementierung.

Hardware Intrinsiken

Hardware Intrinsiken sind Funktionen, welche spezielle Hardware-Funktionen widerspiegeln. So gibt es für die Thread-Kommunikation, Warp-Intrinsiken oder auch Funktionen, welche bei den mathematischen Operationen unterstützen. Für die Berechnung der Hashes, gibt es für die Implementierung keinen Anwendungsfall, welcher die Kommunikation zwischen Threads erfordert. Viel mehr bietet Cuda, für Integer-Operationen Intrinsiken, welche das verschieben von Bits vereinfacht. Darunter zählt __brev und __funnelshift_l. __brev dreht die Reihenfolge der Bits um und __funnelshift_l verbindet zwei Integer zu einem 64-Bit Integer, führt auf diesen Wert ein Shift aus und gibt den linken oder rechten 32-Bit Anteil zurück. Wenn __funnelshift_l mit jeweils zwei gleichen 32-Bit Integer-Werten ausgeführt wird, dann ist das Ergebnis, eine Permutation mit so vielen Schritten, wie es der Shift angibt. Intern ist der Shift auf 31 beschränkt. Ein reiner Hash mit nur den zwei Operationen, ist nicht möglich, da keine Bits miteinander verknüpft und somit nur alte Information verdrängt werden. Die Verdrängung erfolgt in diesem Fall durch die Shift-Operation. Daher werden alle vorher vorgestellten Funktionen, mit den neuen Intrinsiken ausgestattet. Zusätzlich bietet CUDA kein __funnelshift_l für 64-Bit Integer an, weshalb die Adaptierung auf 64-Bit Hashes, nicht einfach durchzuführen ist. __brev_l ist

dagegen eine Funktion, welche auch 64-Bit Zahlenwerte unterstützt. Die Funnel-Shift-Funktion ersetzt die Shift-Funktion und der Offset wird um den 32-Bit Wert erhöht. In den ersten Versionen, wurde der Offset nur um Eins erhöht. Das führte allerdings dazu, dass sich bei sehr kleinen Werte, die Hashes am Rand akkumuliert haben. Mit dem vollen Elementwert, ist die Verteilung deutlich besser ausgefallen. Zusätzlich erfolgt am Ende der Verknüpfung, die `_brev` Funktion, welche den Hash für den nächsten Chunk umdreht. Dadurch sollen auch kleinere Werte, Einfluss auf größere Werte haben und umgekehrt. Alle Funktionen, mit diesen Intrinsiken, bekommen das Kürzel "HW" zugewiesen. Die beschriebene Funktion, ist anhand der XOR-Funktion in Algorithmus 9 abgebildet. Mit Blick auf die Verteilungseigenschaften, ist anzumerken, dass die Permutation nur um 32-Bit Zahlenraum stattfindet. Ein Wert kann nie über diese Grenze hinaus, weshalb die Überschreitung dieser Grenze, nur durch das Umdrehen des Hashes oder der Verknüpfung mit dem Hash erfolgen kann. Das Reihenfolgeproblem spielt für die Implementierung keine Rolle. Genau wie in den Lösungen für die anderen Funktionen, soll die Permutation eine Abhängigkeit zur Position in der Eingabe darstellen. Der Zweck dieser Funktionen ist, die Vorteile aus den Verknüpfungen und den speziellen Hardwarefunktionen zu vereinen und eine performante Hash-Funktion zu repräsentieren.

3.1.3 Speicherverwaltung

Die vorgestellten Verfahren verwenden alle das gleiche Schema, um die einzelnen Oktette oder Bytes zu verarbeiten. Jeder Thread verarbeitet mindestens ein Element und benachbarte Threads, verarbeiten auch benachbarte Elemente, damit Coalesced-Memory-Access gewährleistet ist. Die resultierenden Hashes sollen immer 32- oder 64-Bit umfassen, weshalb der maximale Shift, abhängig vom Hash zu betrachten ist. Die Ergebnisse werden zurück in den Global-Memory geschrieben, weshalb für jeden Hash, auch mit einem Store-Befehl von 32- oder 64-Bit zu rechnen ist. Die Positionen der Hashes ist bereits bekannt, da der Hash-Buffer bereits vor dem Start allokiert wird und genau die Anzahl an Hashes, der Anzahl der in der Eingabe vorhandene Element X entspricht. Der Datentyp für die Eingabe kann frei gewählt werden und sollte mindestens einen Byte umfassen. Jeweils einen Byte nach dem anderen zu laden, ist sehr umständlich und lastet die Speicher-Anbindung ineffizient aus. Wenn der Compiler nicht optimierend eingreift, dann wird für jeweils ein Byte in X, ein Load-Befehl ausgeführt. Was zur Folge hat, dass nicht nur deutlich mehr Speicheranfragen anfallen, sondern auch deutlich mehr Schleifendurchläufe ausgeführt werden müssen. Zusammen treibt das die Instruktion und die Latenzen in die Höhe. Als Ausweg ist es daher besser, größere Datentypen aus dem Speicher zu laden. So steht bei Cuda, neben dem herkömmlichen 32- und 64-Bit-Datentypen, auch Vektoren zur Verfügung, welche bis zu 128-Bit umfassen können. Das entspricht vier Integer-Werten, mit nur einer Anfrage. Hierfür müssen jedoch alle Algorithmen angepasst werden. So reduziert sich der maximale Shift und die Verknüpfungen müssen auf die Datentypen abgestimmt sein. In der C++-API bestehen die Vektoren immer aus den Feldern X, Y, Z, W. Je nach Bit-Größe, steht zum Beispiel nur X und Y zur Verfügung. Wie es aus den Begrenzungen bereits hervorgeht, ist jedes einzelne Feld immer ein 32-Bit Integer (Unsigned). Aus allen Oktetten in den Algorithmen, werden daher 32-Bit Integer, ohne Vorzeichen verwendet. Der Eindimensionale Vektor gleicht dem Integer und ändert damit nicht wirklich den Syntax. Bei einem 2D- oder 4D-Vektor jedoch, ist mehr als nur eine Verknüpfung in einem Iterationsschritt zu verarbeiten. Das führt dazu, dass ein sogenannter Loop-Unroll angewendet wird. Bei einem Unroll werden einzelne Iterationsschritte zu einer Iteration zusammengefasst, wobei die Instruktionen sequentiell in Schleifenrumpf stehen. Ein Unroll für den 2D-Vektor würde somit nur halb so viele Iterationen besitzen, aber doppelt so viele Elemente, in einem Schritt nacheinander verarbeiten. Der 4D-Vektor kann analog integriert werden. Bisher wurden nur über die Vorteile, durch das verwenden von größeren Load-Befehlen gesprochen. Ein wesentlicher Nachteil ist die Vorbereitung der Daten. Während bei einzelnen Bytes, einfach die Werte nacheinan-

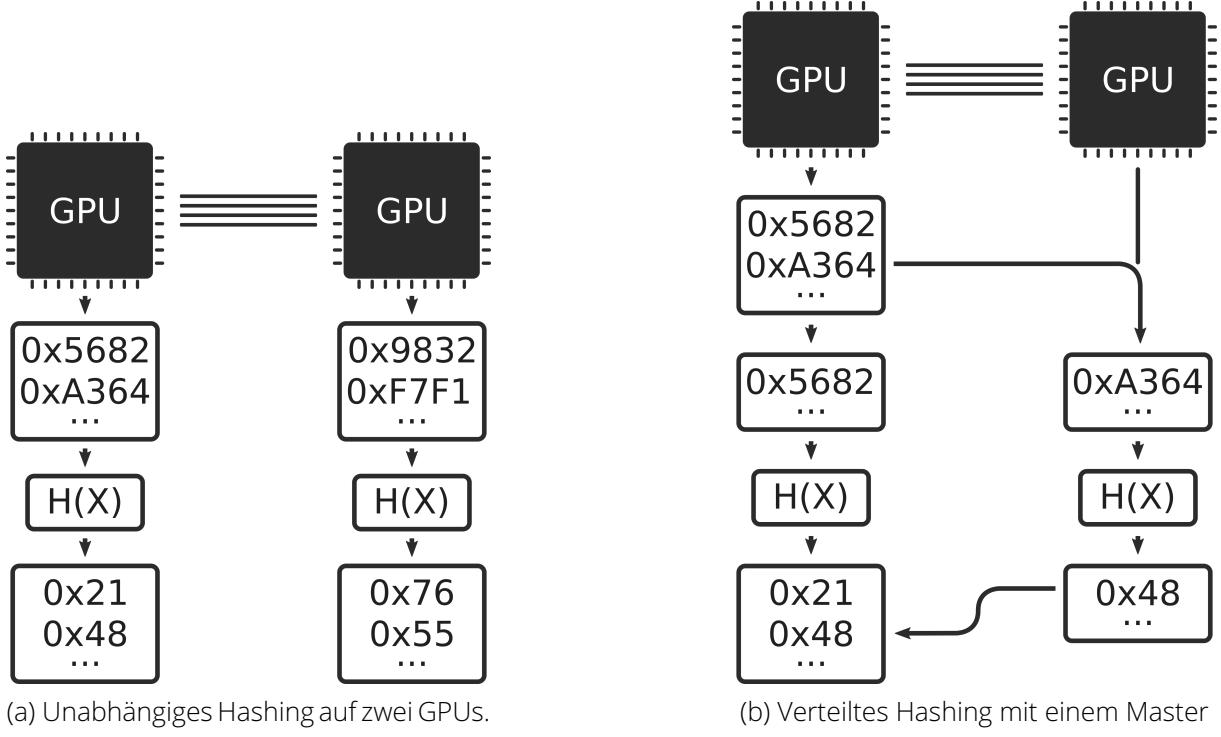


Abbildung 3.1: Hashing auf mehreren GPUs

der in dem Buffer liegen, müssen für die Vektoren, die Daten in einem bestimmten Alignment vorhanden sein. Das Alignment entspricht immer der verwendeten Vektorgröße und darf nicht unterschritten werden. Daraus folgt Padding, welches im Zweifel zu einer schlechten Speichernutzung führen kann. Für Eingabegrößen, welche durch die Vektorgröße teilbar sind, kann der Kernel effizient die Daten aus dem Speicher laden. Bei nicht teilbaren Größen, müssen die Padding Bytes eingefügt werden. Da der Algorithmus deterministisch sein muss, dürfen die Padding-Bytes keine zufälligen Werte besitzen und sind immer auf Null gesetzt. Neben dem ineffizienten Laden spielt auch der Speicherverbrauch eine Rolle. Der schlimmste Fall trifft hier auf einen einzelnen Byte zu. Für das einfache Byte-Modell, ist der Speicherverbrauch immer zu einhundert Prozent Nutzdaten und verschwendet keinen Speicher an Padding Bytes. Für das Vektor-Modell, kann im schlimmsten Fall, nur ein Byte Nutzdaten entsprechen und der Rest fällt an das Padding. Der Anteil an Nutzdaten, bei einem Vektor von 16 Bytes, beträgt in diesem Fall nur sechs Prozent. Das Gute ist, dass sich dieses Verhältnis verbessert, um so länger die Eingabesequenz ist.

$$Padding(X, S) = \lceil \frac{|X|}{|S|} \rceil \times |S| - |X| \quad (3.5)$$

$$P_{X,S} = \frac{Padding(X, S)}{\lceil \frac{|X|}{|S|} \rceil \times |S|} \quad (3.6)$$

Der Anteil lässt sich mit der Gleichung 3.5, durch das Padding berechnen. $|X|$ entspricht den Bytes, die in X enthalten sind und $|S|$ entspricht den Bytes, die der Vektor-Typ umfasst. Der Padding-Anteil wird durch die Gleichung 3.6 bestimmt. Für die Evaluation ist daher nicht nur die Geschwindigkeit, sondern auch die Speicherauslastung ein wichtiges Kriterium.

3.1.4 Multi GPU

Die Kernel sind fertig und das Datenformat vollständig. Für den nächsten Schritt, ist die Anordnung der GPUs an der Reihe. Eine einzelne GPU ist nicht von großem Interesse, da sich die Algorithmen ohne Probleme auf einer GPU ausführen lassen. Sobald mehrere GPUs ins Spiel kommen, muss die Verteilung der Daten beachtet werden. In diesem Fall kann von zwei Möglichkeiten ausgegangen werden, welche in Abbildung 3.1 zu sehen sind. In der einfachsten Form, besitzen beide GPUs einen Teildatensatz. Dieser wird nie zwischen den GPUs geteilt, wodurch auch keine Kommunikation zwischen den GPUs stattfindet. Diese Anordnung wird relevant, wenn der Speicher aller GPUs, wie ein Unified-Memory betrachtet wird. Hierfür müssen sich die GPUs in den Datensatz teilen und die Ausgabe in den Hash-Buffer schreiben. Die Buffer werden in einer Liste gespeichert und der GPU-Index gibt die Position in der Liste an. Am Ende gibt es zwei Listen. In der Einen sind die Speicheradressen für die X-Buffer gespeichert und in der Anderen die Speicheradressen für die finalen Hash-Buffer. Keiner der Buffer wird GPU übergreifend verwendet und liegt somit immer nur auf einer GPU. Der genaue Ablauf ist in Abbildung 3.1a dargestellt. Als Basis dienen zwei GPUs. Das in dieser Arbeit verwendeten System, bietet zwei identische GPUs. Beide allokierten zu Beginn, die Buffer für die Hashes und den Eingaben. Als nächstes wird die Eingabe durch die Hash-Funktion $H(X)$ verarbeitet und der Hash in den Hash-Buffer geschrieben. Im Beispiel ist der Hash nur ein Byte groß und die Eingabe das Doppelte. Für die Evaluation spielt im Grunde nur der Schritt mit der Hash-Funktion eine Rolle, da die Beschaffung der Buffer in allen möglichen Varianten geschehen kann und diese nicht unbedingt vorher erst allokiert werden müssen. So liegen diese bei einem GPU In-Memory-System bereits auf dem GPU-Speicher vor. Komplizierter wird es, wenn die Daten nur von einer GPU kommen. Der Fall wird in Abbildung 3.1b dargestellt. Als Grundlage gibt es einen Buffer, welcher alle Daten für die Eingabe beinhaltet, welcher nur auf einer GPU zu finden ist. Für das GPU-Setup wird nun der NVLink wichtig und ist als Linien zwischen den GPUs gekennzeichnet. Ein Teil der Daten muss zwischen den GPUs verteilt werden und dies geschieht über die teils sehr schnelle Anbindung. Damit das Kopieren sehr leicht ablaufen kann, kopiert sich jede GPU einen bestimmten Block aus den Ausgangsdaten. Da jeder Block gleich groß für eine GPU sein soll, ist der Offset so groß wie die durchschnittliche Blockgröße pro GPU. Der Offset wird dann mit dem GPU-Index multipliziert, woraus der finale Offset für jede GPU, im Datensatz bestimmt werden kann. Das Kopieren geschieht asynchron, weshalb in der Darstellung, die Abläufe auch nebeneinander sind. Für die Berechnung des Hashes, wird ein Intermediate-Buffer, für alle weiteren GPUs benötigt. Die sogenannte "Master-GPU" besitzt bereits den finalen Buffer und kann die Ergebnisse direkt schreiben. Durch die Kopier-Operationen entstehen neue Latenzen, welche im "Single" Betrieb nicht vorhanden sind. Um diese Latenzen zu rechtfertigen, ist es sinnvoll die Hash-Funktion so anzupassen, dass neue Vorteile durch die Funktion gewonnen werden, welche evtl. im späteren Verlauf, die GPU ein paar Abkürzungen nehmen lässt. Zusätzlich können im Shared-Betrieb auch nur eine verminderte Hashanzahl berechnet werden, da alle Daten am Ende auf eine GPU passen müssen. Deshalb sollte die GPU mit dem meisten Speicher immer den Master abbilden. Wenn wie beim in dieser Arbeit verwendeten System, beide Grafikkarten identisch sind, kann davon ausgegangen werden, dass im ersten Modus eine doppelte Menge an Hashes, ohne Kopier-Latenzen berechnet wird.

3.1.5 Zusammenfassung

Für das Hashing, zählt in dieser Arbeit hauptsächlich die Geschwindigkeit. So wurden Hash-Funktionen vorgestellt, welche so schnell wie möglich, einen Hash erzeugen. Neben der Geschwindigkeit wird auch Wert auf die herkömmlichen Hash-Eigenschaften, wie Kollision und Verteilung geachtet. Als Grundlage dienen die Addition, Multiplikation und das XOR. Diese haben alle ihre Vor- und Nachteile, welche mit Shifts überwunden werden. Außerdem werden XOR und die Multiplikation in einer Funktion verbunden. Als Ablösung für den Shift, gibt es

Hardware-Intrinsiken, welche Bit-Operationen auf Integer-Werte durchführen. Diese sind die Permutation für 32-Bit-Integer und das Umdrehen der Bit-Reihenfolge. Alle Funktionen verwenden die gleiche Strategie, wie Daten aus dem Global-Memory geladen werden, was dazu führt, dass die Funktionen einfach austauschbar sind. Aus diesem Grund ist es auch möglich, die Funktionen für das vorgestellte GPU-Verfahren, einzusetzen. Da das Testsystem mehr als eine GPU bietet, erfolgt die Übersetzung auf ein Multi-GPU Modell. In diesem Modell agiert die weitere GPU entweder als Koprozessor oder verarbeitet einen eigenen Hash-Buffer. Beide Varianten wurden ausführlich vorgestellt und Vor- und Nachteile besprochen.

3.2 Hash-Join

In diesen Abschnitt folgt der Einsatz des Hashings. Es wird ein Hash-Join Verfahren implementiert und alle Feinheiten und Tücken genauer betrachtet. Der Algorithmus wird von Grund auf entwickelt und verwendet einzelne Optimierungen aus Sioulas et al. [32] und Rui et al. [33]. Die GPU spielt für den Hash-Join die Hauptrolle. So werden nicht wie bei bisherigen Algorithmen, Daten erst aus dem Hauptspeicher der CPU geladen, stattdessen liegen die Daten bereits auf der GPU und bilden somit einen In-Memory Hash-Join. Der Vorteil des Ansatzes ist, dass keine Daten über den PCIe-Bus, in den Hauptspeicher der GPU kopiert werden muss und somit Kernels nicht auf den Transfer warten müssen. Dem gegenüber steht die Größe des Hauptspeichers der GPU. Wie in späteren Abschnitten zu sehen ist, können bestimmte Operationen sehr viel Speicher beanspruchen, welcher vor dem eigentlichen Ergebnis zur Verfügung gestellt werden muss. Dadurch entstehen Limitierungen, was die Element-Größe betrifft. Wie mit diesen Limitierungen umgegangen werden kann und muss, wird in den betreffenden Abschnitten vorgestellt.

3.2.1 Zielstellung und Annahmen

Das Ziel in dieser Arbeit besteht darin, einen Hash-Join-Algorithmus bereitzustellen, welcher nur auf dem Speicher der GPU arbeitet und sich im Zweifel auf mehrere GPUs verteilen lässt. Um In-Memory zu erreichen, wird davon ausgegangen, dass die Daten bereits auf der GPU vorliegen. Für den Algorithmus wird ein bestimmtes Datenformat erwartet, welches im folgenden Abschnitt genauer betrachtet wird. Die Begrenzung der Datentypen liegt bei 64 Bits. Diese Begrenzung ist nicht zufällig gewählt, sondern von den Cuda-Atomics bestimmt. Eine Atomic-Funktion wird nur für Datentypen mit 16-, 32- oder 64-Bit [20] angeboten, welche in bestimmten Funktionen notwendig sind und daher die Limitierungen diktieren. Deshalb erhalten Primary-Keys (PKs) und Columns, jeweils einen Wert für einen 64-Bit unsigned Integer (Abbildung 3.2). Columns oder Spalten, werden auf reine Zahlen reduziert, was den Vergleich vereinfacht. Die Vektorisierung, wird anhand der vorherigen Hash-Funktionen getestet, sodass der Einfluss von unterschiedlich guten Hash-Funktionen, auf die Laufzeit des Algorithmus bewertet werden kann. Für unterschiedliche Probing-Algorithmen, gibt es auch unterschiedliche Anforderungen, was die Limitierung der maximalen Elemente betrifft. Dafür werden Vektoren aus den Datensätzen gebildet, welche nacheinander von der GPU verarbeitet werden können. Außerdem ist die Ausführung nicht nur auf eine GPU begrenzt und wird auf mehrere GPUs übertragen. Das Finale Produkt des Algorithmus, ist eine Relations-Tabelle aus den übereinstimmenden Zeilen, aus beiden Tabellen. Die Spaltenwerte werden durch die Primary-Keys der jeweiligen Tabelle repräsentiert.

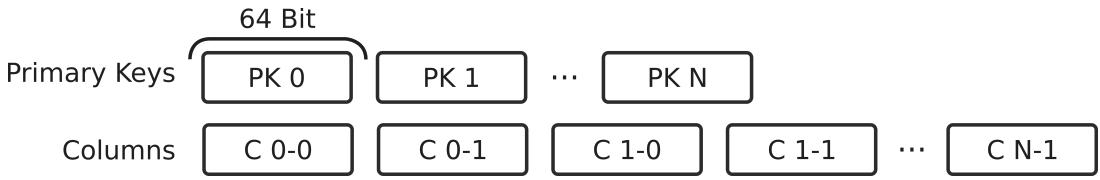


Abbildung 3.2: Tabellen Datenformat

3.2.2 R/S Datensatz

$$|R| \leq |S| \quad (3.7)$$

Für das Durchführen eines Hash-Joins, werden zwei Tabellen benötigt, welche die jeweils zu verknüpfenden Spalten enthält. In der Literatur [32, 34, 35] werden die Tabellen, R und S genannt. R repräsentiert die Tabelle, mit den wenigstens Zeilen und S, die mit den meisten Zeilen. Daher wird auch in dieser Arbeit, die Bezeichnung fortgeführt. Formal ist die Restriktion in Gleichung 3.7 gegeben. Aus dem vorherigen Abschnitt ist bekannt, dass für eine Tabelle, PK und die eigentlichen Werte-Spalten notwendig sind. Für jede Zeile, in einer Tabelle, gibt es jeweils genau einen PK und eine endliche Anzahl an Spalten, welche die Daten für die Zeile beinhalten. Die Darstellung im internen Speicher kann in zwei wesentliche Formen unterschieden werden. Zum einen ist es möglich, eine komplette Zeile in einem Array zu speichern oder für jede Spalte, inklusive PK, einen extra Array anzulegen. Ersteres trägt den Namen Array-of-Structs (AoS) und das andere den Namen Struct-of-Arrays (SoA). Beide finden ihre Anwendungsbereiche, wenn für unterschiedliche Zugriffs-Formen, ein coalesced Speicherzugriff erfolgen soll. AoS eignet sich am besten, wenn alle Elemente in einem Struct an gleicher Stelle benötigt werden. SoA bietet sich somit genau für das Gegenteil an. Sobald nur ein Feld, von allen Elementen verwendet werden soll, bietet sich diese Form am besten an. Zusätzlich ist das Padding der Structs zu beachten. Ein Struct trägt eine endliche Menge an Feldern, wobei alle Felder einen bestimmten Datentyp besitzen. Durch diesen Datentyp ist auch die Größe der Elemente bekannt und alle Felder addiert, ergeben die Größe des gesamten Structs. Hier gilt es zu beachten, dass der Speicherzugriff immer nur in bestimmten Byte-Größen erfolgen kann. So kann es in ungünstigen Konfigurationen dazu führen, dass Padding-Bytes eingeschoben werden, welche den Speicherverbrauch erhöhen. Im Fallbeispiel der Datenbanktabelle, tritt das nicht auf, da alle Datentypen 64-Bit groß sind und dieses Format von der GPU, als Lade-Befehl unterstützt wird (siehe Vektoren). Allerdings werden PKs nur am Ende des Joins benötigt, da kein Hash mit dem Join gebildet oder der PK für Vergleichsoperationen verwendet wird. Im Gegenteil, der PK einer Zeile ist das Resultat aus dem Join. Die neue Tabelle besteht nur aus einzelnen PKs, aus R und S. Deshalb müssten Lade-Operationen immer das Feld des PKs im gesamten Prozess überspringen, was die Kapazität des Caches, im Bezug auf die Elementanzahl mindert. Daher ist es am Besten, wenn der PK seine eigenen Array bekommt. Die Schlussfolgerung wäre dann, dass die Daten als SoA gespeichert werden. Für die Spalten jedoch, ist ein Array-of-Structs vorgesehen. Das ist der Einfachheit geschuldet, welche dieses Konzept mit sich bringt. Da davon ausgegangen werden muss, dass es eine beliebige Anzahl von Spalten geben kann, müssen diese dynamisch erweiterbar sein. Hierfür könnte eine Liste mit jeweils den Speicheradressen für die Column-Buffer vorgesehen werden. Das führt allerdings zu einer unnötigen Komplexität, da eine 2D-Struktur verwaltet werden muss. Viel einfacher ist es, die Spalten-Werte direkt hintereinander in eine Liste zu packen, da für jeden Zugriff in der Regel alle Werte aus einer Zeile verwendet werden. Neben den Metadaten, wie zum Beispiel Länge, werden die Daten daher im Format, wie in Abbildung 3.2 beschrieben dargestellt.

Algorithm 10 Hashing der Tabelle

Require: Table, ElementType, HashType, HashFunction

```
Elements ← Table::Size
HashBuffer ← { $x_1, \dots, x_{Elements} | x \in \{0, 1\}^{|HashType|}\}$ 
ElementBytes ← Table::ColumnCount × |ElementType|
if ElementBytes > |HashType| then
    HashBuffer ← HashFunction(Elements, Table::Columns)
else
    HashBuffer ← Table::Columns
end if
return HashBuffer
```

3.2.3 Speicherlimitierungen

Jegliche Daten müssen auf der GPU vorliegen, so wurde die Anforderung definiert. Aus diesem Grund ist es besonders wichtig, dass die Algorithmen in einem wohl bekannten Speicherlimit arbeiten, damit vor der Ausführung bekannt ist, ob die Operation auf der GPU genügend Speicher zur Verfügung hat. Eine zufällige Fehlermeldung ist nicht benutzerfreundlich und verringert zusätzlich die Möglichkeit, Fehler in der Implementierung oder dem Datensatz zu finden. Der maximale und genutzte Speicher der GPU, kann zu jeder Zeit herausgefunden werden, weshalb Algorithmen frühzeitig blockiert werden können. Ist bekannt, dass der Datensatz keinen Platz im Hash-Join-Verfahren finden wird, so wird die Funktion frühzeitig beendet. Als Resultat wird eine leere Tabelle zurückgegeben.

3.2.4 Hashing

$$\text{Bytes}_{\text{HashBuffer}}(N, \text{HashType}) = N \times |\text{HashType}| \quad (3.8)$$

Das Hashing ist eine grundlegende Operation, damit überhaupt ein Hash-Join ablaufen kann. Da sich in der Arbeit nur auf 32- oder 64-Bit Hashes beschränkt wird, können die vorher vorgestellten Hashing-Algorithmen ohne Probleme wiederverwendet werden. Der Hash ist ein Resultat aus den jeweiligen Daten-Columns für eine Zeile. Am Ende entstehen daher pro Zeile, genau ein Hash. Ein Buffer muss dementsprechend bereit stehen. Die Größe des Buffers lässt sich durch die Gleichung 3.8 bestimmen. N gibt die Anzahl der Zeilen an und HashType , den verwendeten Datentyp für die Hashes. $|\text{HashType}|$ ist dementsprechend die Anzahl der Bytes in dem gegebenen Datentyp. Für das Hashing an sich, kann auf die vorher vorgestellten Hashes verwiesen werden. Das einzige was sich unterscheidet, ist wie die Daten für die Eingabe der Hash-Funktion vorbereitet werden. Hier macht sich die Entscheidung zur Auslagerung der PKs bemerkbar. Wären diese noch direkte Nachbarn im gleichen Array, von den eigentlichen Daten, so müsste die Hash-Funktion mit einem Offset ausgestattet werden, da der PK für den finalen Hash mehr als ungeeignet ist und deshalb übersprungen werden muss. Fließen die PKs mit in den Hash ein, dann wird es das Probing fehlschlagen, da eine Übereinstimmung von R und S fast ausgeschlossen ist. Mit dem festgelegten Format, kann jeder Hash aus einer vortlaufenden Folge von Bytes gebildet werden. Je nach Bytes für einen Hash, kann dies auch Vorteil mit dem Spalten-Datentyp bringen. Für 64-Bit Columns und einem 64-Bit Hash ist es zum Beispiel überflüssig, noch einen Hash zu bilden, da der gegebene Wert bereits alle Eigenschaften des Hashes erfüllt. Das erspart Zeit und Speicher, weil kein extra Hash-Buffer mehr notwendig wird. Sobald der Hash (in Bytes) kleiner als das Datentyp der Column wird, ist ein extra Buffer für den Hash wieder zu verwenden. Der beschriebene Ablauf ist als Algorithmus 10 zu sehen. In dem Dargestellten Fall wird der HashBuffer erst in der Funktion initialisiert. Die einzelnen Werte spielen dabei keine Rolle, es muss nur die Größe des Buffers korrekt sein.

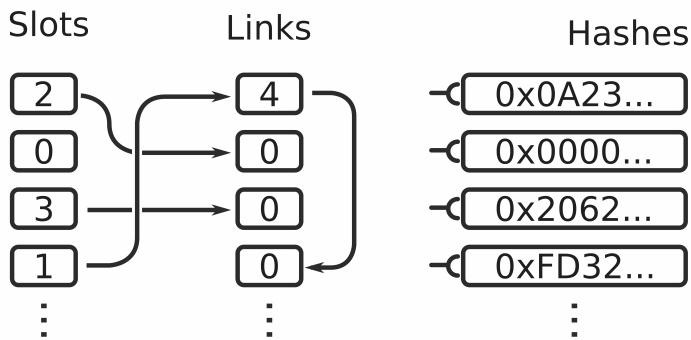


Abbildung 3.3: GPU Hashtabelle Aufbau

Algorithm 11 Probing in Hash-Tabelle

Require: Value, ValueHash, Hashtabelle

```

1: SlotIndex ← ValueHash mod Hashtabelle::SlotCount
2: Link ← Hashtabelle::Slots[SlotIndex]
3: ResultSet ← { }
4: while Link ≠ 0 do
5:   Link ← Link - 1
6:   if ValueHash = Table::Hashes[Link] then
7:     if Value = Table::Values[Link] then
8:       ResultSet ← ResultSet ∪ {Link}
9:     end if
10:   end if
11:   Link ← Table::Links[Link]
12: end while
13: return ResultSet

```

Alle Werte werden in der Hash-Funktion, in jedem Fall überschrieben. *Table* wird wie ein Struct verwendet, was der Implementierung entspricht. Daher sind alle Zugriffe an Structs angelegt und entsprechen einzelnen Feldern im Struct.

3.2.5 Probing

Mit den Hashes aus dem vorherigen Abschnitt, wird das Probing durchgeführt. Für das Probing ist eine Hashtabelle aus den Hashes von R notwendig. Der Aufbau ist in Abbildung 3.3 schematisch sichtbar. Für die Verlinkung ist eine Separate-Chaining Variante vorgesehen. Als alternative gibt es Open-Addressing Methoden [36]. In dieser Arbeit konnten jedoch keine effiziente Implementierung für dieses Verfahren auf der GPU implementiert werden, weshalb sich nur auf das Separate-Chaining konzentriert wird. Die Hashtabelle besteht aus insgesamt drei Komponenten und bedient sich am Konzept des Direct-Chainings. Wie es für eine Hashtabelle üblich ist, ist die Anzahl an Slots endlich. Ein Slot entspricht dem Startpunkt der Verkettenungen. Die Anzahl der Slots ergibt sich durch den gewählten Load-Faktor. Auf der GPU ist es einfacher mit Indizes, als mit reinen Memory-Offsets zu arbeiten, was den Atomics geschuldet ist. Speicheradressen besitzen feste Datentypgrößen und können daher bei einer Komprimierung keine Beitrag leisten. Ein Index dagegen, kann in der Größe variieren und somit auf bis zu 32-Bit reduziert werden. Die Untergrenze ist durch die Atomics gegeben, welche im späteren Verlauf ihre Anwendung finden. Wie es in der Abbildung 3.3 zu sehen ist, besitzt jeder Slot einen Zahlenwert. Dieser Zahlenwert entspricht dem nächsten Element in der Kette. Ist der Zahlenwert Null, dann ist das Ende der Kette erreicht. Ebenso ist es für die Links anzuwenden.

Algorithm 12 Hashtabelle Build-Phase

Require: Hashes, SlotCount

```
1: Slots  $\leftarrow \{0_0, \dots, 0_{SlotCount-1}\}$ 
2: Links  $\leftarrow \{x_0, \dots, x_{|Hashes|-1} | x \in \mathbb{N}\}$ 
3: HashIndex  $\leftarrow 0$ 
4: for Hash in Hashes do
5:   SlotIndex  $\leftarrow$  Hash mod SlotCount
6:   Links[HashIndex]  $\leftarrow atomicExch(Slots[SlotIndex], HashIndex)$ 
7:   HashIndex  $\leftarrow$  HashIndex + 1
8: end for
9: return Hashes, Slots, Links
```

In jeder Hashtabelle gibt es immer genau so viele Links, wie Hashes. Daher sind rechts daneben auch die Hashes abgebildet, die durch eine symbolische Verknüpfung, zu dem jeweiligen Link gehören. Als kleines Beispiel wird überprüft, ob ein Hash mit dem Wert h , in der Tabelle vorliegt. Im ersten Schritt wird der Rest des Hashes durch die Modulo-Operation gebildet. Der Wert des Modulos entspricht dabei der Anzahl, der Slot-Elemente. Für das Beispiel soll der Rest, den Wert Drei bekommen. Der Algorithmus überprüft den Slot und findet die Eins, was dem ersten Hash und Link entspricht. Der Hash wird mit h verglichen, jedoch passen die Werte nicht zueinander. Darauf folgt der Sprung in die Links-Liste, wo Vier als Wert gelesen wird. Der Hash entspricht auch nicht h und folgt der Verkettung in den Letzten Eintrag, da dieser einen Link auf die Null besitzt. Beim Vergleich mit dem Hash, stellt der Algorithmus eine Übereinstimmung fest. Als Reaktion müssen nun die ungehaschten Elemente miteinander verglichen werden, um eine Hash-Collision auszuschließen. Ist diese erfolgreich, so wird der gefundene Eintrag als Ergebnis zurückgegeben. Dieser Ablauf ist die Grundlage für das Probing und durch Algorithmus 11 zusammengefasst.

3.2.6 Build

In der Build-Phase wird die Hashtabelle aufgebaut. Im vorherigen Abschnitt wurde die Funktion und eine finale Hashtabelle genauer betrachtet, daher ist das Grundprinzip der Tabelle bereits bekannt. Zu Beginn startet der Build-Algorithmus mit einer leeren Tabelle. Die Anzahl der Slots wird vorher definiert und danach in der gewünschten Größe allokiert. Wäre die Hashtabelle nicht auf der GPU, so würden die Links und Hashes, Stück, für Stück hinzugefügt werden. Auf der GPU sind die Buffer vorher allokiert, weshalb die Anzahl der Hashes bekannt sein muss. Das ist kein Problem beim Hash-Join, da immer eine fixe Menge an Hashes verarbeitet werden. Daher sind die Buffer für die Links und Hashes, bereits in der Hashtabelle enthalten. Initial müssen alle Slots auf Null gesetzt werden. Wäre das nicht der Fall, könnten Werte aus vergangenen Operationen, im Speicher, die Kette auf eine falsche Route führen. Bei Links ist das nicht notwendig. Links werden mindestens einmal in der Erstellung gesetzt und haben deshalb keine Werte aus vergangenen Speicherfragmenten. Sobald alle Buffer allokiert sind, beginnt das Einfügen der Hashes. Dazu bedient sich der Algorithmus einem Trick aus Sioulas et al. [32]. Auf der GPU werden im Idealfall alle Hashes gleichzeitig in die Liste eingetragen. Dafür wird im ersten Schritt der Index für den Hash, wie aus dem vorherigen Abschnitt berechnet. Ist der Slot bereits größer als Null, also belegt, dann wird der aktuelle Slot-Link-Index durch den neuen Hash-Index ersetzt. Gleichzeitig wird der alte Slot-Link-Index in die Link-Liste, an die Stelle des Hash-Indexes kopiert.

An dieser Stelle gäbe es Probleme, wenn sehr viele Threads an die gleiche Stelle, ihren Hash-Index ablegen möchten. Als Lösung folgt eine *atomicExch(...)* Methode. Sie erwartet einen

neuen Wert und eine Speicheradresse. An der gegebene Speicheradresse (der Slot) erfolgt der Austausch mit dem neuen Wert und der alte Wert wird von der Funktion zurückgegeben. Durch die Atomic-Eigenschaft, werden die Austausch-Operationen Sequentiell durchgeführt. Der gesamte Prozess ist wieder im Algorithmus 12 zu sehen. Im Algorithmus werden nicht die einzelnen Hashes, in eine neue Liste eingetragen. Das liegt daran, dass die Sortierung des Hash-Buffers, im Prozess nicht verändert und deshalb direkt im Probing verwendet wird. Algorithmus 12 könnte direkt für einen Hashtabelle im Global-Memory übernommen werden. Allerdings wird auf Elemente in der Tabelle, mehr als einmal zugegriffen. Dieser Umstand spricht für einen Transfer in den Shared-Memory. Dafür muss Slots und Links und der Hash-Buffer, im Shared-Memory liegen. Der Hash-Buffer wird in der For-Schleife kopiert.

3.2.7 Join

Build und Probe liefern fast alle Werkzeuge, für den finalen Join. In dieser Arbeit werden drei Möglichkeiten untersucht, zum einen die klassische Variante, wo die Hashtabelle in den Global-Memory verweilt und zwei Varianten, welche die Hashtabelle in den Shared-Memory ablegen. Bevor die einzelnen Phasen des Probings durchlaufen werden, müssen bestimmte Buffer vorliegen. Egal für welche Variante sich entschieden wird, es ist ein Buffer für die Probing-Ergebnisse notwendig. Der Nutzen des Buffers folgt später. In diesem Buffer liegen alle möglichen Ergebnisse, welche dem Produkt aus R und S entsprechen. Zusätzlich muss die Anzahl der Elemente festgehalten werden, was in einer extra Variable gespeichert wird. Als Datentyp reicht ein 32-Bit Integer, weil die Grenze von 2^{32} nie überschritten wird, was am limitierenden Speicher der GPU liegt.

$$Buffer_{JoinResults}(R, S) = (|R| \times |S| + 1) \times 4\text{Byte} \quad (3.9)$$

$$HashtabelleSize(R, Slots, HashType) = |R| \times (4\text{Byte} + |\text{HashType}|) + Slots \times 4\text{Byte} \quad (3.10)$$

$$RS_{Table}(Matches) = Matches \times (3 \times 8\text{Byte}) \quad (3.11)$$

Zusammengefasst lässt sich der gesamte Speicher, für den Buffer mit Gleichung 3.9 berechnen. Die Eins entspricht der Zählervariable. Dadurch dass die Größe ein Produkt aus beiden Tabellen ist, wird sichtbar, dass bei größer werdenden Joins, sehr schnell der Platz eng werden kann. Neben dem Results-Buffer, wird für den Global-Memory-Join eine Hashtabelle benötigt. Für die Slots und den Links, gibt es 32-Bit Integer und die Hashes sind je nach Datentyp, 32- oder 64-Bit. Auch wenn nur sehr kleine Slot-Zahlen zu erwarten sind, so ist kein kleinere Datentyp, als 32-Bit möglich. Hierfür ist das *atomicExch(...)* verantwortlich, was nur für 32- und 64-Bit Integer angeboten wird. Wenn die Hashtabelle in den Shared-Memory angelegt wird, dann entfällt diese Allokation. Die Buffer-Größe ist dann im Kernel-Launch zu definieren. Als letztes Objekt fehlt noch die finale Tabelle aus dem Join. Sie besteht immer aus einem PK und zwei Columns, für die PKs aus R und S. Die Berechnung der Größe erfolgt durch Gleichung 3.11. Jedes Feld in der Tabelle ist ein 64-Bit Integer, weshalb mit Drei multipliziert wird. Die finale Größe ist erst nach dem Start bekannt, weshalb die Berechnung des schlimmsten Fall, dem Kreuzprodukt aus R und S erfolgt. Weitere Buffer sind für die drei Varianten nicht notwendig. Als erstes wird für die Einfachheit, die Global-Hashtabelle betrachtet. Wie es der Name schon sagt, liegt die erstellte Hashtabelle im Global-Memory. Dadurch kann ein gesamter Kernel, an einer Hashtabelle arbeiten und zusätzlich bietet der Global-Memory genügend Platz. Build- und Probing-Phase sind durch zwei Kernel voneinander getrennt. Der Build-Kernel baut die Hashtabelle auf und erwartet deshalb, den Hash-Buffer von R, den Links-Buffer und den Slots-Buffer. Alle Ressourcen liegen im Global-Memory. Der Code des Kernels ist ähnlich zu Algorithmus 12, nur wird der Slots-Buffer außerhalb auf Null gesetzt und die einzelnen Hashes, werden via Striding-Loop in die Hashtabelle eingefügt. Die fertige Hashtabelle dient im

nächsten Schritt, als Basis für den Probing-Kernel. Dieser verwendet neben der Hashtabelle, die jeweiligen Hash- und Tabellen-Buffer von R und S. Außerdem ist für das Ergebnis, der Join-Results-Buffer notwendig. Das Probing läuft wie im Algorithmus 11 ab, nur wird auch hier, über die die Hashes von S, via Striding-Loop iteriert. Die Join-Zwischenergebnisse sind im Join-Results-Buffer vorhanden, wobei die Offset-Berechnung für jedes neue Element, durch ein *atomicAdd(. . .)* stattfindet. Bei den Shared-Memory-Varianten, sind Build- und Probing-Phase in einem Kernel vereint. Da die Hashtabelle im Shared-Memory liegt und nicht von anderen Blöcken im Kernel erreichbar ist, muss jeder Block, eine eigene Hashtabelle aufbauen. In diesem Prozess unterscheiden sich beide Varianten. Zum einen kann sich jeder Block einen vollständige Hashtabelle von R aufbauen oder jeder Block bearbeitet immer nur einen Teil von R. Erstes führt dazu, dass die Größe von R, durch den maximalen Shared-Memory begrenzt wird. Jedoch teilen sich alle Blöcke, den Datensatz S auf. Teilen sich alle Blöcke den Datensatz R, so muss ein Block den gesamten Datensatz S testen, was durch die maximale Anzahl an Threads pro Block, zu einem weiteren Engpass führen kann. Letzteres eignet sich daher nicht für Datensätze, wo S einen deutlich größere Anzahl an Elementen, als R besitzt. Das Arbeiten mit dem Shared-Memory, stellt in diesem Fall keinen Unterschied zu einem Global-Memory-Buffer dar. In der Regel sollte der Shared-Memory nur in einem bestimmten Pattern aufgerufen werden, damit keine Bankkonflikte entstehen. Allerdings ist die Sortierung der Hashes nicht bekannt, weshalb auf das Pattern kein Einfluss genommen werden kann. Da für den Kernel-Start immer der Shared-Memory angegeben werden muss und der für alle Blöcke gleich ist, muss immer die größte Tabelle als Referenz genommen werden. In der ersten Variante (Shared-R), ist die Tabellen-Größe für alle Gleich. Die zweite Variante (Shared-S), hat im Zweifel unterschiedliche Tabellengrößen, jedoch ist die letzte Tabelle immer die kleinste. Für die Aufteilung, wird $|R|$ durch die Anzahl der Blöcke geteilt (Gleichung 3.12). Der letzte Block erhält somit die restlichen Elemente. Innerhalb des Kernels, findet die eigentliche Aufteilung statt. In jedem Block ist der erste Thread, für die Offset-Bestimmung der Daten zuständig.

$$\text{Shared - } R_{\text{Elements}}(R, \text{Blocks}) = \lceil \frac{|R|}{\text{Blocks}} \rceil \quad (3.12)$$

Der Aufbau der Hashtabelle lässt sich recht gut für die GPU implementieren. Beim Probing sieht es ein wenig anders aus. So können zwar alle Element ohne Probleme von jedem Thread, unabhängig voneinander verarbeitet werden, aber für die Warp-Struktur, hat dies negative Auswirkungen. Da Threads in einem Warp im Lock-Step agieren, müssen bestimmte Threads eine unbestimmte Zeit warten. Lock-Step führt alle 32-Threads auf der gleichen Instruktion aus. Threads, welche nicht die gleiche Instruktion erwarten, zum Beispiel durch ein If-Statement, werden für den Zyklus ausgesetzt. Übertragen auf das Probing, entstehen solche Aussetzer durch die unterschiedlichen Kettenlängen in den einzelnen Slots. Im schlimmsten Fall würde daher ein Thread allein im ersten Slot arbeiten und die anderen Threads, würden in den anderen leeren Slots, auf den ersten Thread warten. Zusätzlich muss jeder Thread bei der Überprüfung, auch im Anschluss die echten Werte überprüfen, weshalb Global-Memory-Anfragen abgeschickt werden, welche hohe Latenzen mit sich bringen und den Leerlauf andere Threads verschlimmert. Da der Shared-Memory bereits sehr knapp ist, können die zusätzlichen Spalten nicht in Shared-Memory übernommen werden. Damit das Schreiben der Ergebnisse nicht noch eine zusätzliche Last für den einzelnen Thread darstellt, schreibt jeder Thread nur den Index, des Matching-Pairs in den Result-Buffer. Der Index beschreibt eine eindeutige Position im Kreuzprodukt aus S und R. Gleichung 3.13 zeigt die Indexbildung. X_R und Y_S entsprechen den jeweiligen Hash-Indizes aus den Tabellen R und S. Nach jedem neuen Index, wird der Offset erhöht, wodurch eine dicht gepackte Liste aus Indizes entsteht.

$$\text{Matching - Index}(R, X_R, Y_S) = Y_S \times |R| + X_R \quad (3.13)$$

Im finalen Schritt, findet das Kopieren, der PKs aus R und S, in die Join-Tabelle RS statt. Hierfür wird die Größe aus den Ergebnissen genommen und RS allokiert. Darauf folgt eine Copy-

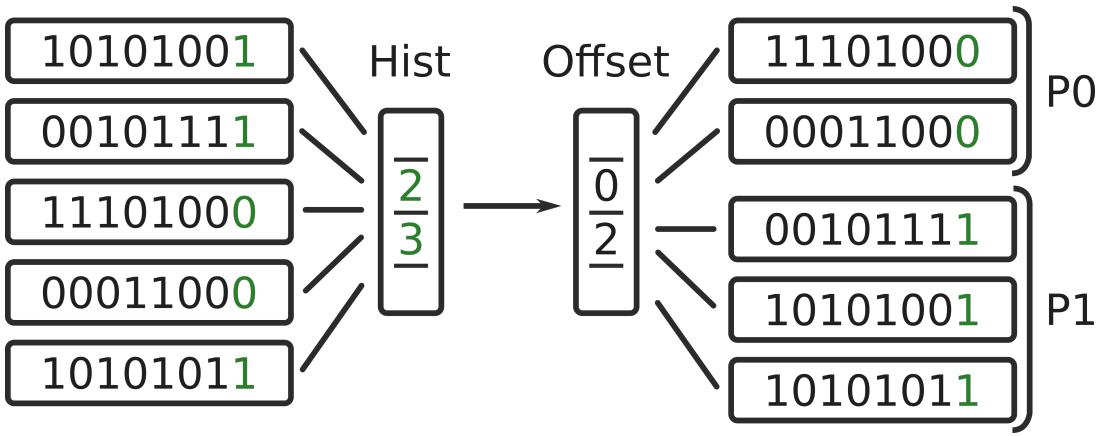


Abbildung 3.4: Radix Partitionierung

Kernel, der die Elemente aus dem Result-Buffer nimmt, die Indizes interpretiert und anschließend die PKs aus R und S, in die einzelnen Spalten einfügt. Die PK-Bildung für die RS-Tabelle, kann direkt im Kernel erfolgen. Alle Kernels in diesem Abschnitt, können asynchron, im gleichen Stream ablaufen. Für die Konfiguration der Threads und Blöcke, wird im Ersten Schritt immer durch die Anzahl der Elemente geteilt, damit die Kernel mit steigender Element-Anzahl skalieren. Die Kernel-Konfigurationen sind flexibel, über die Anzahl der Threads und wie viele Elemente ein Thread bearbeiten soll steuerbar.

3.2.8 Vektorisierung

Der Join ist bereits vollständig, jedoch schränken die einzelnen Buffer-Größen, die maximalen Tabellen-Größen deutlich ein. In der Regel wir der meiste Speicher vom Results-Buffer verbraucht. Daher sollte er den kleinst möglichen Integer-Datentyp bekommen. Um dennoch größere Tabellen zu verarbeiten, ist die Vektorisierung der Tabellen notwendig. In der Vektorisierung, findet die Reduzierung, der Tabellen, auf eine Teilmenge statt. Diese Teilmenge besteht aus R- und S-Zeilen. Auch wenn es verlockend klingt, alle Daten in kleine Abschnitte zu unterteilen, muss gewährleistet sein, dass das gleiche Join-Ergebnis zu erwarten ist. Hierfür muss jeder Hash aus R, auch jeden Hash mit dem gleichen Wert in S überprüfen. Für ein kleines Beispiel, wird die Vektorgroße auf maximale zwei Elemente begrenzt. Bei N Elementen in R und M Elementen in S, entstehen insgesamt $N \times M$ Paarungen. Es sind nun pro Paar weniger Iterationen notwendig, allerdings gibt es deutlich mehr Paarungen oder Join-Aufrufe. Mit Blick auf den Results-Buffer gibt es somit eine Chance, mehr Speicher für andere oder parallele Aufgaben zu schaffen. Abhilfe schafft das Radix-Partitioning und ist ein in der Literatur gern verwendetes Verfahren, um auf Makro-Ebene, den Datensatz in kleinere Partitionen aufzuteilen [32, 34, 35]. Durch den Algorithmus werden die Hashes aus der jeweiligen Tabelle, wie in Abbildung 3.4 neu angeordnet. Als Unterscheidungsmerkmal dient hierfür der Radix. Dieser repräsentiert eine endliche Anzahl an Bits (Grün), wodurch die Zugehörigkeit der Hashes zu den einzelnen Partitionen festgestellt wird. Im ersten Schritt wird die Anzahl der Partitionen festgelegt. Die Anzahl muss immer ein Zweier-Potenz sein. Dadurch ergibt sich die breite des Radix. Bei 2^N Partitionen, sind daher N Bits notwendig. In Abbildung 3.4 beginnt der Radix am Least-Significant-Bit (LSB). Gleiches gilt auch in der Implementierung in dieser Arbeit. Als nächstes wird das Histogramm der Radix-Werte berechnet. Dafür erfolgt eine Logische-Und-Verknüpfung mit der Radix-Maske. Das Histogramm gibt an, wie viele Elemente in jede Partition erwartet werden. Darauf folgt die Übertragung der Elemente in die Partitionen. Durch das Histogramm ist der Offset in jeder Partition bekannt, wodurch ohne Problem die Elemente zufällig in die neuen Listen geschrieben werden können. Die Radix-Partitionierung erfolgt nie in einer und der selben

Algorithm 13 Rekursives Radix-Partitioning

Require: $S, R, Hashes_S, Hashes_R, \text{RadixWidth}, \text{Bucket}, \text{MaxVectorSize}, \text{Depth}$

```
1: Increase Depth
2: if Size of S and R is still greater than MaxVectorSize then
3:   Calculate histogram of R
4:   Calculate offset of R
5:   Swap R
6:   Calculate histogram of S
7:   Calculate offset of S
8:   Swap S
9:   for each bin in histogram do
10:     Create sub bucket
11:     Set sub bucket size and data offset
12:     Add sub bucket to Bucket
13:     Start radix partitioning with sub bucket config and swapped buffers
14:   end for
15: end if
16: if Bucket has no sub buckets and is not empty then
17:   Vectorize bucket and add to final bucket list
18: end if
```

Liste. Für den Tausch der Elemente, ist immer ein weitere Buffer mit der gleichen Größe notwendig. Beim Join reicht der Hash allein nicht aus. Es fehlen noch die Tabellen-Zeilen. Dafür kann der Histogramm-Schritt übersprungen und die Offsets für die Hashes wiederverwendet werden. Da eine zweite Tabellen-Kopie genau den gleichen Speicher, wie die ursprüngliche Tabelle einnimmt, ist die Reduzierung auf Indizes, welche vertauscht werden eine Option. Das erfordert dass die Tabellen und Hashes noch im gleichen Speicher liegen und nicht zum Beispiel auf einer anderen GPU, was im nächsten Abschnitt der Fall sein kann. Abbildung 3.4 zeigt nur den ersten Schritt. In der Regel sind die einzelnen Partitionen noch zu groß für den Join. Aus diesem Grund werden weitere Iterationen, über die einzelnen Partitionen durchgeführt. Mit jedem Iterationsschritt verschiebt sich der Radix, um die Bit-Breite zum Most-Significant-Bit (MSB). Dadurch entstehen neue Verteilungen in der Partition, welche in neue Unter-Partitionen sortiert werden. Dies geschieht so lange, bis die minimale Partitionsgröße oder der maximale Shift erreicht ist. Die maximale Partitionsgröße wird durch die Vektorgröße festgelegt. Der Algorithmus 13 zeigt die Implementierung des rekursiven Radix-Partitionings, in dieser Arbeit. Im Gegensatz zu den anderen Algorithmen, ist dieser etwas größer formuliert, da der Algorithmus sonst von zu vielen C++ Konstrukten überhäuft wäre. Gestartet wird immer mit der Tiefe Null. In dieser besitzen alle Buffer noch ihre ursprüngliche Ordnung und sind dem Root-Bucket zugeordnet. Die *RadixWidth* ergibt sich aus der Bit-Anzahl vom Radix. Die erste Zeile inkrementiert die Tiefe, damit die maximale Tiefe überprüfbar ist. Wie vorher beschrieben, ist eine erneute Partitionierung möglich, wenn die Größen der Buffer zu groß sind oder die maximale Tiefe erreicht ist. Die Größe der Buffer wird durch die potentielle Hashtabellengröße von R und der Tabellen- und Hash-Buffer-Größe von S bestimmt. Sollten beide Kriterien noch nicht erfüllt sein, so werden drei Kernel für die Partitionierung ausgeführt. Jeweils für R und S. Die resultierenden Offsets sind relativ von der aktuellen Buffer-Adresse und dienen für die Zuweisung, der Partitionen in den neuen Sub-Buckets, im nächsten Schritt. Buckets sind am Ende nichts anderes als Partitionen, jedoch wird ersteres häufig in der englischen Literatur gefunden. Darauf folgt ein neuer Aufruf der Partitions-Funktion, nur dass jetzt der Sub-Bucket als Root-Bucket dient. Die Buffer werden alternierend mit ihrem Gegenstück ausgetauscht, so dass das Radix-Partitioning immer auf den aktuellen Partitionen arbeitet. Zusammenge-

Algorithm 14 Bucket-Vectorization

Require: $Buckets_S$, $Bucket_R$, MaxVectorSize

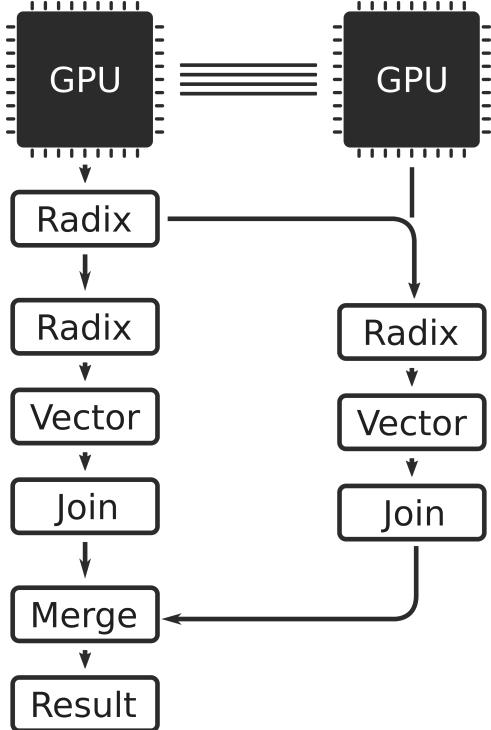
```
1: Get total size of R dataset
2: Set R as item of sub chunks
3: if R size is larger than half of the vector size then
4:   Split R into equally sized chunks, fitting in the vector
5:   Remove R from sub chunks and replace it by new sub chunks
6: end if
7: Calculate maximum bytes for S from largest sub bucket of R
8: Creates sub buckets from S according to byte limitations
9: for each sub bucket from R do
10:   for each sub bucket from S do
11:     Create Pair of both sub buckets
12:   end for
13: end for
return Sub bucket pairs
```

fasst teilt der Algorithmus eine Partition des Datensatzes so lange auf, bis sie klein genug ist oder nicht mehr weiter partitioniert werden kann. Ist eins von beiden Kriterien erfüllt, wird die nächste Partition Stück für Stück reduziert. Wenn die Partitionen die gewünschte Größe oder Tiefe erreicht haben, können sie der zu verarbeitenden Join-Liste hinzugefügt werden. Das Resultat ist immer eine Paarung aus S und R Daten. Wie bereits angedeutet, kann es passieren, dass ein Datensatz nicht weiter reduzierbar ist. Das ist der Fall, wenn zum Beispiel alle Elemente den gleichen Hash und somit den gleichen Radix-Key besitzen. In diesem Fall ist die gewünschte Vektorgröße nicht erreicht und der Datensatz muss noch zusätzlich vektorisiert werden. Dieser Schritt wird in dieser Arbeit Bucket-Vectorization genannt und ist in Algorithmus 14 zu finden. Damit die bestehenden Buckets aufgeteilt werden können, ist die Größe des $Buckets_R$ entscheidend. In jedem Vektor darf der R-Datensatz nicht mehr als einen bestimmten Anteil übernehmen. Sollte $Buckets_S$ nur einen sehr kleinen Anteil bekommen, führt das zu sehr vielen kleinen Buckets, welche es zu vermeiden gilt. Daher wird $Bucket_R$ in gleich große Teil-Buckets aufgeteilt. Anhand der Bucket-Größe von R, sind die restlichen Bytes, von S zu füllen. Als Endergebnis findet die Paarung der einzelnen Sub-Buckets statt. Hierbei werden insgesamt $|Sub-Bucket_R| \times |Sub-Buckets_S|$ Paare erzeugt. Wenn R und S direkt in den Vektor passen, dann wird auch nur ein Paar erzeugt. Zum Schluss erfolgt durch den Merge-Prozess, die Zusammenführung aller Join-Teilergebnisse in einen Buffer. Hier gilt es zu beachten, dass für den finalen Join-Buffer, auch ein extra Buffer angelegt wird. Dieser hat die gleiche Größe, wie die Teilergebnisse zusammen, was eine weitere Limitierung in Tabellengrößen schafft. Die Implementierung beachtet diese Einschränkung erst kurz vor dem Merge und bricht diesen gegeben falls ab. Die PK-Berechnung im Probing muss nun im Merge-Prozess stattfinden, da im Probing-Prozess nur die Teilgröße des erzeugten RS-Ergebnis bekannt ist.

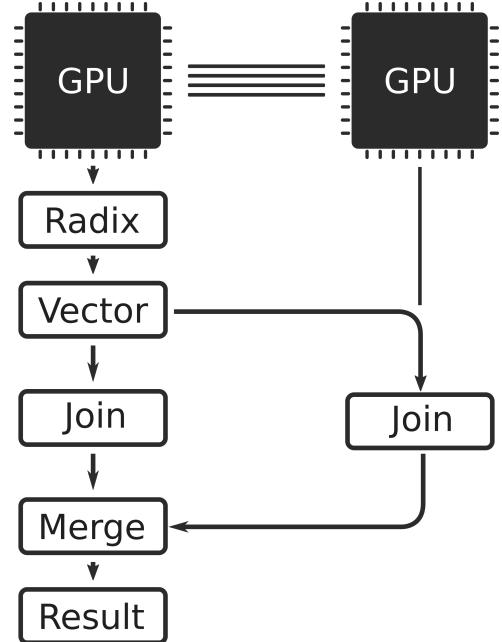
Streaming

Der gesamte Partitionierungs-Prozess läuft noch sequentiell ab. Bei genauer Betrachtung der einzelnen Partitionen und den Buffer-Offsets. Wird eine Unabhängigkeit zwischen den einzelnen Partitionen sichtbar. Die Größe der Partition wird nie größer als die in der Hierarchie darüber liegende Partition, weshalb benachbarte Partitionen ohne Probleme parallel verarbeitet werden können. Um dieses Ziel zu erreichen, startet jeder Radix-Aufruf in einem neuen Thread, wodurch die anderen Buckets nicht geblockt sind.

Das löst die Wartezeiten im CPU-Code, jedoch werden alle Kernel noch auf dem selben Stream ausgeführt und blockieren sich dadurch gegenseitig. Das Problem wird durch eine Liste von



(a) Radix Split



(b) Vector Split

Abbildung 3.5: Hash-Join auf Multi-GPU-Systemen

Streams behoben. Aus dieser Liste bedient sich jeder Thread, bevor er eine Sequenz von Kernels startet. Die Streams werden im Round-Robin-Verfahren verteilt, wodurch jeder Thread in etwa die gleiche Chance hat, einen Platz auf der GPU zu bekommen. Die einzelnen Streams werden nur am Ende der Radix-Operation synchronisiert, damit die Histogramm- und Offset-Information, zum Zeitpunkt der For-Schleife bekannt ist.

Radix Kernel

Die einzelnen Kernels, welche auf den Streams laufen, wurden bisher nur als gegeben behandelt. Insgesamt handelt sich um die drei Funktionen Histogramm-, Offset- oder Präfix-Sum- und Swap-Kernel. Im Histogramm-Kernel erfolgt die Zählung aller Hashes, welche in eine bestimmte Partition passen. Der Zugriff erfolgt durch eine Striding-Loop und die einzelnen Bins werden durch die Verknüpfung mit der Radix-Mask berechnet. Der Berechnete Wert dient als Bin-Index und inkrementiert den Wert, mit einem Atomic-Add, des Bins um Eins. Damit ein Bin entlastet wird, wenn viele Threads auf genau einen zugreifen wollen, wird das Histogramm für jeden Block, in den Shared-Memory verlegt. In diesem berechnen alle Blöcke ein temporäres Histogramm und der erste Thread in jedem Block, addiert die finalen Werte in den Global-Memory. Das berechnete Histogramm dient im Offset-Kernel als Grundlage für die Präfix-Sum [37]. Im letzten Schritt, folgt der Austausch in die Partitionen. In diesem Kernel werden Tabellen- und Hash-Daten, vom Swap und Original benötigt. Erneut findet die Berechnung des Radix-Keys über die Maske statt. Im Anschluss erfolgt die Übertragung von Hash und Tabellen-Zeile, in das Swap-Gegenstück. Damit die Sub-Buckets für die neuen Partitionen erstellt werden können, erfolgt die Kopie, des Histogramms zum Host.

3.2.9 Multi GPU

Wie beim Hashing, gibt es auch für den gesamten Join-Prozess, eine Multi-GPU Implementierung. Für einzelnen Joins, ist die Aufteilung zwischen GPUs nicht möglich, da immer nur einer der Datensätze geteilt werden kann und nicht beide zusammen. Würden Join-Operationen in der Mitte geteilt werden, so findet kein Vergleich zwischen den Werten, aus beiden Gruppen statt und das Resultat wäre unvollständig. Daher fällt die Aufteilung in den Vektorisierungs-Prozess. Eine GPU kann hier an zwei Stellen einen Teildatensatz zugeordnet bekommen, ohne dass das Endergebnis davon beeinflusst wird. Die Erste ist das Radix-Partitioning, wo einzelne Sub-Trees auf eine andere GPU verlagert und weiter verarbeitet werden können. Zu sehen ist der Ablauf in Abbildung 3.5a und wird in dieser Arbeit als Radix-Split bezeichnet. Neben der Reduzierung der Vektorgröße, ist die Verteilung auf verschiedene Prozessoren, auch ein Kern-Argument für das Radix-Partitioning, was in der Literatur auch als Radix-Hash-Join bezeichnet wird [4, 38]. Das Ziel soll es sein, mindestens eine weitere GPU für die Aufteilung der Daten in weitere Partitionen zu verwenden und mit dieser auch das Probing für den Datensatz durchzuführen. In der Implementierung ändert sich nicht viel. Die weitere GPU wird für die Übertragung zwischen beiden GPUs vorbereitet, wie es für CUDA, wenn kein UVM verwendet wird, üblich ist. Nach diesem Schritt können die Partitionen über NVLink übertragen werden. Die Aufteilung findet nach der Ersten Partitionierung statt. Ein Teil der Partitionen wird von GPU 0 weiter verarbeitet und der andere Teil von GPU 1, welcher explizit kopiert wird, was bei GPU 0 nicht der Fall ist, da hier nur Offsets angepasst werden. Danach läuft die rekursive Aufteilung wie bereits erläutert ab. Hier gilt es zu beachten, dass für jede Partitionierung, auch die richtige GPU aktiviert ist. Am Schluss liegen auf beiden GPUs, eine Anzahl an Vektoren vor, welche sich zwischen beiden GPUs unterscheiden. Das Ungleichgewicht ist eine Folge aus der Verteilung der Daten. Liegen alle Daten konzentriert bei einem Wert, so fallen die Partitionen unterschiedlich groß aus, was eine ungleiche Auslastung zur Folge hat. Das Ungleichgewicht wird im weiteren Verlauf keine weitere Rolle spielen und als gegebener Nachteil betrachtet. Nach der Aufteilung folgt das vorgestellte Probing. Hierfür wird wieder für jeden Vektor ein Thread gestartet. Die Abläufe unterliegen der jeweiligen GPU und deren Einschränkungen. Jeder einzelne Prozess, welcher einen Vektor verarbeitet, setzt zu Beginn den GPU-Kontext. Diesen Kontext erhält jeder Vektor bei der Aufteilung der Daten, von der jeweiligen GPU. Nachdem für alle Vektoren eine RS-Tabelle entstanden ist, folgt das Zusammenführen der Tabellen. Der Prozess ähnelt dem vorgestellten Merge, nur erfolgt zusätzlich das Kopieren der Daten von den anderen GPUs auf die ausgehende GPU. Eine weitere Option ist der Vector-Split (Abbildung 3.5b). Beim Vector-Split wird erst nach der Partitionierung und der Vektorbildung, die Vektoren zwischen den GPUs aufgeteilt. Dadurch erhalten alle GPUs eine ausgeglichene Anzahl an Vektoren. Die Zuteilung erfolgt abwechselnd an die jeweilige GPU, welche durch den Rest mit der Modulo-Operation, über den Vektor-Index zu bestimmen ist. Wenn zusätzlich die Liste mit den Vektoren nach der Größe sortiert wird, profitiert nicht nur die Allokation der Probe-Buffer. Dadurch erhält jede GPU die gleiche Menge an Vektoren mit ähnlicher Größe, wodurch das Verfahren als ausgeglichen betrachtet wird. Das verarbeiten der einzelnen Joins und dem finalen Merge, erfolgt identisch zum Radix-Split. In der Theorie schließen sich beide Verfahren nicht aus. So könnte zuerst ein Radix-Split erfolgen und anschließend durch den Vector-Split das Ungleichgewicht wieder angepasst werden. Diese Variante wird allerdings nicht weiter betrachtet und dient eher der Veranschaulichung, wie beide Verfahren in dieser Arbeit implementiert wurden.

3.2.10 Parameter

Der gesamte Hash-Join lässt sich an unzähligen Stellen anpassen. Jeder Kernel bietet die Möglichkeit, dass die Anzahl der Elemente pro Thread und die Anzahl der Threads pro Block sich verändern lassen. Des weiteren ist die Anzahl der Streams flexibel. Die Stream-Anzahl beschreibt wie viele Radix-Partitionierungen, Probing und wie viele Merges, parallel ablaufen sollen. Natürlich ist dieser Parameter von der Hardware beschränkt und die gewünschte Anzahl ist nicht immer von der Hardware erreichbar, wenn einzelnen Kernel zu viele Ressourcen belegen. Für das Probing lassen sich die Modi einstellen. In der Arbeit wurden drei Varianten vorgestellt, zwischen denen vor Beginn entschieden wird. Durch die Veränderung der Probe-Funktion, ändert sich auch die maximale Vektorgröße. Wird ein Verfahren verwendet, welches durch den Shared-Memory begrenzt wird, so erfolgt auch die Limitierung, des Vektors im Algorithmus automatisch. Dennoch lässt sich die Vektorgröße von außen konfigurieren, jedoch mit den genannten Einschränkungen. Als Letztes gibt es noch die Anzahl der Buckets. Diese muss immer aus den genannten Gründen, eine zweier Potenz abbilden und kann beliebig groß gewählt werden. Entscheidend ist diese Größe, wenn mehrere GPUs ihre Anwendung finden sollen.

3.2.11 Zusammenfassung

In diesen umfassenden Abschnitt, wurde ein Hash-Join für In-Memory GPU-Systeme vorgestellt. Für die Erläuterung wurde insbesondere auf Implementierungs-Details eingegangen, welche eine Rolle für die folgende Evaluation spielen. Da der Hash-Join nicht nur aus einer einfachen Operation besteht, wurden alle wichtigen Teilschritte betrachtet. Der erste Schritt, erfolgt durch das Hashing, wobei die Hash-Algorithmen im vorherigen Abschnitt genauer betrachtet wurden. Nachdem die Hashes vorhanden sind, folgt das Probing. Für das Probing gibt es in dieser Arbeit drei Varianten, welche eine Hashtabelle bilden und sich in der Speicherung der Hashtabelle unterscheiden. Hierbei wurde zwischen Global- und Shared-Memory unterschieden und zusätzlich eine Aufteilung im Shared-Memory vorgestellt. Da alle Verfahren bestimmte Limitierungen, was den Speicher betrifft mit sich bringen, wird vor dem Probing eine Partitionierung eingeschoben. Diese erfolgt durch das Radix-Partitioning. Aus der Partitionierung erfolgen Vektoren, welche vorher durch die Speicher-Limitierungen oder dem Anwender, in ihrer Größe limitiert sind. Jeder Vektor wird anschließend in den Probing-Algorithmus übergeben und gibt eine RS-Tabelle, mit allen Paarungen zurück. Jede einzelne RS-Tabelle wird erfasst und im Merge-Prozess zu einer großen RS-Tabelle zusammengefasst. Am Ende des Hash-Joins wird eine Tabelle mit den jeweiligen Primary-Key-Paaren ausgegeben. Der gesamte Prozess unterstützt Streaming, was beim Radix-Partitioning, Probing und dem Merge-Prozess seine Anwendung findet. Steht mehr als eine GPU zur Verfügung, können das Radix-Partitioning und das Probing auf den weiteren GPUs ausgeführt werden. Hierbei werden die Daten bereits beim Radix-Partitioning (Radix-Split) oder nach der Vektorbildung (Vector-Split), auf die GPUs übertragen.

4 Evaluation

Im letzten Kapitel wurden alle Implementierungen, in dieser Arbeit vorgestellt und Thesen für die Ausführung aufgestellt. Dieses Kapitel greift die Implementierungen auf und evaluier alle Algorithmen und deren Parameter. Als Hauptmerkmal dient hierbei die Geschwindigkeit. Anhand der Geschwindigkeit wird der Einfluss der jeweiligen Parameter evaluiert. Zusätzlich sind alle Funktionen als Kernels präsent, welche mit dem Nvidia-Profiler genauer betrachtet werden. Am Ende findet eine Evaluation aller Funktionen zusammen im Hash-Join-Algorithmus statt. Aus diesem Grund folgen zuerst kleinere Teifunktionen, worunter auch das Hashing fällt. Das Hashing hat in dieser Arbeit einen besonderen Platz bekommen, da nicht nur Geschwindigkeit, sondern auch Eigenschaften von Interesse sind. Die einzelnen Eigenschaften wurden bisher nur theoretisch betrachtet und in diesem Kapitel quantitativ untersucht.

4.1 Aufbau

Für das Testsystem steht ein Aufbau, wie in Abbildung 2.10 zur Verfügung. Beide GPUs werden durch jeweils eine Quadro 8000 RTX [26] repräsentiert und sind via NVLink miteinander verbunden. Die Datenrate ist bei einer Konfiguration von zwei Quadros 8000 RTX, auf 100GB/s limitiert. Für die Steuerung der Abläufe, steht ein Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz zur Verfügung. Auf diesem System werden alle Laufzeitanalysen durchgeführt. Für genauere Analysen der einzelnen Kernels mit dem Nvidia-Profiler, wird eine GTX 1650 [39] verwendet. Beide Grafikkarten unterstützen die gleiche Cuda-Architektur und können daher verglichen werden. Mit den Quadro-GPUs stehen jeweils bis zu 50GB an Global-Memory zur Verfügung, weshalb nur auf diesem System größere Datensätze testbar sind. Die interne Speicher-Bandbreite liegt bei etwa 600GB/s, weshalb bei den späteren Betrachtung, diese Limitierung zu beachten ist.

4.2 Hashing

Im Vergleich zu den anderen vorgestellten Kernels, sind die Hashing-Kernels gesondert zu betrachten. In diesen werden nicht nur vordefinierte Prozesse gearbeitet, sondern auch verschiedene und neue Verfahren eingebunden, und deren Leistung und Eigenschaften auf einem Multi-GPU-System getestet. Im ersten Schritt ist es daher wichtig, sich mit den Eigenschaften der Hash-Funktionen vertraut zu machen. Im Gegensatz zu einem Join, ist hier immer ein anderes Ergebnis zu erwarten, weshalb die Hashes auf die einzelnen Eigenschaften, wie sie bereits genannt wurden zu überprüfen sind. Ein Hash mit sehr schlechten Eigenschaften

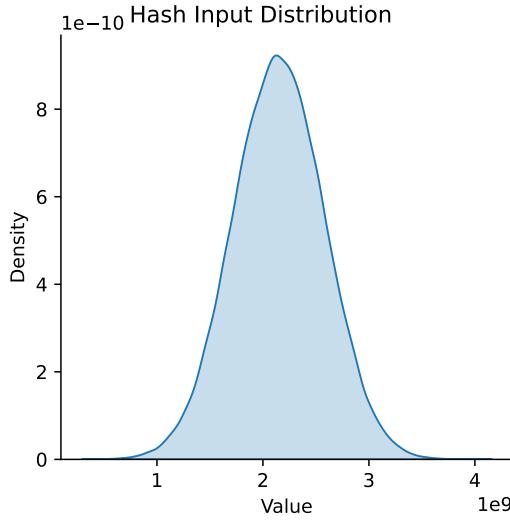


Abbildung 4.1: Verteilung der Testwerte für die Eigenschaftstests

hat unter Umständen einen negativen Einfluss auf die abhängigen Operationen, weshalb diese vor der Anwendung bekannt sein müssen. Im Anschluss erfolgt die Analyse von einzelnen Benchmarks, aller Parameter, welche für den Kernel entscheidend sind. Diese setzen sich nicht nur aus reinen Kernel-Parametern zusammen, sondern werden auch von der Größe einzelner Datentypen bestimmt. In der Implementierung werden die einzelnen Hash-Funktionen, von einem Wrapper versteckt, weshalb die Funktionen auch als ein Parameter zu betrachten sind. Mit der Übersicht lassen sich die besten Parameter für die Kernels ableiten, weshalb im folgenden Schritt eine Konfiguration auf das Multi-GPU-Setup übertragen und analysiert wird. Für die Laufzeit spielt immer die Metrik Hash/Sekunde die Hauptrolle. Zusätzlich ist der Durchsatz in GB/Sekunde relevant, um bei unterschiedlichen Datentypen die Leistung zu vergleichen. Weil die Datenrate limitiert ist, ist davon auszugehen, dass bei einer bestimmten Eingabegröße auch der Durchsatz in Hash/Sekunde stagniert.

4.2.1 Eigenschaften

Die immer wieder angesprochen Eigenschaften sind die Verteilung, Abstand zwischen benachbarten Werten und die Kollision. Jede Eigenschaft lässt sich mit einem unterschiedlichen Datensatz sinnvoll testen. Für die Kollision und der Verteilung lassen sich nur Näherungswerte bilden. Für das Testen von 32- oder 64-Bit Werten, muss jeder Zahlenwert in diesem Spektrum abgebildet sein, um eine umfassende Analyse durchführen zu können. Die Generierung ist praktisch noch für 32-Bit-Werten möglich, allerdings wird bei 64-Bit sehr schnell der Speicher knapp. Aus diesem Grund findet in dieser Arbeit nur die Analyse auf einer ausgewählten Verteilung, mit einer begrenzten Datenmenge statt. Am einfachsten wäre es, alle Datenpunkte von Null bis N zu verwenden. Jedoch fällt dann ein signifikanter Teil der Werte aus dem Test heraus. Die Wahl ist deshalb auf eine Normalverteilung gefallen. In dieser wird ein enger Zahlenbereich abgebildet, aber auch Werte aus allen Bereichen im Zahlenraum verwendet. Der Erwartungswert liegt bei 0.5, die Standardabweichung bei 0.1 und die resultierende Werte sind auf den Zahlenbereich von 2^{32} skaliert. Dadurch ergibt sich eine Eingabe von jeweils 32-Bit Integer-Zahlen, ohne Vorzeichen.

Die finale Verteilung besteht aus 100.000 Elementen. Diese Zahl ist in erster Linie willkürlich gewählt, allerdings sind die dadurch entstandenen Ergebnisse nicht mehr von Ausreißern dominiert und ermöglichen somit eine sinnvolle Entscheidung, für die Verwendbarkeit zu treffen.

Wie die jeweilige Verteilung aussieht, ist in Abbildung 4.1 zu sehen. Im weiteren Verlauf wird nicht nur ein Wert aus der Verteilung für jede Eingabe genommen, sondern es werden auch Eingaben mit mehreren Teilwerten entnommen. Diese Teilwerte sind jeweils frei aus der Verteilung entnommen und haben keine Korrelation. Egal ob 32-Bit oder 64-Bit Hash, die Verteilung ist immer gleich und liegt im 32-Bit Bereich. Für den weiteren Verlauf spielen eigentlich nur 64-Bit Hashes eine Rolle, jedoch gibt die Abbildung auf einen 32-Bit Hash einen guten Überblick, wie sich die unterschiedlichen Zahlenräume auf das Ergebnis auswirken. Beim Testen jeder Funktion, gibt es jeweils die Eingaben mit 1,2 und 4 Elementen, welche in den Abbildung entsprechend gekennzeichnet sind. Jeder 32-Bit entspricht einem Chunk, weshalb auch dieser Name für die Legende getroffen wurde.

4.2.2 Verteilung

Beginnend mit dem 32-Bit Hash, sind alle Funktionen in Abbildung 4.2, zu sehen. Da jeder Operations-Typ genau drei Variationen besitzt, ist jede Zeile eine Operation. Ganz zum Schluss folgt der FNV-Algorithmus als Referenz. Jedes Diagramm zeigt die Verteilung der Hashes, im Zahlnbereich von 0 bis 2^{32} an. Gute Hash-Funktionen verteilen die Werte gleichmäßig über den Wertebereich und sind daher durch einen linearen Verlauf über den gesamten Zahlnbereich zu erkennen. Für die Addition trifft das nur in einem von drei Fällen zu. Die reine und mit einem Shift versehene Addition, spiegeln in etwa die Verteilungsfunktion der Eingabe wieder. Wenn mehr als nur ein Element vorhanden ist, dann verschiebt sich der Wert, um die Hälfte des Wertebereichs. Dies liegt daran, dass die meisten Werte genau bei der Hälfte des maximalen Wertes liegen. Somit landen sie in Kombination, entweder beim Maximum oder durch die Modulo-Operation beim Minimum. Shifts funktionieren beim 32-Bit Hash nicht, da sonst Daten verloren gehen würden, was nicht erwünscht ist. Aus diesem Grund sind die Funktionen mit dem Shift, auch nur Funktionen ohne Shift. Die HW-Variante erreicht durch die Permutation und dem Umdrehen des Hashes, genau das Ergebnis, was gefordert ist. Auch wenn es bei einem Element leichte Schwankungen gibt. Für die Multiplikation wird immer ein weiteres Element benötigt, damit die Ausgabe nicht der Eingabe entspricht, was auch in den einzelnen Graphen zu sehen ist. Ab mehr als einem Element ist das gewünschte Ergebnis sichtbar, allerdings gibt die HW-Variante einen unsauberen Verlauf zurück, was auf eine Schwäche in der Kombination hindeutet. Kombiniert mit dem XOR, bietet die Funktion mit mehr als nur einer Iteration, über die gesamte Eingaben, das beste Ergebnis. Die Iteration wurde auf zwei Runden begrenzt, was die Anforderungen bereits erfüllt. Eine einzelne Iteration liefert das gleiche Ergebnis, wie die XOR-Funktion, weshalb die zusätzliche Multiplikation auch komplett ignoriert werden kann. Wenn der Shift durch die HW-Variante ausgetauscht wird, ähnelt das Ergebnis der reinen Multiplikation mit der HW-Variante, daher könnte auch hier auf das XOR verzichtet werden. Mit Blick auf die XOR-Funktionen, ist der bereits vorher angetroffene Knick zu beobachten. Diese lässt sich durch folgendes Beispiel erklären. Der in den einzelnen Diagrammen zu sehende Zahlenraum, lässt sich perfekt auf die Position der einzelnen Bits und deren Wertigkeit übertragen. Beim höchsten Wert liegt der Most-Significant-Bit und beim niedrigsten Wert der Least-Significant-Bit. Da die ausgehende Verteilung eine Normalverteilung ist, liegen die meisten Werte auch in der Mitte, weshalb bei den meisten Werten, die Bits in der Mitte gleich sind. Für die XOR-Funktion bedeutet dies, dass viele Werte in der Mitte verschwinden, da die Bits in der Mitte häufiger auf Null gesetzt werden. Die Bit-Werte am Rand, sind weniger konzentriert und haben daher auch eine höhere Wahrscheinlichkeit, dass die Einsen erhalten bleiben. Im Vergleich dazu kann die HW-Varianten, diese Eigenschaft überwinden und eine gute Verteilung liefern. Gleiches gilt für die FNV-Funktion.

Wird der Hash auf 64-Bit erhöht, so findet die Modulo-Operation erst viel später statt. Abbildung 4.3 liefert hierfür die Ergebnisse. Für die reine Addition hat das die Konsequenz, dass

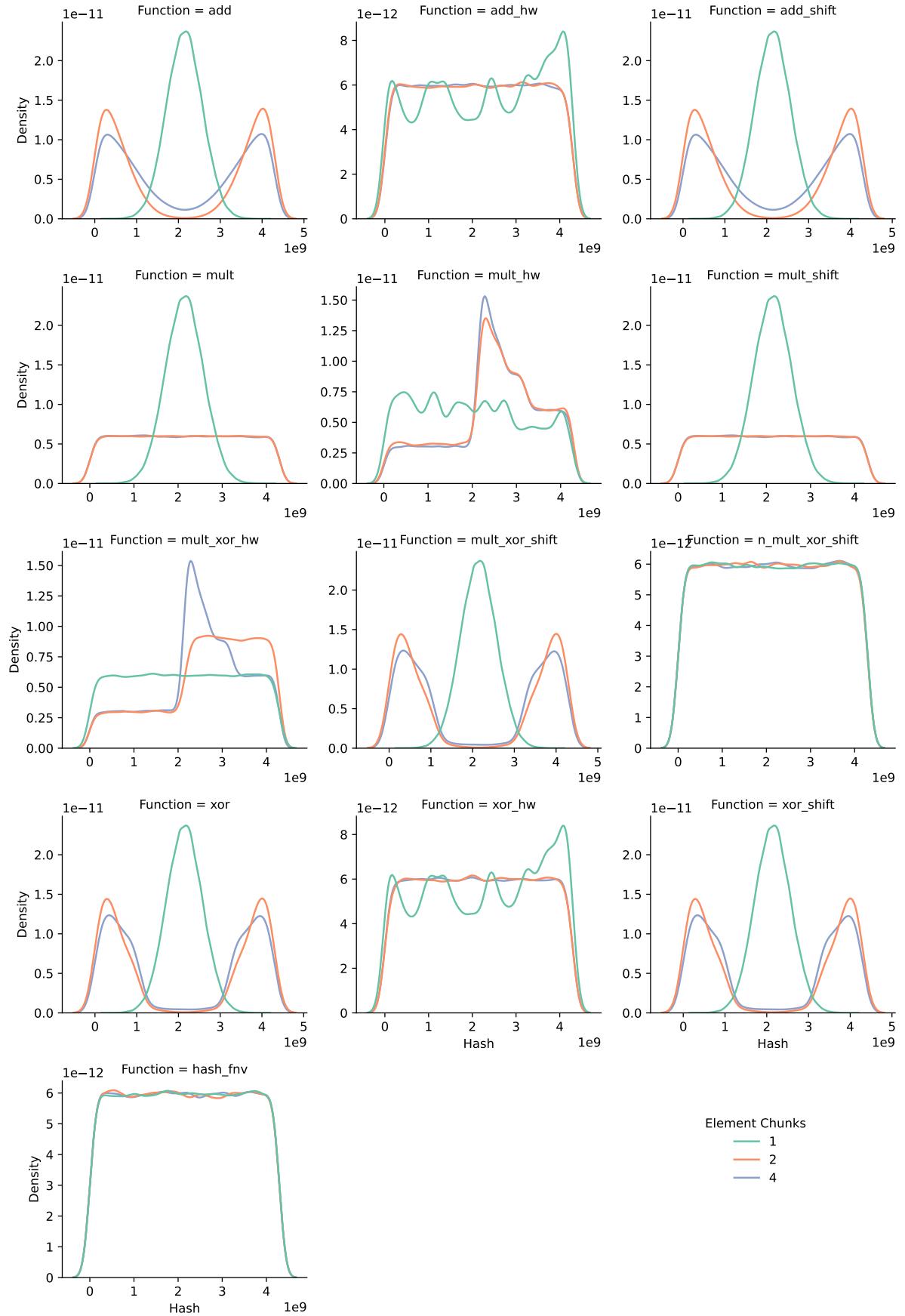


Abbildung 4.2: Hash-Verteilung 32-Bit

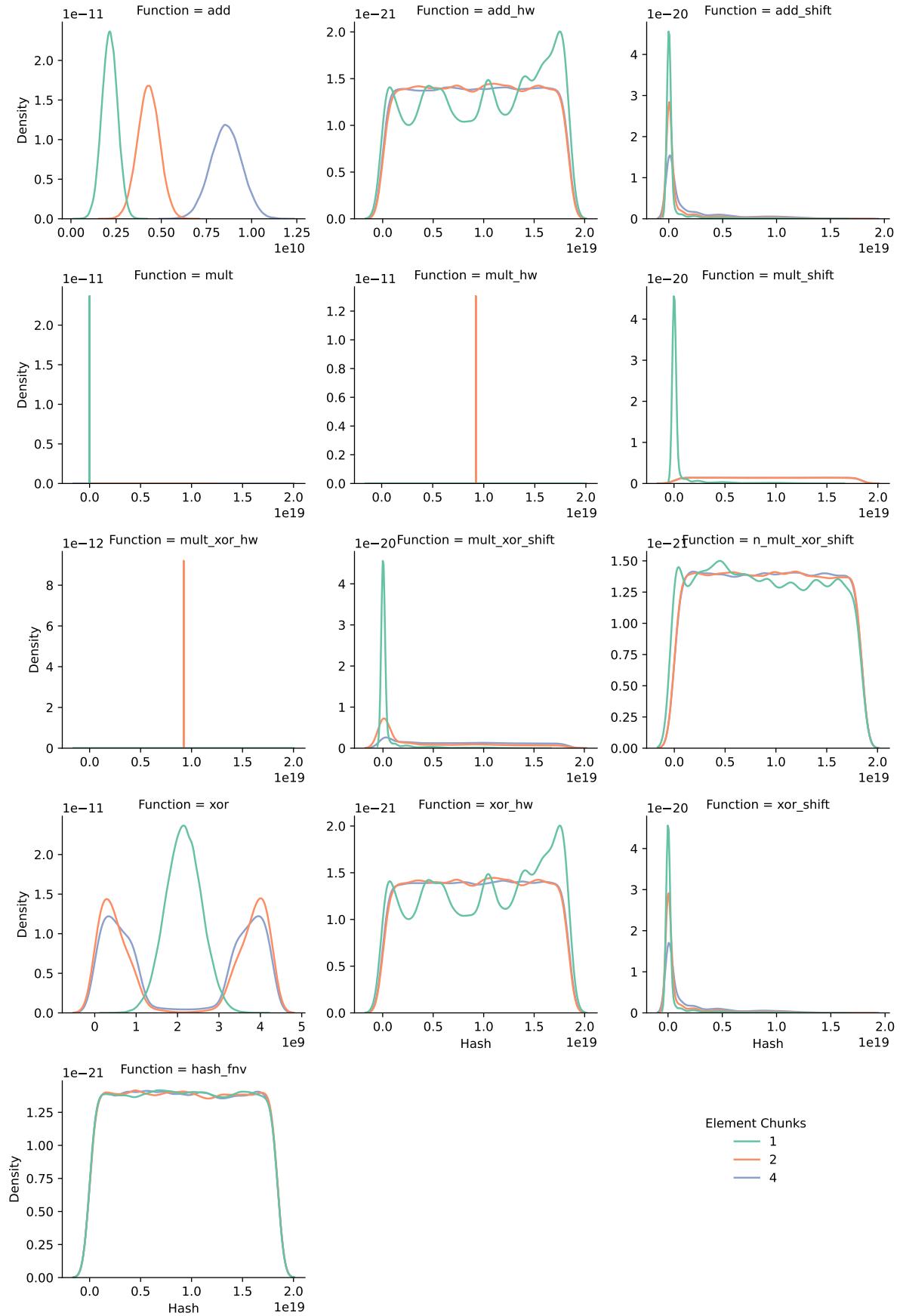


Abbildung 4.3: Hash-Verteilung 64-Bit

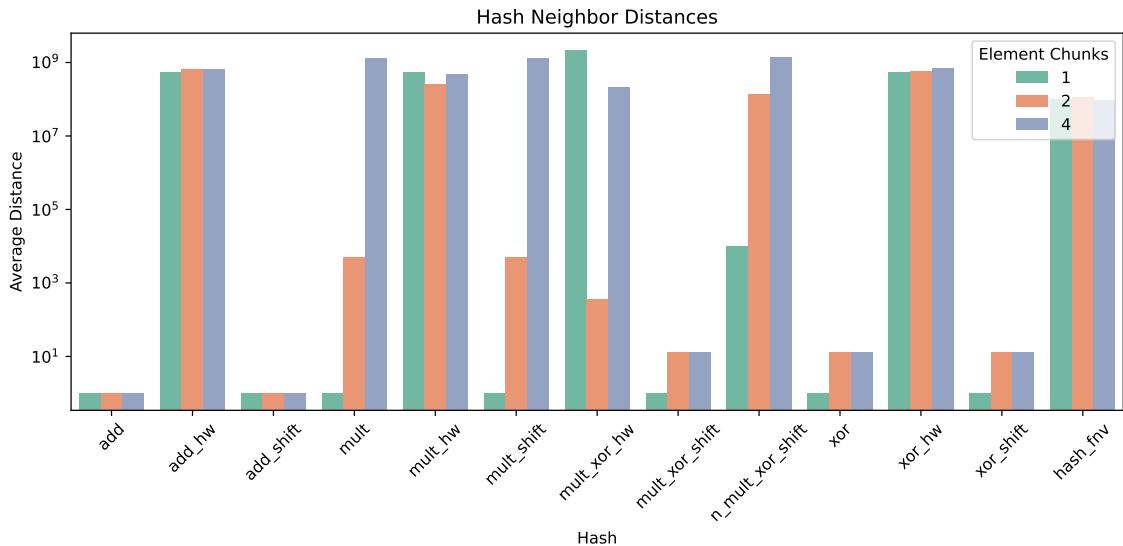


Abbildung 4.4: Hash-Abstand 32-Bit

die generierten Hashes, nicht mehr in den 64-Bit Zahlenraum eindringen. Nur durch den Shift ist es möglich, dass die Hashes sich weiter über den gesamten Zahlenbereich erstrecken. Hier gilt es allerdings zu beachten, dass der Shift von der Chunk-Anzahl abhängig ist und sich einer guten Verteilung nur bei vielen Chunks annähert. Für die HW-Variante gibt es auch hier keine Probleme. Hashes aus der Multiplikation unterliegen mit nur einem Chunk dem gleichen Problem, wie es bei der Addition zu finden ist. Bei nur einem Chunk wird die Eingaben zurückgegeben und das Ergebnis sieht aus wie die Verteilung in Abbildung 4.1. Wird die Anzahl erhöht, so verbessert sich die Verteilung und kann sogar das Optimum erreichen. Das gilt für die reine Multiplikation und die Shift-Variante, wobei mit Shift das Ergebnis auch mit weniger Element erreichbar ist. Die HW-Variante weiß die gleichen Probleme, wie beim 32-Bit Hash auf und zusätzlich übernimmt die Kombination, mit dem XOR wieder die Probleme. Sobald der Shift durch den 64-Bit Hash möglich ist, nimmt die Qualität der Verteilung mit jedem Chunk mehr zu. Eine Erhöhung der Iterationen ändert auch hier, wie beim 32-Bit Hash, die Qualität bei weniger Chunks. Durch die XOR-Operation finden nur Bit-Manipulationen, auf der Länge des Chunks statt. Daher agiert das reine XOR nur um 32-Bit Bereich, was im Diagramm auch zu sehen ist. Wenn ein Shift beigefügt wird, ist das Ergebnis ähnlich zur Addition mit Shift und konvergiert langsam mit jedem Element, an eine bessere Verteilung. Die 64-Bit haben keinen Einfluss auf die HW-Variante und die Verteilung bleibt nach wie vor sehr gut. Dank des Shifts unterliegt das XOR nicht mehr den gleichen Problemen, wie im vorherigen Schritt. Daher lässt es darauf schließen, dass ein besseres Shift auch zu einer ausgewogenen Bit-Manipulation führt, welche nicht nur in einer Region stattfindet. Der FNV-Algorithmus bietet wie im vorherigen Schritt, das gleiche Ergebnis und somit eine gute Verteilung.

Abstand

Als ein weiteres Kriterium dient die Verteilung von benachbarten Werten im Zahlenraum. Hierfür liegt eine Liste von Werten vor, welche aus zwei Teilen besteht. In der ersten Hälfte liegt die Basis. Sie besteht aus einer endlichen Anzahl an Chunks und diese haben alle den gleichen Wert. Neben der Basis gibt es den Nachbarn. Der Nachbar hat die gleiche Chunk-Anzahl und nur das letzte Element ist um Eins inkrementiert. Die Chunk-Anzahl ist wieder auf 1,2 und 4 Chunks begrenzt und in 32-Bit und 64-Bit Hashes unterteilt. Die Verteilung der Werte spielt in diesem Aufbau keine Rolle und beginnt bei Null und endet bei N der Elemente. Die finalen

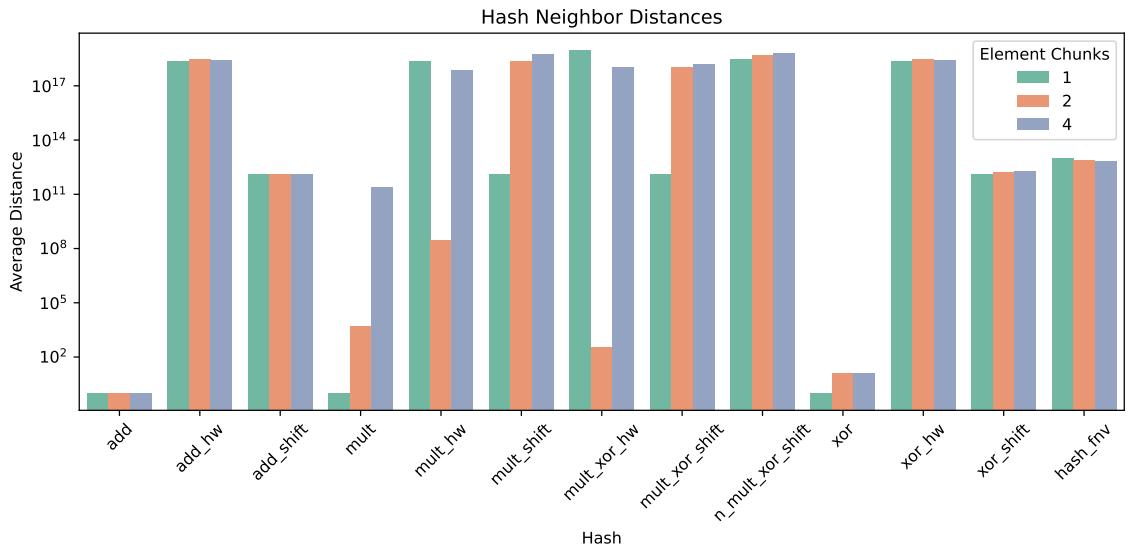


Abbildung 4.5: Hash-Abstand 64-Bit

Hash-Distanzen aus den insgesamt 10.000 Werten, wird durch den Durchschnittswert aller Teilwerte repräsentiert. Alle Distanzen mit 32-Bit Hashes sind in Abbildung 4.4 sichtbar. Je die Funktion bekommt für jede Chunk-Konfiguration einen Balken. Da die Werte sehr stark schwanken, ist die Distanz auf der Y-Achse logarithmisch dargestellt. Wie zu erwarten ist, haben die reine Addition (mit und ohne Shift) eine sehr kurze Distanz. Nur die HW-Variante verteilt Nachbarn fast über den gesamten Wertebereich. Mit der Multiplikation steigt die Distanz mit jedem weiteren Chunk. Im Vergleich zum Add, kann hier fast der gesamte Wertebereich abgebildet werden. Jedoch hat die Permutation und das Umdrehen des Hashes, einen negativen Einfluss auf die Distanz. Mit dem XOR zusammen, sind bei der HW-Variante fast die gleichen Ergebnisse zu sehen. Wenn stattdessen ein Shift verwendet wird, fällt die Distanz deutlich geringer, als bei der Multiplikation mit Shift aus. Die Distanzeigenschaft lässt sich auf die XOR-Funktion direkt übertragen, da beide in etwa die gleichen Abstände erreichen. Was auch bei den Verteilungen zu beobachten ist. Als Lösung ist daher die zusätzliche Iteration zu betrachten, welche den Abstand noch einmal deutlich erhöht und zum maximalen Abstand konvergiert. Dieses Verhalten ist jedoch wieder von der Elementanzahl abhängig und ist nicht endgültig. Der Verlauf könnte eine Ähnlichkeit zum Verlauf, wie beim Add-HW, XOR-HW und FNV aufweisen. Ähnlich zur Add-HW-Funktion schneidet auch die XOR-HW-Funktion ab und beide überbieten die Distanz von der FNV-Funktion, welche auch eine gute Distanz erzeugt. Für alle drei genannten Funktionen gilt, dass bei steigender Elementanzahl, der Abstand sich verringert. Da der Hash-Abstand in Log-Scale ist, kann eine Reduzierung um eine Größenordnung beobachtet werden. Mit Blick auf die vorgestellten Verteilung in Abbildung 4.2, ist eine Korrelation zwischen guter Verteilung und guter Distanz zu erkennen. Allerdings ist auch zu sehen, dass wie bei der N-Mult-XOR-Shift-Funktion ein geringere Hash-Abstand ausreicht, um eine gute Verteilung, zu erzielen.

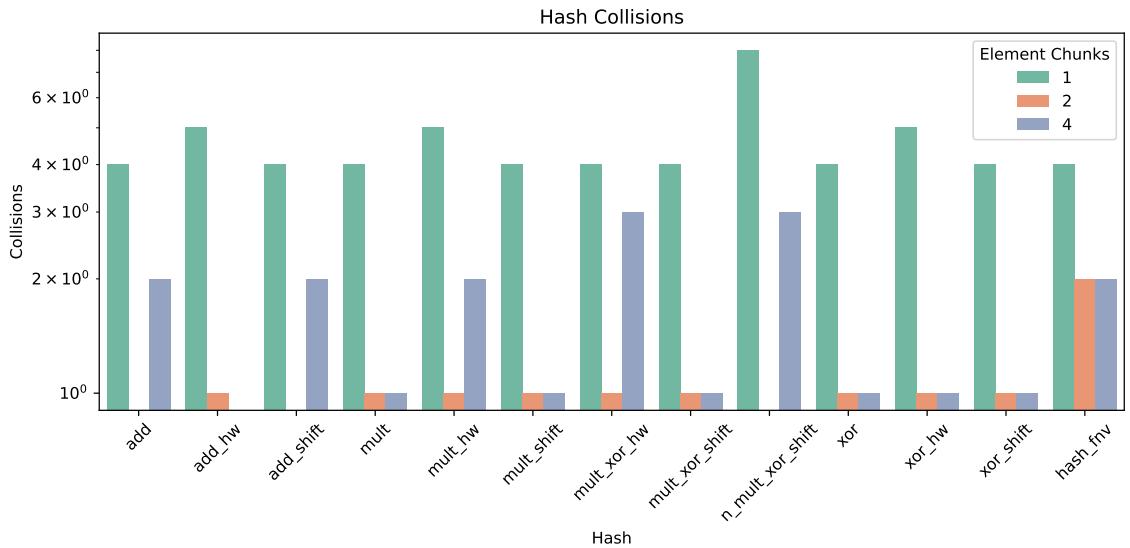


Abbildung 4.6: Hash-Kollision 32-Bit

Das gleiche Prinzip wird wieder auf den 64-Bit Hash in Abbildung 4.5 übertragen, was am Maximum im Hash-Abstand zu sehen ist. Die Funktionen, die auch bei den 32-Bit Hashes bereits gute Distanzen erreicht haben, erzielen das gleiche Ergebnis für den 64-Bit-Hash. Nur die FNV-Funktion reduziert die Distanz. Alle Funktionen, welche ein Shift als Zusatz besitzen, können diesen nun verwenden und damit den Hash-Abstand deutlich erhöhen. Allerdings gibt es nun keine wirkliche Korrelation zwischen Abstand und Verteilung. Mit Blick auf die Add-Shift-Funktion ist ein Abstand, wie bei der FNV-Funktion zu erkennen, doch ähneln sich die Verteilung in Abbildung 4.3 in keinem Punkt. Daher ist der Korrelationsgedanke nur eingeschränkt nutzbar und liefert maximal einen Hinweis auf eine sinnvolle Hash-Funktion.

Kollisionen

Als letzte Eigenschaft soll die Kollision, der aus der Verteilung generierten Werte, einen Überblick über die Güte der Funktionen verschaffen. Da die Daten zufälligen mit der Normalverteilung generiert sind, ist es nicht unwahrscheinlich, dass es Kollisionen gibt, da Wertepaare doppelt vorkommen. Diese Dopplungen lassen sich erkennen, wenn alle Funktionen in etwa gleich viele Kollisionen aufweisen. Die Kollisionen für den 32-Bit Hash, sind in Abbildung 4.6 dargestellt. Auf der X-Achse sind wieder alle Funktionen aufgelistet, welche jeweils unterschiedliche Chunk-Konfigurationen besitzen. Die Y-Achse ist logarithmisch skaliert und gibt die gezählten Kollisionen an. Ein Chunk hat nicht viel Aussagekraft, da viele Funktionen den Wert dann einfach wieder zurückgeben. Allerdings zeigt diese Konfiguration auf, wie viele Dopplungen im Datensatz zu finden sind. Die Quote liegt hierbei bei etwa einem Prozent. Nur die N-XOR-Mult-Shift-Funktion liefert ein leicht schlechteres Ergebnis, als die anderen Funktionen. Bei zwei oder mehr Chunks, nimmt die Entropie des Diagramms zu. Mit Blick auf die Mult-HW-Funktion, ist zu erkennen, dass es keine Dopplungen mehr im Datensatz gibt. Im Bereich der Addition liegen alle Funktionen gleich auf. Obwohl die HW-Variante in allen anderen Bereichen gute Ergebnisse erzielen kann, ist sie in diesem Fall nicht bessere als die anderen Formen. Schlechter wird die Kollisionsrate bei der Multiplikation. Egal ob mit oder ohne Shift, liegt die Anzahl um eins höher als beim Add. Nur die HW-Variante, welche in den vorherigen Eigenschaften, keine vergleichbar guten Ergebnisse erzielen kann, ist besser als die bereits genannten Funktionen. Für die Kombination zwischen XOR und Multiplikation, sind die Kollisionen am niedrigsten, wenn nur zwei Chunks verwendet werden. Danach steigen die Kollisionen deutlich an. Am schlimmsten

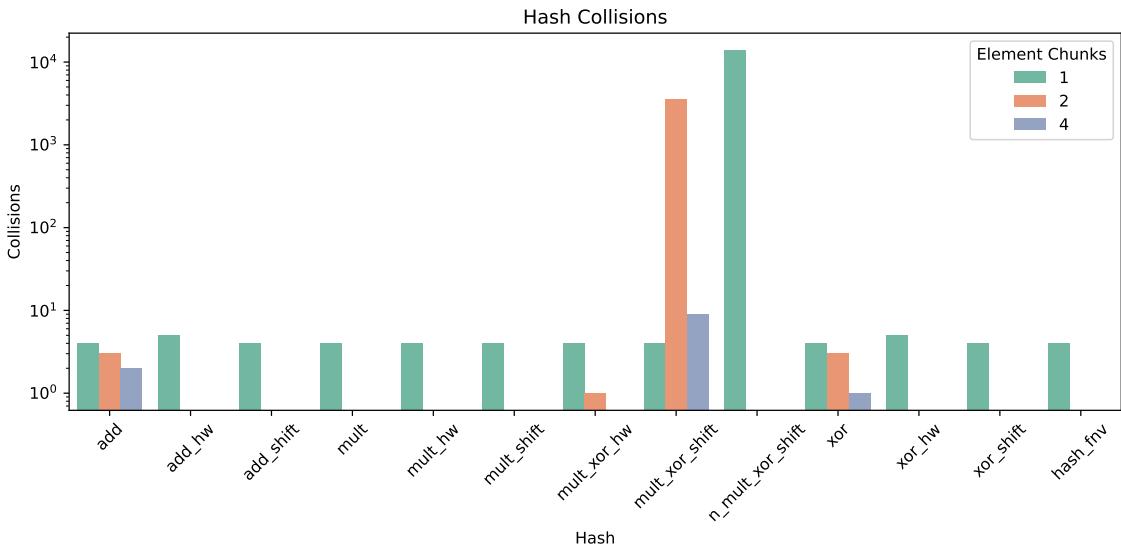


Abbildung 4.7: Hash-Kollision 64-Bit

schniedet diese Funktionen ab, wenn es weitere Iterationen über die Eingabe gibt. Das reine XOR liefert für nur zwei Chunks, keine Kollisionen. Werden die Chunks auf Vier erhöht, dann folgt ein Anstieg der Kollisionen auf die höchste Anzahl. Nur durch das Ersetzen des Shifts, durch die HW-Funktionen, erfolgt ein gleichbleibend gutes Ergebnis. Somit schließen Mult-HW und XOR-HW die Generierung von 32-Bit-Hashes, mit dem verwendeten Datensatz am besten ab.

Die Erhöhung auf 64-Bit hat für fast alle Funktionen den Vorteil, dass die Kollisionen fast komplett verschwinden. Dies liegt zum einen daran, dass der Zahlenbereich größer wird, aber auch an den generierten Werten, welche kaum eine Aussagekraft für den Zahlenbereich liefern können. Dennoch bieten die Ergebnisse leichte Tendenzen und sind in Abbildung 4.7 aufgeführt. Es sind wieder die üblichen Dopplungen, bei jeweils nur einem Chunk zu sehen. Die Kollisionen für die Addition, Multiplikation, XOR-Operationen und der FNV-Funktion, sind quasi nicht vorhanden, da einzelne Werte eher auf zufällige Treffer schließen lassen und keine Systematik aufzeigen. Heraus sticht die N-Mult-XOR-Shift-Funktion. Diese bietet in Abbildung 4.3 eine gute Verteilung an, jedoch sind die Kollisionen bei nur einem Chunk am größten, im Vergleich mit den anderen Funktionen. Diese Tatsache verdeutlicht, dass auch hier keine eindeutige Entscheidung getroffen werden kann, wenn eine von beiden Eigenschaften gut erfüllt ist. Gleicher gilt für die Funktion ohne der Iteration. Im Verteilungs-Diagramm steigt die Verteilungs-Güte mit jedem weiteren Chunk, wohingegen die Kollisionsrate mit weiteren Chunks steigt. Der Höchstwert liegt bei zwei Chunks und lässt darauf schließen, dass die alternierende Verwendung von XOR und Multiplikation keinen guten Hash erzeugen lässt oder bestimmte Anomalien zulässt, bei den die Hash-Erzeugung besonders schlechte Ergebnisse liefert.

Auswahl

Das Ziel der drei Eigenschaften ist eine gute Hash-Funktion zu finden. Für die Auswahl spielen nur 64-Bit Hashes eine Rolle, weswegen sich die Analyse auf diese Kategorie beschränkt. Im Bereich der Verteilungen erzielen Add-HW, XOR-HW, N-Mult-XOR-Shift und FNV die besten Ergebnisse. Allerdings sind leichte Schwankungen bei N-Mult-XOR-Shift zu erkennen. Der Hash-Abstand reduziert die Auswahl auf den Add-HW, XOR-HW und N-Mult-XOR-Shift, wenn

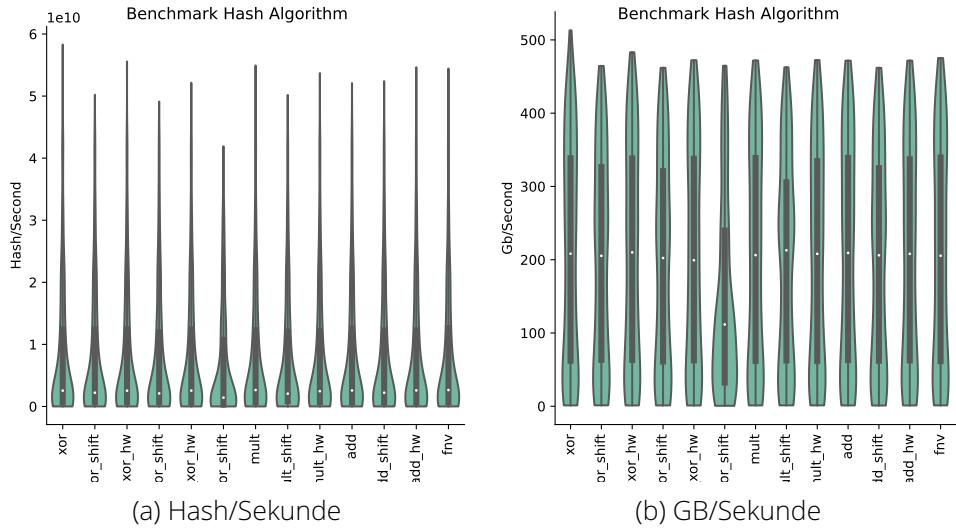


Abbildung 4.8: Einfluss der Vektorgröße auf die Hash-Funktionen

der maximale Abstand genommen wird. Allerdings ist diese Eigenschaft bedingt Aussagekräftig und liefert nur ein Indiz, wie sehr der Wertebereich für die Hashbildung verwendet wird. Aus diesem Grund wird der FNV-Algorithmus auch weiterhin in Betracht gezogen. Als finales Kriterium dient die Kollisionsrate. Hier liefert die N-Mult-XOR-Shift-Funktion das schlechteste Ergebnis, von allen Funktion und fällt somit aus den möglichen Kandidaten heraus. Add-HW, XOR-HW und FNV, nehmen sich in dieser Kategorie keinen Unterschied. Da alle drei Funktionen in allen drei Kategorien gute Ergebnisse erzielen, sind diese Funktionen gleichwertig für die weitere Verwendung geeignet. Wenn ein höherer Hash-Abstand gewünscht ist, dann sollte die Wahl auf Add- oder XOR-HW fallen. Es gilt allerdings zu beachten, dass die Ergebnisse nur aus Stichproben stammen und nur für die Verteilung getestet sind. Andere Verteilungen können durchaus ein anderes Ergebnis liefern.

4.2.3 Vektor- und Datengröße

Die Eigenschaften beziehen sich nur auf die Güte des Hashes. Im weiteren Verlauf wird die Geschwindigkeit untersucht. Hierbei spielt die Art und Weise, wie die Daten geladen werden eine wesentlich Rolle, weshalb diese Konfiguration zuerst untersucht wird. Ein Element besteht aus unterschiedlich vielen Chunks, wie es bereits im vorherigen Abschnitt verwendet wurde. Diese Chunks können eine unterschiedliche Anzahl an Bytes besitzen, wie zum Beispiel vier Bytes, für 32-Bit aus den vorherigen Daten. Damit ein Chunk aus dem Global-Memory geladen werden kann, muss dieser in einem Buffer liegen, welcher für einen bestimmten Datentypen vorgesehen ist. Diese Datentypen werden in dieser Arbeit als Chunks bezeichnet. Für die Konfiguration liegen insgesamt vier Typen vor. 4, 8, und 16Byte entsprechen den Vektortypen Vec1, Vec2 und Vec4 in CUDA. Ein Byte ist ein einfacher Char-Typ, wie er in C++ üblich ist. Die verwendeten Daten wurde auf der GPU generiert. Die Verteilung spielt für diesen Aufbau keine Rolle. Insgesamt werden bis zu einer Millionen Elemente erzeugt, welche jeweils aus bis zu 256 Bytes bestehen können. Alle Eingaben haben die gleichen Größen, wodurch keine umständliche Offset-Berechnung stattfinden muss. Die finalen Hashes sind entweder als 32- oder 64-Bit im finalen Buffer vorzufinden. Alle Tests laufen auf nur einer GPU und jede Konfiguration wird genau Fünf mal ausgeführt und gemittelt. Da die Ausgabedaten immer ein großes Spektrum an Parametern abdecken, werden alle Ergebnisse gebündelt und für die jeweiligen untersuchten Attribute, in einem Violin-Plot dargestellt. Dadurch ist es möglich, den Einfluss des jeweiligen Parameters zu erkennen. Jede Funktion verarbeitet pro Schritt maximal einen

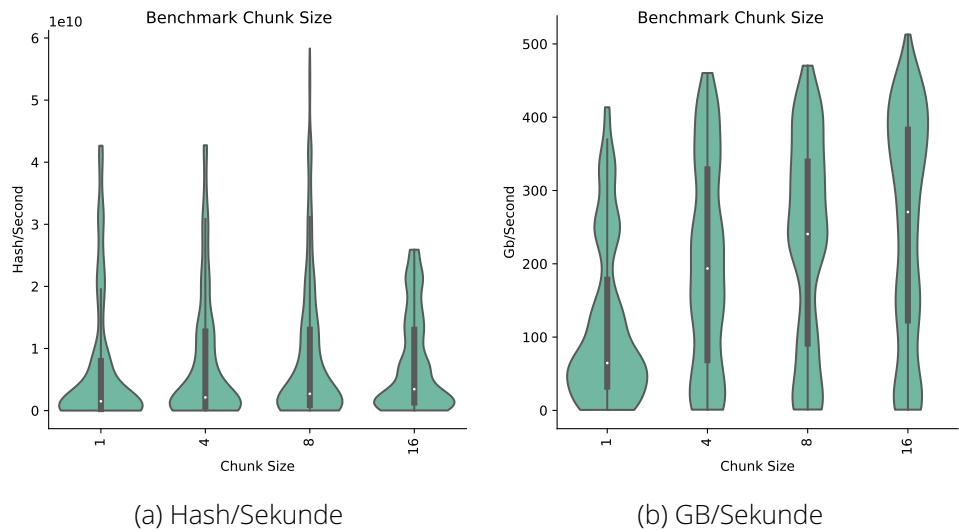


Abbildung 4.9: Einfluss der Vektorgröße auf die Geschwindigkeit

Chunk Size (Bytes)	PTX Code
1	ld.global.s8 %r19, [%rd29+-1];
4	ld.global.u32- %r20, [%rd31+-8];
8	ld.global.v2.u32- {%r8, %r9}, [%rd67+-16];
16	ld.global.v4.u32- {%r20, %r21, %r22, %r23}, [%rd31+-32];

Tabelle 4.1: PTX Code von Load-Befehlen, für jeden Chunk-Typ

32-Bit-Wert. Alle anderen 32-Bit-Werte werden mit der gleichen Operation aneinander gereiht. Für den Kernel hat es den Vorteil, dass die Threads effizienter arbeiten können und weniger auf Speicher-Transfers warten müssen. In Abbildung 4.8 sind alle Algorithmen aufgelistet. Jeder Algorithmus hat jede Vektor-Konfiguration und jede Hash-Konfiguration durchlaufen. Die Gesamtergebnisse sind in 4.8a, in Hashes pro Sekunde angegeben. Bis auf ein paar wenige Ausreißer hat keine Funktion besondere Merkmale, welche sie von den anderen Funktionen unterscheidet. Nur die N-Mult-XOR-Funktion, mit der erhöhten Iterations-Anzahl, liegt im Median hinter den anderen Funktionen. Für diese Funktion wurde wie bei den Verteilungen, die Iterationen auf Zwei begrenzt. Rechts in Abbildung 4.8b sind die gleichen Ergebnisse für die Speicherbandbreite angegeben. Hier können die Algorithmen im Peak bis zu 500GB/s erreichen, was ziemlich nah an die Peak-Performance der GPU kommt. Bei den Hashes kann mit bis zu zehn Milliarden Hashes pro Sekunde gerechnet werden.

Chunk-Size

Der Einfluss, der einzelnen Vektor- oder Chunk-Größen, ist in den Abbildungen 4.9 zusammengefasst. Die Hashes pro Sekunde sehen auf den ersten Blick gleich aus. Allerdings liegt die Median-Hash-Rate beim Chunk mit nur einem Byte, leicht unter den anderen Konfigurationen. Die Peaks der anderen Konfiguration, kann die 16Byte-Variante nicht erreichen, jedoch liegt die grundlegende Verteilung im selben Bereich. Ein wenig deutlicher wird das Ergebnis in Abbildung 4.9b. Hier findet wieder das Mapping auf die Speicherbandbreite statt und in dieser Kategorie, ist der 16Byte-Typ um rund 50GB/s vor den anderen Konfigurationen. Diese Steigerung ist mit den Load-Instruktionen, welche CUDA unterstützt zu begründen. Ein 16Byte-Type kann in genau gleich vielen Instruktionen, wie ein Byte-Type geladen werden, was den Overhead verringert und dadurch das schnellere Laden ermöglicht. Zum Vergleich ist in Tabelle 4.1,

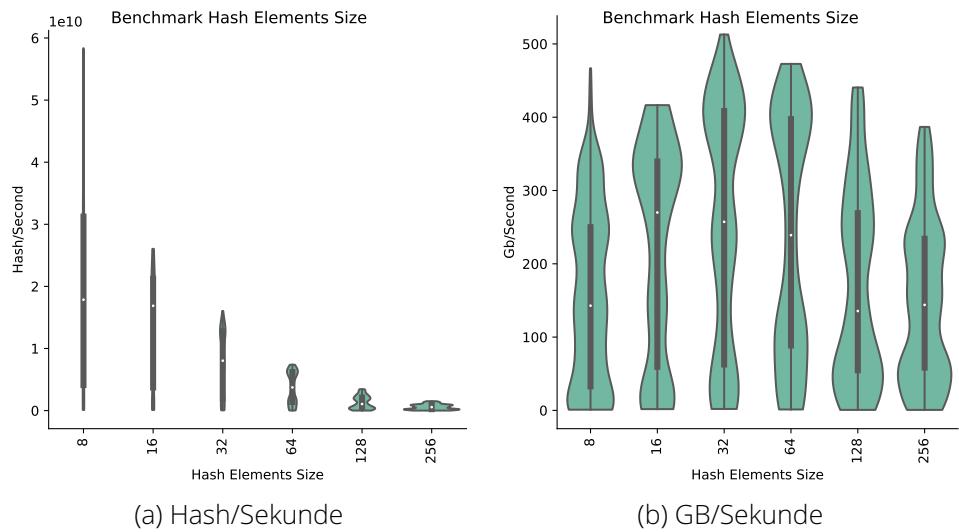


Abbildung 4.10: Einfluss von Elemente/Hash, auf die Geschwindigkeit

für jeden Chunk-Typ der entsprechende PTX-Befehl aufgelistet. Jeder Befehl stammt direkt aus einer Analyse mit dem NVProfiler. Anhand der Befehle für 8 und 16Byte Werte ist gut zu sehen, dass es für Grafikkarten spezielle Instruktionen gibt, welche das Laden von Vektor-Type ermöglichen.

Element Size/Bytes

Größere Chunks sind natürlich dann von Vorteil, wenn die Bytes pro Element zunehmen. Um so größer die Bytes pro Element sind, um so weniger Overhead entsteht bei den einzelnen Chunk-Typen. Jedoch müssen durch größere Elemente, auch größere Datenmengen transferiert und verarbeitet werden, was sich zusätzlich auf die Iterationen pro Hash auswirkt. Ein Verhältnis zwischen Durchsatz und Element-Größe, ist in den Abbildungen 4.10 sichtbar. Eindeutig zu erkennen ist, dass bei steigender Element-Größe auch der Durchsatz sinkt. Mit Blick auf die reinen Peak-Werte, halbiert sich die Hash-Rate bei jeder Verdopplung der Element-Größe. Ein anderes Bild ist in Abbildung 4.10b vorzufinden. Hier liegen die Peak-Wert bei 16 bis 64Byte. Es liegt der Verdacht nahe, dass in diesen Bereichen, der Cache besonders gut genutzt werden kann. 128Byte würde zwar auch noch einer L1-Cache-Line entsprechen, jedoch könnte hier der dahinter liegende L2-Cache negative beeinträchtigt sein. Aus diesen Daten lässt sich dazu keine genau Angabe anfertigen. Aus diesem Grund folgt in einem späteren Abschnitt, die Evaluation mit dem NVProfiler, welcher auf L1- und L2-Metriken eingehen kann.

Hash Size/Bytes

Für die Ausgabe soll es immer ein Hash mit 64-Bit sein, so wurde es bereits öfters definiert. Allerdings ist noch unklar, inwiefern die Hash-Größe einen Einfluss auf den Durchsatz hat. Abbildung 4.11 liefert dazu eine klare Antwort. Die Hash Bytes haben keinen wirklichen Einfluss auf den Durchsatz. In Abbildung 4.11a und 4.11b, sind beide Konfiguration gegenübergestellt. Sowohl für die Hash-Rate als auch für die Speicherbandbreite, liefern beide das in etwa gleiche Ergebnis. Ein 4Byte Hash kann zwar einen leicht höhere Peak Hash-Rate erreichen und liegt bei der Median-Speicherbandbreite marginal vorne, allerdings sind die Werte so dicht beieinander, weshalb nichts dafür spricht, eher einen 4Byte Hash als einen 8Byte Hash zu verwenden. Das einzige Argument wäre der Speicherverbrauch, allerdings fällt dieser im weiteren Verlauf nicht signifikant auf.

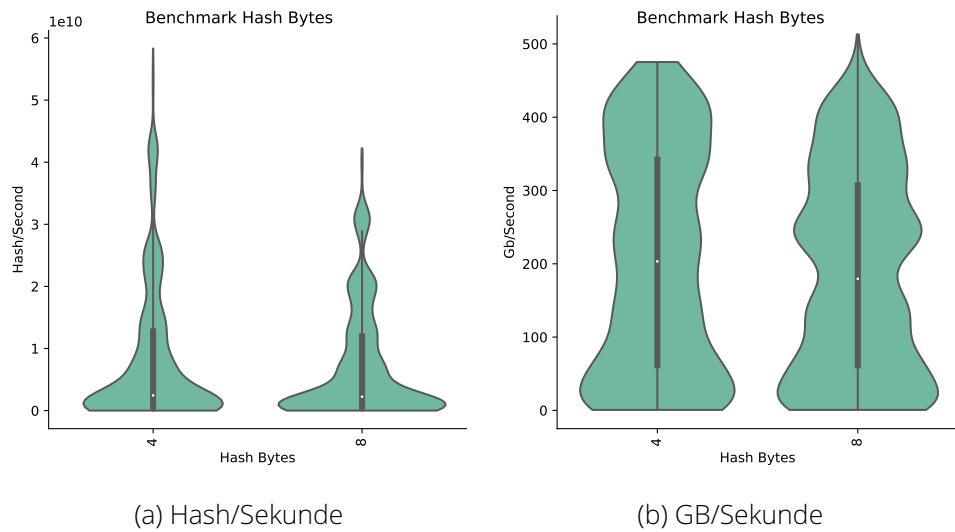


Abbildung 4.11: Einfluss der Hash-Größe, auf die Geschwindigkeit

4.2.4 Kernel Parameter

Alle Chunk-Konfigurationen verwendeten immer die gleichen Kernel-Parameter. Dieser Abschnitt dreht diesen Prozess um und setzt für alle folgenden Konfigurationen, immer einen Chunk von 4Byte und einen Hash von 8Byte ein. Um einen Kernel zu starten, werden immer Threads pro Block, Hashes pro Thread und der Algorithmus benötigt. Für N-Mult-XOR-Shift gibt es noch die Iterationsanzahl, jedoch ist diese für die Benchmarks fix auf zwei Iterationen, da bereits mit dieser Einstellung, die Reduzierung der Hash-Rate sichtbar ist und zusätzlich die Qualität des Hashes nicht ausreichend ist. Für jeden Parameter wurde auf dem Lokalen-System der Profiler auf alle Konfigurationen angewendet und die daraus entstandenen Plots sind in den jeweiligen Abschnitten näher erläutert. Der Profiler bietet viele mehr Metriken an, für diesen Abschnitt sind jedoch nur vier von Interesse. Dazu zählen die Auslastung des Streaming Multi-Prozessors (SM Usage (%)), die Auslastung des Memory-Interfaces (Memory Usage (%)), die Nutzung des L1 Caches (L1 Usage (%)) und die Nutzung des L2 Caches (L2 Usage (%)). "Usage" für Caches bedeutet, dass Speicherzugriffe Elemente im Cache gefunden haben. Eine hohe Prozentangabe bedeutet eine hohe Hit-Rate.

Hash Funktionen

Der Einfluss der Funktions-Auswahl ist bereits aus Abbildung 4.8 bekannt. In dieser liefert die keine Funktionen eine schlechtere Hash-Rate, als eine andere Funktion, außer wenn mehrere Iterationen durchgeführt werden. Die Auslastung einzelner Komponenten für jede Funktion, ist in Abbildung 4.12 dargestellt. Links oben befindet sich die SM-Auslastung, daneben die Memory-Auslastung und darunter die L1- und L2-Nutzung. Auf der X-Achse sind immer die Parameter und auf der Y-Achse die Auslastung/Nutzung in Prozent. Für die einzelnen Algorithmen gibt es keinen Unterschiede, bei der SM- und Memory-Auslastung. Die Peaks variieren ein wenig, allerdings ist für die Auswertung nur der Median und die jeweiligen Quartil-Grenzen relevant. Diese bewegen sich alle in der selben Region. Mit Blick auf die darunter liegenden Cache-Nutzungen, liegen fast alle Funktionen gleich auf. Der Median der Mult-XOR-Shift-Funktion weicht ein wenig zu seinen Nachbarn ab, alle Werte liegen aber dicht beieinander. Was aus den Diagrammen abzulesen ist, dass die generelle Hash-Funktion auf der GPU eine SM-Auslastung von rund fünf bis zehn Prozent erreichen kann und die Speicherauslastung bei 30% liegt. In der Regel ist gewünscht, dass diese Kategorien bis zu 100% erreichen. Die Tendenzen im Memory-Bereich lassen darauf schließen, dass die Funktion sehr speicher-

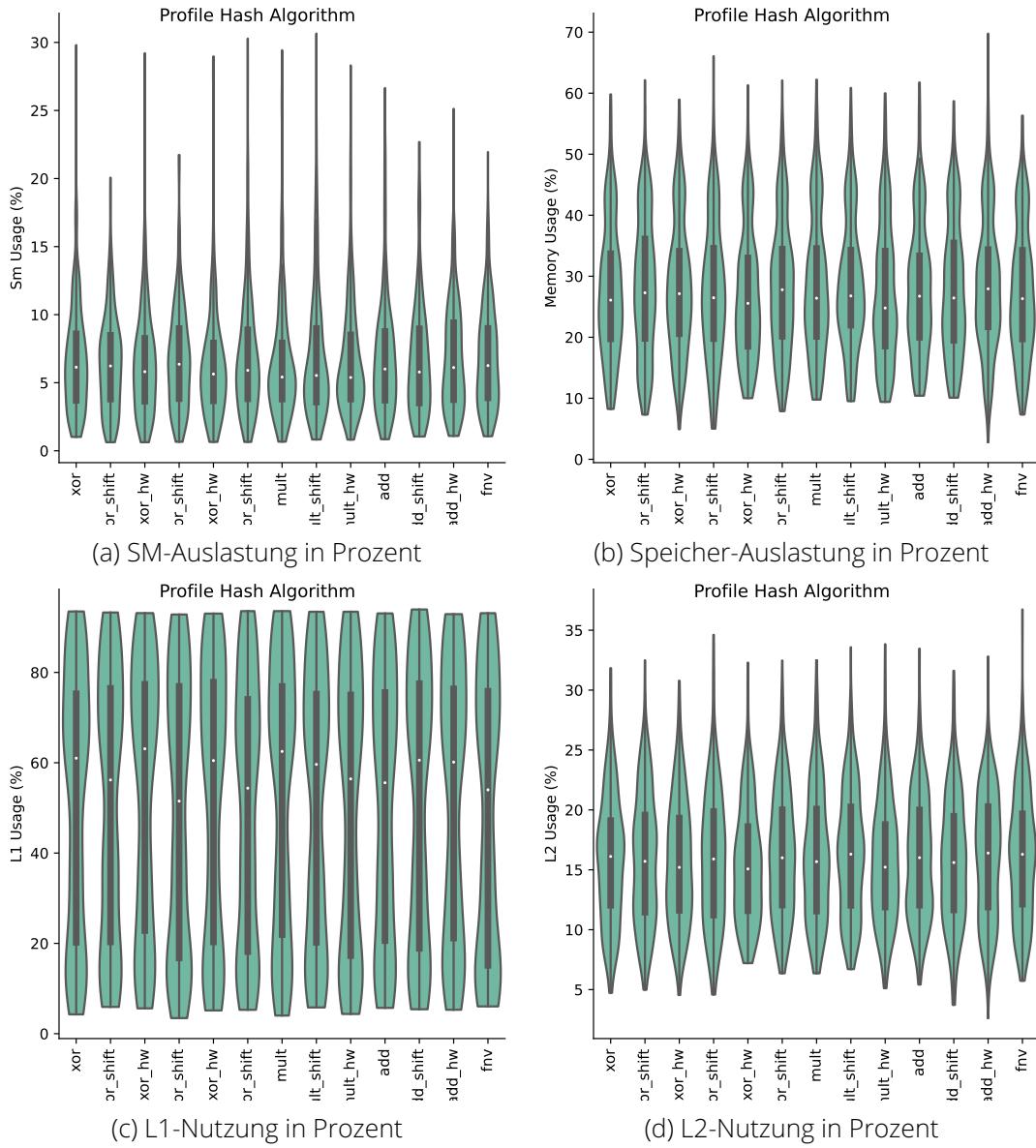


Abbildung 4.12: Kernel-Profile für einzelne Hash-Funktionen

lastig ist, aber diesen auch nicht effizient nutzen kann. Für diese Eigenschaft spielt die Nutzung der Caches, eine wichtige Rolle. Der L1-Cache in Abbildung 4.12c zeigt eine Median-Auslastung von rund 60% schwankt aber stark zwischen 20 und 80%. Beim L2-Cache (Abbildung 4.12d) sieht es deutlich schlechter aus, denn dort liegt der Median nur bei 15%. Für alle Funktionen sind Cache-Ergebnisse gleich, was daran liegt, dass alle das gleiche Zugriffs-Modell verwenden. Jeder Thread liest alle seine Chunks nacheinander und diese Daten sind auch angeordnet, wie die Threads. Daher liegt bereits ein Coalesced-Access-Pattern, welches auf GPUs zu bevorzugen gilt vor. Sind zu viele Threads für einen Kernel aktiv, ist es durchaus möglich, dass diese gegenseitig die Elemente aus dem L2-Cache entfernen. Das sogenannte Thrashing findet statt. Als Lösung für das Problem kann entweder die Chunk-Anordnung so angepasst werden, dass ein Struct-of-Arrays vorliegt und jeder Chunk-Abschnitt in seinen eigenen fortlaufenden Buffer liegt oder es wird die Thread-Anzahl reduziert. Ersteres wird rein für den Aufwand nicht weitere in Betracht gezogen, da hierfür erst ein Kernel die Daten in die jeweiligen Sektionen kopieren müsste und letzteres skaliert im Zweifel schlecht mit steigender Elementanzahl.

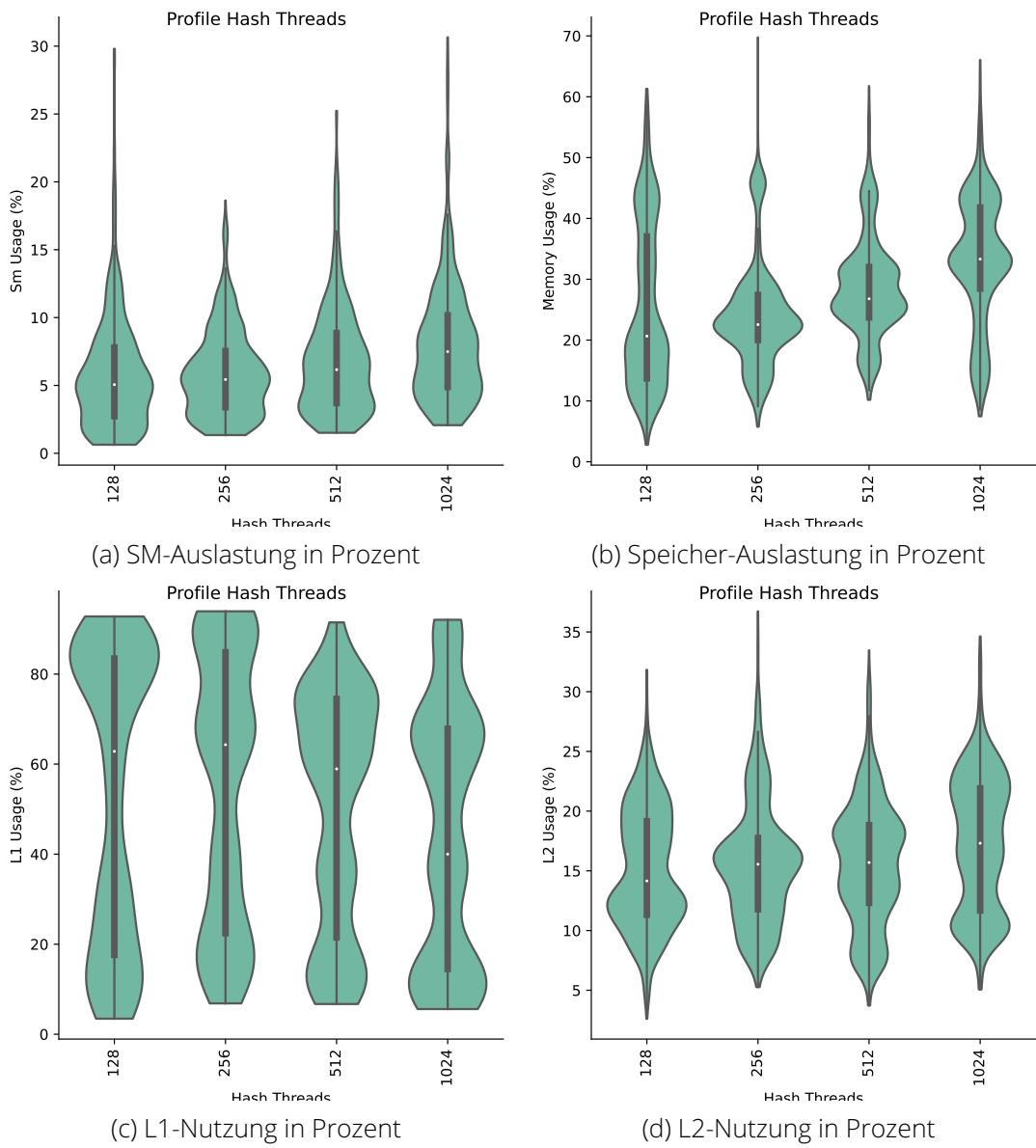


Abbildung 4.13: Kernel-Profile für einzelne Thread/Block-Konfigurationen

Threads

Eine weitere Option bieten die Thread-Konfigurationen für den Kernel. In Abbildung 4.14 sind die gleichen Metriken, wie für die einzelnen Funktionen aufgeführt. Auf der X-Achse befindet sich in diesen Abschnitt, die jeweilige Thread-Anzahl pro Block. Die maximale Anzahl an Threads pro Block, ist immer 1024 [20]. Daher werden alle typischen Werte von 128, bis 1024 abgebildet. Anzumerken ist, dass die Zahl immer durch 32-teilbar sein muss, da sonst die Kernel-Ausführung scheitert. Es ist leicht zu erkennen, dass die Speicherauslastung und die SM-Auslastung mit der Thread-Anzahl steigt. Gleiches gilt auch für die L2-Nutzung. Nur der L1-Cache lässt mit jeden weiteren Thread nach.

Daraus lässt sich schließen, dass mit erhöhter Thread-Anzahl pro Block, die Leistung des Kernels fällt. Auch wenn die L2-Nutzung steigt, deutet die L1-Nutzung darauf hin, dass bei mehr Threads auch Cache-Misses häufiger auftreten. Der Zusammenhang besteht darin, dass mehr Threads auch einen größeren Abschnitt im Buffer bearbeiten. Wenn dieser Abschnitt nicht in den Cache passt, dann erfolgt das sogenannte Thrashing. Threads werfen Elemente, welche ein anderen Thread benötigt aus dem Cache, was zu einem Miss für den anderen Thread führt.

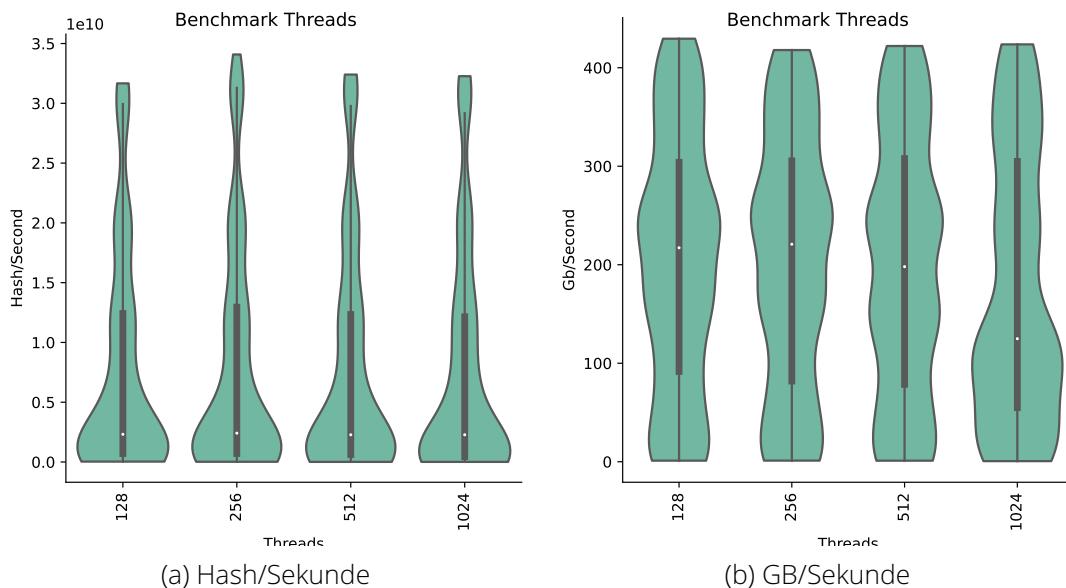


Abbildung 4.14: Hashing mit unterschiedlichen Thread pro Block

Daraus resultieren deutlich mehr Anfragen an den L2-Cache und somit die L2-Nutzung steigt. Für den Kernel bedeutet das eine höhere Wartezeit und dadurch auch eine höhere Speicher- auslastung, da der Zugriff auf den L2-Cache der langsamer ist. Für die Hash-Rate hat dieses Verhalten keinen großen Einfluss. In Abbildung 4.14 ist für die Hash-Rate und der Speicher- bandbreite, die Thread-Anzahl eigentlich egal. Wenn zusätzlich die Peaks betrachtet werden, so liegt die 256-Thread-Konfiguration marginal vorn.

Hashes pro Thread

Da ein Thread nicht nur ein Element verarbeiten muss, sondern auch mehrere Elemente hintereinander verarbeiten kann, gibt es den Parameter Hashes pro Thread. Dieser gibt an, wie es der Name bereits sagt, wie viele Hashes ein Thread erzeugen soll. Hierfür gibt es eine Analyse mit dem Profiler in Abbildung 4.15. Die Konfiguration wurde auf bis zu vier Elemente begrenzt, da bereits beim vierten Element das Verhalten sichtbar wird. Für die Auslastung von SMs und Memory-Interface, führt eine Erhöhung der Hashes pro Thread, zu einer Verringerung der Auslastung. Hier trifft, wie beim Thread-Parameter, das Verhalten im L1-Cache zu. Wenn die Hashes pro Thread erhöht werden, steigt die Effizienz des L1-Caches. Dadurch sinkt die Nutzung des L2-Caches und das Memory-Interface wird entlastet. Die Schwankungen sind jedoch deutlich geringer, als bei der Thread-Konfiguration, weshalb dieser Parameter keine besondere Rolle bei der Optimierung spielt.

Element Bytes

Als letzten Parameter ist die Bytes-Anzahl, welche für einen Hash verarbeitet wird zu nennen. Für jeden Hash wird immer ein Element benötigt und diese Element besteht aus N Bytes. Daraus ergibt sich der Parameter "Hash Element Size" oder "Chunks". Den Parameter kann der Kernel nicht kontrollieren, dennoch ist es sinnvoll den Einfluss der einzelnen Byte-Größen zu vergleichen. Da mit den Hashes keine Texte verarbeitet werden sollen, findet eine Reduzierung auf bis zu 256Bytes statt. Die Schrittweite ist immer eine zweier Potenz und beginnt bei 2^3 . Acht Bytes entsprechen in dieser Arbeit, genau einem Spalten-Wert in der Datenbanktabelle und sind daher besonders interessant für die spätere Evaluation des Hash-Joins. Die

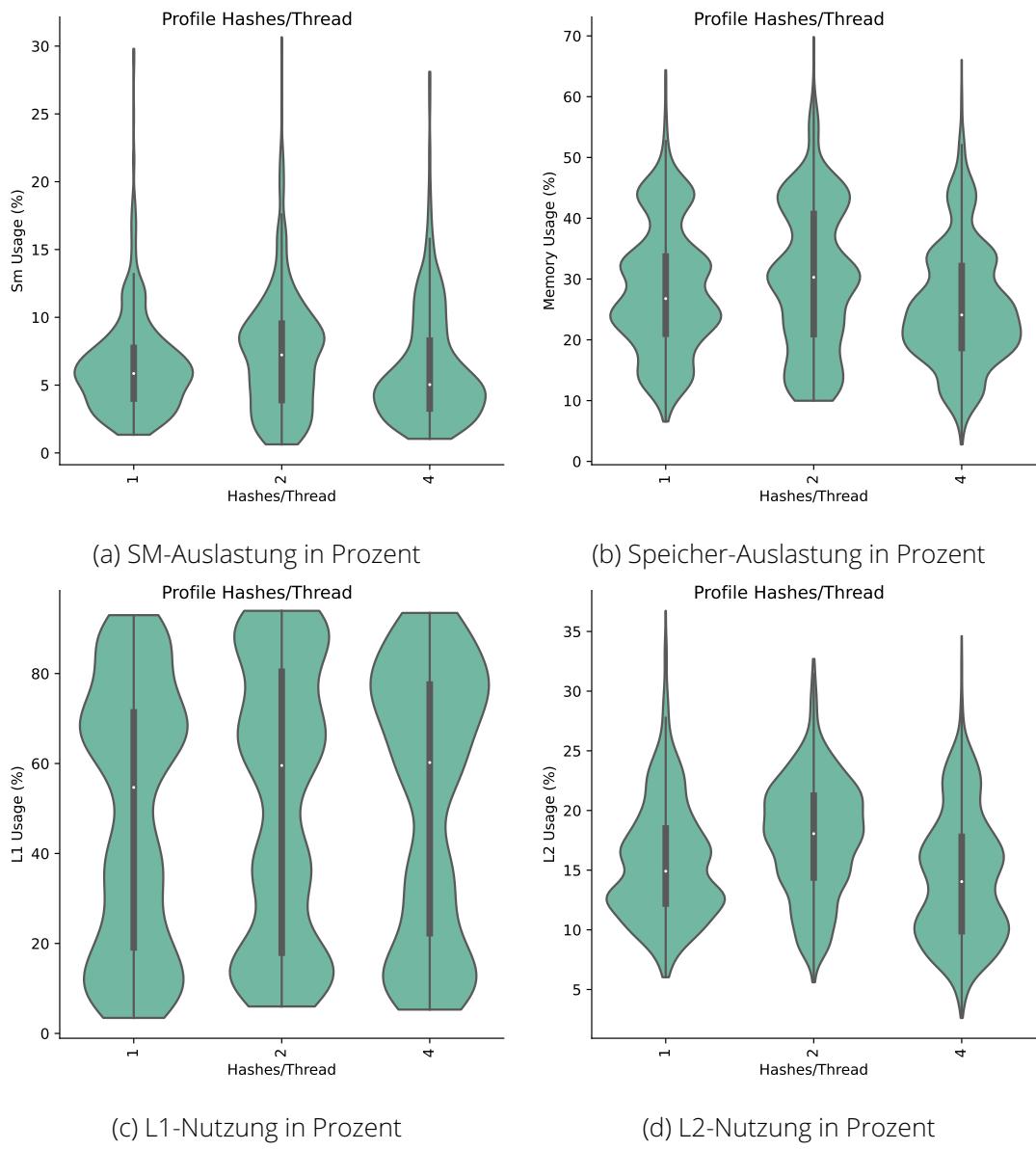


Abbildung 4.15: Kernel-Profile für berechnete Hashes pro Thread

Metriken sind in Abbildung 4.16 zu sehen. Auch wie in den vorherigen Kernels, spielt die Relation zwischen L1-Cache-Nutzung und Memory-Interface-Auslastung eine wichtige Rolle. Bei genauerer Betrachtung erzeugt die L1-Darstellung (4.16c), eine umgedrehte Glockenkurve und die Memory-Interface-Darstellung (Abbildung 4.16b) den inversen Verlauf, was auf eine Korrelation schließen lässt. Allerdings lässt in diesen Fall eine hohe L1-Nutzung, nicht auf eine höhere Datenrate schließen, wie es in Abbildung 4.10b zu sehen ist. Hier führt eine höhere L1-Nutzung zu einer schlechteren Datenrate. Die L2-Nutzung in Abbildung 4.15d, liefert auch keine genaueren Informationen, da der Verlauf recht gleich für alle Varianten ist. Die aller erste Vermutung deutet darauf hin, dass der Grund für den Peak am L2-Cache liegt, welcher mit genau 32Byte Cache-Zeilen auftritt. Allerdings sprechen die Daten nicht für diese These. Aus diesem Grund lässt sich zu diesem Verhalten keine nähere Aussage treffen und sollte in zukünftigen Arbeiten genauer betrachtet werden.

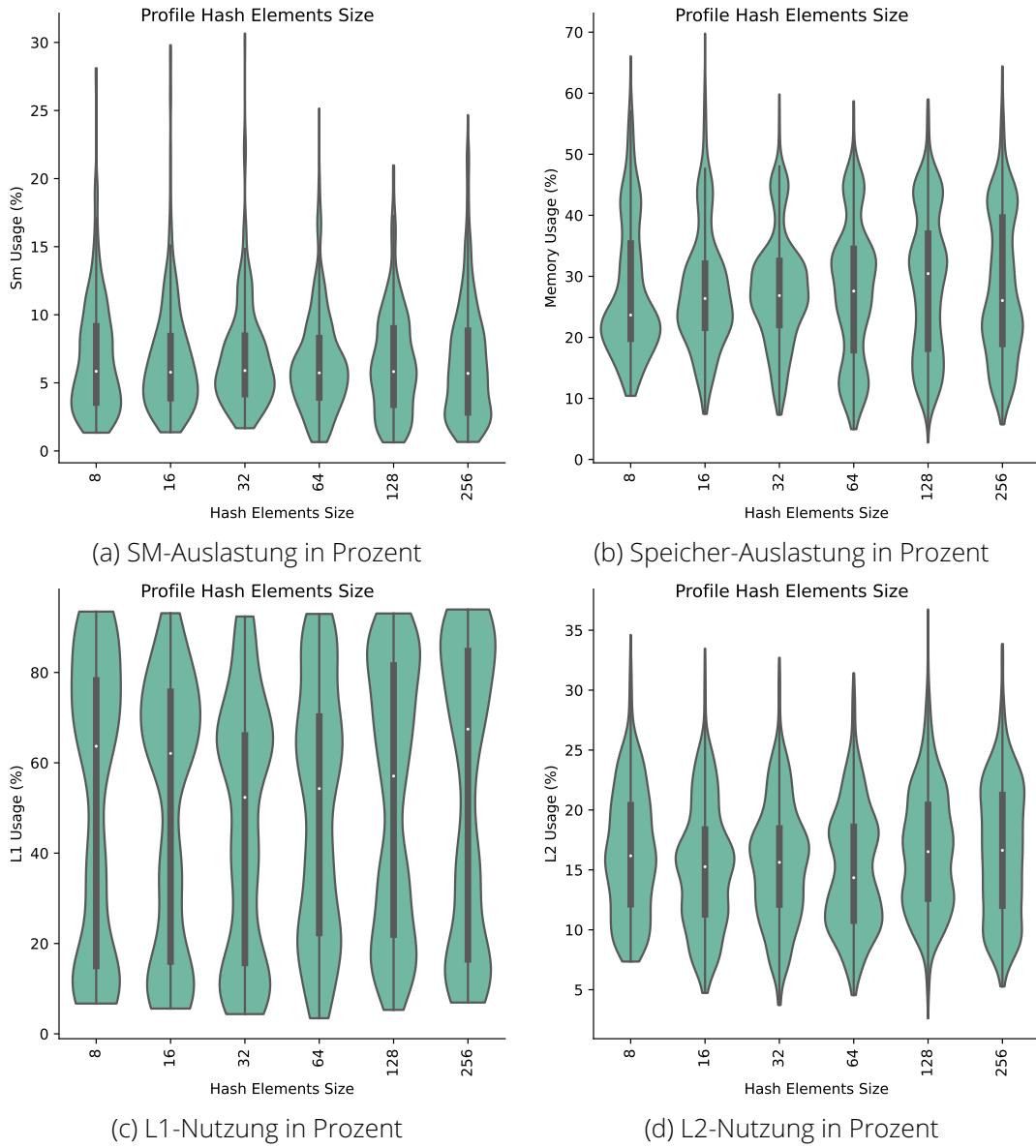


Abbildung 4.16: Kernel-Profile für einzelne Eingabe-Größen-Konfigurationen

4.2.5 Multi GPU

Die Höchstwerte mit einer GPU sind bereits durch die vorherigen Ergebnisse bekannt. Dieser Abschnitt beschäftigt sich mit den Erweiterungen, für die Mult-GPU-Anwendung. Da es in den Versuchen nicht mehr um die einzelnen Kernel-Parameter geht, werden diese auf fixe Werte festgelegt. Chunk- und Hash-Größen werden jeweils auf 4 und 8Byte gesetzt. Für die Kernel-Startparameter werden 256 Threads pro Block und zwei Hashes pro Thread angegeben. Die Hash-Funktion hat keinen nennenswerten Einfluss und die Hash-Güte ist für die Laufzeit nicht relevant. Daher führt der Kernel immer die XOR-Funktion aus. Der Hash erfolgt durch die Verknüpfung von 64Byte Eingabe-Daten. Die Eingabe beginnt mit nur 1000 Element und Endet mit 100 Millionen Elementen. Dieser Datensatz wird für die weitere GPU, in gleich große Segmente geteilt und kopiert oder direkt geniert, was von dem Modus abhängt. Die Messergebnisse umfassen den gesamten Zeitraum, wie er in den einzelnen Grafiken in Abbildung 3.1 angegeben ist. In der Abbildung 4.17 sind beide Modi gegenübergestellt. Auf der X-Achse befindet sich jeweils die Nummer, welche intern für die Modi vergeben wurde. Die Y-Achse gibt die jeweilige Kenngröße an. Es wird wieder in Hash-Rate und Datenrate unterschieden. Die Null

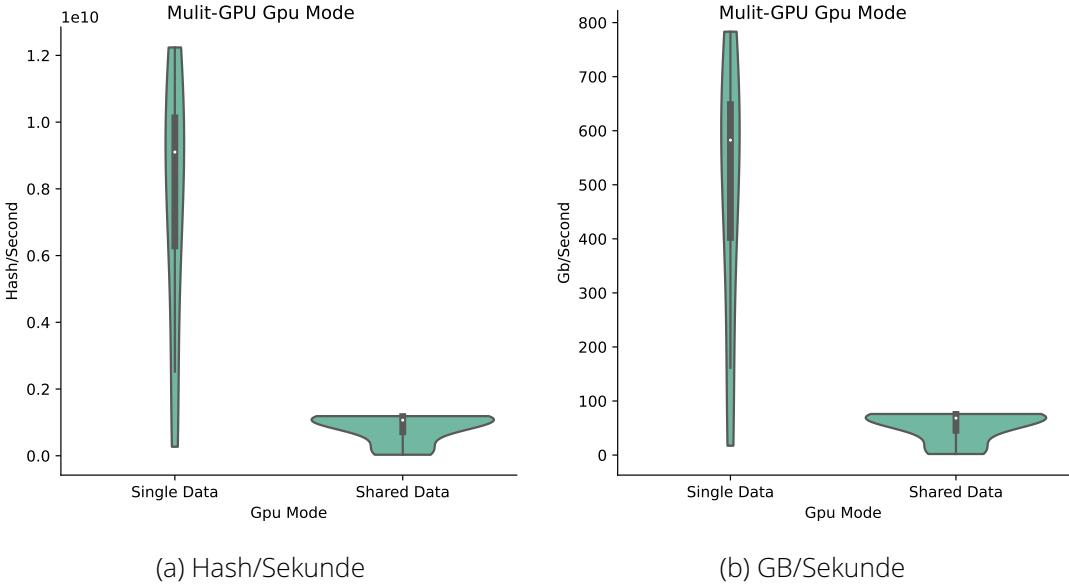


Abbildung 4.17: Hashing mit unterschiedlichen GPU-Strategien

steht intern für den Single-Betrieb. Jede GPU verarbeitet den eigenen Datensatz und die Zeit wird gestoppt, wenn alle GPUs das Ziel erreicht haben. Aus diesem Grund bekommen auch alle GPUs den gleich großen Datensatz. Mit der Eins ist der Shared-Betrieb gemeint. In diesem Fall wird der Datensatz auf der ersten GPU generiert und die Buffer für alle GPUs vorbereitet. Die Zeit startet, sobald die zweite GPU den Datensatz auf den internen Speicher kopiert. Der einzige Parameter welcher sich in der Abbildung 4.17 verändert, ist die Element-Anzahl. Jeder Modus ist speziell nur für mehrere GPUs gedacht und daher fällt der letzte freie Parameter, welche die GPU-Anzahl bestimmt, für diese Vergleich raus. Was für den Vergleich wichtig ist, ist ein Referenzwert. Dieser beträgt rund 10^{10} Hashes pro Sekunde, bei einer Konfiguration mit 64Bytes pro Element. Die Erwartung an die Modi ist, dass eine Konfiguration mit mindestens einer GPU mehr, auch einen höheren Durchsatz erzielt. Im Idealfall wäre dies der doppelte Wert, da beide GPUs identisch sind. Für den shared Betrieb ist diese Ziel in Abbildung 4.17a auch fast erreicht. Die Datenrate liegt auch ungefähr beim doppelten Wert und reizt fast das Maximum beider GPUs aus (Abbildung 4.17b). Variante zwei liefert ein deutlich schlechteres Ergebnis ab. Hier liegen die Werte bei etwa dem Viertel vom normalen Betrieb mit nur einer GPU. Der Grund kann an zwei Faktoren liegen. Zum einen ist die maximale Auslastung für jede GPU, durch die Speicherlimits der Haupt-GPU begrenzt, wodurch jede GPU nur die Hälfte von der eigentlichen Menge verarbeiten kann und zum anderen müssen die Daten erst zwischen den GPUs kopiert werden. Die Datenrate ist von NVLink auf 100GB/s begrenzt und liegt somit fernab von den möglichen 1TB/s aus der shared Konfiguration. In einzelnen Konfiguration wurden Übertragungswerte von rund 45GB/s für die Verteilung und 6GB/s für das einsammeln der Hashwerte erreicht. Als Lösung könnte die Menge für die zweite GPU so reduziert werden, bis sich beide Verarbeitungsprozesse von beiden GPUs zeitlich annähern. Dadurch würde die Latenz wegfallen, allerdings steigt der Workload auf der Haupt-GPU. Dieses Konzept wird nicht näher betrachtet und fällt in das Aufgabengebiet für folgende Arbeiten. Im ersten Eindruck wirkt diese Lösung jedoch wenig praktikabel, da erst die Latenzen ausgemessen werden müssen und diese für jede GPU anders ausfallen. Auch wenn das Ergebnis von mehreren GPUs nicht immer vielversprechend aussieht, ist dennoch eine Steigerung möglich und diese ist in Abbildung 4.18 zusammengefasst. Diese bestätigt die Aussage, dass die Hash- und Datenrate sich zu der Einzel-GPU-Variante verdoppelt haben. Zur Einordnung der Datenrate, ist in Abbildung 4.18b gut zu sehen, dass für eine GPU in etwa 500GB/s erreicht werden und diese für zwei GPUs auf 1TB/s steigt. Die gleiche Steigerung gilt auch für die Hash-Rate in Abbildung

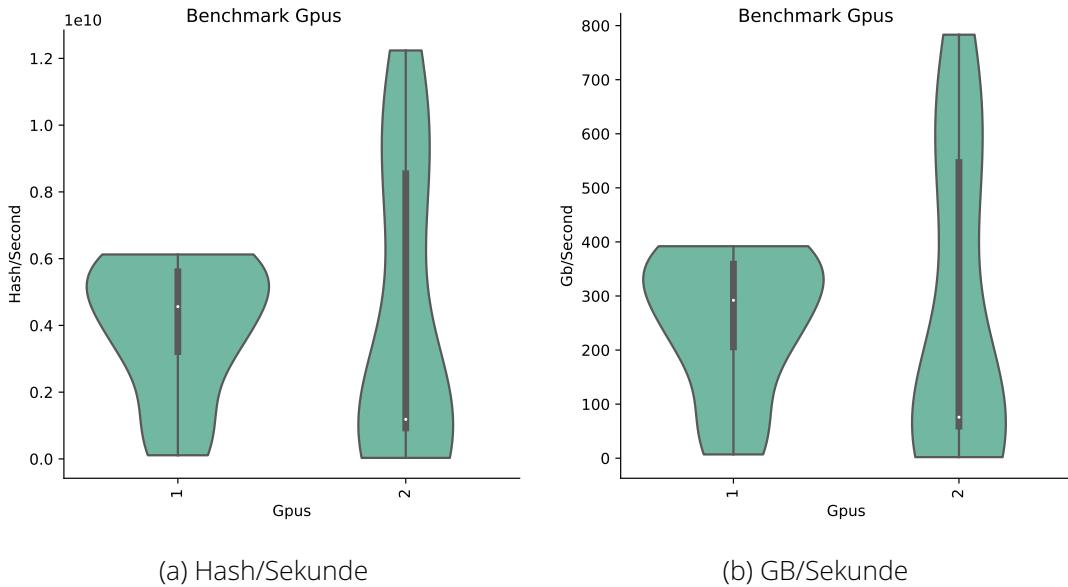


Abbildung 4.18: Hashing mit mehreren GPUs

4.18a, allerdings ist in dieser Darstellung keine Verdopplung sichtbar. Mit diesen Eigenschaften bietet zumindest eine Multi-GPU-Variante eine attraktive Steigerung an, jedoch ist der shared Modus nur begrenzt einsetzbar. Da im weiteren Verlauf die Eingaben und die Ergebnisse immer auf einer GPU liegen, würde nur der single Modus in Betracht kommen. Da dieser jedoch einen relative schlechten Durchsatz erzielt. Wird dieser nicht im weiteren Verlauf eingesetzt.

4.2.6 Zusammenfassung

Die einzelnen Eigenschaften wurden bereits einer kleinen Auswertung unterzogen und die Wahl fiel auf die zwei neuen Add-HW und XOR-HW-Funktionen, welche im gegebenen Datensatz eine gute Konkurrenz für den etablierten FNV-Algorithmus darstellten. Beide lieferten eine gute Verteilung, einen guten Abstand zwischen benachbarten Werten und eine geringe Kollisions-Rate. Die anderen Funktionen, welche im Hashing-Kapitel vorgestellt wurden, haben in mindestens einer Eigenschaft, ein unzureichendes Ergebnis erzielt. Fällt das Hauptkriterium auf die Geschwindigkeit, dann eignen sich fast alle Funktionen für die Anwendung. Nur eine Funktion mit mehreren Iterationen, reduziert die Geschwindigkeit spürbar. Ist die Funktion gewählt, so folgt im nächsten Schritt die Auswahl des Datenformats. Hier bieten alle Funktionen ein Spektrum von bis zu 16Byte Datentypen an. Diese werden alle von der GPU nativ unterstützt und reduzieren die Auslastung des Memory-Interfaces. Auch wenn die Datentypen größere Chunks unterstützen, sinkt mit jedem weiteren Byte die Hash-Rate. Für die Auswahl der Hash Bytes ist das nicht der Fall. Hier konnte kein Unterschied zwischen 32- und 64-Bit Hashes festgestellt werden. Das sind die Parameter, welche in der Praxis mit der Eingabe definiert werden. Für die reine Kernel-Optimierung gibt es Parameter wie Threads pro Block und Hashes pro Thread. Mit der Steigerung der Threads pro Block, sinkt auch die L1-Cache Nutzung, was sich wiederum negativ in der Hash-Rate widerspiegelt. Die Erhöhung der Hashes pro Thread, liefert genau das Gegenteil und steigert marginal die L1-Nutzung. Um die Hash-Rate zu erhöhen, werden zwei GPU-Modi untersucht, welche jeweils sehr stark unterschiedliche Ergebnisse liefern. Im shared Betrieb steigt der Durchsatz auf fast das Doppelte und für den single Betrieb reduziert sich der Durchsatz auf fast 25% zum normalen Kernelaufruf, mit nur einer GPU. Die Erhöhung des Durchsatzes ist mit zwei GPUs somit durchaus möglich, allerdings nur für den Anwendungsfall, dass beide GPUs die Daten nicht austauschen müssen. Aus diesen Erkenntnissen wird ein Hash-Kernel mit 256 Threads pro Block, vier Hashes pro Thread, einem

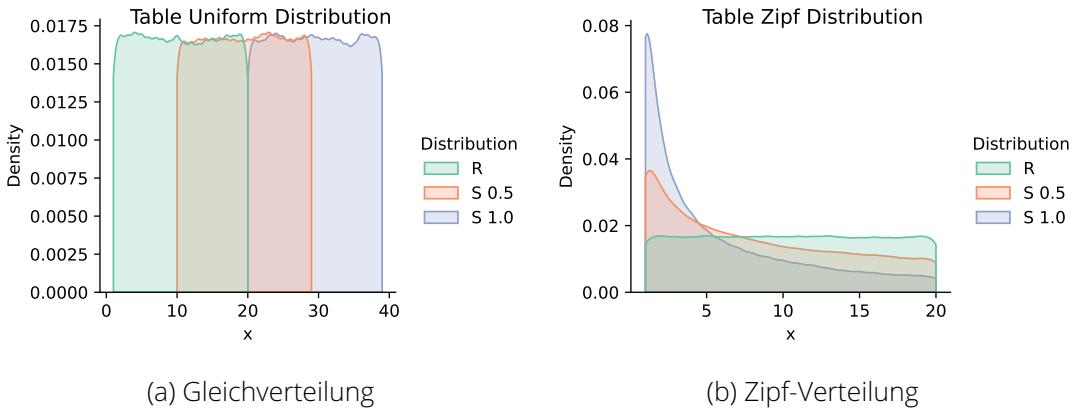


Abbildung 4.19: Verteilungen der Testdaten

8Byte Chunk und einem 8Byte Hash gebildet, welcher perfekt für das Anwendungsgebiet im nächsten Abschnitt angepasst ist.

4.3 Hash-Join

$$|R| = \frac{|S|}{RS_{Scale}} \quad (4.1)$$

$$\frac{\text{Tuples}}{\text{Second}} = \frac{|S| + |R|}{T} \quad (4.2)$$

Anders als beim Hashing, besteht der Hash-Join nicht nur aus einem Kernel. Für die Berechnung des Joins wird erst die Hash und danach die Probe-Funktion aufgerufen. Zwischen diesen zwei Schritten, liegt noch die Vektorisierungs-Funktion, wenn die Daten zu große für das Probing sind. Der Vektorisierungs-Schritt wird hauptsächlich von der CPU durchgeführt, nur die GPU übernimmt beim Radix-Teilschritt die Partitionierung. Für die Eingabe sind immer zwei Tabellen nötig. R beschreibt die kleinste und S die größte Tabelle. Das Verhältnis zwischen den beiden Tabellen ist, durch den RS-Scale Parameter gekennzeichnet. Die Anzahl der Zeilen für S, lässt sich mit Gleichung 4.1 berechnen. Neben der Zeilenanzahl ist die Konfiguration der Spaltenanzahl möglich. Die Anzahl ist auf ein bis zwei Spalten beschränkt. Die Partitionierung lässt sich in unterschiedlich viele Sub-Tasks parallel abarbeiten. Aus diesem Grund gibt es für jede Konfiguration ein Stream-Limit. Das Limit beschreibt, wie viel Streams jede GPU zur Verfügung hat und beschränkt dadurch die maximalen aktiven Sub-Tasks. Natürlich kann eine Funktion mehr Aufgaben starten, allerdings unterliegen diese dann der jeweiligen Stream-Warteschlange. Damit jedes Ergebnis repräsentativ ist und keins Schwankungen im System unterliegt, wird jede Konfiguration mindestens fünfmal ausgeführt. Die Einheit der Ergebnisse ist Tuple pro Sekunden und wird aus der Kombination von R und S, wie Gleichung 4.1 berechnet. Wie bei der Hash-Analyse, liegen alle Ergebnisse auch als GB/s vor. Die Datengröße ergibt sich aus der Kombination von R und S.

4.3.1 Verteilung

Egal wie große die Tabellen ausfallen, jeder Spaltenwert unterliegt immer einer Gleich- oder Zipf-Verteilung. Mit der Gleichverteilung wird die Überlappung von R und S simuliert. R und S sind jeweils gleichverteilt und erstrecken sich über einen definierten Wertebereich. Mit dem Skew wird der Wertebereich von der S-Tabelle angehoben. Hierbei verschiebt sich der Offset für Start- und Endwert, um die Differenz zwischen Start- und Endwert, mal dem Skew-Faktor. Bei einem Skew von Null, würden beide Datensätze zu 100% überlappen und bei 0.5

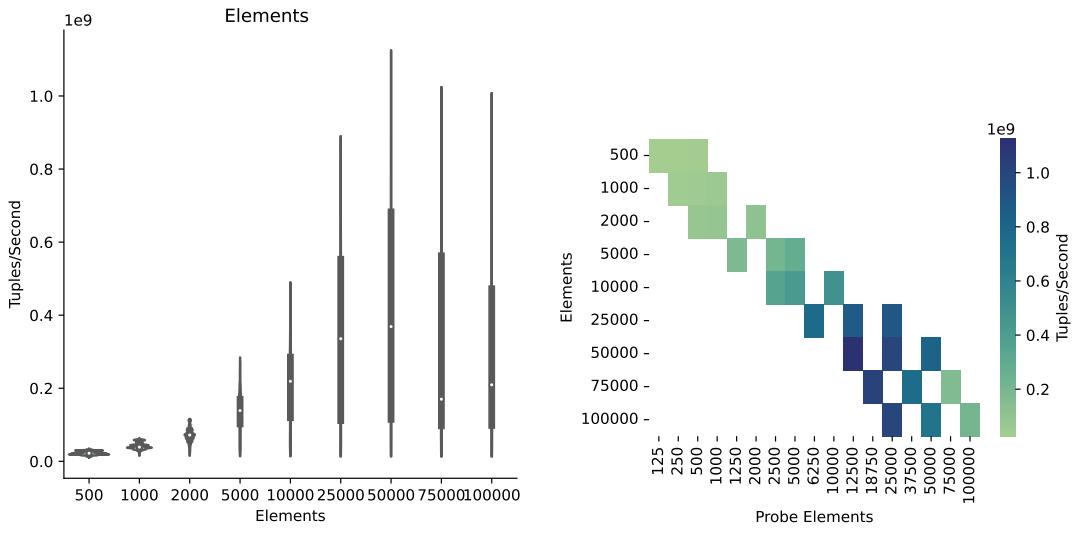
nur noch zu 50%. Die Zipf-Verteilung wird nur auf dem R-Datensatz angewendet. R behält die Gleichverteilung und hat das gleiche Maxima, wie die resultierende Verteilung von S. Mit dem Skew-Faktor wird die Konzentration von S, auf einen bestimmten Wert erhöht. Der Wert Null gleicht bei der Zipf-Verteilung, einer Gleichverteilung. Wird der Skew erhöht, so erhöht sich die Wahrscheinlichkeit für einen bestimmten Wert. Das Ziel der unterschiedlichen Verteilungen liegt darin, dass das Verhalten bei unterschiedlich vielen Paarung untersucht werden soll. Abgesehen vom Hashing, sind alle weiteren Funktionen von der R/S-Verteilung abhängig. So reduziert sich die Partitions-Tiefe, wenn die Daten gleichmäßig verteilt sind und jede Partition gleich gefüllt werden. Liegt dagegen nur ein einziger Wert für alle Datensätze vor, wird unter Umständen die maximale Partitions-Tiefe erreicht, da kein Wert in eine anderen Partition übertragen wird. Auf das Probing wirkt sich das Verhalten ähnlich aus. Für die Werte von R, wird eine Hashtabelle angelegt. Liegen die Daten sehr konzentriert für einem Wert vor, entstehen lange Ketten in der Struktur und ein Thread muss diese Kette vollständig abarbeiten, um den Join zu berechnen. Bei einer gleichmäßigen Verteilung, wird auch die Hashtabelle gleichmäßig gefüllt, was die Reduzierung der Kettenlänge zur Folge hat. Verteilungen spielen daher eine wichtige Rolle, in der Evaluation von den folgenden Algorithmen

4.3.2 Probe

Die zentrale Aufgaben ist das Finden von Paaren, welche in den ausgewählten Spalten übereinstimmen. Für diese Aufgabe wurde der Hash-Join ausgewählt. Im Hash-Join findet das Probing auf einer Hashtabelle von R statt, welche in drei unterschiedlichen Varianten, für die GPU implementiert und in den vorherigen vorgestellt wurde. Ziel dieses Abschnittes ist es daher, die Varianten gegenüber zu Stellen und äußere Einflüsse, wie Kernel-Parameter und Dateneigenschaften zu vergleichen. In allen drei Formen folgt zu erst die Build-Phase, danach die Probe-Phase und zum Schluss die Extraktions-Phase. Die ersten Zwei, sind mal als einzelner oder als zwei Kernel implementiert. Für die Zeitmessungen, wird die Zeit erst nach der Probe-Phase gestoppt. Für den Extraktions-Schritt erfolgt eine separate Zeitmessung. Die vollständige Zeitmessung umfasst alle Schritte und die Allokation der Buffer, welche für die einzelnen Schritte erforderlich sind. Da diese unter Umständen wiederverwendet werden können, erfolgt eine unterschiedliche hohe Anzahl an Durchläufen mit der gleichen Konfiguration, um den Einfluss der Allokation zu messen. Neben der Anzahl der Durchläufe, spielen die bereits genannten Parameter, wie Verteilung und Eigenschaften von R und S, einen wichtige Rolle. Jeder Kernel besitzt die gleichen Start-Parameter, wie beim Hashing. Diese sind Threads pro Block und Elemente pro Thread. Die Kombination aus beiden, geteilt durch den gesamten Datensatz, ergibt die Block-Anzahl. Streams und die dadurch entstehende parallele Verarbeitung, wird in den folgenden Abschnitten nicht untersucht und ist Teil der vollständigen Join-Evaluation.

R / S Abhängigkeit

Neben den Parametern für die einzelnen Kernels, definieren unterschiedliche weitere Parameter, die Ausführungsform des Probings. Einer davon ist die Buffer-Größe. In Abbildung 4.20 reicht diese bis 100.000 Elemente. Auf der X-Achse sind alle validen Konfigurationen, aus den Versuchen zu sehen. Die Y-Achse gibt die Tuples pro Sekunde an. Wie es unschwer zu erkennen ist, steigt der Durchsatz mit jedem weiteren Element an. Der Peak ist bei 5.000 Elementen in der S-Tabelle erreicht. Die Schwankungen für jeden Eintrag, sind durch die einzelnen Kernel- und Datenkonfigurationen zu erklären. Da das Diagramm alle Modi zusammenfasst, ist nicht zu erkennen, welcher Modus für den Peak verantwortlich ist. Diese Thematik wird in einem späteren Abschnitt aufgegriffen. Die genannte Abbildung stellt nur die S-Größen vor, in der Abbildung 4.20b, wird als weitere Dimension, die R-Größen hinzugefügt. Die Tupel-Rate ist nun durch Farben gekennzeichnet. Um so dunkler das Quadrat, desto höher ist die Tupel-Rate. In



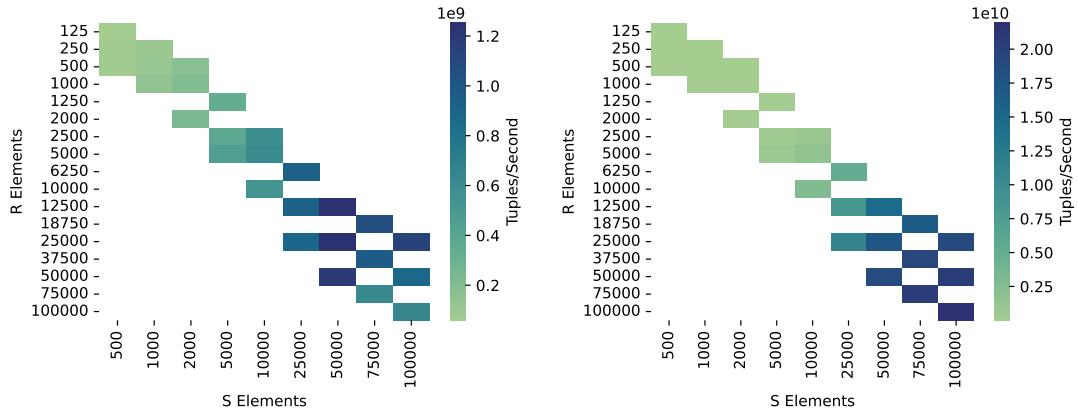
(a) Tuples/Sekunde in Abhängigkeit zur Elementanzahl
(b) Tuples/Sekunde in Abhängigkeit zur Größe von R und S

Abbildung 4.20: Tuples/Sekunde in Abhängigkeit zur Buffer-Größe

diesem Versuch wurden für die Konfiguration nicht jede Kombination zwischen R und S getestet. Die möglichen Kombinationen, sind auf Größen für S, auf 1.000 bis 100.000 Elemente beschränkt und für jede Größe, gibt es einen RS_{Scale} -Faktor, welcher die Größe von R bestimmt. Dieser kann 1, 2 oder 4 betragen. Durch die gewählten Parameter lassen sich die Lücken in der Heatmap erklären. Das Diagramm zeigt deutlich, dass die Tupel-Rate bis zu 50.000 Elementen im S-Buffer wächst und danach stagniert. Die Größe von R lässt darauf schließen, dass bei einem sehr großen Wert für S und einen kleinen Wert für R, die Tupel-Rate steigt. Was aus Datenbanksicht heißt, dass der Join zwischen zwei sehr unterschiedlich großen Tabellen schneller läuft, als bei gleich großen Tabellen.

Verteilung

Da nicht nur die Tabellen-Größe eine Rolle spielt, sondern auch die Verteilung der Werte in jeder Spalte, folgt als nächstes ein Blick auf die bereits genannten Verteilungen und deren Skew. Der Versuch wurde auf einem etwas reduzierten Datensatz getestet und umfasst nur die Parameter Skew, maximale Elemente für S und der Maximalwert für die Verteilung. Da alle Parameter in Kombination getestet wurden, ist es zu erwarten, dass die Verläufe in Abbildung 4.1, in etwa die gleichen Verteilungen wiedergeben. Die Zipf- und Gleichverteilung sind als separate Farben angeben und nebeneinander dargestellt. Beide Funktionen verfolgen ein anderes Ziel, was mit fortschreitend Skew, auf der X-Achse sichtbar ist. Für die Y-Achse wurde die Tupel-Rate gewählt, um den Einfluss auf diese aufzuzeigen. Wenn bei der Gleichverteilung der Skew angehoben wird, reduziert sich die Überlappung zwischen R und S. In der Darstellung hat dies einen Anstieg, der Tupel-Rate zur Folge. Das ist nicht nur durch den Peak zu sehen, sondern auch durch die Zerrung der Verteilungsfunktion. Im Vergleich zwischen 0.75 und 0.0, ist das untere Segment bei 0.75 deutlich dünner. Das liegt hauptsächlich daran, dass für eine sehr kleine Ergebnis-Tabelle, der Extract-Kernel nur noch wenig erledigen muss und bei 1.0 sogar komplett weg fällt. Ohne dem Kernel ist auch keine Allokation der Tabelle nötig, welche zusätzlich ins Gewicht fallen würde. Mit der Zipf-Verteilung tritt genau der umgekehrte Effekt auf. Bei steigenden Skew reduziert sich die Vielfalt in der S-Tabelle und es treffen viel mehr Elemente auf genau einen R-Wert. Des weiteren werden die Ketten in der Hashtabelle deutlich länger, wenn viele Hashes den gleichen Wert besitzen. Beide Faktoren zusammen ergeben



(a) Build/Probe Kernel Tupel-Rate, in Abhängigkeit von R und S
(b) Extract Kernel Tupel-Rate, in Abhängigkeit von R und S

Abbildung 4.21: Einfluss der Probing Kernel-Parameter auf die Laufzeit

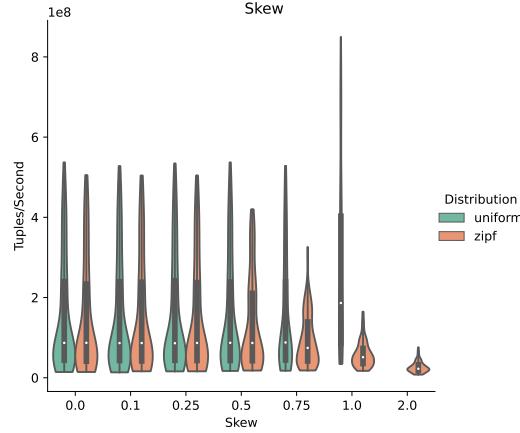


Abbildung 4.22: Probing mit unterschiedlichen Verteilungen

erschwerte Bedingungen. Die einzelnen Threads im Probing-Kernel müssen deutlich längere Ketten ablaufen und der Extract-Kernel eine deutlich größere Ergebnis-Tabelle zusammensetzen. Hinzu kommt die Allokation der Tabelle. Anhand der gezeigten Ergebnisse wird deutlich, wie entscheidend eine Verteilung in einem Datensatz auf die Geschwindigkeit Einfluss nimmt. Für dieses Problem gibt es allerdings keine Lösung, welche den Algorithmus an den Skew anpassen kann. Neben der Verteilung, können die Tabellen sich auch in zwei weiteren Punkten unterscheiden. Zum einen ist das die Anzahl der Spalten und zum anderen das Verhältnis zwischen den Tabellen-Größen. Beide Eigenschaften wurden aus den vorherigen Experimenten extrahiert und in Abbildung 4.23 dargestellt. Für die Spaltenanzahl, wurden jeweils die Werte Eins und Zwei getestet. Bei zwei Spalten müssen immer beide Werte, mit den jeweils in der anderen Tabelle vorliegenden Wert übereinstimmen. Diese Operation entspricht somit dem logisch Und. Diese Variante erfordert zum einen mehr Speicherplatz und erfordert eine Iteration mehr im Probe und Hash-Kernel. Im späteren Verlauf auch im Swap-Kernel. Die einzelnen Konfigurationen sind auf der X-Achse abgetragen und stehen in Relation zu der Tupel-Rate auf der Y-Achse. Es ist zu sehen, dass ein weitere Spalte einen Einfluss auf die Laufzeit ausübt, dieser allerdings nur gering ausfällt.

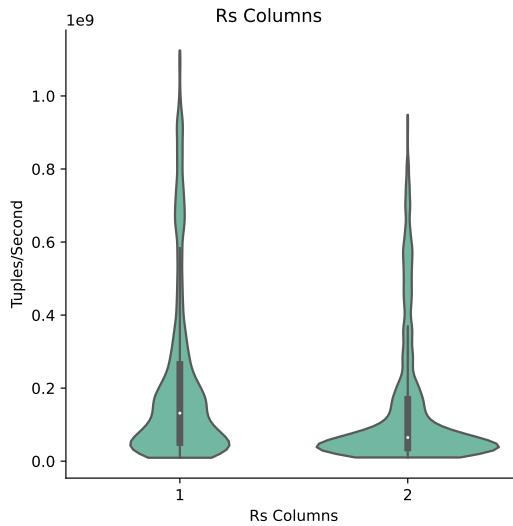


Abbildung 4.23: Probing mit mehreren Spalten

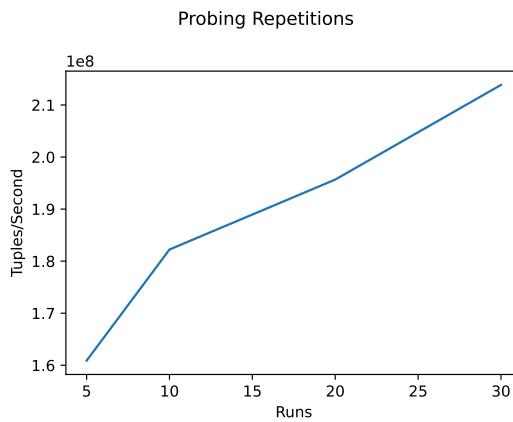


Abbildung 4.24: Tuples/Sekunde in Abhängigkeit zur Elementanzahl

Buffer-Optimierung

Das Probing findet in der Regel häufiger statt, wie es auch bei der Vektorisierung der Fall ist. Hierfür wäre es nicht sehr effizient, wenn die temporären Buffer immer wieder neu angelegt werden. Aus diesem Grund bleibt der Buffer für eine bestimmte Probe-Konfiguration, immer so lange erhalten, bis keine größere Anfrage einen größeren Buffer benötigt oder die Konfiguration gelöscht wird. Diese Verhalten wurde mit einem Datensatz getestet, welcher N -mal das Probing aufgerufen hat, ohne die Buffer in der Größe zu verändern. Dieser Parameter wird "Runs" genannt und ist in Abbildung 4.24 auf der X-Achse angegeben. Als Kennwert dient die Tupel-Rate. Es ist zu sehen, dass bei steigender Wiederholung, auch die Tupel-Rate steigt. Der erste Wert in der Kette allokiert immer die genannten Buffer und erzeugt somit die größte Latenz. Alle nachfolgenden Aufrufe können die Buffer wiederverwenden und ziehen somit die Tupel-Rate langsam nach oben, was in der Darstellung gut zu sehen ist.

Kernel-Parameter

Jede Funktion, egal mit welchen Probe-Modus, hat eine Probe/Build- und Extraktions-Phase. Für beide Phasen gibt es die Kernel-Konfiguration Threads pro Block und Elemente pro Thread. Für den Global-Probe-Modus gibt es zwei separate Kernel, welche jeweils eine Phase reprä-

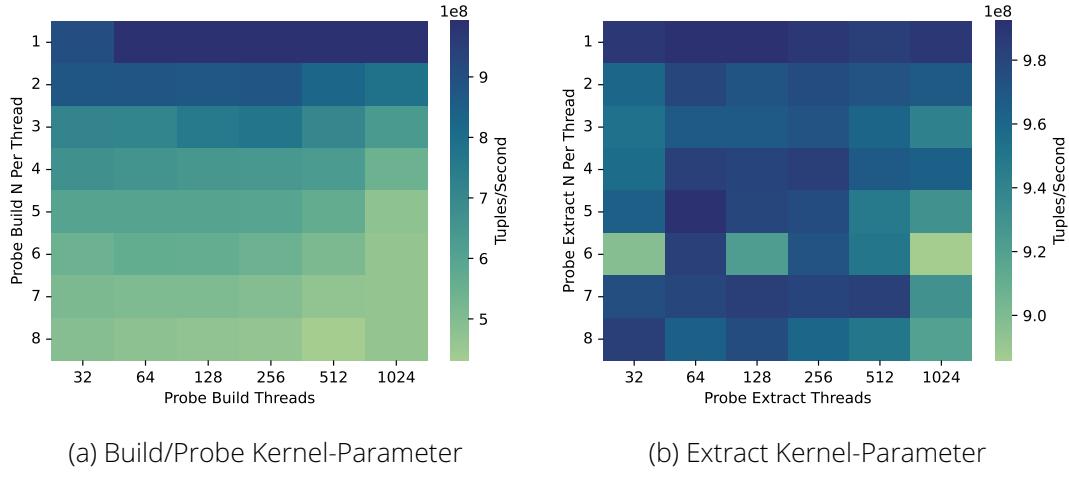


Abbildung 4.25: Einfluss der Probing Kernel auf die Laufzeit

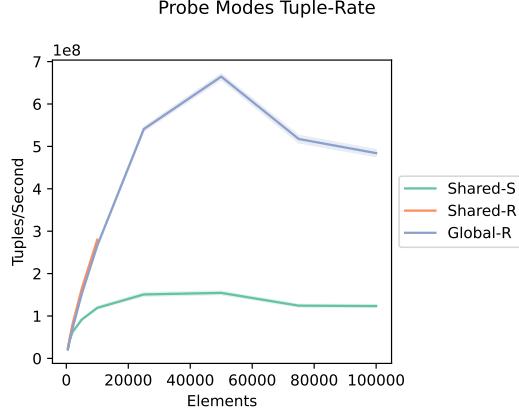


Abbildung 4.26: Tuples/Sekunde in Abhangigkeit zum Probe-Modus

sentieren, jedoch verwenden diese die gleiche Probe/Build Konfiguration. Genau wie beim Hash-Kernel, werden so viele Blocke verwendet, dass jeder Thread die definierten N Elemente abarbeiten kann. Um die beste Konfiguration zu finden, gibt es eine Gegenuberstellung bei der Parameter, fur den jeweiligen Kernel in Abbildung 4.25. Links in Abbildung 4.25a, sind die besten Kombinationen fur die Build- und Probe-Phase zu sehen und rechts daneben, die besten Kombinationen fur die Extraktions-Phase (Abbildung 4.25b). In beiden Abbildungen ist der beste Wert, durch die hochste Tuple-Rate zu erkennen. Der Build- und Probe-Kernel arbeitet am schnellsten mit nur einem Element pro Thread. Die Anzahl der Threads spielt fast keine Rolle, da sich die maximale Thread-Anzahl nur leicht abhebt. Auch fur die Extraktion reicht ein Element pro Thread, jedoch liegen andere Konfigurationen sehr dicht beieinander, sodass die Auswahl fast keine Rolle spielt. Nur im Bereich von sechs Elementen, liegen einige Werte deutlich unter den anderen Werten, weshalb diese Konfiguration zu vermeiden ist. Zusatzlich neigt die Thread-Anzahl von 1024 zu einer schlechteren Tupel-Rate und sollte daher auch nicht konfiguriert werden.

Probe Modus

Aus den vorherigen Abschnitten sind bereits die drei Probe-Modi bekannt. Jeder hat seine eigene Limitierung und deshalb sind auch unterschiedliche Ergebnisse zu erwarten. Eine Gegenuberstellung findet in Abbildung 4.26 statt. Die einzelnen Modi sind in Abhangigkeit zur

Element-Anzahl angegeben und durch die Tupel-Rate auf der Y-Achse eingeordnet. Die Shared-R und Global-R-Varianten liefern für alle Konfigurationen Ergebnisse. Nur die Shared-S-Methode ist limitiert, durch die maximale Hashtabelle-Größe. Alle angegeben Werte repräsentieren gemessene Maxima. Da jede Konfiguration fünfmal ausgeführt und gemittelt wird, ist somit nicht von Ausreißern zu sprechen. Die Shared-R-Variante erzielt einen Höchstwert von rund 300 Millionen Tupeln pro Sekunde. Darunter liegt die Shared-S-Variante mit rund 100 Millionen Tupeln. Beide Ergebnisse zusammen unterliegen dem Maximum von der Global-R-Variante, mit 650 Millionen Tupeln pro Sekunde. Es fehlt natürlich das Verhältnis zum R-Wert, dieser ist je nach Konfiguration nur ein Viertel oder genau so groß wie der Datensatz von S. Das Diagramm gibt keine vollständige Übersicht, aber liefert einen kleinen Ausblick, wie hoch die einzelnen Tupel-Raten gehen können.

4.3.3 Vektorisierung

Für das Probing wurde ausführlich jede Eigenschaft betrachtet. Jedoch kann das Probing allein, nur einen geringen Datensatz verarbeiten. Diese Limitierung wurde mit dem Vektorisieren aufgegriffen und ermöglicht den Einsatz von beliebig großen Daten, so lange der interne Speicher ausreicht. Der internen Speicher ist auf rund 48GB begrenzt und erstreckt sich nur über eine Quadro 8000 RTX. Die verwendeten Tabellen sind genau so beschaffen, wie im Probing-Abschnitt. Für die Versuche wird die Spalten-Anzahl auf nur eine Spalte begrenzt. Dies reduziert den Speicherverbrauch und ermöglicht im Zweifel, ein wenig größere Testdatensätze. Genau wie beim Probing, ist die wichtigste Einheit Tupels pro Sekunde. Jeweils ein Tupel entspricht einer Tabellen-Zeile. Das Umfasst den 8Byte Primary-Key und den 8Byte Tabellen-Wert. Für die Evaluation werden mehre Rubriken durchlaufen. Als erstes ist die Beschaffung des Datensatzes von Interesse. Darauf folgen Eigenschaften für die Vektorbildung und darauf die Übertragung auf die Multi-GPU-Algorithmen und ein Überblick zu den restlichen Nebenfunktionen.

Verteilung

Die Generierung des Datensatzes, erfolgt wie beim Probing in Abbildung 4.19. Die Testdaten erstrecken sich von 10.000 bis vier Millionen Tupel für S und zwei Millionen Tupel für R. Der höchste Spaltenwert liegt bei 100.000, wodurch nur ein Bruchteil des Zahlenbereichs von 2^{64} verwendbar ist. Das Ziel der Einschränkung ist die Erhöhung der Trefferraten, zwischen den Tupeln. Für die Vektorisierung spielt die Verteilung eine wichtige Rolle. Im ersten Schritt erfolgt die Zuteilung der Werte in die gleiche Radix-Partition. Die Partition ergibt sich aus dem Radix-Key, welcher sich mit der Tiefe verschiebt. Liegen viele Werte im gleichen Zahlenbereich, so ist es nicht unwahrscheinlich, dass diese auch in die gleiche Partition fallen. Der Radix-Key reicht allerdings nur bis zum Ende des letzten Bits. Ist der letzte Bit-Wert erreicht, erfolgt die Aufteilung mit der erweiterten Vektorbildung. Dieser Schritt ist entscheidend für das Probing. Denn liegen sehr große Partition vor, welche nicht mehr aufgeteilt werden können, folgt die Aufteilung in kleinere Datensätze. Alle R und S-Teilpartitionen werden miteinander kombiniert, sodass keine Paarung beim Probing verloren geht. Sind R und S bereits klein genug, entsteht nur ein Vektor aus beiden Datensätzen. Durch dieses Verhalten unterscheidet sich die Last, für das Vektorisieren und dem Probing signifikant.

Das Ergebnis aus dem Versuch, ist in Abbildung 4.27 für den vollständigen Join zu sehen. Die durchschnittliche Tupel-Rate lässt sich wie beim Probing, durch die Summe von R und S bilden und ist auf der Y-Achse abgebildet. Als X-Wert sind die einzelnen Skew-Werte angegeben. Jedes Teilergebnis unterliegt der jeweiligen Verteilung. Die durchschnittliche Tupel-Rate liegt unter 10 Millionen Tupeln pro Sekunde und ist somit deutlich unter dem erreichte-

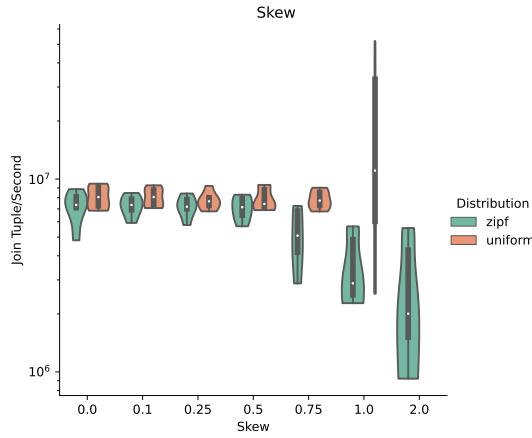


Abbildung 4.27: Einfluss der Gleich- und Zipf-Verteilung, auf den Hash-Join

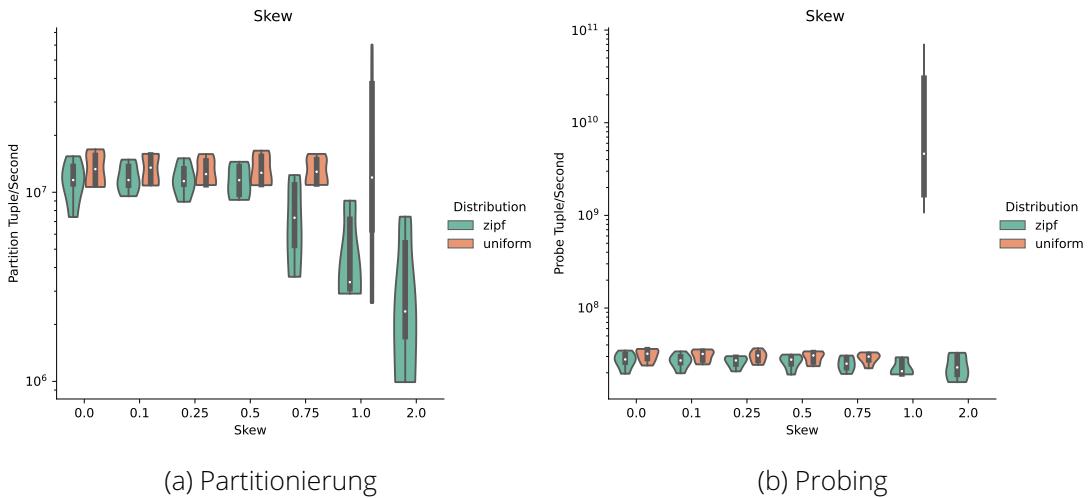


Abbildung 4.28: Einfluss der Verteilungen auf das Vektorisieren und Probing im Join

ten Wert vom Probing, aus den vorherigen Versuchen. Auch wenn die Global-R-Variante als Probing-Strategie gewählt wurde. Diese ist primär dafür da, dass die Vektoren nicht durch die Speicher-Limitationen beeinflusst werden. Daher ist zu erwarten, dass die Ergebnisse in etwa der gleichen Größenordnung liegen. Unterschiede sind durch die neuen Funktionen und Ressourcen zu erwarten gewesen allerdings nicht in dieser Größenordnung. Der Probing-Test hat nur die Funktion und deren Kernel getestet. Funktionen wie das Hashing oder die Vektorisierung haben für die Messung keine Rolle gespielt. All diese Faktoren wirken sich negativ auf die Join-Laufzeit aus. Die Zeitmessung startet daher bereits vor dem Hashing und endet, wenn alle benötigten Buffer wieder gelöscht sind, um ein Gesamtbild für den Hash-Join zu schaffen.

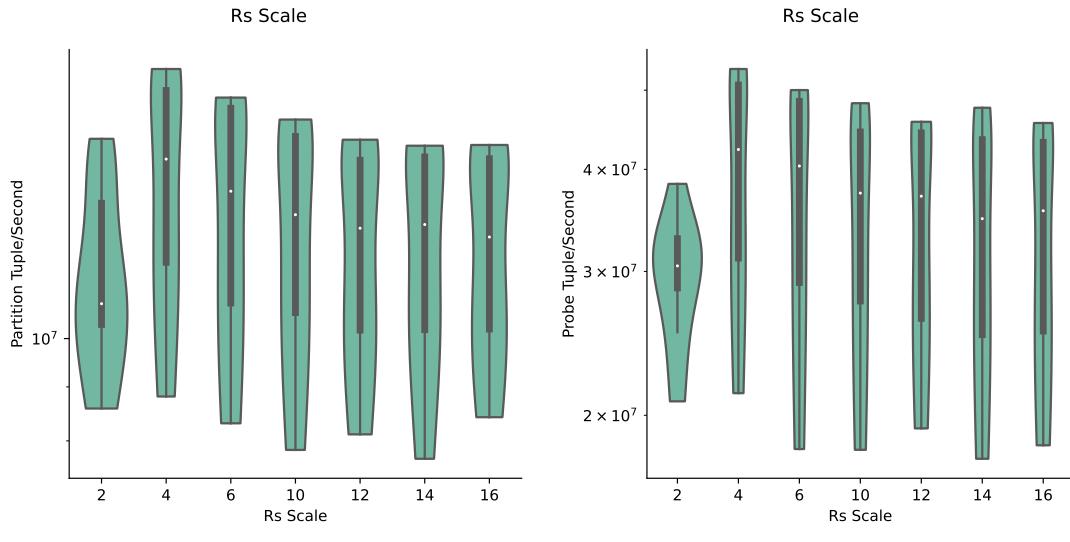
Dass die Allokation von Ressourcen kein großer Faktor ist, ist in Abbildung 4.28a dargestellt. Hier liefert die Partitionierung und die damit verbundene Vektorisierung genau das gleiche Ergebnis, wie es für den gesamten Join zu sehen ist. In diesem Kernel werden keine neuen Daten, außer die resultierenden Vektoren angelegt, welche nur die Speicheradressen beinhalten und daher keiner wirklichen Allokation von Buffern entspricht. Durch die geringe Tupel-Rate von der Vektorisierung, kann der Hash-Join auch keine höhere Tupel-Rate erreichen. Allgemein gilt, dass der niedrigste Wert von allen vier Teil-Prozessen, auch die Laufzeit des Hash-Joins bestimmt. Für das Probing in Abbildung 4.28b liegen die Tupel-Raten etwas höher, aber die Größenordnungen stimmen auch nicht mit den Werten aus den vorherigen Tests überein.

Hier liegt der Grund in der Vektor-Menge. Erhöht sich die Vektor-Anzahl nicht proportional zur Tabellengröße von R und S, sinkt oder steigt die Tupel-Rate für das Probing. In diesem Fall sinkt die Tupel-Rate, da sehr viele einzelne Vektoren getestet werden müssen und die Größe von R und S nicht steigt. Mit Blick auf die Verteilung, ist diese Eigenschaft für das Probing erkennbar. Durch die Gleichverteilung ändert sich nicht viel, bis zu einem Skew von 0.75, aber mit 1.0, steigt die Tupel-Rate, da nun keine Werte übereinstimmen. Durch das Hashing wird der Skew verzehrt, da es trotzdem zu Überlagerungen kommt, wenn Hashes kollidieren oder in den gleichen Vektor zugeordnet werden. Würde eine Hash-Funktion jeden Wert auf Eins abbilden, dann hätte der Skew keinen Einfluss. In diesem Fall gäbe es eine 100% Kollisions-Rate, wodurch alle Werte in den selben Vektor geworfen werden, egal wie weit die Original-Werte auseinander liegen. Das Ergebnis ist eine extrem steigende Anzahl an Vektoren und somit eine sehr geringe Tupel-Rate für das Probing. Abschwächt ist das Verhalten für einen Skew von 1.0 zu sehen. Auch wenn die Hash-Funktion eine niedrige Kollisions-Rate besitzt, kollidieren einzelne Bit-Werte, da diese nur Null oder Eins betragen. Daher findet auch ohne Übereinstimmung des Wertebereichs, das Probing über bestimmte Wert statt. Nur innerhalb des Probing erfolgt der frühere Ausschluss von Werten, da die Hashes nicht übereinstimmen.

Als alternative Verteilung dient die Zipf-Verteilung, mit der Konzentration der S-Tabelle auf einen bestimmten Wert. Aus dem Skew verschieben sich somit die Partitionen auf einen bestimmten Wertebereich und einzelne Radix-Keys bleiben leer. Als Folge ist für den Join eine leichte Steigerung, aber auch eine deutliche Reduzierung der Tupel-Rate sichtbar. Die Schwankungen sind eine Folge der zufälligen Generierung. Der Wertebereich ist gerade bei kleineren Datensätzen nicht vollständig ausgefüllt, wodurch bei der Gleichverteilung von R, nicht immer alle Möglichkeiten abgedeckt sind. Als Folge gibt es für den Peak von S, keine Treffer und ein signifikanter Probing-Aufwand fällt weg. Die Erhöhung der Tupel-Rate ist die Folge. Wenn der Peak von R getroffen wird, dann fallen Tests für sehr viele Werte von S an, was im Zweifel viele Vektoren bedeutet und dadurch die Tupel-Rate reduziert. Die Last beim Partitionieren erhöht sich zusätzlich, da nun viele Werte die gleichen Bit-Werte aufweisen, weshalb es nach vielen Schritten noch keine passende Aufteilung für die Hashes gibt. Bereits der Blick auf die Partitionierung und Vektorisierung, zeigt den negativen Einfluss von der Zipf-Verteilung. In der Gleichverteilung liegt bei egal welchen Skew, eine sinnvolle Bit-Verteilung vor. Zipf dagegen, konzentriert die Werte auf einen bestimmten Bit-Wert an bestimmten Positionen und verhindert somit eine schnelle Aufteilung.

Der Vergleich zwischen Gleichverteilung und Zipf-Verteilung lässt darauf schließen, dass ein Skew von Null, jeweils eine gleiche Gleichverteilung wiedergibt. In den Abbildungen ist jedoch zu sehen, dass sich beide Verteilungen bei einem Skew von Null in der Tupel-Rate unterscheiden. Der Unterschied lässt sich daraus begründen, dass die Zipf-Verteilung an den Enden keine saubere Gleichverteilung abbildet. Es ist jeweils in Abbildung 4.19b, ein kleiner Knick zu sehen. Dieser lässt darauf schließen, dass sich Werte nicht über den kompletten Wertebereich gleich verteilen und ein wenig konzentrierter vorliegen.

R und S sind wie bereits angemerkt, nicht immer gleich groß. Die R-Tabelle ist bei Größenunterschieden immer die kleinere Tabelle. Der Unterschied zwischen den Tabellen-Größen, ist die letzte Verteilungs-Eigenschaft, welche betrachtet wird. Abbildung 4.29 betrachtet unterschiedlichen RS-Konfigurationen für das Vektorisieren und dem Probing. Hierbei liegen jeweils Messwerte, für eine Unterschied von 2 bis 16 vor. Die Zahlen entsprechen dem Faktor, welcher der Datensatz R, kleiner als S ist. Das Probing aus dem letzten Abschnitt hat deutliche Unterschiede zwischen den einzelnen Werte aufgezeigt, allerdings verschwinden sie beim Hash-Join mit der Vektorisierung. Es gibt einen leichten Peak bei einem RS-Scale von Vier, welcher sich aber nur gering von den anderen Werten unterscheidet. Nur bei gleicher Größe liegen die



(a) Einfluss auf das Partitionieren

(b) Einfluss auf das Probing

Abbildung 4.29: Unterschiedliche RS-Verhältnisse im Vergleich

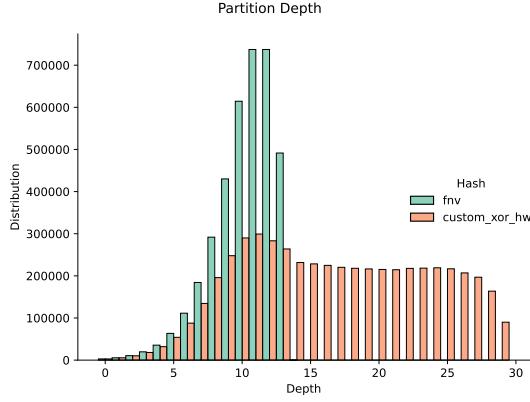


Abbildung 4.30: Der Einfluss der Hash-Funktion auf die Partitionstiefe

Ergebnisse aus allen Konfigurationen dichter beieinander und ergeben insgesamt eine etwas schlechtere Tupel-Rate. Das Verhalten für den RS-Scale ist auf die resultierenden Matches zurückzuführen. Mit der Menge an Tupels, steigen die Vektoren proportional an. Dadurch erhöht sich der Aufwand bei der Vektorisierung und beim Probing proportional. Dass bei der Erhöhung von RS-Scale, die Werte nicht besser werden, folgt aus der reduzierten Menge an Tupels. Die GPU arbeitet am effizientesten, wenn ein Kernel sehr große Datenmengen verarbeitet. Bei geringeren Datenmengen nimmt die Effizienz ab, was sich auf die Tupel-Rate niederschlägt.

Partitionen

Ein weitere Aspekt sind die Konfigurationen, welche die Hash-Funktion und die Anzahl der Partitionen betreffen. Der Einfluss der Hash-Funktion ist in Abbildung 4.30 visualisiert. Die Informationen sind aus einer vollständigen Konfiguration extrahiert und bilden daher nicht die wirkliche Partitionen-Anzahl für einen Durchlauf ab, sondern nur einen Summe aus allen Konfigurationen. Auf der X-Achse ist Partitionstiefe angeben, welche die Anzahl der Partitionen mit der jeweiligen Tiefe angibt. Für den Versuch wurden nur zwei Partitionen verwendet, weshalb jeder Schritt, einem weiteren Bit im Zahlenwert entspricht. Der Vergleich zwischen den beiden Hash-Funktionen verdeutlicht, wie wichtig eine gute Bit-Verteilung für das Partitionie-

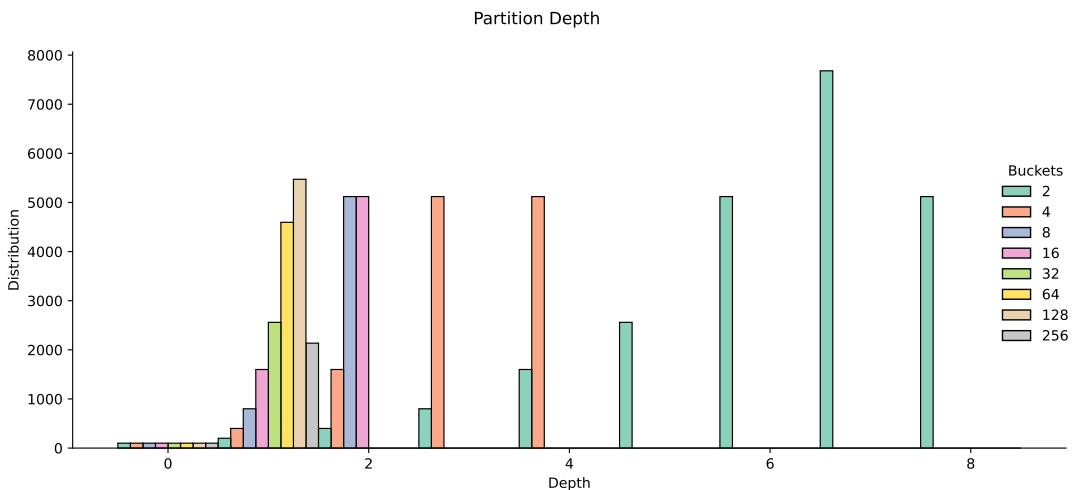


Abbildung 4.31: Der Einfluss der Partition-Buckets auf Partitionstiefe

ren ist. Die FNV-Funktion verteilt die Bits bereits in den ersten zehn Bits gleichmäßig, wodurch frühzeitig die gewünschte Partitions-Größe erreicht ist. Das XOR-HW-Verfahren hat in den vorherigen Analysen sehr gute Werte erzielt, allerdings liegt bei dieser Funktion, eine schlechte Bit-Verteilung bei den ersten Bits vor. Als Folge gibt es für gewisse Tiefen keine sinnvolle Aufteilung und die Suche läuft doppelt so lang, als beim FNV-Verfahren. Aus diesem Grund, wird die FNV-Funktion für weitere Tests verwendet.

Ein weiteres Beispiel für die Verteilung, ist die Partitions-Anzahl in Abbildung 4.31, welche auf der X-Achse zu finden ist. Als Schrittweite gibt es die Konfigurationen 2 bis 256, wobei jeder Zwischenschritt, eine Zweierpotenz ist, wie es für den Radix-Key erforderlich ist. Es ist leicht zu sehen, dass sehr große Partitions-Anzahlen, auch schneller eine finale Tiefe erreichen. Mit der Erhöhung der Partitionen, verlängert sich auch der Radix-Key. Durch die Vergrößerung ist es möglich, die Hashes breiter zu verteilen, was die Partitionen schneller auf die geforderte Vektorgröße reduziert. Der Einfluss auf die Geschwindigkeit, ist in Abbildung 4.32 zu sehen. Bei der Interpretation der Grafik ist zu beachten, dass nur die Partitionen, welche einen Kernel-Aufruf zur Folge haben, gezählt werden. Daher fehlt immer die letzte Tiefe, welche nicht mehr aufgeteilt wird.

Die Partitionierung zeigt deutlich erhöhte Tupel-Raten, wenn die Partitionen-Anzahl erhöht wird. Dies gilt nicht für jede Erhöhung, was zwischen den Werten 8 und 32, erkennbar ist. Der Grund für die Partitionierung lässt sich aus den vorherigen Abbildungen erklären. Mit größeren Partitionen, werden die Daten schneller aufgeteilt und dadurch endet die Operation schneller. Zusätzlich bleiben bei geringeren Tiefen, die einzelnen Partitionen für den Kernel größer, wodurch weniger Kernel mit größeren Datensätzen gestartet werden. Auf das Probing hat die Konfiguration keinen Einfluss, da es hauptsächlich durch die Beschaffenheit der Daten beeinflusst wird.

Die Partitionstiefe ist allerdings nicht nur schlecht für die Laufzeit an sich, sondern führt in der Regel auch dazu, dass die Kernel auf geringeren Datensätzen arbeiten. Für diese Analyse sind in Abbildung 4.33, Der Swap- und Histogramm-Kernel, mit Bezug auf die Datenmengen dargestellt. Die Daten stammen aus der Streaming-Konfiguration und gelten aber auch für alle Konfigurationen. Um die Laufzeit der Kernels zu verstehen, ist es wichtig die Datengrundlage der Kernels zu betrachten. Diese ist auf der X-Achse abgebildet und über der Punktwolke in einer Verteilung zusammengefasst. Diese gibt an, wie viele Kernel mit der Datenmenge gestar-

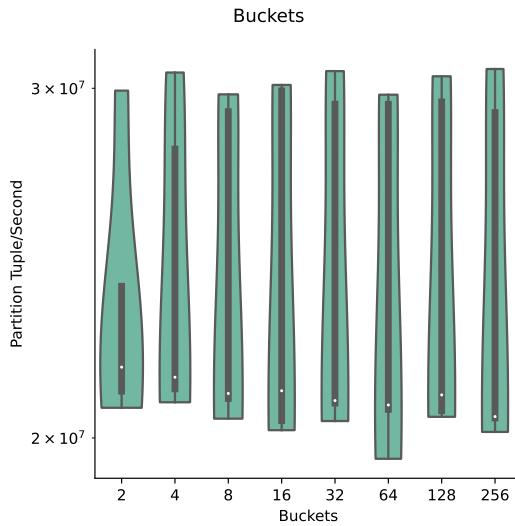


Abbildung 4.32: Unterschiedliche Partitionenanzahl bei der Partitionierung

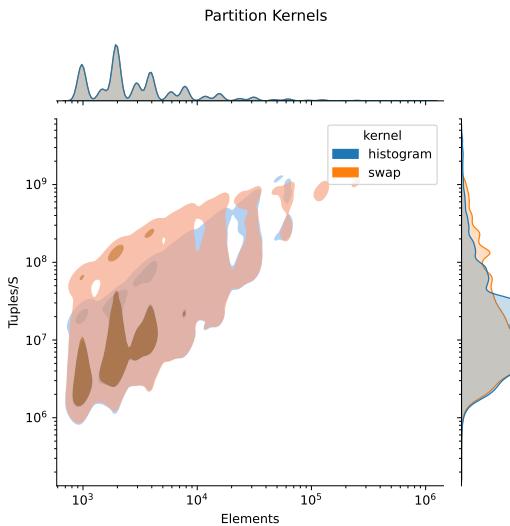


Abbildung 4.33: Histogram- und Swap-Kernel im Vergleich

tet sind. Daraus ergibt sich auf der Y-Achse die Tupel-Rate und die Verteilung, der Tupel-Raten auf der anderen Seite. Es zu sehen, dass viele Kernel mit relativ kleinen Datensätzen arbeiten müssen und dadurch die Tupel-Rate sehr niedrig bleibt. Diese Darstellung erklärt, wieso die Partitionierungs-Phase nicht die gewünschten hohen Tupel-Raten erreichen kann.

Streaming

Die Partitionierung und das Probing haben jeweils das Potential, einzelne Datensätze parallel abzuarbeiten. Für diesen Fall eignen sich Streams, um auch die Kernel parallel zu starten. Jede gemeinsame R- und S-Partitionierung und jedes Vektor-Probing findet jeweils in einem Stream statt. Für diesen Zweck wird zu Beginn, ein Pool an Streams und Threads allokiert und bereitgestellt. Die Verteilung findet im Round-Robin-Verfahren statt, sodass jeder Stream gleichmäßig verwendet wird. Diese Gleichmäßigkeit ist in Abbildung 4.35 visualisiert. Die Abbildung zeigt einen Auszug aus einem Benchmark. Jede Partition zeichnet für diesen Zweck, den zugeordneten Stream auf, welche im Anschluss zu akkumuliert sind. Aus dem Diagramm

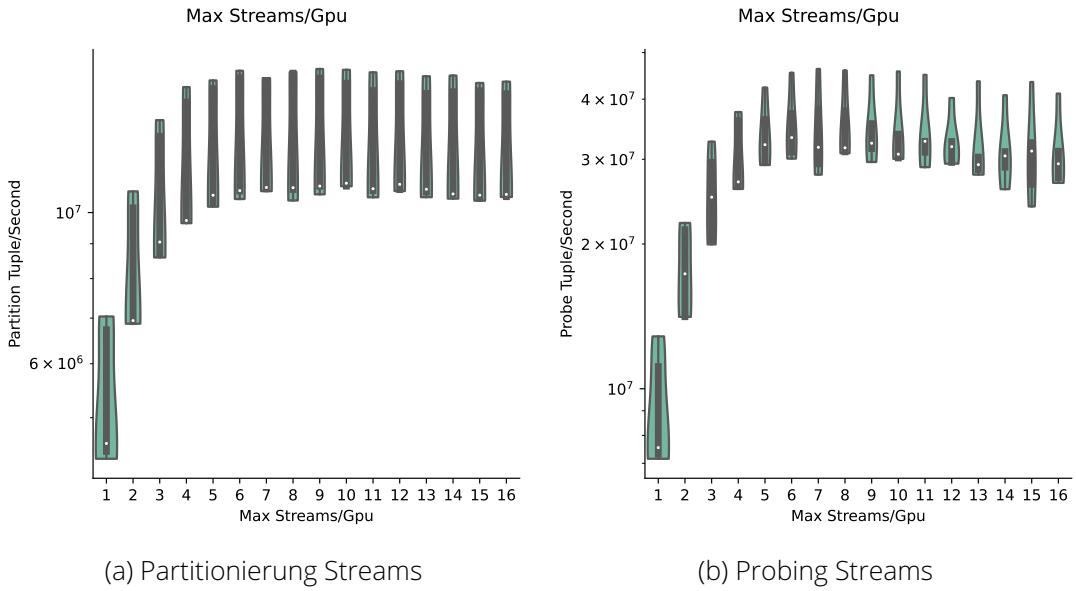


Abbildung 4.34: Einfluss der Stream-Anzahl auf die Tupel-Rate

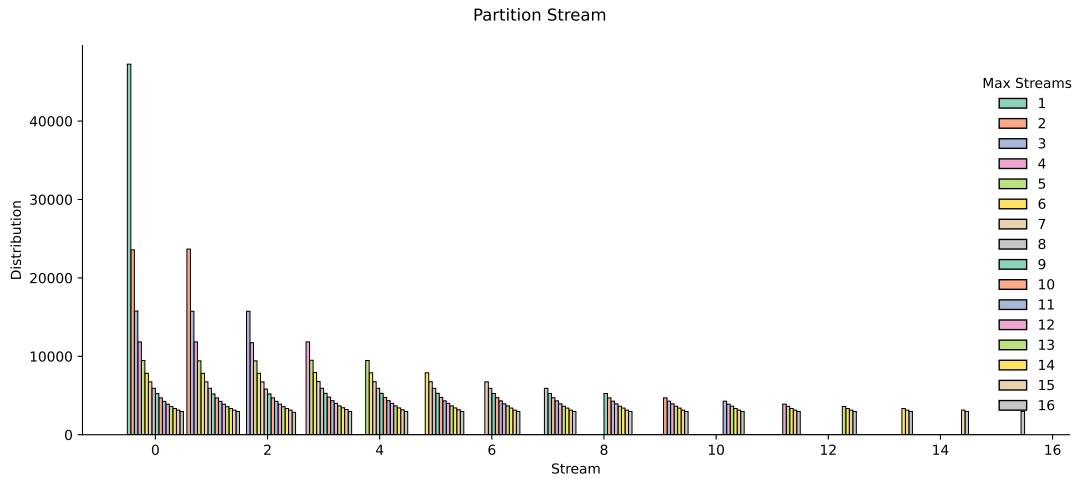
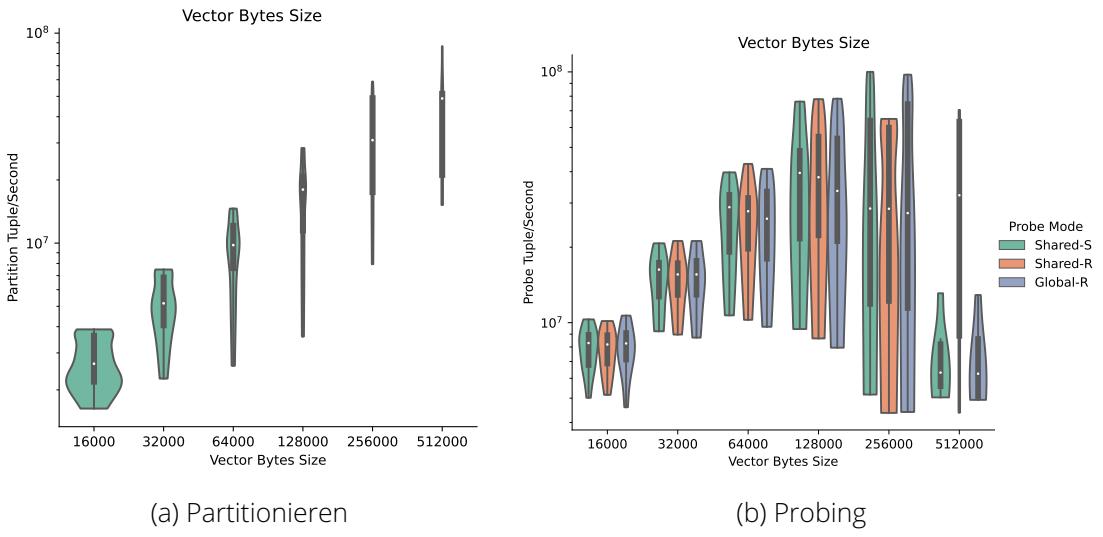


Abbildung 4.35: Stream-Verteilung in der Partitionierung

geht daher hervor, dass jeder Stream von gleich vielen Partitionen verwendet wird, egal bei welcher maximalen Stream-Konfiguration. Neben der parallelen Ausführung, entsteht bei der Synchronisierung ein Vorteil. Wenn auf eine bestimmte Operation gewartet werden muss, erfordert dies nur noch das Warten auf den Stream und nicht auf die vollständige GPU. Um die beste Stream-Konfiguration zu finden, wurde eine Konfiguration mit 1 bis 16 Streams getestet. Das Ergebnis in Abbildung 4.34 deutet sowohl für die Partitionierung, als auch für das Probing, auf ein Optimum bei acht Streams. Danach reduziert sich die Tupel-Rate ein wenig und davor steigt sie wie eine Wurzelfunktion an. Der Anstieg ist möglich, da noch nicht alle GPU-Ressource beansprucht sind, wodurch weitere Kernel auf der GPU ihren Platz finden. Die Sättigung findet dann statt, wenn die GPU keinen freien Kapazitäten aufweisen kann. Der leichte Einbruch, ist die Folge der Überlastung des Schedulings auf der GPU. Daher gilt, dass viele Streams nicht unbedingt eine Leistungssteigerung bedeuten und dies immer von den verwendeten Kernel-Ressourcen abhängig ist.



(a) Partitionieren

(b) Probing

Abbildung 4.36: Einfluss der Vektorgröße auf die Funktionen

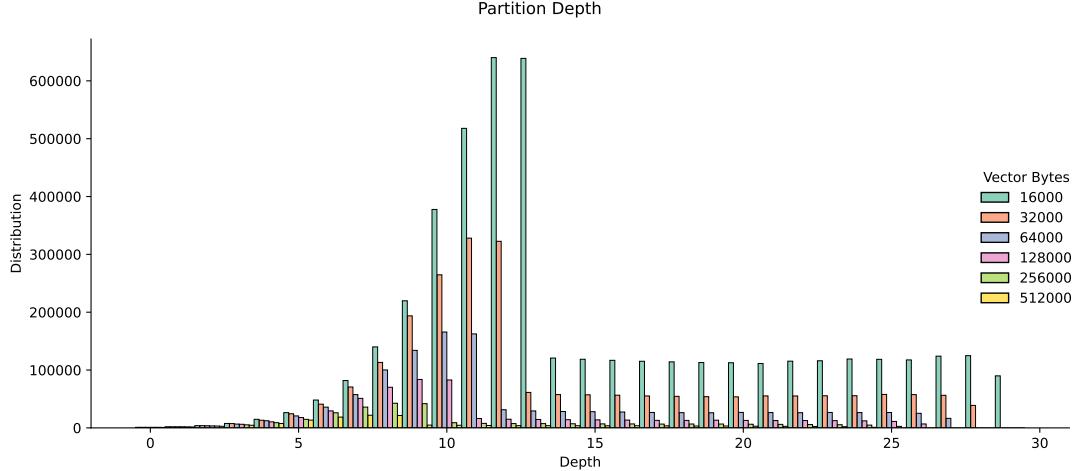


Abbildung 4.37: Partitionstiefen in Abhängigkeit zur Vektorgröße

Vektorgrößen

Das Ziel der Partitionierung ist die Reduzierung von R und S, auf eine maximale Größe. Die maximale Größe entspricht dem Konfigurations-Parameter "Vector Bytes Size" und gibt die Vektorgröße in Bytes an. Für die Durchführung liegen diese zwischen 16.000 und 512.000 Bytes und es sind alle Probe-Modi involviert. Was das genau für die maximalen Elemente pro Vektor bedeutet, ist durch die Abbildungen 7.1 im Anhang aufgeschlüsselt. Die einzigen Parameter die sich für jede Konfiguration ändern, sind die Elemente in der R- und S-Tabelle, wobei das Verhältnis immer gleich ist. Mit der Vektorgröße steigt auch die Tupel-Rate. Diese Verhalten ist bereits aus vorherigen Diagrammen sichtbar gewesen und wird in Abbildung 4.37, in Bezug auf die Vektorgröße verdeutlicht. In dem Diagramm werden für einen Versuch alle Partitionen bezüglich ihrer Tiefe und maximalen Vektorgröße aufgelistet. Dabei erreichen die größten Vektoren nur eine geringe Tiefe und sehr kleine Vektoren, erreichen fast die Hälfte des maximalen Radix-Shifts. Die Tiefen entsprechen den Werten aus Abbildung 4.30, da beide Diagramme aus dem gleichen Versuch stammen. Im Vergleich zeigt sich, dass auch bei großen Vektoren, die Hash-Funktion die Verteilung dominiert. Der Peak bei der Partitionierung stammt aus der Elementanzahl. Da ab einer bestimmten Vektorgröße es vorkommen kann, dass R und S bereits komplett hinein passen, ist der Partitionierungs-Schritt nicht notwendig und liefert daher den

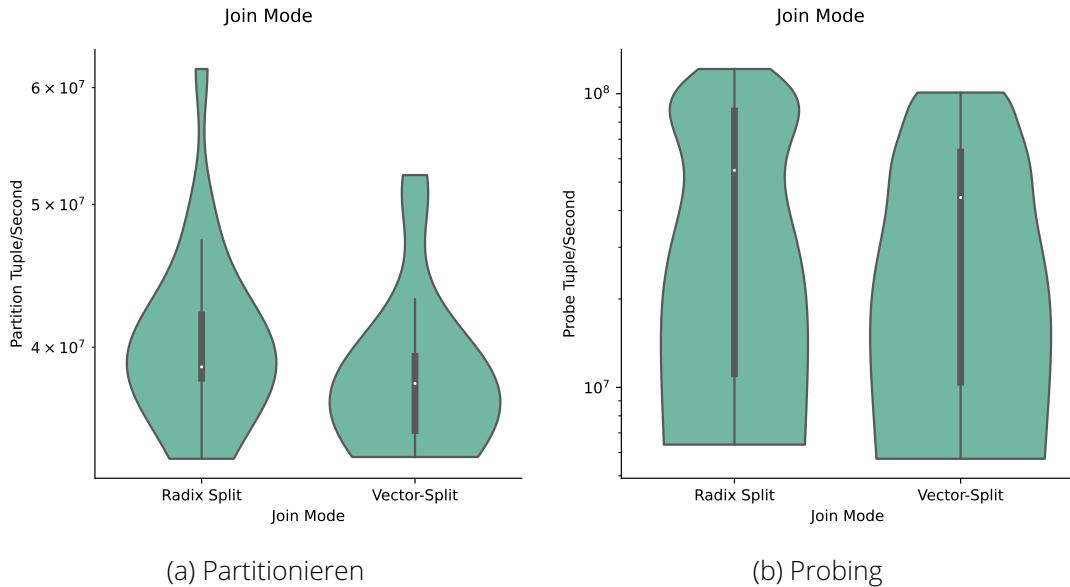


Abbildung 4.38: Der Einfluss des Join-Modes auf die einzelnen Teilfunktionen

hohen Durchsatz.

Beim Probing in Abbildung 4.36b, sieht der Verlauf ein wenig anders aus. Generell lässt sich sagen, dass das Probing nicht immer durch die Vektorgröße profitiert. Es gibt einen Peak bei 256KB und die niedrigsten Werte liegen ungefähr bei der größten und kleinsten Vektorgröße. Die maximale R-Größe für den S-Partitioning Probing-Modus, liegt bei 48KB, was der verwendeten Shared-Memory-Konfiguration entspricht. Daher muss bei größeren Werten, R aufgeteilt werden und es entstehen mehr Vektoren, als bei den anderen Verfahren. Bei der größten Vektorgröße, liegen viele Werte von der Shared-R-Variante, über den Werten der anderen Verfahren ohne Speicherlimit. Der Unterschied besteht darin, dass die mittlere Konfiguration mehr Vektoren, als die anderen beiden Verfahren besitzen. Der gesamte Probing-Prozess läuft parallel mit acht Streams ab, weswegen hier die Vermutung nahe liegt, dass die GPU schneller kleinere Vektoren und damit kleinere Hash-Tabellen abarbeiten kann. Größere und dafür weniger Tabellen, schneiden deutlich schlechter ab. Es darf jedoch nicht vergessen werden, dass bei sehr vielen kleinen Vektoren, auch die Tupel-Rate sinkt, was vorherige Ergebnisse zeigen.

Multi GPU

Um den Durchsatz zu erhöhen, wird erneut eine weitere GPU hinzugezogen. Diese dient der Haupt-GPU, wie im Radix- und Vektor-Split beschrieben. Beide Varianten haben nur einen Einfluss auf die Partitionierung und dem Probing, weshalb beide Ergebnisse in Abbildung 4.38 dargestellt sind. Der Versuchsaufbau verwendet die gleichen Tabellengrößen, da trotz der weiteren GPU, die erste GPU das Speicherlimit setzt. Die Daten sind Gleichverteilt, mit einem Skew von Null. Als Vektorgröße wurde 256KB gewählt. Beide Konfigurationen wurden zufällig gewählt, da es keinen Unterschied im Verhältnis zwischen den beiden Varianten erzeugt. Es ist nur zwingend erforderlich, dass beide Varianten die gleiche Konfiguration verwenden. Zu sehen ist, dass egal bei welcher Funktion, der Radix-Split einen höheren Maxima und ein geringeres Minima, in der Tupel-Rate besitzt. Die großen Schwankungen lassen sich durch die Verteilung erklären. So liefern unterschiedlich viele Werte, auch unterschiedliche Verteilungen, die durch das Hashing entstehen. Anhand dieser Verteilung entscheidet sich die Effizienz des Radix-Splits. Für den Vektor-Split ist das nicht der Fall. Hier entscheidet die Menge an Kopier-Operationen. Liegen nur wenige Vektoren vor, so nehmen die Latenzen, durch das Kopieren

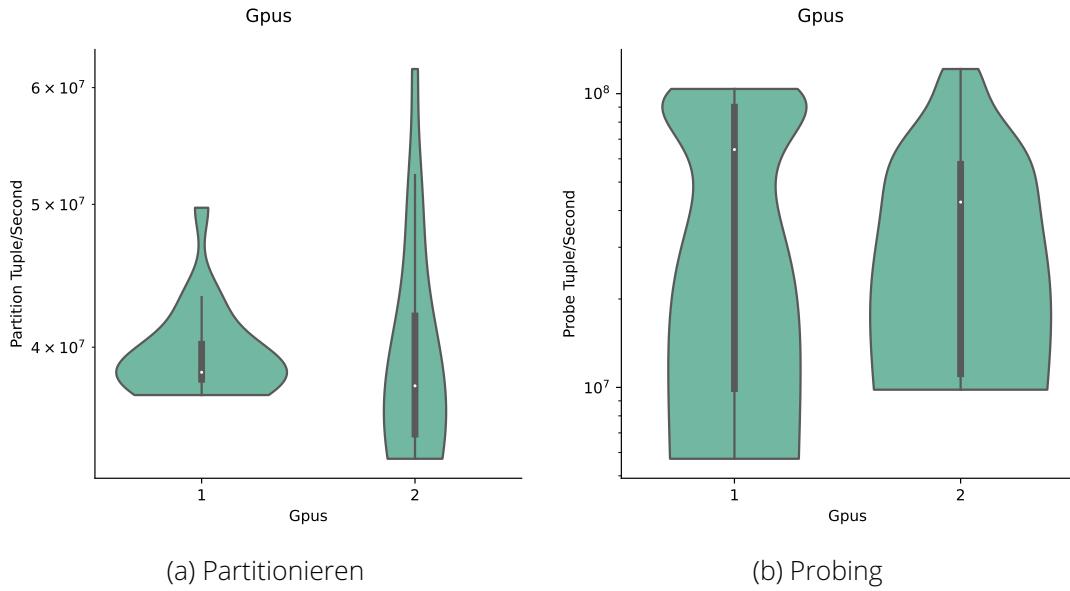


Abbildung 4.39: Einfluss der GPU-Anzahl auf die Vektorisierung

mehr Einfluss auf die Laufzeit und reduzieren die Tupel-Rate. Aus den Ergebnissen lässt sich daher kein klarer Favorit ermitteln, weshalb die Funktion für den Anwendungsfall angepasst werden muss.

Im Vergleich zwischen nur einer und zwei GPUs, profitieren beide Algorithmen. Dieses Ergebnis zeigen die Abbildungen 4.39. Beide Funktionen erreichen einen höheren Peak in der Tupel-Rate. Nur beim Partitionieren fällt das Ergebnis mit zwei GPUs, bei manchen Konfigurationen leicht nach unten. Das liegt hauptsächlich daran, dass die GPUs nicht gleichmäßig die Daten aufteilen und dadurch der Overhead, durch beide Geräte, stärker ins Gewicht fällt. Mit Blick auf die vorherigen Ergebnisse, sind die Parallelen zu nur einer GPU und dem Vector-Split zu erkennen. Der Vector-Split arbeitet im Partitionieren nur mit einer GPU, weshalb das Ergebnis identisch sein muss. Die leichte Stauchung entsteht nur durch das Ergebnis mit zwei GPUs. Allgemein lässt sich daher sagen, dass beide Varianten, die einzelnen Prozesse beschleunigen können.

Zusammenföhrung

Als letzte Funktion fehlt nur noch der Merge-Prozess. Dieser wandelt alle Teilergebnisse in eine große Relations-Tabelle um und ist daher unerlässlich. Das Zusammenführen besitzt keine große Komplexität. Und besteht hauptsächlich aus Cuda-Memory-Operationen. Jede Kopier-Operation bedient sich im Stream-Pool der GPU und verschiebt asynchron den Speicher an vorgesehenen Stelle, welche aus den vorherigen Tabellen-Größen bekannt ist. Bei einer weiteren GPU, müssen die Teilergebnisse auf die Haupt-GPU übertragen werden. Diese Übertragung ist von der NVLink-Anbindung limitiert und um das sechsfache langsamer als der interne Global-Memory. Daher ist mit einer Reduzierung, der Tupel-Rate zu rechnen. Abbildung 4.40 gibt die Ergebnisse für den Merge, aus dem vorherigen Versuch wieder. Der zu erwartende Einbruch in der Tupel-Rate, fällt im Vergleich nur minimal aus. Da die anderen Funktionen eine deutlich niedrigere Bandbreite besitzen, fällt der Effekt kaum ins Gewicht.

Final ist ein Blick auf den gesamten Join-Prozess zu werfen. Bereits in vorherigen Abschnitten wurden Zahlen für Konfigurationen mit einer GPU dargestellt, weshalb nur noch der Vergleich zu einer weiteren GPU fehlt. In der Abbildung 4.41 wird dieser vollzogen. Die Zahlen orientieren

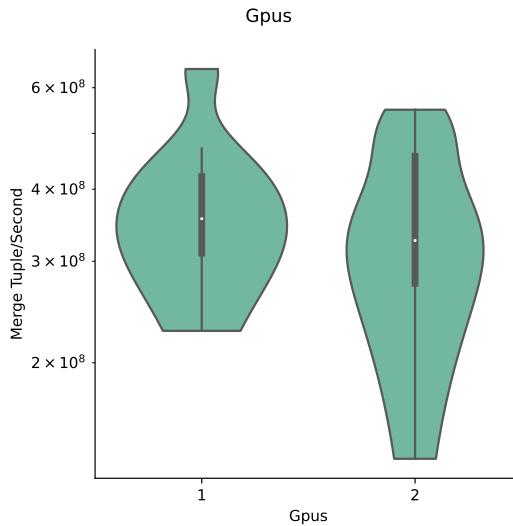


Abbildung 4.40: Merging im Multi-GPU Vergleich

sich immer stark, an der langsamsten Funktion, im Hash-Join. Weshalb die Höchstwerte nicht weit vom Partitionieren abweichen und nur bei rund drei Millionen Tupeln pro Sekunde liegen. Diese Höchstwerte sind nur durch die Verwendung einer weiteren GPU zu erreichen.

4.3.4 Zusammenfassung

Dieser Abschnitt hat sich mit der umfassenden Analyse des Hash-Joins beschäftigt. Der vorgestellte Algorithmus konnte die erwarteten Ergebnisse von bis zu einer Milliarde Tupels [32] pro Sekunde nicht erreichen, weshalb auf die Schwächen und Eigenschaften genauer eingegangen wurde. Das Herzstück ist das Probing-Verfahren. Dies unterteilt sich in dieser Arbeit in drei verschiedene Varianten, welche unterschiedliche Speicherlimitierungen mit sich bringen und durch verschiedene RS-Verhältnisse profitieren. Im ersten Schritt erfolgte die Analyse der Eigenschaften, durch unterschiedliche Verteilungen und Verhältnisse. Daraus ging hervor, dass die Tupel-Rate steigt, wenn die Überlappung der Datensätze abnimmt und es insgesamt weniger Paare in der Relations-Tabelle gibt. Für weitere Optimierungen, wurden die Kernel-Parameter genauer betrachtet und einzelne Konfigurationen vorgestellt und verglichen. Am Ende des Probing liegt immer eine Paarung vor, welche aus den Tabellen R und S, extrahiert werden. Damit das Probing die Paare speichern kann, liegt immer ein Buffer für alle Varianten vor, was dazu führt, dass bei größer werdenden Tabellen, der Probing-Buffer extrem größer wird. Aus diesem Grund, folgt im nächsten Schritt, die Vektorisierung. Bestehend aus der Radix-Partitionierung und der anschließenden Vektorbildung, wurde die Vektorisierung auf einzelne Eigenschaften untersucht. Am schnellsten liefen die Partitionierungs-Schritte ab, wenn ein gute Hash-Funktion, mit mehr als zwei Partitionen und Vektoren von bis 128KB verwendet werden. Zusätzlich kann das System auf bis zu acht Streams erweitert werden, wodurch die Tupel-Rate stetig steigt. Im Gegensatz zum Hashing, verbessert das kooperative Arbeiten zwischen GPUs die Tupel-Rate und erhöht diese auf bis zu 35 Millionen Tupel pro Sekunde. Die Hauptfaktoren für die schlechte Bandbreite sind die zu kleinen Partitionen und zu viele Vektoren für das Probing. Die kleinen Partitionen wirken sich negativ auf die Tupel-Rate im Histogramm- und Swap-Kernel aus. Zu viele Vektoren erhöhen die Latenz vom Probing-Verfahren, da die Streams begrenzt verfügbar sind. Daher ist es notwendig, für weiterführende Arbeiten, eine alternative Vektorisierungs-Strategie zu entwickeln.

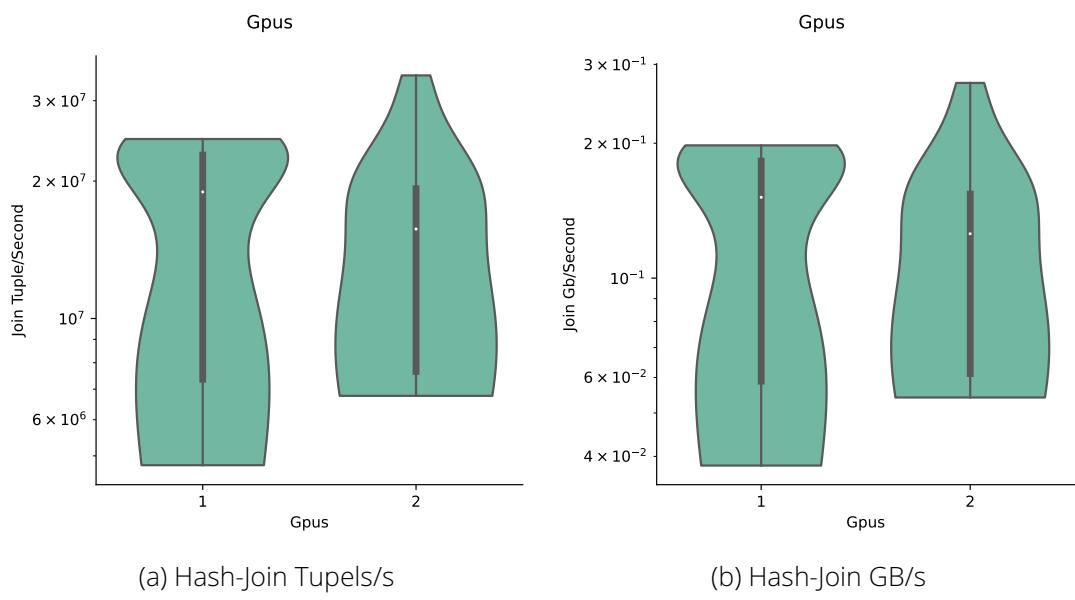


Abbildung 4.41: Hash-Join Tupel-Rate im Multi-GPU-System

5 Related Work

Die in der Arbeit vorgestellten Hash-Verfahren, sind sehr einfach gehalten. Jede Operation verwendet so wenig Teilschritte wie möglich, wodurch die die Berechnung eine geringe Latenz aufweist. Weit aus mehr Teilschritte verwendet das SHA-3 Verfahren und findet seine Anwendung in der Kryptographie. Choi et al. [40] nutzen in ihrer Arbeit die GPU, um das Verfahren zu beschleunigen und erreichen eine Bandbreite von bis zu 88.51Gb/s mit den SHA3-(512). Als Operation kommen hauptsächlich Bit-Operationen zum Einsatz. Zusätzlich sind die Hashes im Vergleich mit 256 bis 512-Bits, deutlich größer als die 64-Bit Hashes, welche in dieser Arbeit ihre Anwendung finden. Ein weitere Bereich ist das Gebiet der Hashtabellen, welches in Brenton et al. [41] für bereits bestehende GPU-Implementierungen untersucht wird. Wichtige Kriterien beziehen sich dabei auf die Geschwindigkeit und Implementierung. Data-Access-Patterns, Kommunikation mit der CPU und die allgemeine Parallelisierbarkeit sind Faktoren, welche die Geschwindigkeit beeinflussen. Darüber hinaus ist der sinnvolle Einsatz, der GPU-Hardware-Funktionen von Vorteil. Bausteine in diesem Gebiet sind zum Beispiel Speicher-Hierarchien, Atomics, oder Inter-Thread-Kommunikation mit Warp-Intrinsiken.

Die Anwendung der Hashtabelle befindet sich bereits im Titel der Arbeit. Der Hash-Join ist eine Variante, um die für Datenbanken typische Join-Operation durchzuführen. Wozniak et al. [42], stellen hierfür ein In-Memory Hash-Join-Verfahren für die GPU vor und vergleichen dieses mit einem heterogenen System, wo die Partitionierung nur auf der CPU stattfindet. Mit der vorgestellten Implementierung erreicht der Join auf der GPU, bis zu fünf Millionen Tupel pro Sekunde, was im Vergleich zum Ergebnis in dieser Arbeit, nur ein fünftel entspricht. Auch in dieser Arbeit wird die Aussage getroffen, dass die Partitionierung auf der GPU, schlechter als auf einem CPU-System verläuft, was die Aussage in dieser Arbeit verstärkt.

Einen anderen Ansatz bieten Guo et al. [43] und Sioulas et al. [32], indem die Partitionierung auf die CPU verlagert wird. Das Ergebnis ist eine Steigerung auf bis zu vier Milliarden Tupels pro Sekunde, was die in dieser Arbeit vorgestellte In-Memory-Implementierungen, um den Faktor 100x übertrifft. In den vorgestellten Implementierungen, wird wie in dieser Arbeit die Radix-Partitionierung verwendet, nur dass diese rein auf der CPU statt findet. Die GPU erhält dadurch die reinen Vektoren und muss keine Vorarbeit mehr leisten. Als Einschränkung tritt dabei die Datenrate des PCIe-Bus auf. Weshalb die obere Grenze, durch diese Schnittstelle definiert ist. Für das Probing wird nur die GPU verwendet. Der implementierte Kernel bedient sich dabei der *atomicExch(. . .)* Funktion, wodurch eine Linked-List für die Hash-Tabelle erstellt werden kann. Für die Generierung der Hashtabelle, wird genau dieser Ansatz verwendet. Daher stellen beide Arbeiten, die Grundlage für die Implementierung, des In-Memory Hash-Joins in dieser Arbeit.

6 Fazit

Im Verlauf der Arbeit entstanden viele verschiedenen Algorithmen, im Bereich Hashing und Hash-Join. Für das Hashing war das Ziel, eine schnelle und stabile Hash-Funktion zu finden. Die Anforderungen bestehen aus guter Verteilung, geringe Kollisionsrate und große Abstände zwischen benachbarten Werten. Für die Implementierung gibt es immer das gleiche Data-Access-Pattern, was flexibel zwischen verschiedenen Chunk-Größen wechseln kann. Die einzelnen Hash-Funktion verwendeten jeweils eine oder zwei Operationen. Als Operationen stehen die Addition, Multiplikation, XOR, Funnel-Shift und Bit-Reverse, zur Verfügung. Die ersten zwei Operation bekommen Unterstützung durch die Shift-Operation, um den Hash besser auf den gesamten Zahlenbereich zu verteilen. Funnel-Shift und Bit-Reverse sind beides Intrinsiken und stehen auf der GPU-Hardware zur Verfügung. Ersteres nimmt zwei Integer-Werte entgegen und ermöglicht somit eine 32-Bit-Permutation. Abgesehen von den Cuda-Vektoren, gab es keine weiteren Möglichkeiten, Cuda-Spezifische Sachen, wie die Warp-Kommunikation, in den Hash zu integrieren. Der finale Hash-Kernel von jeder Hash-Funktion, wurde auf bis zu zwei GPUs, mit zwei unterschiedlichen Methoden getestet. Es gibt immer eine Haupt-GPU, welche die weite GPU einbindet. Im ersten Modus verteilt eine GPU die Daten zwischen beiden GPUs. Der zweite Modus berechnet zwei geteilte Datensätze, auf den jeweiligen GPUs. Für die Evaluation stehen die Eigenschaften im Vordergrund. Jede Hash-Funktion wird auf eine Normalverteilung getestet und anhand der Ergebnisse, Verteilungs- und Kollisions-Eigenschaften evaluiert. In beiden Kriterien haben alle Funktionen, außer der weit verbreitete FNV- und neuen den HW-Funktionen sichtbare Einschränkungen. Im Abstands-Test, erzielten die HW-Funktionen eine deutlich höhere Distanz als das FNV-Verfahren. Im Test mit nur einer Quadro 8000 RTX und 64Byte Eingaben, erzielten die Hash-Funktionen einen Durchsatz von bis zu einer Milliarden Hashes (64-Bit) pro Sekunde. Die Datenrate liegt bei rund 550GB/s. Mit zwei Quadro 8000 RTX, erfolgt die Kommunikation über einen 100GB/s NVLink. Im Single-Betrieb erzielen die Hash-Funktionen einen Durchsatz von 1.75 Milliarden Hashes pro Sekunde und mehr als 1TB/s. Im Gegensatz dazu, liefert der Shared-Betrieb, eine deutlich geringere Datenrate mit einer Milliarde Hashes pro Sekunde. In diesem Fall limitiert das Kopieren über den NVLink die Bandbreite auf bis zu 100GB/s. Neben den GPUs, liefern unterschiedliche Chunk-Größen und Element-Größen auch unterschiedliche Hash-Raten. Ein Chunk mit 128Bit erzielt die höchste Datenrate, ist aber nur effizient, wenn die Element Bytes auch den Chunk ausfüllen. Bei kleineren Element-Größen ist es daher notwendig, die Chunk-Größe zu reduzieren. Die implementierten Kernel wiesen einen sehr hohe L1-Nutzung auf. Allgemein nutzen die Kernels um 30 Prozentpunkte mehr das Speicher-Interface. Als Ergebnis präsentierte diese Arbeit zwei Hash-Verfahren, welche jeweils Integer-Intrinsiken verwenden und einen sehr hohen Durchsatz auf der GPU erzielen. Das entwickelte Hash-Verfahren findet seinen Einsatz im Hash-Join. Die Hashes dienen dem Aufbau einer Hashtabelle aus dem Datensatz der R-Tabelle. Die S-Tabelle liegt gehasht

vor, und sucht nach Übereinstimmungen in der Hash-Tabelle. Die Tabellen können endlich viele Spalten besitzen, werden in der Arbeit aber auf maximal zwei Spalten begrenzt. Jede Spalte und der Primary-Key, umfassen acht Byte Integer-Werte. Die Hash liegen auch als acht Byte Buffer vor. Der Hash-Join besitzt als Herzstück, drei verschieden Probing-Verfahren, welche sich in der Durchführung ähneln und nur in der Speicher-Nutzung unterscheiden. Durch die immer größer werdenden Buffer für die Probing-Ergebnisse erfolgt in einem Schritt davor, das Partitionieren und Vektorisieren. Dieser Schritt teilt die Daten in vordefinierte Vektoren auf, sodass die GPU auf jedem Datensatz, das Probing durchführen kann. Für jeden Schritt erfolgt die Analyse der Tupel-Rate, bei unterschiedlichen Verteilungen der Datensätze. Die Datenerzeugung sieht Tabellen mit einer Gleich- und Zipf-Verteilung vor, welche jeweils um einen Skew angepasst werden und dadurch das Verhalten von den Algorithmen beeinflussen. Zusätzlich findet die Analyse der einzelnen Kernel-Parameter des Probing-Kernels statt, um die perfekte Konfiguration zu finden. Mit all den Parametern wurde die höchste Tupel-Rate, von knapp mehr als eine Milliarden Tupel pro Sekunde, mit der globalen Hashtabelle gemessen. Die Partitionierung verwendet das Radix-Verfahren und unterteilt, bei zu großen Partitionen, im Anschluss die Daten in der Vektorbildung. Als Resultat liegt am Ende, eine endliche Menge an Vektoren vor. Um das Verfahren zu beschleunigen, wird in dieser Arbeit, die Partitionierung auf zwei GPUs übertragen, welche sich die Partitionen (im Radix-Split) oder die Vektoren (im Vector-Split) teilen. Die erreichte Tupel-Rate lag mit zwei GPUs, bei rund 35 Millionen Tupel pro Sekunde. Da das Probing im Zusammenhang mit den Vektoren deutlich langsamer wurde, erfolgte eine breite Analyse der Partitionierung. In diesem Zusammenhang finden Analysen zu der Partitionstiefe statt, welche durch Vektorgröße, Hash-Funktion, und Partitionen-Anzahl beeinflussbar ist. Aus den Ergebnissen lässt sich ableiten, dass die Aufteilung der Daten auf der GPU, eine sehr große Herausforderung darstellt, welche in dieser Arbeit zu deutlich schlechteren Ergebnissen geführt hat, als in heterogenen Systemen. Alle Systeme wurden in der Arbeit von Grund auf implementiert.

6.1 Ausblick

Für zukünftige Arbeiten gibt es viele Teilbereiche, welche in dieser Arbeit nicht näher betrachtet wurden. Diese Betreffen zum einen das Hashing, das Probing und die Vektorisierung.

Hashing Für die Analyse der Hash-Funktionen, erfolgten die Tests auf einer Normalverteilung, welche Werte bis zu 2^{32} beinhalten. Anhand dieser Kriterien wurden die Hashes bewertet und für die weitere Nutzung klassifiziert. Allerdings stellte sich mit den Verteilungen aus der Vektorisierung heraus, dass die Normalverteilung nicht immer die schlechtesten Eigenschaften aufzeigen kann. Dies ist zum Beispiel anhand der Partitionstiefe in Abbildung 4.30 sichtbar. Aus diesem Grund ist es für zukünftige Arbeiten sinnvoll. Die vorgestellten HW-Funktionen auf weitere Datensätze zu analysieren.

Hash-Element-Size Als weitere Punkt für das Hashing, ist die Anomalie mit 32- bis 64Byte Elementen zu untersuchen. Mit dem NVProfiler konnten keine genauen Erkenntnisse errungen werden, weshalb die Analyse für diese Byte-Region breiter untersucht werden muss.

Shared-Mode Die Erweiterung des Hash-Verfahrens auf mehr als eine GPU, hat nicht nur höhere Datenraten erzielt. Der Shared-Mode wies sehr schlechte Werte auf und ist für zukünftige Arbeiten zu überdenken und anzupassen. Als gravierender Faktor erwies sich die Datenanbindung. Als Lösung könnte daher die Datenmenge auf der zweiten GPU reduziert werden oder die Datenpakete in kleinere Chunks aufgeteilt werden.

Vektorisierung Mit der Vektorisierung treten auf der GPU deutlich Probleme auf. Gerade kleinere Datensätze, welche unweigerlich entstehen, reduzieren den Durchsatz sehr stark. Aus die-

sem Grund fällt die Annahmen, dass das Radix-Partitioning, in der vorgestellten Implementierungsform, nicht geeignet für eine GPU Anwendung ist. Als Lösung könnten die einzelnen Partitionen zusammen in einem Kernel verarbeitet werden, wodurch sich die Kernel-Anzahl im Algorithmus reduziert. Hierfür ist es notwendig, dass die Partitionen in einer beliebigen Struktur gesammelt werden. Wenn diese Hürde überwunden ist, liegt eine Bandbreite, wie sie in heterogenen System zu sehen ist, nicht mehr all zu weit entfernt.

Literatur

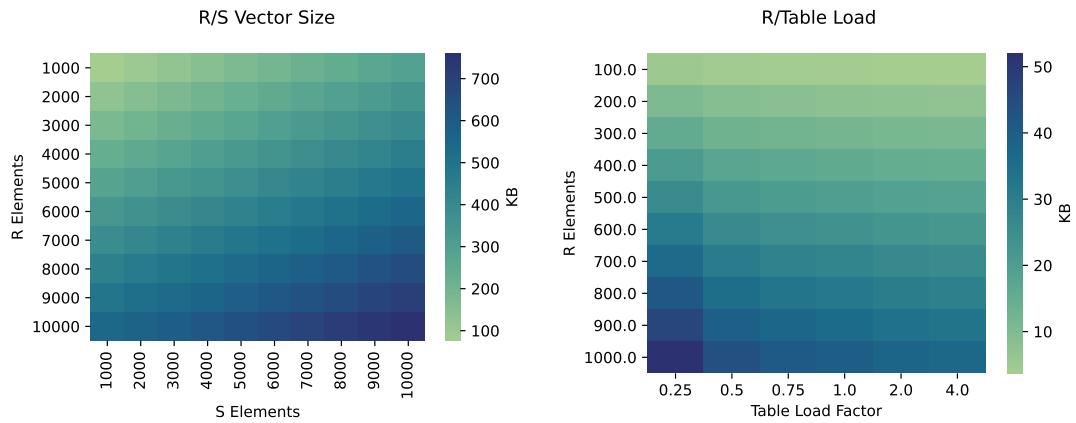
- [1] Priti Mishra und Margaret H. Eich. „Join Processing in Relational Databases“. In: 24.1 (März 1992), S. 63–113. ISSN: 0360-0300. DOI: 10.1145/128762.128764. URL: <https://doi.org/10.1145/128762.128764>.
- [2] Oliver Fluck u. a. „GPU Histogram Computation“. In: *ACM SIGGRAPH 2006 Research Posters*. SIGGRAPH '06. Boston, Massachusetts: Association for Computing Machinery, 2006, 53–es. ISBN: 1595933646. DOI: 10.1145/1179622.1179683. URL: <https://doi.org/10.1145/1179622.1179683>.
- [3] Jens-Peter Dittrich u. a. „Chapter 27 - Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm**This work has been supported by grant no. SE 553/2-2 from DFG.“ In: *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*. Hrsg. von Philip A. Bernstein u. a. San Francisco: Morgan Kaufmann, 2002, S. 299–310. ISBN: 978-1-55860-869-6. DOI: <https://doi.org/10.1016/B978-155860869-6/50034-2>. URL: <https://www.sciencedirect.com/science/article/pii/B9781558608696500342>.
- [4] Claude Barthels u. a. „Distributed Join Algorithms on Thousands of Cores“. In: Bd. 10. Aug. 2017. DOI: 10.14778/3055540.3055545.
- [5] Yujun Wen und Jie Kang. „Mobile Data Collection Strategy Research Based on the FNV Hash Algorithm“. In: *2015 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence*. 2015, S. 474–477. DOI: 10.1109/ACIT-CSI.2015.90.
- [6] *Python Dictionary Implementation*. <https://hg.python.org/cpython/file/10eea15880db/Objects/dictobject.c>. Accessed: 2021-10-23.
- [7] *Java Hash Map*. <https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html>. Accessed: 2021-10-23.
- [8] National Institute of Standards und Technology. *Secure hash standard (SHS)*. Aug. 2015. URL: <https://csrc.nist.gov/publications/detail/fips/180/4/final>.
- [9] Widmayer Ottmann T. *Algorithmen und Datenstrukturen*. Berlin, Heidelberg: Springer, 2017.
- [10] Yiou Zhao. „Optimizing Hash Strategy to Avoid Birthday Attack“. In: 1486 (Apr. 2020), S. 032004. DOI: 10.1088/1742-6596/1486/3/032004. URL: <https://doi.org/10.1088/1742-6596/1486/3/032004>.
- [11] Satyendra Gurjar u. a. *An empirical comparison of widely adopted hash functions in digital forensics: Does the programming language and operating system make a difference?* URL: <https://commons.erau.edu/adfs1/2015/tuesday/6>.

- [12] Rajeev Sobti und Geetha Ganesan. „Cryptographic Hash Functions: A Review“. In: *International Journal of Computer Science Issues, ISSN (Online): 1694-0814 Vol 9* (März 2012), S. 461–479.
- [13] Sandeep Kumar und Er Piyush Gupta. „A Comparative Analysis of SHA and MD5 Algorithm“. In: *International Journal of Computer Science and Information Technologies* 5 (Juni 2014), S. 4492–4495.
- [14] Thomas Mailund. *The Joys of hashing: hash table programming with C*. Apress., 2019.
- [15] Manik Sharma, Gurdev Singh und Rajinder Singh Virk. „Analysis of Joins and Semi Joins in a Distributed Database Queries“. In: *International Journal of Computer Applications* 49 (2012), S. 14–18.
- [16] Serge Abiteboul, Richard Hull und Victor Vianu. *Foundations of databases*. Addison-Wesley, 1996.
- [17] Mingxian Chen und Zhi Zhong. „Block Nested Join and Sort Merge Join Algorithms: An Empirical Evaluation“. In: Dez. 2014, S. 705–715. ISBN: 978-3-319-14716-1. DOI: 10 . 1007/978-3-319-14717-8_56.
- [18] Jens-peter Dittrich u. a. „Progressive Merge Join: A Generic and Non-Blocking Sort-Based Join Algorithm“. In: (Aug. 2002).
- [19] Spyros Blanas, Yinan Li und Jignesh M. Patel. „Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs“. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. SIGMOD '11. Athens, Greece: Association for Computing Machinery, 2011, S. 37–48. ISBN: 9781450306614. DOI: 10 . 1145/1989323 . 1989328. URL: <https://doi.org/10.1145/1989323.1989328>.
- [20] *Cuda Programming Guide*. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessed: 2021-10-23.
- [21] *Nvidia Turing Architektur*. <https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>. Accessed: 2021-10-23.
- [22] *Nvidia Fermi Architektur*. https://www.nvidia.de/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf. Accessed: 2021-10-23.
- [23] *ROCM Platform*. <https://rocmdocs.amd.com/en/latest/>. Accessed: 2021-10-23.
- [24] Jason Sanders, Edward Kandrot und Jack J. Dongarra. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley/Pearson Education, 2015.
- [25] *Nvidia NVLink/Switch*. <https://www.nvidia.com/en-us/data-center/nvlink/>. Accessed: 2021-10-23.
- [26] *Nvidia Quadro RTX 8000*. <https://www.nvidia.com/en-us/design-visualization/quadro/rtx-8000/>. Accessed: 2021-10-23.
- [27] Ariel Duarte-López, Arnau Prat-Pérez und Marta Pérez-Casany. „Using the Marshall-Olkin Extended Zipf Distribution in Graph Generation“. In: *Euro-Par 2015: Parallel Processing Workshops*. Hrsg. von Sascha Hunold u. a. Cham: Springer International Publishing, 2015, S. 493–502. ISBN: 978-3-319-27308-2.
- [28] *Zipf C Implementierung*. <https://www.csee.usf.edu/~kchriste/tools/genzipf.c>. Accessed: 2021-10-23.
- [29] Glenn Fowler u. a. *The FNV Non-Cryptographic Hash Algorithm*. Internet-Draft draft-eastlake-fnv-17. Work in Progress. Internet Engineering Task Force, Mai 2019. 119 S. URL: <https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-17>.

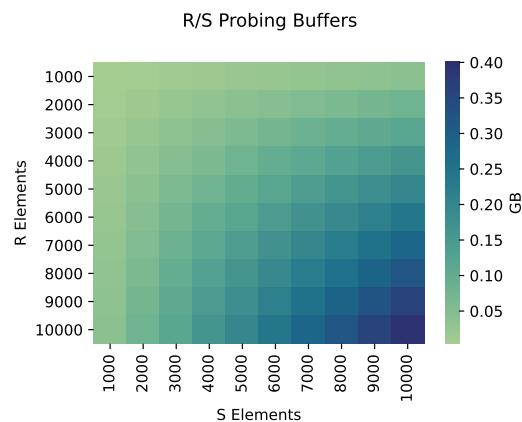
- [30] César Estébanez u. a. „Performance of the most common non-cryptographic hash functions“. In: *Software: Practice and Experience* 44 (2014).
- [31] H. Vandierendonck und K. De Bosschere. „XOR-based hash functions“. In: *IEEE Transactions on Computers* 54.7 (2005), S. 800–812. DOI: 10.1109/TC.2005.122.
- [32] Panagiotis Sioulas u. a. „Hardware-Conscious Hash-Joins on GPUs“. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, S. 698–709. DOI: 10.1109/ICDE.2019.00068.
- [33] Ran Rui und Yi-Cheng Tu. „Fast Equi-Join Algorithms on GPUs: Design and Implementation“. In: *SSDBM '17*. Chicago, IL, USA: Association for Computing Machinery, 2017. ISBN: 9781450352826. DOI: 10.1145/3085504.3085521. URL: <https://doi.org/10.1145/3085504.3085521>.
- [34] Cagri Balkesen u. a. „Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware“. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013, S. 362–373. DOI: 10.1109/ICDE.2013.6544839.
- [35] Johns Paul u. a. „Revisiting Hash Join on Graphics Processors: A Decade Later“. In: *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. 2019, S. 294–299. DOI: 10.1109/ICDEW.2019.00008.
- [36] Muhammad A. Awad u. a. „Better GPU Hash Tables“. In: *CoRR* abs/2108.07232 (2021).
- [37] Koji Nakano. „An Optimal Parallel Prefix-Sums Algorithm on the Memory Machine Models for GPUs“. In: *Algorithms and Architectures for Parallel Processing*. Hrsg. von Yang Xiang u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 99–113. ISBN: 978-3-642-33078-0.
- [38] Wei He u. a. „Massive Parallel Join in NUMA Architecture“. In: *2013 IEEE International Congress on Big Data*. 2013, S. 219–226. DOI: 10.1109/BigData.Congress.2013.37.
- [39] Nvidia GTX 1650. <https://www.nvidia.com/de-de/geforce/graphics-cards/gtx-1650/>. Accessed: 2021-10-23.
- [40] Hojin Choi und Seog Chung Seo. „Fast Implementation of SHA-3 in GPU Environment“. In: *IEEE Access* 9 (2021), S. 144574–144586. DOI: 10.1109/ACCESS.2021.3122466.
- [41] Brenton Lessley und Hank Childs. „Data-Parallel Hashing Techniques for GPU Architectures“. In: *IEEE Transactions on Parallel and Distributed Systems* 31.1 (2020), S. 237–250. DOI: 10.1109/TPDS.2019.2929768.
- [42] Kinga Anna Wozniak und Erich Schikuta. „Classification Framework for the Parallel Hash Join with a Performance Analysis on the GPU“. In: *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. 2017, S. 675–682. DOI: 10.1109/ISPA/IUCC.2017.00106.
- [43] Chengxin Guo und Hong Chen. „In-Memory Join Algorithms on GPUs for Large-Data“. In: *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 2019, S. 1060–1067. DOI: 10.1109/HPCC/SmartCity/DSS.2019.00151.

7 Anhang

Der Hash-Join verwendet verschiedene Buffer-Strukturen. Jeder Buffer führt zu einem unterschiedlichen hohen Speicherverbrauch, bei unterschiedlichen Konfigurationen. Da in den vergangenen Kapiteln immer von bestimmten Limitierungen gesprochen wurde, wirft die Abbildung 7.1 eine Blick auf die Buffer-Größen. Abbildung 7.1a visualisiert die Vektorgrößen, bei unterschiedlichen R- und S-Elementen-Anzahl. Ein Vektor umfasst die Bytes in S und R und zusätzlich die benötigten Bytes, in der Hashtabelle. Abbildung 7.1b vergleicht die Hashtabellen-Größe in den jeweiligen Probing-Kernels. Diese ist abhängig von der Menge an R-Elementen und dem Load-Faktor. Steigt der Load-Faktor an, verringert sich die Slot-Anzahl und die Hash-Table wird kleiner. jeder weitere Hash aus R, findet auch einen Platz in der Hashtabelle, wenn der Shared-Memory verwendet wird. Im Global-Memory ist dieser Buffer bereits durch die ursprüngliche Tabelle gegeben. Den größten Einfluss wirkt der Probing-Buffer aus. Dieser gewinnt sehr stark an Größe, wenn sich R und S nur leicht erhöhen. Aus diesem Grund ist in Abbildung 7.1c, die Skala auf Gigabytes angepasst.



(a) Größe des Vektors in Abhängigkeit von R und S
 (b) Größe der Hashtabelle in Abhängigkeit von R und dem Load-Faktor



(c) Größe der Probing-Buffer in Abhängigkeit von R und S

Abbildung 7.1: Einfluss der Vektorgröße auf die Funktionen