Paralleler Cuckoo-Filter

Jeremias Neth

22. November 2017

Zusammenfassung

Diese Seminararbeit befasst sich mit dem Probe-Algorithmus eines Cuckoo-Filters. Danach wird eine Mehrkernimplementierung mit Software-Threads als Vergleich untersucht. Die Ergebnisse eines Testes mit 200 Millionen 64-Bit Zahlen lassen, bei der Benutzung einer 4-Kern CPU, eine Beschleunigung des Verfahrens um den Faktor 2,9x im Vergleich zu einer Seriellen Implementierung erahnen. Für weitere Verbesserungen eines parallelen Cuckoo-Filters wird die Compute Unified Device Architecture (CUDA) verwendet um einen GPGPU Ansatz zu verfolgen. Testresultate lassen eine Beschleunigung um den Faktor 13,4x erahnen, bei der Benutzung einer 1.664-Kern CUDA GPU.

1 Einleitung

Inspiriert von dem Kuckuck, einem Brutparasiten, wurde 2001 mit Cuckoo-Hashing von Rasmus Pagh und Flemming Friche Rodler, ein neuer Algorithmus, zum Lösen von Hash Kollisionen, vorgestellt.

Darauf basierend entwickelten Fan et al. eine, praktisch bessere, Alternative zum Bloom Filter, dieser Cuckoo-Filter bietet eine platzeffizientere und performantere Datenstruktur für Set-Membership Abfragen. [4]

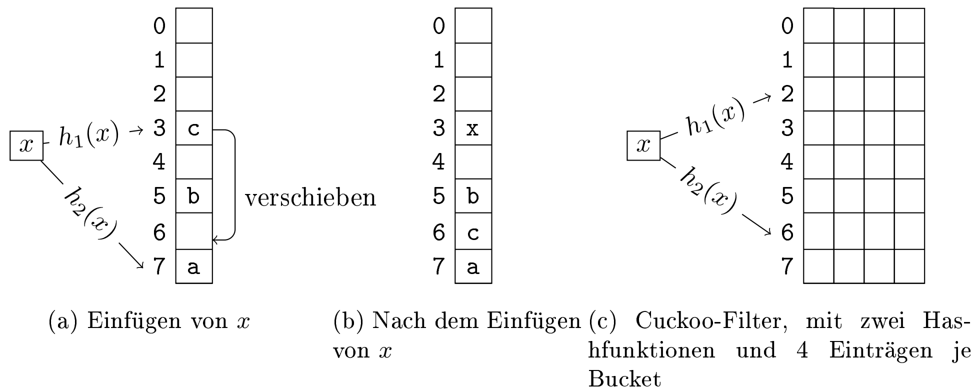
Dabei zielt diese Arbeit auf die Beantwortung, einer der offenen Fragen in Bezug auf Cuckoo-Hashtabellen ab, der Konstruktion und Analyse von Hashtabellen in Parallelen Architekturen. [7]

Diese Arbeit ist wie folgt organisiert. Abschnitt 2 untersucht die Performance eines seriellen Cuckoo-Filters für große Datenmengen. Abschnitt 3 beschreibt eine parallele Implementierung auf einem Mehrkernprozessor. Abschnitt 4 befasst sich mit der parallelen Implementierung auf der CUDA-Platform. Abschnitt 5 fasst diese Arbeit zusammen.

2 Analyse eines seriellen Cuckoo-Filters

2.1 Cuckoo-Filter-Algorithmus

Cuckoo Hashing Grundlagen: Eine Cuckoo Hashtabelle [2] besteht aus einem Array an Buckets, bei der jeder Eintrag x zwei mögliche Buckets, bestimmt durch zwei verschiedene Hashfunktionen $h_1(x)$ und $h_2(x)$, zugewiesen hat. Das nachschauen eines Eintrages x wird durch Berechnung und überprüfung beider Buckets mit $h_1(x)$ und $h_2(x)$ durchgeführt und erfolgt damit in konstanter Laufzeit. Das Einfügen eines neuen Eintrages wird durch Abbildung 1a illustriert. In diesem Beispiel wird versucht x in eine Hashtabelle der Größe 8 einzufügen, x kann in 3 oder 7 eingefügt werden. Wenn einer davon frei ist, kann x in diesen eingesetzt werden, da dies hier nicht der Fall ist, wird ein Bucket ausgewählt (bspw. Bucket 3) und dieser Eintrag verschoben (hier:"a") und dieser in seinen alternativen Bucket (hier:



(a) Einfügen von x

(b) Nach dem Einfügen von x

(c) Cuckoo-Filter, mit zwei Hashfunktionen und 4 Einträgen je Bucket

Abbildung 1: **Illustration von Cuckoo Hashing**

6) verschoben. Wenn dieser wieder voll ist, wird dieser wieder verschoben. Dies wird so lange wiederholt bis kein Eintrag mehr verschoben werden muss, illustriert durch Abbildung 1b, oder eine maximale Anzahl an Wiederholungen (bswps. 500 in der Implementierung) erreicht wurde. In diesem Fall ist die Hashtabelle zu voll um weitere Einträge einfügen zu können. Die amortisierte Laufzeit der Einfügeoperation ist konstant [3].

Cuckoo Filter: Ein Cuckoo-Filter basiert auf einer Cuckoo Hashtabelle und ist eine probalistische Datenstruktur zur Festellung ob ein Datum in einem Datensatz bereits vorgekommen ist. Dabei werden anstatt vollständigen Gegenständen nur die "Fingerabdrücke" dieser abgespeichert, dieses sogenannte "'partial-key cuckoo hashing"' erlaubt eine Platzersparniss bei dem Bau einer Cuckoo-Hashtabelle. Der alternative Bucket wird dann ausschließlich auf dem Fingerabdruck und dem primären Index basierend berechnet. Des Weiteren wird die Anzahl an möglichen Einträgen in einem Bucket erhöht um eine höhere Belegung zu ermöglichen.[1]

2.2 Implementierung und Performance Analyse

Die serielle Implementierung des Cuckoo-Filters wurde mit Visual C++ implementiert. Die Hardware Spezifikationen belaufen sich auf einen Intel Core i5-2500K Prozessor mit einer Taktrate von 3,30 GHz, 4-physischen Prozessorkernen und 6MB an L3-Cache, sowie 12 GB an Systemspeicher. Die verwendete Bucket-Größe beläuft sich auf 4 Einträge mit jeweils 16-Bit großen Fingerabdrücken.

Implementierung Algorithmus 1 stellt das Einfügen eines Gegenstandes in den Filter dar. Dieser basiert auf der Implementierung von Fan et al.[1]. Dafür werden der Fingerabdruck von x, als auch der Hash von x berechnet:

$$i_1(x) = hash(x). \tag{1}$$

Ausgehend von diesen kann auch die alternative Position berechnet werden:

$$i_2(x) = i_1(x) \oplus hash(f). \tag{2}$$

In diesen Buckets wird dann nach einem leeren Platz gesucht, sollte einer gefunden sein, wird der Fingerabdruck dort eingefügt und der Algorithmus terminiert. Andernfalls wird ein Eintrag in einem dieser Buckets mit dem einzufügenden Fingerabdruck ausgetauscht und für diesen Fingerabdruck, mit Gleichung 2, der alternative Bucket berechnet.

[1] Die Implementierung von Fan et al. findet man auf Github

Algorithm 1 Einfügen eines Gegenstand

```
 1: procedure EINFÜGEN(Gegenstand)
 2:     f ← Fingerabdruck von Gegenstand
 3:     i_1 ← Hash von Gegenstand
 4:     i_2 ← i_1 ⊕ Hash von f
 5:     if bucket[i_1] oder bucket[i_2] haben einen freien Eintrag then
 6:         füge f dort ein
 7:         return true
 8:     for Maximale Anzahl an Versuchen noch nicht erreicht do
 9:         wähle zufälligen Eintrag e in bucket[i_1]
10:         tausche f und den Fingerabdruck in e
11:         i_1 ← i_1 ⊕ hash(f)
12:         if bucket[i_1] hat einen freien Eintrag then
13:             füge f dort ein
14:             return true
15:     return false
```

Algorithm 2 Enthaltensein eines Gegenstand

```
 1: procedure PROBE(Gegenstand)
 2:     f ← Fingerabdruck von Gegenstand
 3:     i_1 ← Hash von Gegenstand
 4:     i_2 ← i_1 ⊕ Hash von f
 5:     if bucket[i_1] oder bucket[i_2] enthalten f then return true
 6:     else return false
```

Algorithmus 2 stellt abstrakt die Überprüfung, ob ein Gegenstand x in dem Filter enthalten ist, dar. Analog zum Einfügen werden mit Gleichungen 1,2 die beiden möglichen Buckets, sowie der Fingerabdruck des Gegenstandes berechnet. Diese beiden Buckets werden dann auf Vorhandensein des Fingerabdrucks überprüft.

Performance Analyse Ausgehend von dieser Implementierung werden drei Experimente ausgeführt um die Leistung des seriellen Cuckoo-Filters zu analysieren:

- Zuerst wird eine, zwischen 3M und 200M variierende Menge, M_1 an unterschiedlichen 64-Bit Werten in den Filter F eingefügt, bis der Filter zu 75% gefüllt ist.
- Danach wird F, nach M_1 durchsucht.
- Zuletzt wird F, auf einen anderen Datensatz $M_2 | M_1 \cap M_2 = \emptyset$ durchsucht.

Sowohl die Einfügungen, als auch die Durchsuchungen werden jeweils zehn Mal wiederholt und die durchschnittliche Rechenzeit aufgezeichnet.

Abb. 2 illustriert die aufgezeichnete Durchschnittliche Zeit für eine zunehmende Menge an beobachteten Gegenständen n. Die Ergebnisse von Abb. 2 deuten darauf hin, dass mit steigendem n auch eine höhere benötigte Rechenzeit beobachtet werden kann. Der höchste gemessene Wert 16569ms konnte beim Einfügen von 200M Elementen festgestellt werden. Für CPU-Nutzung wurde durchschnittlich 25% gemessen, was auf die Nutzung von nur 1/4, der zur Verfügung stehenden Prozessorkernen schließen lässt. Da dieser Algorithmus seriell implementiert wurde, ist dies auch nichts Verwunderliches.

Um die verfügbaren Ressourcen besser auszunutzen, müssen sämtliche Prozessorkerne benutzt werden. Und damit beschäftigt sich das folgende Kapitel.

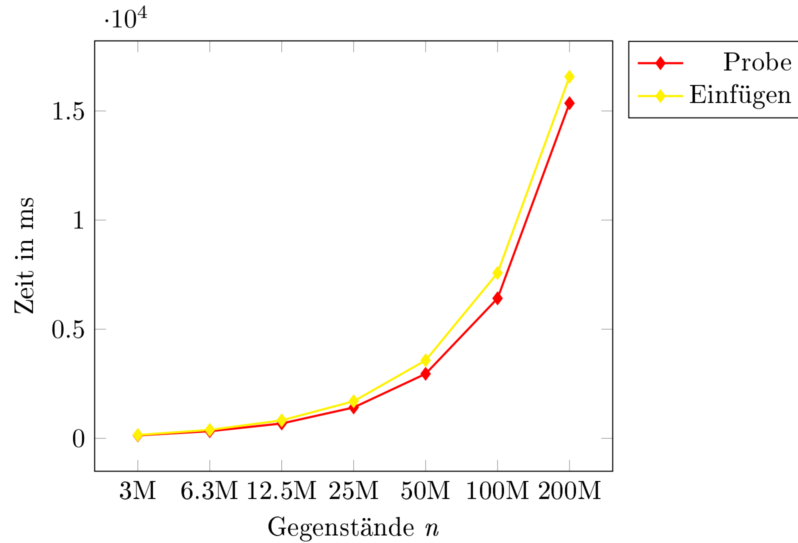


Abbildung 2: Performance einer seriellen Cuckoo-Filter Implementierung (Probe und Insert) für ein zunehmendes n

3 Paralleler Cuckoo-Filter auf einem Mehrkernprozessor

Um die Performance des seriellen Cuckoo-Filters für große Datensätze zu verbessern, wird hier ein paralleler Cuckoo-Filter mithilfe der C++ Standard Threads und Semaphoren implementiert. Der Programmablauf ist ähnlich zu der seriellen Implementierung. Der Datensatz wird auf mehrere Threads aufgeteilt und der kritische Bereich durch Semaphoren abgesichert.

Einfügen Der Algorithmus wird insofern abgeändert, dass T Threads (In unserem Fall 4) gestartet werden und jedem Thread $\frac{1}{T}$ großer Teil, des zu bearbeitenden Datensatzes, zugewiesen wird. Jeder Thread arbeitet nun wiederum sequentiell seinen Datensatz ab. Um die Integrität des Filters zu garantieren, wird Alg. 1 angepasst.

Die kritischen Abschnitte im Einfügen-Algorithmus befinden sich zwischen dem Finden eines kandidaten Eintrages und dem Einfügen oder Tauschen des Fingerabdrucks. Dies findet genau dann statt, wenn die Hashfunktionen in unterschiedlichen Threads kollidieren und auf denselben Bucket verweisen. Um dies zu verhindern, werden mehrere Semaphoren S auf den Filter F verteilt.

Abb. **??** stellt dies für zwei Threads und zwei Semaphoren dar. t_1 blockiert die ersten vier Elemente durch s_1 und verhindert somit eine Thread-Kollision mit t_2.

Jede Semaphore $s \in S$ für $\frac{|F|}{|S|}$-viele Buckets zuständig ist. Angenommen die verwendete Hashfunkton verteilt die Gegenstände gleichmäßig auf den Filter, dann ist die Wahrscheinlichkeit $P(B)$, dass ein Thread eine Semaphore sperrt:

$$P(B) = \frac{1}{|F|} * \frac{|F|}{|S|} = \frac{1}{|S|}$$

Damit ist die Wahrscheinlichkeit $P(A)$, dass alle Threads T, zur selben Zeit[2],

[2] Annahmen: Es greifen gerade alle Threads auf eine Semaphor zu

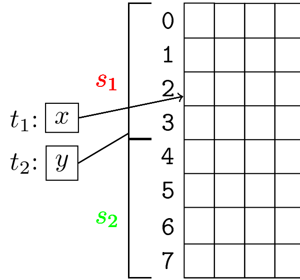


Abbildung 3: Thread Kollision: Zwei Threads versuchen in denselben Bucket zu schreiben

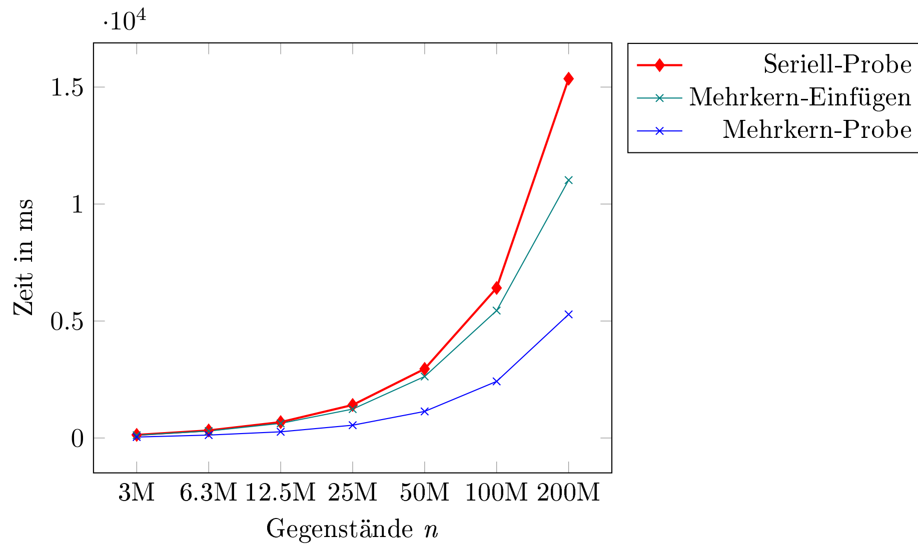


Abbildung 4: Performance einer mehrkern Cuckoo-Filter Implementierung (Probe und Insert) für ein zunehmendes n, im Vergleich mit einer seriellen Implementierung

eine unterschiedliche Semphor $s \in S$ sperren wollen, wie folgt bestimmt:

$$P(A) = \prod_{k=0}^{|T|-1} \frac{|S| - k}{|S|} \tag{3}$$

In der Implementierung wurden 4 Threads und 256 Semaphoren verwendet, was zur Kollision von 3,4% aller kongruenten Zugriffe führt.

Probe Der verwendete Algorithmus ist identisch zu der sequentiellen Implementierung, lediglich der zu bearbeitende Datensatz wird auf mehrere Threads verteilt.

Analog zu dem Experiment in 2.2, wird die selbe Testumgebung verwendet, mit dem selben Datensatz um die gerade vorgestellten Algorithmen zu benchmarken. Abb. 4 stellt die Ergebnisse dieses Experiment dar. Die höchste Zeit von 11023ms, für den Einfüge-Algorithmus wurde bei 200M Gegenständen gemessen. Damit wird eine 1,5x bessere Performance als bei der sequentiellen Implementierung erreicht. Beim Probe-Algorithmus wurde eine 2,9x schnellere Abarbeitung des Datensatzes, als mit nur einem Thread, gemessen. Des Weiteren wurde eine CPU-Auslastung von 100% gemessen. Weitere Verbesserungen könnten mit mehr Threads erreicht werden. Dies wirft die Nutzung von "many-core"-Prozessoren auf, wie sie in GPU Prozessoren zu finden sind.

4 Paralleler Cuckoo-Filter mit CUDA

Um den Cuckoo-Filter noch schneller zu gestalten wird in diesem Kapitel ein paralleler Cuckoo-Filter konstruiert, unter der Nutzung der CUDA-Platform. Ein mit CUDA entwickeltes Programm ist in Host (CPU) und Device (GPU) aufgeteilt. Dabei haben host und device, jeweils einen eigenen unterschiedlichen Speicher. Der Host kopiert dabei die Eingabedaten in den Speicher des Devices. Danach startet er den Kernel auf dem Device. Mit der Benutzung der CUDA-Platform werden Threads in Blöcken organisiert, diese Blöcke werden dann auf einer Mehrkernprozessoreinheit (engl. "streaming multiprocessor unit" (SM)) ausgeführt.

Die Implementierung des Cuckoo-Filters auf der CUDA-Plattform funktioniert ähnlich wie auf einem Mehrkernprozessor, dabei wird jedoch für jeden Gegenstand ein eigener Thread gestartet.

Beim Insertion-Algorithmus wird der kritische Bereich, durch die atomare Ausführung von "Compare-And-Swap" abgesichert:

```
1: procedure AtomicCAS(zieladresse, erwartet, swap)
2:     old = *zieladresse
3:     if erwartet == old then *zieladresse = swap
4:     return old
```

Der auf dem Device ausgeführten Teil des Probe-Algorithmus ist grundlegend ähnlich zu der Mehrkernimplementierung gestaltet. Der Probe-Algorithmus führt dabei bis zu drei Zugriffe auf den globalen Speicher aus. Bei dem ersten Zugriff wird der Gegenstand aus dem globalen Speicher geladen, gecached und gehasht. Die Gegenstände sind 1:1 auf die Threads gemappt und mit nur einem "coalesced Memory Transfer" können Daten für einen Warp übertragen werden. Der zweite Zugriff ist auf den primären Bucket, berechnet durch die primäre Hashfunktion. Durch die zufällige Verteilung der Hashfunktion können keine "coalesced Memory-Transfers" ausgeführt werden und jeder einzelner Gegenstand einzeln kopiert werden. Der dritte Zugriff ist der alternative Bucket und hat die selben Limitationen wie der Zweite. Dadurch haben wir eine starke Einschränkung durch die Speicherlatenz, welche sich in einem klassischen Cuckoo-Filter nicht lösen lässt. Alcantara et al. umgehen die hohe Latenz des globalen Speichers indem sie nur kleine Cuckoo-Hashtabellen ($<$512 Bytes) im geteilten Speicher verwenden. 6

Bei der verwendeten GPU handelt es sich um eine Nvidia GTX 970 mit 1.664 Kernen (jeder mit einer Taktrate von 1329 MHz), aufgeteilt auf 13 SMs und 4 GB an GPU Speicher. Tabelle 1 summiert die Spezifikationen der verwendeten GPU.

Zahl der Mehrkernprozessoren (SM)	13
Zahl der CUDA Kernen je SM	128
Zahl der Threads pro Warp	32
Maximale Zahl der Threads pro SM	2048
Maximale Zahl der Warps pro SM	64
Maximale Zahl der Blöcke pro SM	32
Maximale Zahl der Register je Block	64k
Maximale Zahl der Register je Thread	255

Tabelle 1: Nvidia GTX 970 Spezifikationen

Mit den aufgeführten Spezifikationen in Tabelle 1 limitiert sich die maximale Anzahl an Threads je SM auf 2048. Mit der, in unseren Tests, verwendeten Blockgröße von $t = 256$ wird somit eine gute Balance, zwischen residierenden Threads je SM und Registernutzung, erreicht.

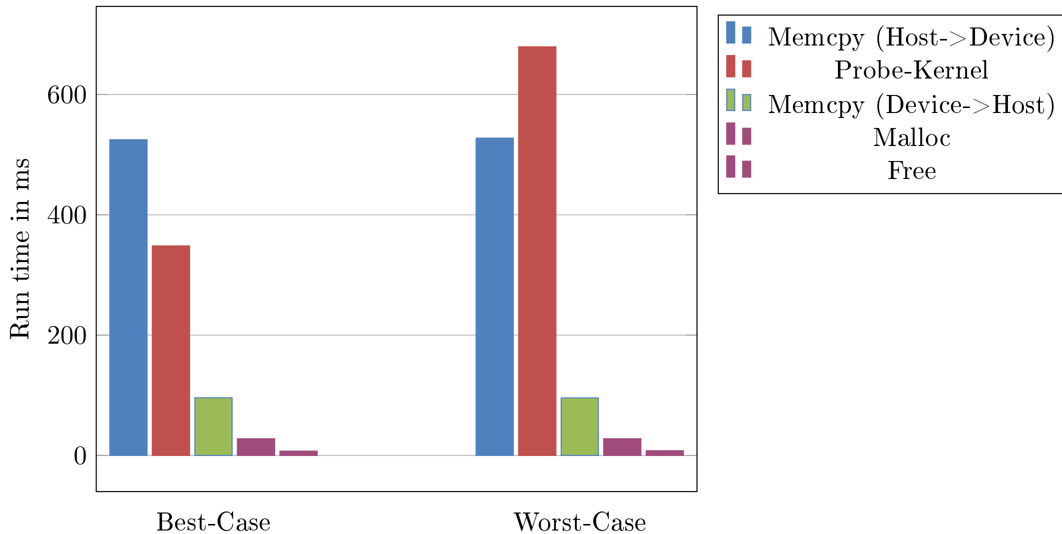


Abbildung 5: Analyse von Best- und Worst-Case für Probing von 200M Gegenstände

Verwendete Performance Richtlinien:
Warp Issue Efficiency: Wie viele der Warps sind aktiv.
Gründe dafür:
Memory Dependency: Laden oder Speichern nicht möglich, weil die angeforderte Ressource noch nicht verfügbar ist.
Memory Throttle: zu viele ausstehende Speicheranfragen.

Analog zu den vorherig ausgeführten Experimenten, wird zusätzlich zu der bisherigen Testumgebung, die vorher aufgeführte GPU verwendet. Der verwendete Datensatz ist identisch. Die Ergebnisse sind in Abb. 6 aufgeführt. Dabei wurde die Zeit, von vor dem Kopieren der Eingabedaten, bis die Ausgabedaten verfügbar waren, gemessen. Die für den Probe gemessenen Zeiten sind der Durchschnitt aus dem Best-,Average- und Worst-Case. Die Probe Zeiten für 200M betragen durchschnittlich 1169ms. Damit bieten wir einen Speedup von 13,3x gegenüber der Seriellen Implementierung und um 4,5x gegenüber der Mehrkern-Implementierung.

Abb. 5 zeigt die zeitliche Aufteilung des Probe-Vorgangs in Speichertransaktionen (Memcpy), Speicherallokationen (Malloc,Free) und den Kernel, auf. Ein großer Teil der benötigten Zeit (Bis zu 52%) wird dabei für Speichertransaktionen und Speicherallokationen (Bis zu 11%) verwendet. In unserem Setup limitiert dabei der PCIe 2.0 BUS zwischen Host und Device, mit einer maximalen Übertragungsrate von 8 GBs.

Dabei wurde jedoch ein starker Unterschied, von 330ms, zwischen Worst- und Best-Case gemessen. Dabei befinden sich die Daten zu hoher Wahrscheinlichkeit in dem primären Bucket im Best-Case und in keinem Bucket im Worst-Case. Dies lässt sich durch die zusätzlichen Speicherzugriffe bei der Durchsuchung des alternativen Buckets erklären. Die gemessene Warp Issue Efficiency [3] liegt bei 12,66%, dies liegt größtenteils an der hohen gemessenen Memory Dependency von 85,91% und Memory Throttle von 9,80%. Unter Beachtung dieser Zahlen und der benötigten Zeit für Speichertransaktionen, wartet die GPU 94% der gemessenen Zeit auf Daten.

[3] Mithilfe von NVIDIA Nsight

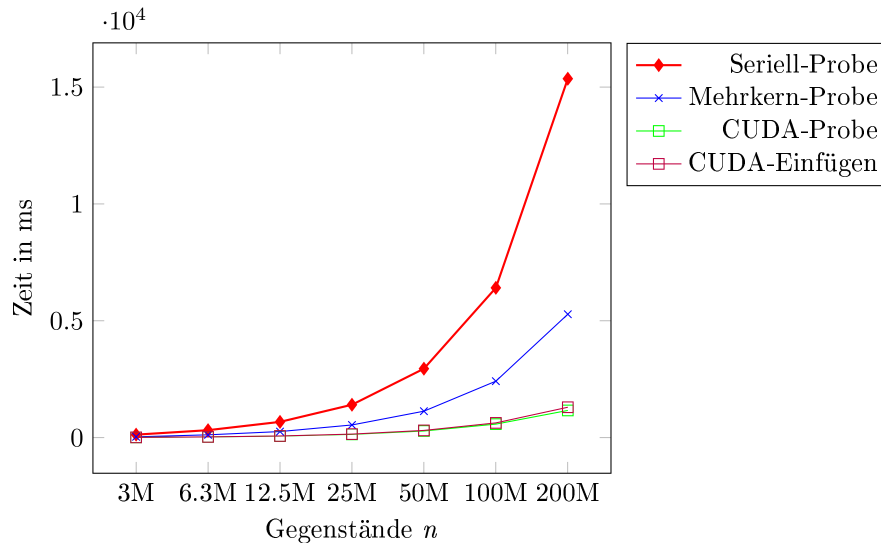


Abbildung 6: Performance einer CUDA Cuckoo-Filter Implementierung (Probe und Insert) für ein zunehmendes n, im Vergleich mit einer seriellen- und mehrkern-Implementierung

5 Fazit

Die gezeigte serielle Implementierung zeigt, bei großen Datensätzen, auf moderner, mehrkernorientierter Hardware eine schlechte Ausnutzung der vorhandenen Ressourcen. Dies lässt sich gerade in der Probe-Phase durch die Verwendung mehrerer Threads verbessern. Wie unsere Mehrkernimplementierung zeigt, sind bereits mit 4 Threads, Speedups von 2,9x möglich. Weitere Verbesserungen kann man durch die Einbindung der CUDA-Plattform erreichen, die vorgeschlagene Implementierung erreicht 13,3x die Performance der seriellen Implementierung. Dabei limitieren sowohl die Latenz des Device-Speichers, als auch die Übertragungsrate zwischen Host und Device. Diesen Limitierungen sollte, bei weiterführender Arbeit, Beachtung geschenkt werden.

Literatur

[1] Bin Fan, David G. Andersen, Michael Kaminsky, Michael D. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. 2014

[2] Rasmus Pagh, Flemming Friche Rodler. Cuckoo Hashing. 2001

[3] Michael Drmota and Reinhard Kutzelnigg. A precise analysis of Cuckoo hashing. ACM Trans. Algorithms 8, 2, Article 11 (April 2012), 36 pages. 2012.

[4] Bin Fan, David G. Andersen, Michael Kaminsky. The Cuckoo Filter: It's Better Than Bloom.

[5] KK Yong, Wen Mei Ong, Visnu Monn Baskaran, Poh Kit Chong, Ettikan Kadasamy Karuppiah. A Parallel Bloom Filter String Searching Algorithm on a Many-core Processor. 2013.

[6] Dan A. Alcantara, Andrei Sharf, Fatemeh Abbasinejad, Shubhabrata Sengupta, Michael Mitzenmacher, John D. Owens, Nina Amenta. Real-Time Parallel Hashing on the GPU. 2009

[7] Michael Mitzenmacher. Some Open Questions Related to Cuckoo Hashing.