Paralleler Cuckoo-Filter

Seminar: Implementierungstechniken für Hauptspeicherdatenbanksysteme

Jeremias Neth

München, 21. November 2017

Paralleler Cuckoo-Filter

- Cuckoo-Hashtabelle
- Serieller Cuckoo-Filter
  - Algorithmus
  - Performance
- Mehrkern Cuckoo-Filter
- CUDA Cuckoo-Filter

Cuckoo-Hashtabelle

Berechnung des Index in einer Hashtabelle

(Quelle: Wikipedia)

Cuckoo-Hashtabelle

Berechnung des Index in einer Hashtabelle

Verwendung von zwei Hashfunktionen

(Quelle: Wikipedia)

Cuckoo-Hashing

Einfügen von x in eine Cuckoo-Hashtabelle:

$h_1(x) = 3$

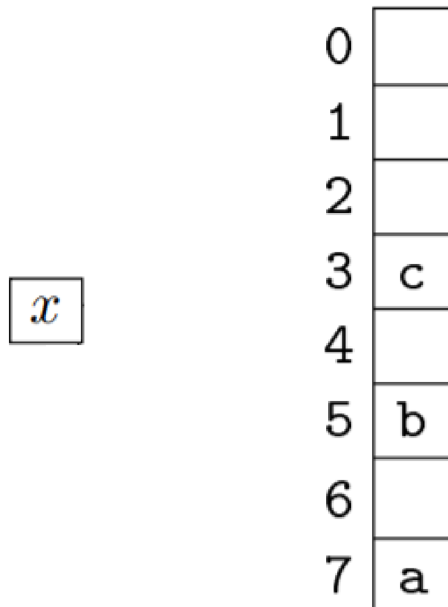
$h_2(x) = 7$

Cuckoo-Hashing

Einfügen von x in eine Cuckoo-Hashtabelle:

$h_1(x) = 3$

$h_2(x) = 7$



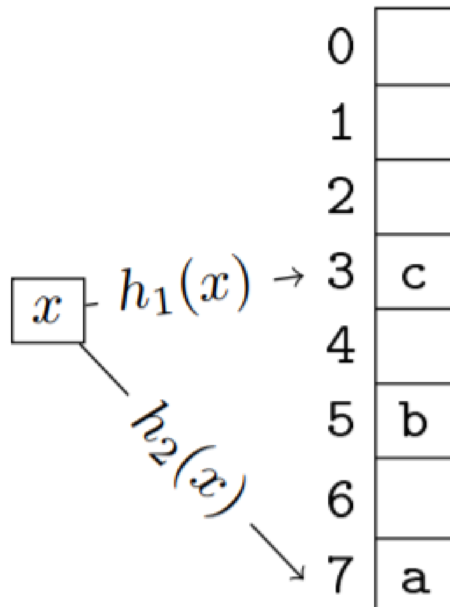
(a) Einfügen von x

Cuckoo-Hashing

Einfügen von x in eine Cuckoo-Hashtabelle:

$h_1(x) = 3$

$h_2(x) = 7$



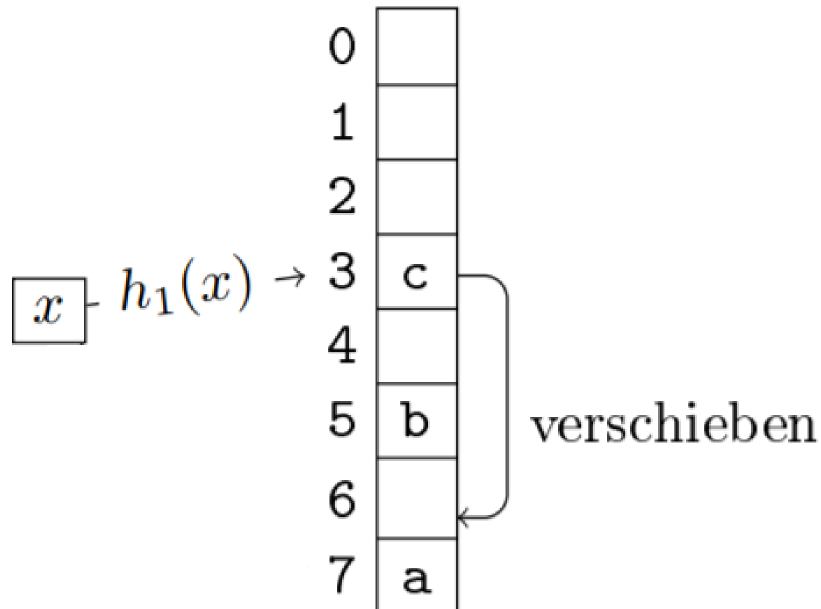
(a) Einfügen von x

Cuckoo-Hashing

Einfügen von x in eine Cuckoo-Hashtabelle:

$h_1(x) = 3$

$h_2(x) = 7$



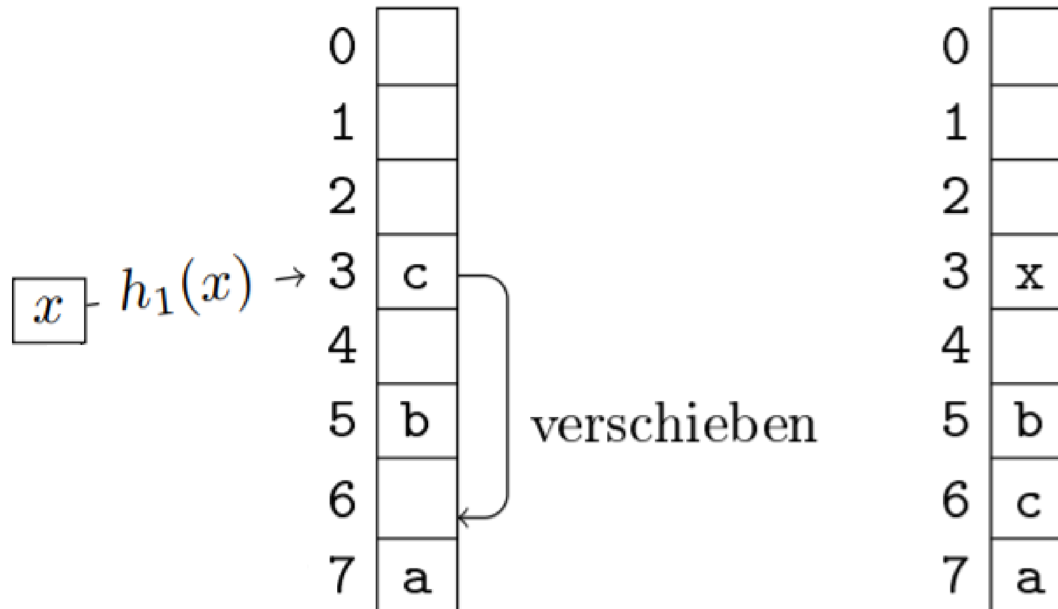
(a) Einfügen von x

Cuckoo-Hashing

Einfügen von x in eine Cuckoo-Hashtabelle:

$h_1(x) = 3$

$h_2(x) = 7$



(a) Einfügen von x (b) Nach dem Einfügen von x

Cuckoo-Filter

Probalistische Datenstruktur

Kann mit hoher Wahrscheinlichkeit sagen ob ein Gegenstand in den Filter eingefügt wurde, ohne false negatives

Cuckoo-Filter

Probalistische Datenstruktur

Kann mit hoher Wahrscheinlichkeit sagen ob ein Gegenstand in den Filter eingefügt wurde, ohne false negatives

Geringer Speicherplatzbedarf

Cuckoo-Filter

Probalistische Datenstruktur

Kann mit hoher Wahrscheinlichkeit sagen ob ein Gegenstand in den Filter eingefügt wurde, ohne false negatives

Geringer Speicherplatzbedarf

Alternative zum Bloom-Filter

Cuckoo-Filter

Probalistische Datenstruktur

Kann mit hoher Wahrscheinlichkeit sagen ob ein Gegenstand in den Filter eingefügt wurde, ohne false negatives

Geringer Speicherplatzbedarf

Alternative zum Bloom-Filter (Praktisch besser)
- Unterstützt das dynamische Löschen von Einträgen

Cuckoo-Filter

Probalistische Datenstruktur

Kann mit hoher Wahrscheinlichkeit sagen ob ein Gegenstand in den Filter eingefügt wurde, ohne false negatives

Geringer Speicherplatzbedarf

Alternative zum Bloom-Filter (Praktisch besser)
- Unterstützt das dynamische Löschen von Einträgen
- Schnellere Lookup Performance

Cuckoo-Filter

Probalistische Datenstruktur

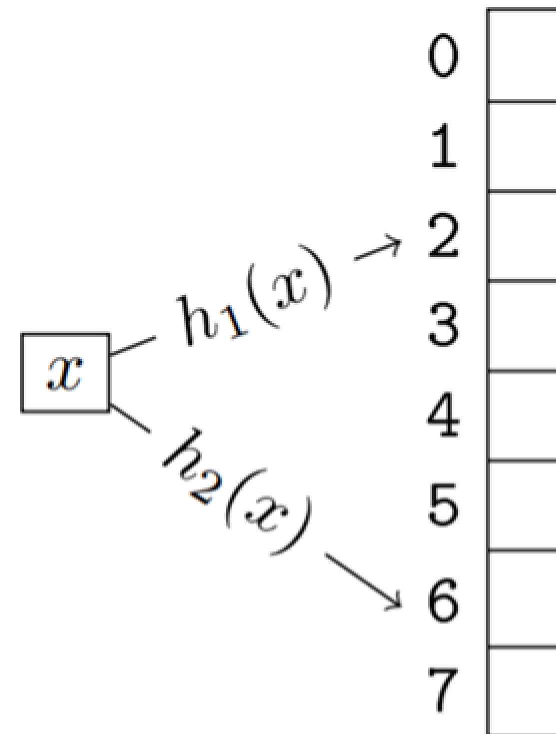
Kann mit hoher Wahrscheinlichkeit sagen ob ein Gegenstand in den Filter eingefügt wurde, ohne false negatives

Geringer Speicherplatzbedarf

Alternative zum Bloom-Filter (Praktisch besser)
- Unterstützt das dynamische Löschen von Einträgen
- Schnellere Lookup Performance
- Effizientere Platznutzung bei Anwendungen mit niedrigen false-positive-rates (<3%)

Cuckoo-Filter

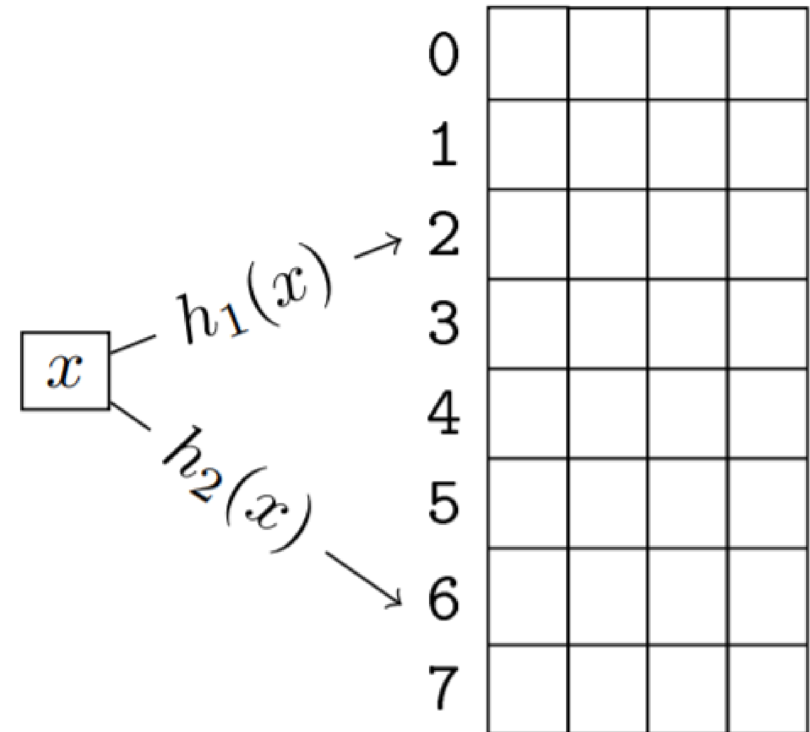
Cuckoo-Hashtabelle



Cuckoo-Filter

Cuckoo-Hashtabelle

Vier Einträge je Bucket

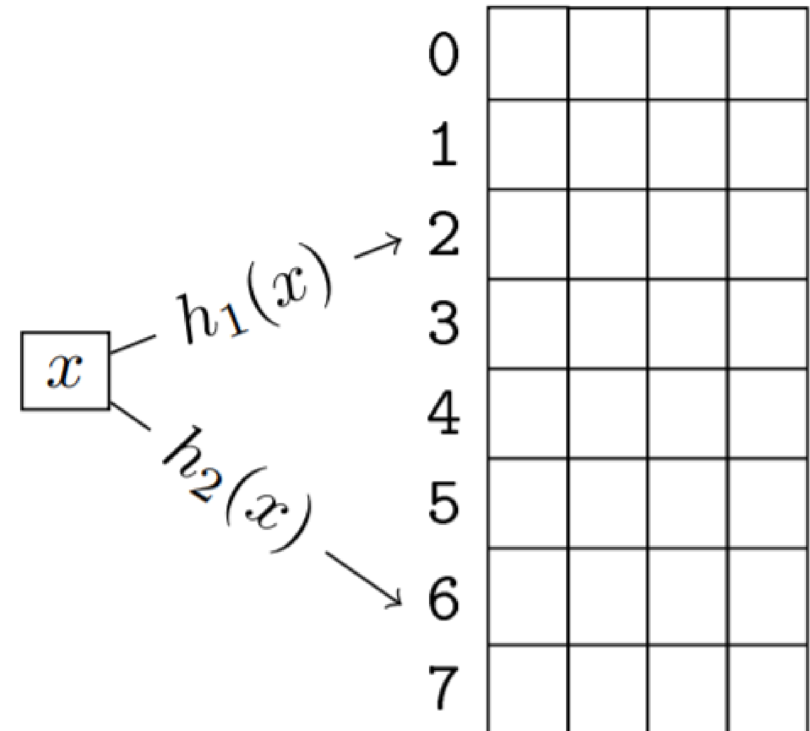


Cuckoo-Filter

Cuckoo-Hashtabelle

Vier Einträge je Bucket

“partial-key cuckoo hashing“

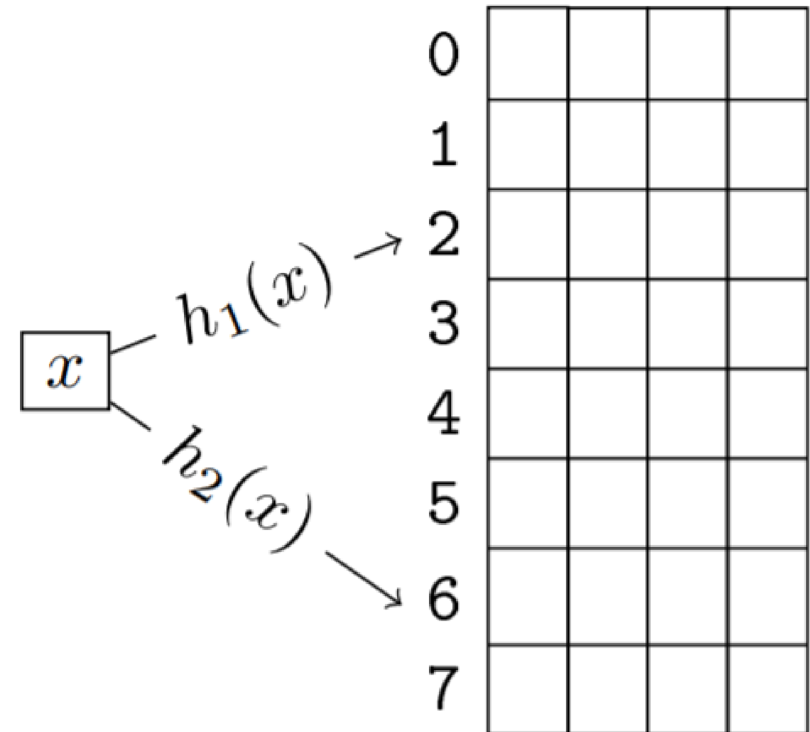


Cuckoo-Filter

Cuckoo-Hashtabelle

Vier Einträge je Bucket

“partial-key cuckoo hashing“
- Abspeichern von Fingerabdrücken

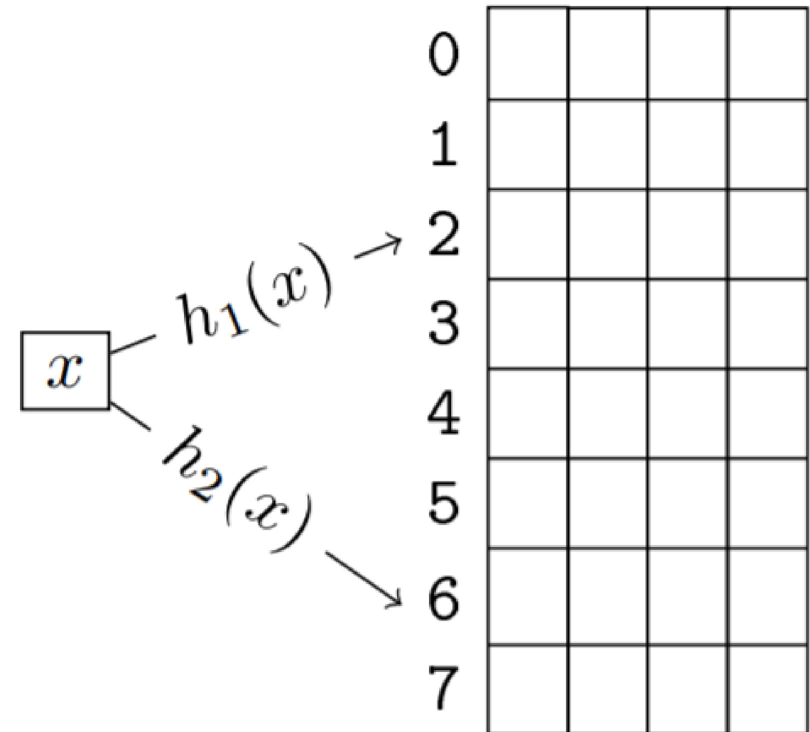


Cuckoo-Filter

Cuckoo-Hashtabelle

Vier Einträge je Bucket

“partial-key cuckoo hashing“
- Abspeichern von Fingerabdrücken
- Alternativer Bucket durch Index und Fingerabdruck bestimmbar



Cuckoo-Filter

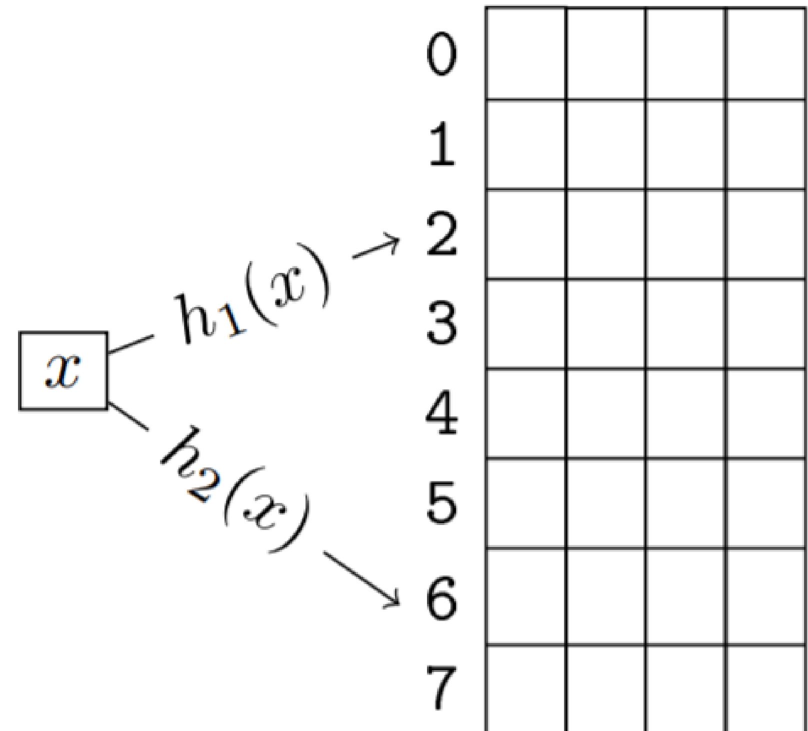
Cuckoo-Hashtabelle

Vier Einträge je Bucket

“partial-key cuckoo hashing“
- Abspeichern von Fingerabdrücken
- Alternativer Bucket durch Index und Fingerabdruck bestimmbar

$$i_1(x) = hash(x).$$

$$i_2(x) = i_1(x) \oplus hash(f).$$



Cuckoo-Filter

Cuckoo-Hashtabelle

Vier Einträge je Bucket

Fingerabdrucklänge: 16-bit

95% Fillrate

0,01% False Positive Rate

Cuckoo-Filter

Einfügen eines Gegenstand in den Filter

Algorithm 1 Einfügen eines Gegenstand

1: **procedure** EINFÜGEN(*Gegenstand*)
2: $f \leftarrow$ Fingerabdruck von *Gegenstand*
3: $i_1 \leftarrow$ Hash von *Gegenstand*
4: $i_2 \leftarrow i_1 \oplus$ 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: $i_1 \leftarrow i_1 \oplus \text{hash}(f)$
12: **if** $bucket[i_1]$ hat einen freien Eintrag **then**
13: füge f dort ein
14: **return** true
15: **return** false

Cuckoo-Filter

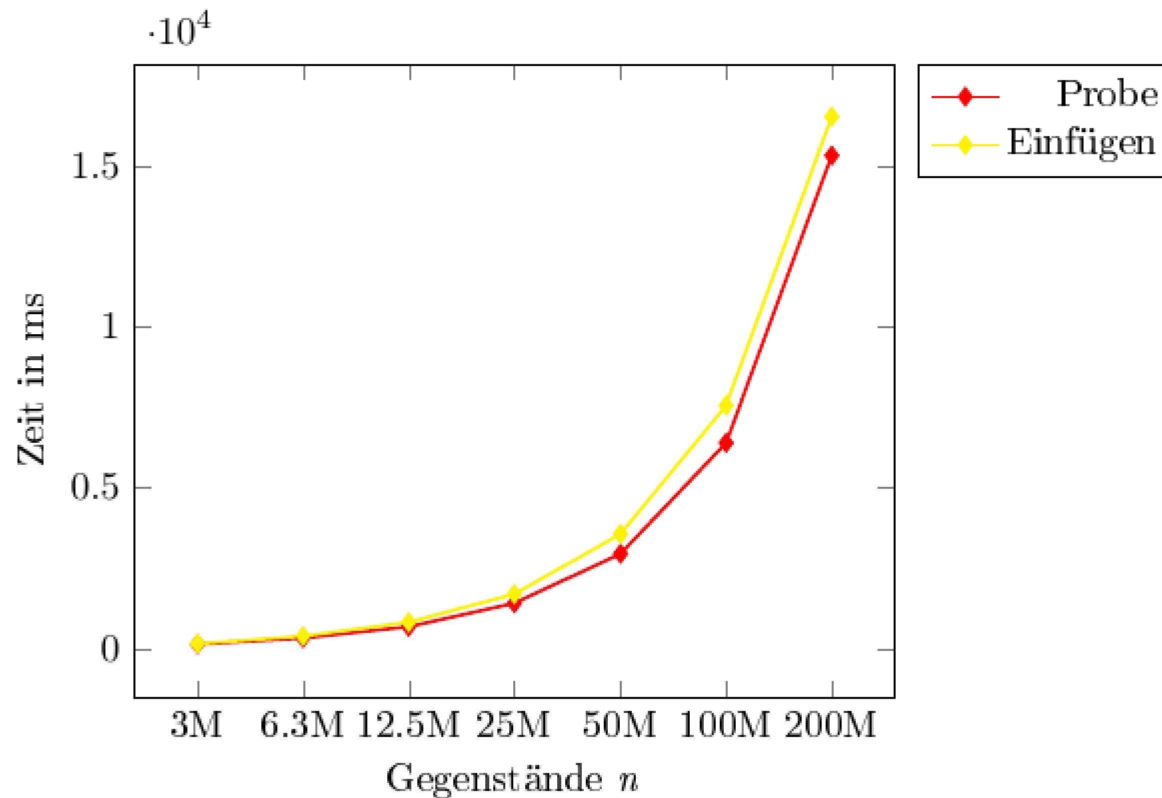
Lookup eines Gegenstand in dem Filter

Algorithm 2 Enthaltensein eines Gegenstand

1: **procedure** PROBE(*Gegenstand*)
2: $f \leftarrow$ Fingerabdruck von *Gegenstand*
3: $i_1 \leftarrow$ Hash von *Gegenstand*
4: $i_2 \leftarrow i_1 \oplus$ Hash von f
5: **if** $bucket[i_1]$ oder $bucket[i_2]$ enthalten f **then return** true
6: **else return** false

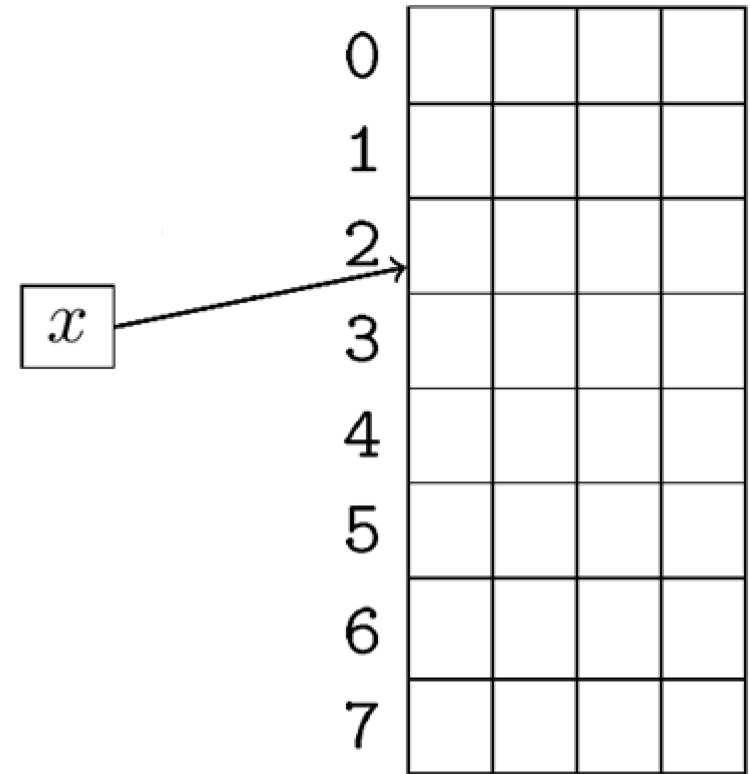
Cuckoo-Filter

Performance



Mehrkern Cuckoo-Filter

Algorithmus analog zur seriellen Implementierung



Mehrkern Cuckoo-Filter

Einfügen eines Gegenstand in den Filter

Algorithm 1 Einfügen eines Gegenstand

```
 1: procedure EINFÜGEN(Gegenstand)
 2:     f ← Fingerabdruck von Gegenstand
 3:     i1 ← Hash von Gegenstand
 4:     i2 ← i1 ⊕ Hash von f
 5:     if bucket[i1] oder bucket[i2] 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[i1]
10:         tausche f und den Fingerabdruck in e
11:         i1 ← i1 ⊕ hash(f)
12:         if bucket[i1] hat einen freien Eintrag then
13:             füge f dort ein
14:             return true
15:     return false
```

Mehrkern Cuckoo-Filter

Einfügen eines Gegenstand in den Filter

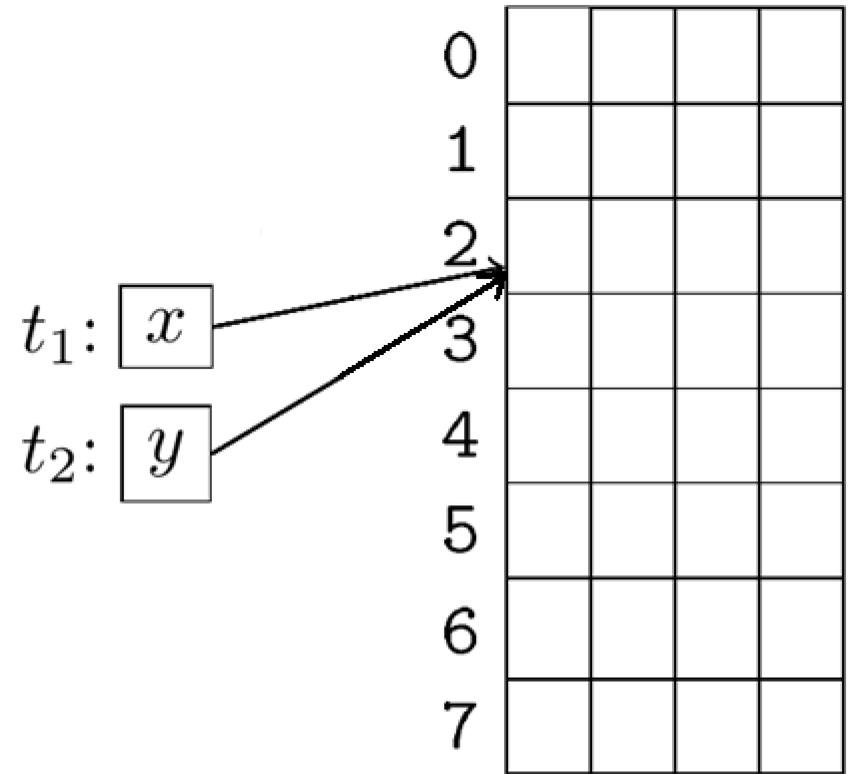
Algorithm 1 Einfügen eines Gegenstand

1: **procedure** EINFÜGEN(*Gegenstand*)
2: $f \leftarrow$ Fingerabdruck von *Gegenstand*
3: $i_1 \leftarrow$ Hash von *Gegenstand*
4: $i_2 \leftarrow i_1 \oplus$ 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: $i_1 \leftarrow i_1 \oplus \text{hash}(f)$
12: **if** $bucket[i_1]$ hat einen freien Eintrag **then**
13: füge f dort ein
14: **return** true
15: **return** false

Mehrkern Cuckoo-Filter

Algorithmus analog zur seriellen Implementierung

Verteilung des abzuarbeitenden Datensatzes auf mehrere Threads mit *std::thread*

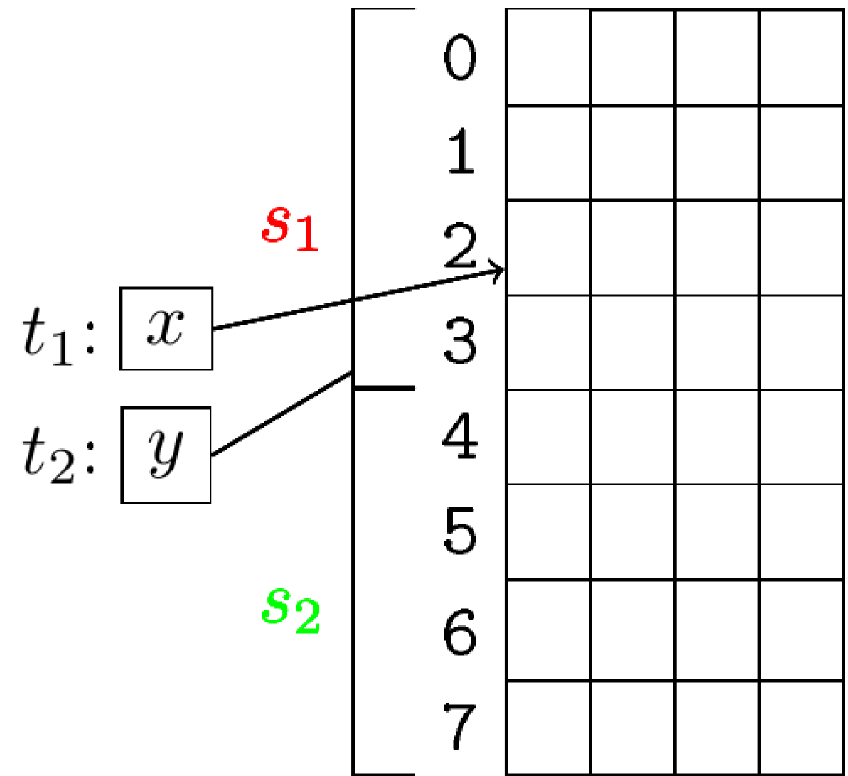


Mehrkern Cuckoo-Filter

Algorithmus analog zur seriellen Implementierung

Verteilung des abzuarbeitenden Datensatzes auf mehrere Threads mit *std::thread*

Absicherung des kritischen Bereichs durch Locks mit *std::mutex*



Mehrkern Cuckoo-Filter

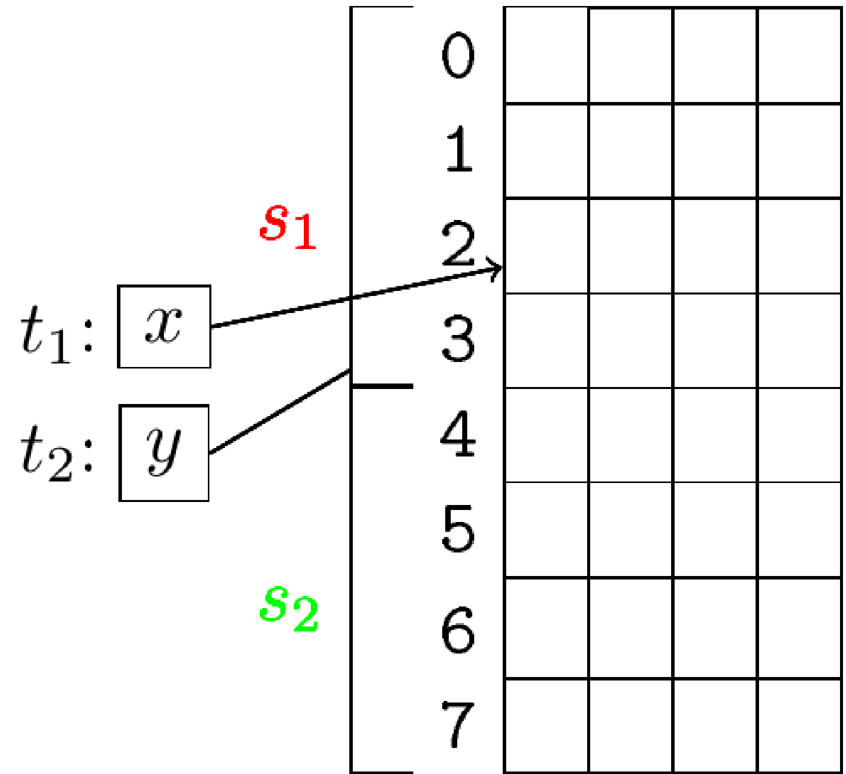
Algorithmus analog zur seriellen Implementierung

Verteilung des abzuarbeitenden Datensatzes auf mehrere Threads mit *std::thread*

Absicherung des kritischen Bereichs durch Locks mit *std::mutex*

Problem:

- Eine Lock: schlechte Performance
- Viele Locks: viel Overhead



Mehrkern Cuckoo-Filter

Algorithmus analog zur seriellen Implementierung

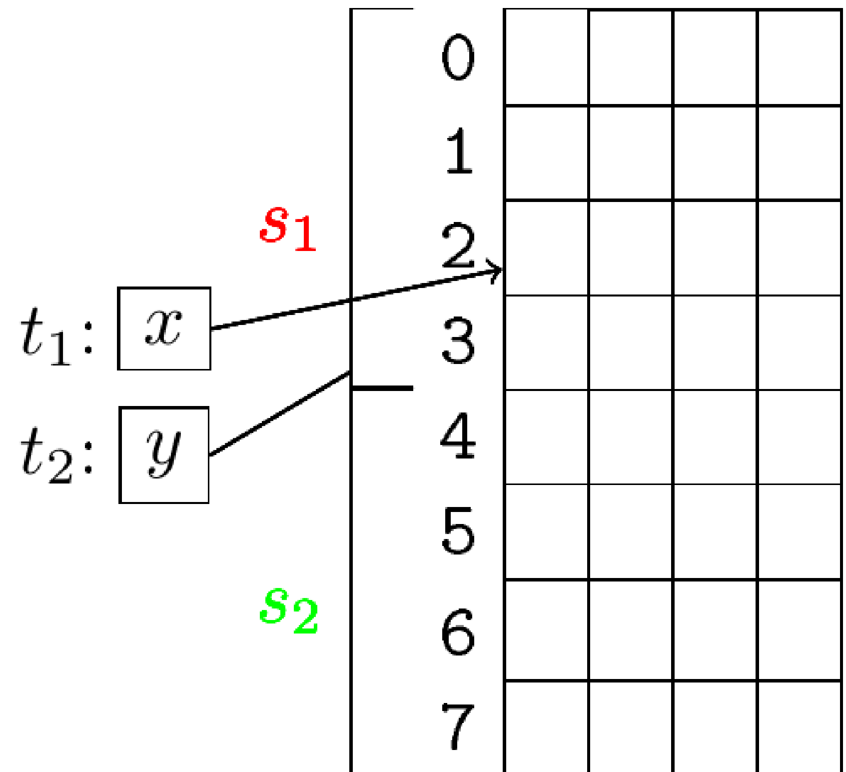
Verteilung des abzuarbeitenden Datensatzes auf mehrere Threads mit *std::thread*

Absicherung des kritischen Bereichs durch Locks mit *std::mutex*

Problem:

- Eine Lock: schlechte Performance
- Viele Locks: viel Overhead

A = Alle Threads T versuchen unterschiedliche Locks s€S zur gleichen Zeit zu sperren



Mehrkern Cuckoo-Filter

A = Alle Threads T versuchen ein unterschiedliches Lock sϵS zur gleichen Zeit zu sperren

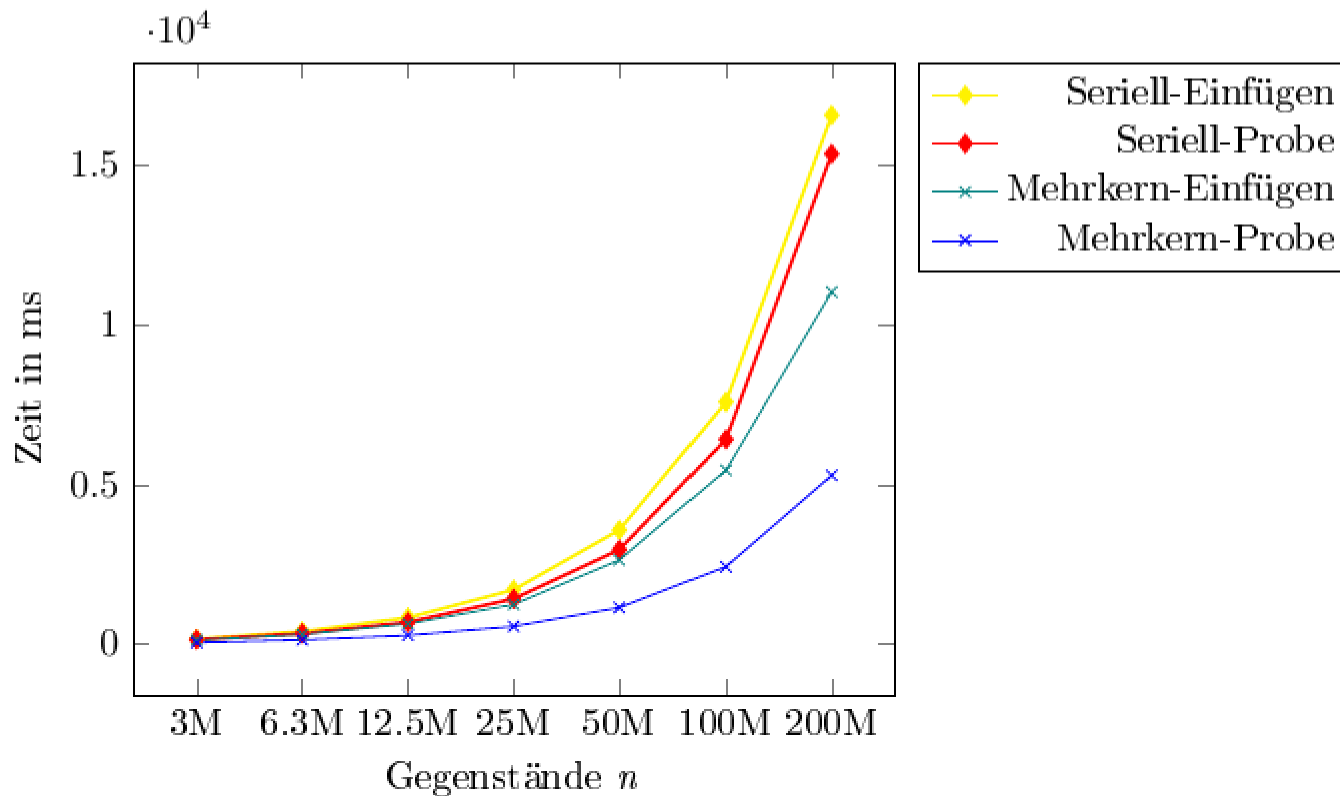
$$P(A) = \prod_{k=0}^{|T|-1} \frac{|S| - k}{|S|}$$

Beispiel: 4 Threads und 256 Locks: 96.6%, Speicherbedarf ~ 10 kB

Mehrkern Cuckoo-Filter

Performance

Speedup: 1.5x (**insert**), 2.9x (**probe**)



Mehrkern Cuckoo-Filter

Mögliche Verbesserungen:
Statt std::mutex, POSIX_MUTEX unter Linux oder CRITICAL_SECTION unter Windows
(Bis zu 2x schneller als std::mutex)

Nur einem Thread Schreibzugriff geben.

Mehrkern Cuckoo-Filter

Mögliche Verbesserungen:
Statt std::mutex: POSIX_MUTEX unter Linux oder CRITICAL_SECTION unter Windows
(Bis zu 2x schneller als std::mutex)

Nur einem Thread Schreibzugriff geben.

Mehr Kerne

Mehrkern Cuckoo-Filter

Mögliche Verbesserungen:
Statt std::mutex: POSIX_MUTEX unter Linux oder CRITICAL_SECTION unter Windows
(Bis zu 2x schneller als std::mutex)

Nur einem Thread Schreibzugriff geben.

Mehr Kerne

Grafikkarte stellt viel mehr Rechnerkerne zur Verfügung (Bsp: GTX 970, 1664)

CUDA Cuckoo-Filter

Compute **U**nified **D**evice **A**rchitecture

- Aufteilung des Programms für Host (CPU) und Device (GPU)
- Getrennter Speicher

CUDA Cuckoo-Filter

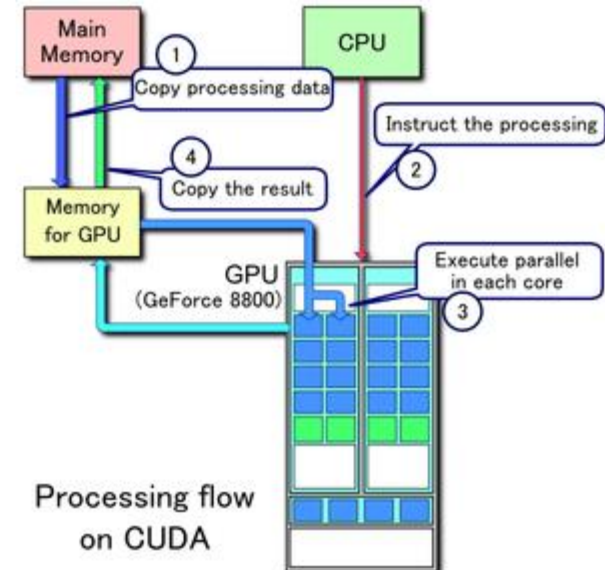
Compute **U**nified **D**evice **A**rchitecture

- Aufteilung des Programms für Host (CPU) und Device (GPU)
- Getrennter Speicher
- Serieller Part auf CPU, Paralleler Part auf GPU
- Aufteilung der Aufgabe in einzelne Threads

CUDA Cuckoo-Filter

Compute **U**nified **D**evice **A**rchitecture

- Aufteilung des Programms für Host (CPU) und Device (GPU)
- Getrennter Speicher
- Serieller Part auf CPU, Paralleler Part auf GPU
- Aufteilung der Aufgabe in einzelne Threads

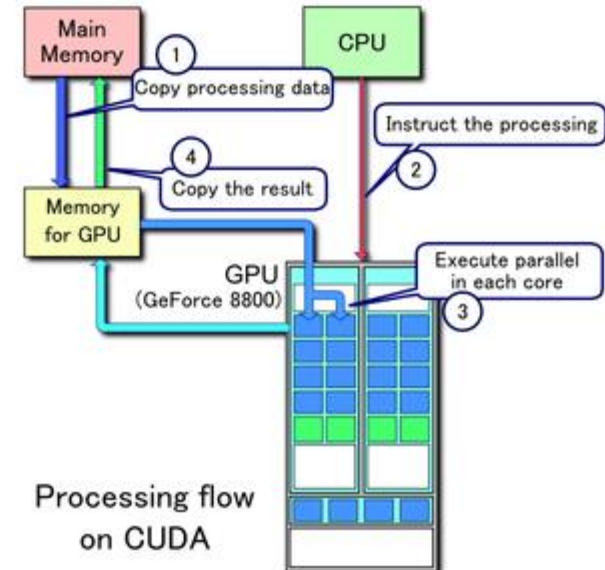


(Quelle: Wikipedia)

CUDA Cuckoo-Filter

Compute **U**nified **D**evice **A**rchitecture

- Aufteilung des Programms für Host (CPU) und Device (GPU)
- Getrennter Speicher
- Serieller Part auf CPU, Paralleler Part auf GPU
- Aufteilung der Aufgabe in einzelne Threads
- SIMT: Single Instruction Multiple Threads
- 32 Threads werden zu einem Warp zusammengefasst

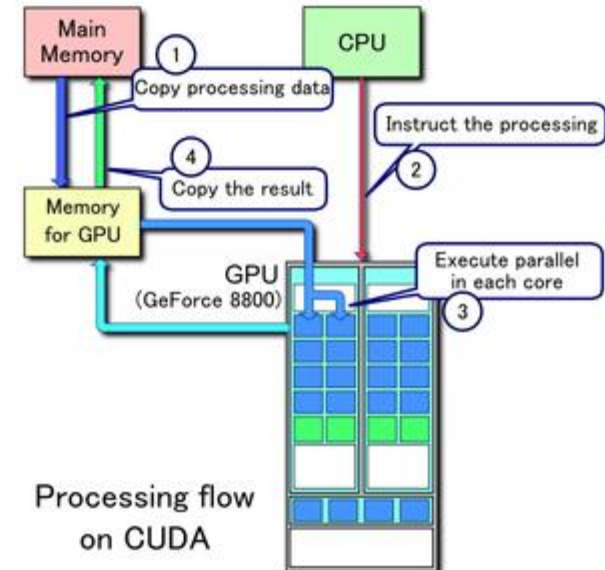


(Quelle: Wikipedia)

CUDA Cuckoo-Filter

Compute **U**nified **D**evice **A**rchitecture

- Aufteilung des Programms für Host (CPU) und Device (GPU)
- Getrennter Speicher
- Serieller Part auf CPU, Paralleler Part auf GPU
- Aufteilung der Aufgabe in einzelne Threads
- SIMT: Single Instruction Multiple Threads
- 32 Threads werden zu einem Warp zusammengefasst
- Bis zu 64 Warps residieren gleichzeitig auf einer Streaming Multiprocessorunit (SM)

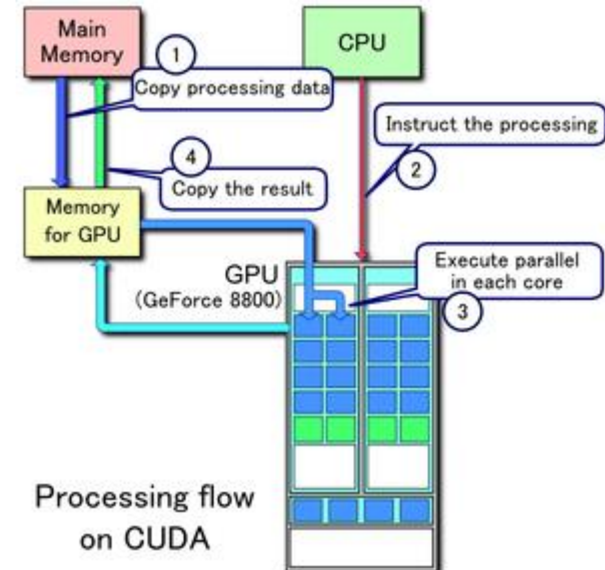


(Quelle: Wikipedia)

CUDA Cuckoo-Filter

Compute **U**nified **D**evice **A**rchitecture

- Aufteilung des Programms für Host (CPU) und Device (GPU)
- Getrennter Speicher
- Serieller Part auf CPU, Paralleler Part auf GPU
- Aufteilung der Aufgabe in einzelne Threads
- SIMT: Single Instruction Multiple Threads
- 32 Threads werden zu einem Warp zusammengefasst
- Bis zu 64 Warps residieren gleichzeitig auf einer Streaming Multiprocessorunit (SM)
- Aber nur 8 werden pro SM gleichzeitig ausgeführt



(Quelle: Wikipedia)

CUDA Cuckoo-Filter

Host

CUDA Cuckoo-Filter

Host

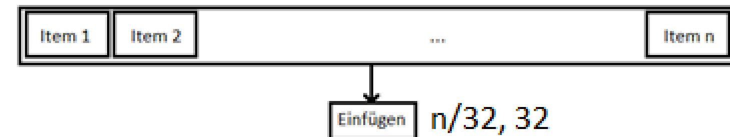
Kopiere Datensatz in Device Speicher

CUDA Cuckoo-Filter

Host

Kopiere Datensatz in Device Speicher

Starte Kernel *Einfügen* mit benötigter Anzahl an Threads, aufgeteilt in Blöcke

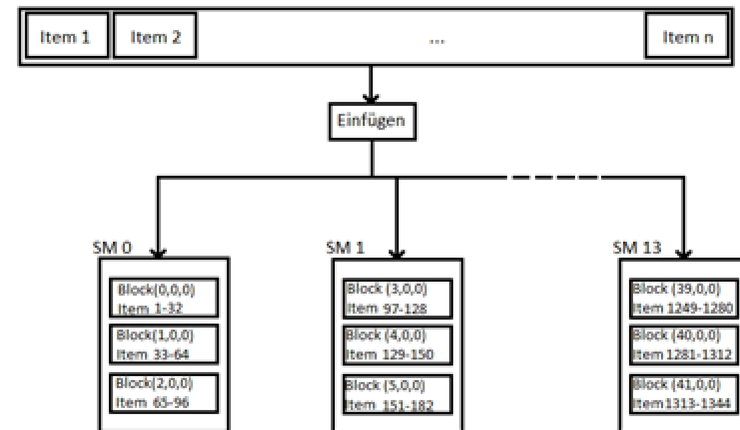


CUDA Cuckoo-Filter

Host
Kopiere Datensatz in Device Speicher

Starte Kernel *Einfügen* mit benötigter Anzahl an Threads, aufgeteilt in Blöcke

Device
Mappe momentanen Thread auf Items



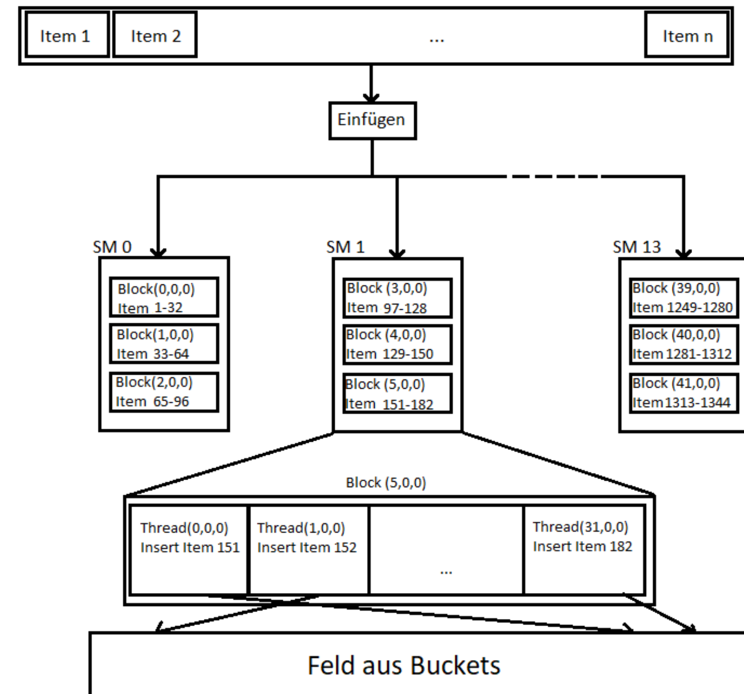
CUDA Cuckoo-Filter

Host
Kopiere Datensatz in Device Speicher

Starte Kernel *Einfügen* mit benötigter Anzahl an Threads, aufgeteilt in Blöcke

Device
Mappe momentanen Thread auf Items

Insert-Algorithmus analog zur Mehrkern-Implementierung



CUDA Cuckoo-Filter

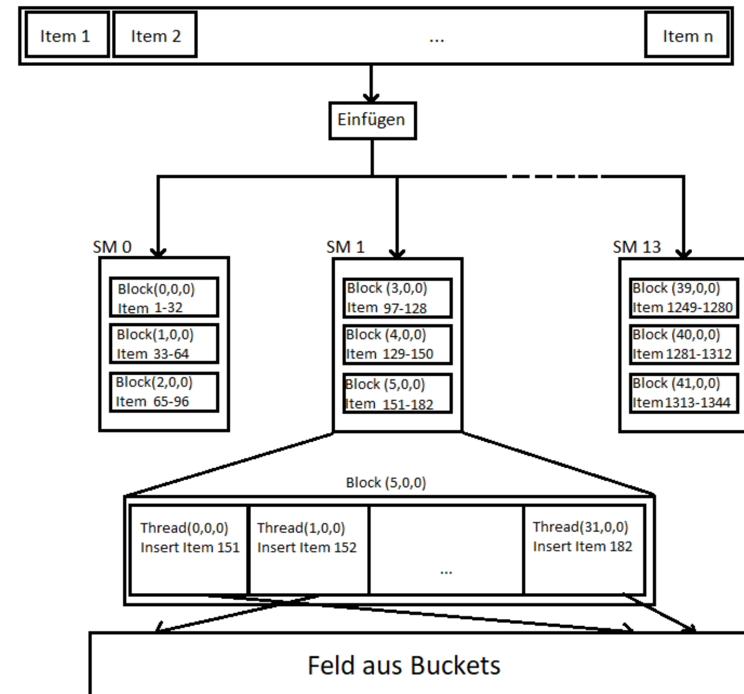
Host
Kopiere Datensatz in Device Speicher

Starte Kernel *Einfügen* mit benötigter Anzahl an Threads, aufgeteilt in Blöcke

Device
Mappe momentanen Thread auf Items

Insert-Algorithmus analog zur Mehrkern-Implementierung

Andere Absicherung des kritischen Bereichs (~32M Semaphoren bei 1664 Threads für ähnlich viele „Kollisionen“ wie bei 4 Threads)



CUDA Cuckoo-Filter

Device

Andere Absicherung des kritischen Bereichs

AtomicCompareAndSwap

AtomicCAS(zieladresse, erwartet, swap)

old = *$zieladresse$
if $erwartet$ == old
do *$zieladresse = swap$
return old

CUDA Cuckoo-Filter

Device

Andere Absicherung des kritischen Bereichs

AtomicCompareAndSwap

Angepasster Algorithmus:

AtomicCAS(zieladresse, erwartet, swap)

$$\text{old} = *\textit{zieladresse}$$
$$\textbf{if}\ \textit{erwartet} == \text{old}$$
$$\textbf{do}\ *\textit{zieladresse} = \textit{swap}$$
$$\textbf{return}\ \text{old}$$

CUDA Cuckoo-Filter

Device

Andere Absicherung des kritischen Bereichs

AtomicCompareAndSwap

Angepasster Algorithmus:
- Finde Kandidatenbucket

AtomicCAS(zieladresse, erwartet, swap)

old = **zieladresse*
if *erwartet* == old
do **zieladresse* = *swap*
return old

CUDA Cuckoo-Filter

Device

Andere Absicherung des kritischen Bereichs

AtomicCompareAndSwap

Angepasster Algorithmus:
- Finde Kandidatenbucket
- AtomicCAS(Kandidatenbucket, 0, Fingerprint)
- Ergebnis == 0?

AtomicCAS(zieladresse, erwartet, swap)

old = *zieladresse*
if *erwartet* == old
do *zieladresse* = *swap*
return old

CUDA Cuckoo-Filter

Device

Andere Absicherung des kritischen Bereichs

AtomicCompareAndSwap

Angepasster Algorithmus:

- Finde Kandidatenbucket
- AtomicCAS(Kandidatenbucket, 0, Fingerprint)
- Ergebnis == 0?
  - Ja, Erfolg

AtomicCAS(zieladresse, erwartet, swap)

old = **zieladresse*
if *erwartet* == old
do **zieladresse* = *swap*
return old

CUDA Cuckoo-Filter

Device

Andere Absicherung des kritischen Bereichs

AtomicCompareAndSwap

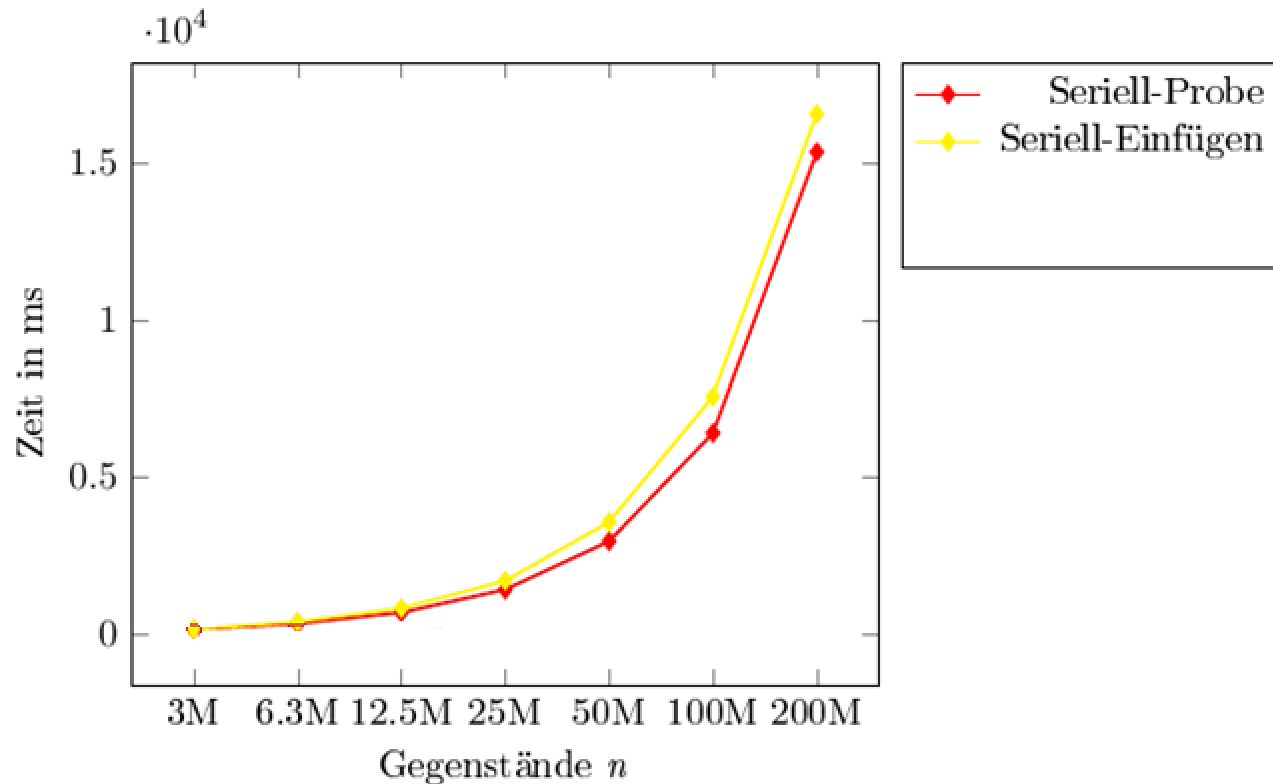
Angepasster Algorithmus:
- Finde Kandidatenbucket
- AtomicCAS(Kandidatenbucket, 0, Fingerprint)
- Ergebnis == 0?
  - Ja, Erfolg
  - Nein, verfahre wie wenn kein Kandidatenbucket gefunden wurde

AtomicCAS(zieladresse, erwartet, swap)

old = **zieladresse*
if *erwartet* == old
do **zieladresse* = *swap*
return old

CUDA Cuckoo-Filter

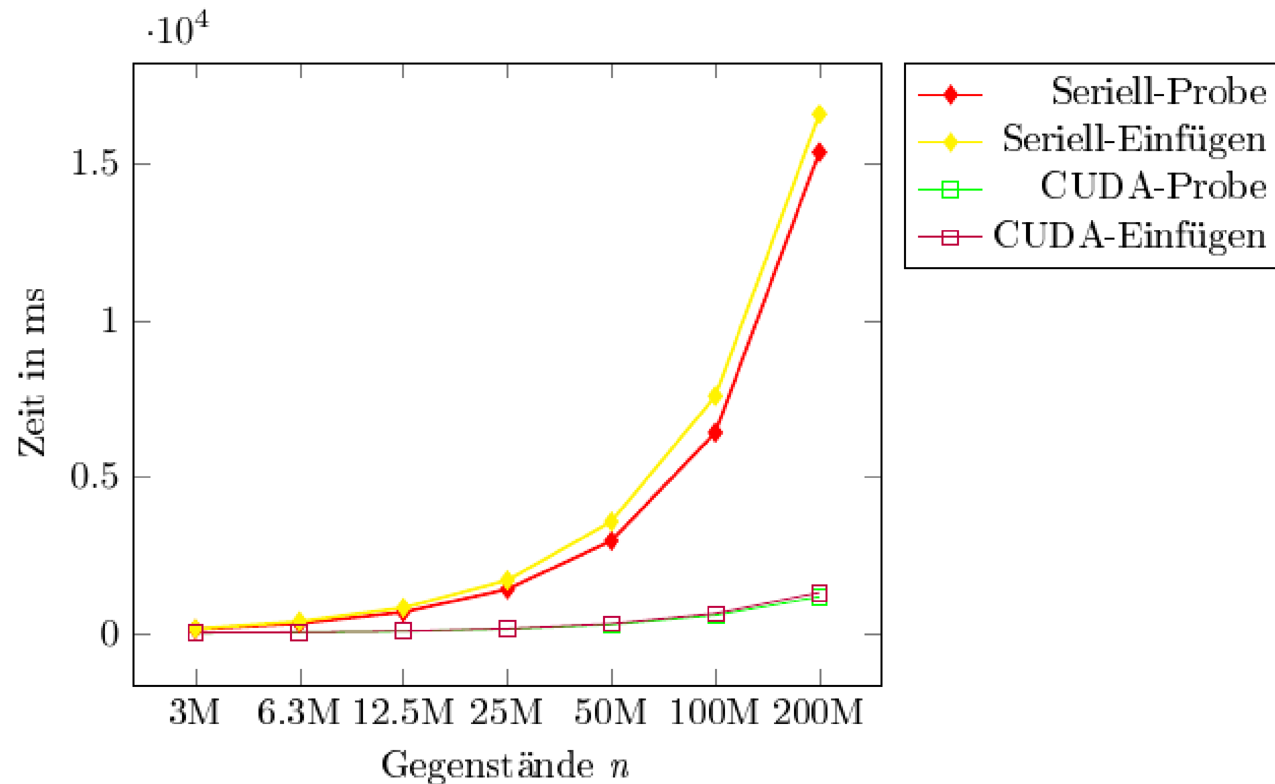
Performance?



CUDA Cuckoo-Filter

Performance

Speedup: 12.7x (**insert**), 13.4x (**probe**)



CUDA Cuckoo-Filter

Mögliche Verbesserungen:
Wahl einer schnelleren Hashfunktion (Abhängig von der Hardware)

CUDA Cuckoo-Filter

Mögliche Verbesserungen:
Wahl einer schnelleren Hashfunktion (Abhängig von der Hardware)

Caching mit Shared Memory

CUDA Cuckoo-Filter

Mögliche Verbesserungen:
Wahl einer schnelleren Hashfunktion (Abhängig von der Hardware)

Caching mit Shared Memory

Anderer Algorithmus? (Auflösen des Cuckoo-Graphen)