Parallele Algorithmen mit OpenCL

Universität Osnabrück, Henning Wenke, 2013-06-12

Kapitel

Synchronization (Fortsetzung)

Beispiel für Race-Condition

- Operation: Erhöhe Wert an Speicheradresse `adress`
  1. Read: Liest Wert an Speicheradresse `adress`
  2. Operation: Erhöht diesen um 1
  3. Write: Schreibt Ergebnis an Speicheradresse `adress` zurück
- 2 Threads führen diese Operation für gleiche `adress` aus
- Dortiger Wert sei anfänglich 0
- Einige Szenarien:

Zyklus	Thread 0			
0	Liest 0	Liest 0		
1	Berechnet 0+1	Berechnet 0 + 1	Liest 0	
2	Schreibt 1	Schreibt 1	Berechnet 0 + 1	
3			Schreibt 1	
4				Liest 1
5				Berechnet 1 + 1
				Schreibt 2
Ergebnis	?	1	1	2

Atomic Operations

- Führen folgende Sequenz von Operationen untrennbar aus:
  - Lade Wert
  - Operation auf Wert
  - Schreibe Wert zurück
- Bis Abschluss der Atomic Operation werden Zugriffe anderer Threads auf diese Daten verzögert
- Kann Konflikte des Veränderns eines Werts durch mehrere Threads lösen
- Wichtig: Garantiert keine Reihenfolge
- OpenCL:
  - Built-in Atomic Functions
  - Funktionieren global für ein Device

Atomic Functions in OpenCL C

- Angewandt auf Adresse im Global- oder Local-Memory
- Zu manipulierende Werte i.d.R. 32 Bit Integer
- 64 Bit Integer als Extensions
- Beispiele hier: Global-Memory & Datentyp int

```
// Führt untrennbar aus:
//    R  : Liest Wert an Adresse *p im Global-Memory
//    Op : Berechnet Wert + 1
//    W  : Schreibt Ergebnis an Adresse *p zurück
//    Und: Funktion alten Wert zurück
int atomic_inc(volatile global int *p)
```

- Reihenfolge beachten:
  - n Threads wenden Funktion auf Adresse an ⇒ Wert um n erhöht
  - Aber: Rückgaben (oldVal, …, oldVal+n-1) in „zufälliger“ Reihenfolge

Weitere Atomic Functions

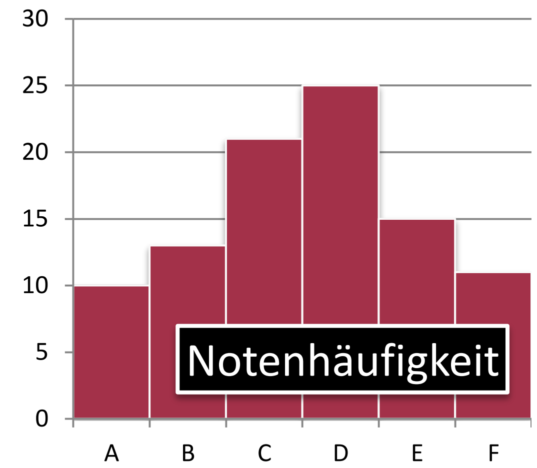
```
// Verringert Wert an Stelle von p & liefert Ergebnis
int atomic_dec(volatile global int *p)

// Verknüpft Wert an Adresse von p mit Übergebenem (hier: Addition) &
//  liefert alten Wert
// Analog: atomic_sub, atomic_max, atomic_min, atomic_and,
//  atomic_or, atomic_xor
int atomic_add(volatile global int *p, int val)


// Schreibt Wert val an Adresse von p
// Liefert alten Wert an Adresse von p
// Hinweis: Variante für float existiert
int atomic_xchg(volatile global int *p, int val)

// Liest Wert old an Adresse von p
// Berechnet new = (old == cmp) ? val : old
// Schreibt new an Adresse von p
// Liefert old
int atomic_cmpxchg(volatile global int *p, int cmp, int val)
```

Algorithmus

Histogramm Berechnung

Histogramm

- Gibt Häufigkeitsverteilung bestimmter Merkmale an
- Histogramme auf Bildaten:
  - Fotografie: Helligkeitsverteilung
  - Computer Vision, …
- Berechnung einer Helligkeitsverteilung
  - H_SIZE: Histogrammgröße (Anzahl Graustufen)
  - int hist[]: Histogramm, H_SIZE Werte, initial 0
  - VAL_CNT: Pixelanzahl
  - int vals[]: Bilddaten. Werte in [0, …, H_SIZE-1]
  - Typisch: VAL_CNT >> H_SIZE



Sequentiell

```
for(int i = 0; i < VAL_CNT; i++){
   hist[ vals[ i ] ] ++;
}
```

Parallel ϟ

```
For Each (i | i in {0,...,VAL_CNT-1}) in parallel do
   hist[ vals[ i ] ] ++;
}
```

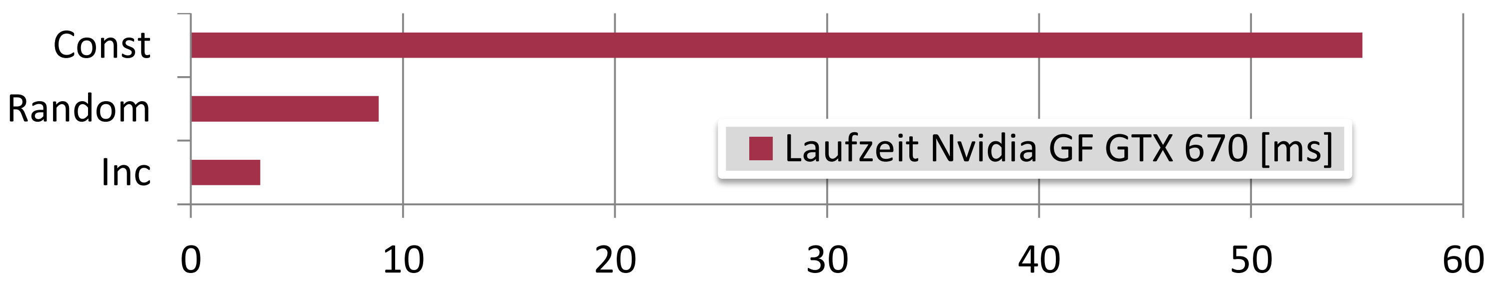
OpenCL C Lösung mit Atomic Functions

```
// Erzeuge VAL_CNT work-items
kernel void histogram(
      global int* hist,
      global int* vals)
{
      int pixelId = get_global_id(0);
      atomic_inc(hist + vals[pixelId]);
}
```

Laufzeitmessungen

- Größen
  - VAL_CNT: 2^26
  - HIST_SIZE: 1024
- Datensätze

inc	0	1	…	1023	0	1	…	1023	…	…	1023
rand	90	374	3	90	3	123	2	213	…	11	159
const	90	90	90	90	90	90	90	90	…	90	90



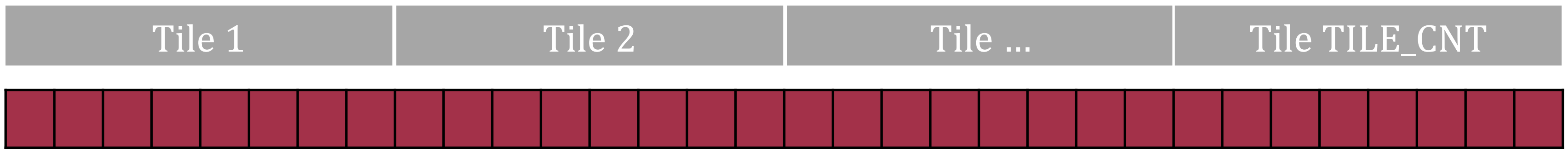
- *const:* Worst-case, erzwingt vollständige Serialisierung
- *inc:* Best-case, maximal HIST_SIZE parallel ausführbar

Verringerung der Konflikte I

Definiere Tile:

Repräsentiert TILE_SIZE = VAL_CNT / TILE_CNT Datenelemente

VAL_CNT Elemente



Stufe I

Berechne lokales Histogramm per Tile

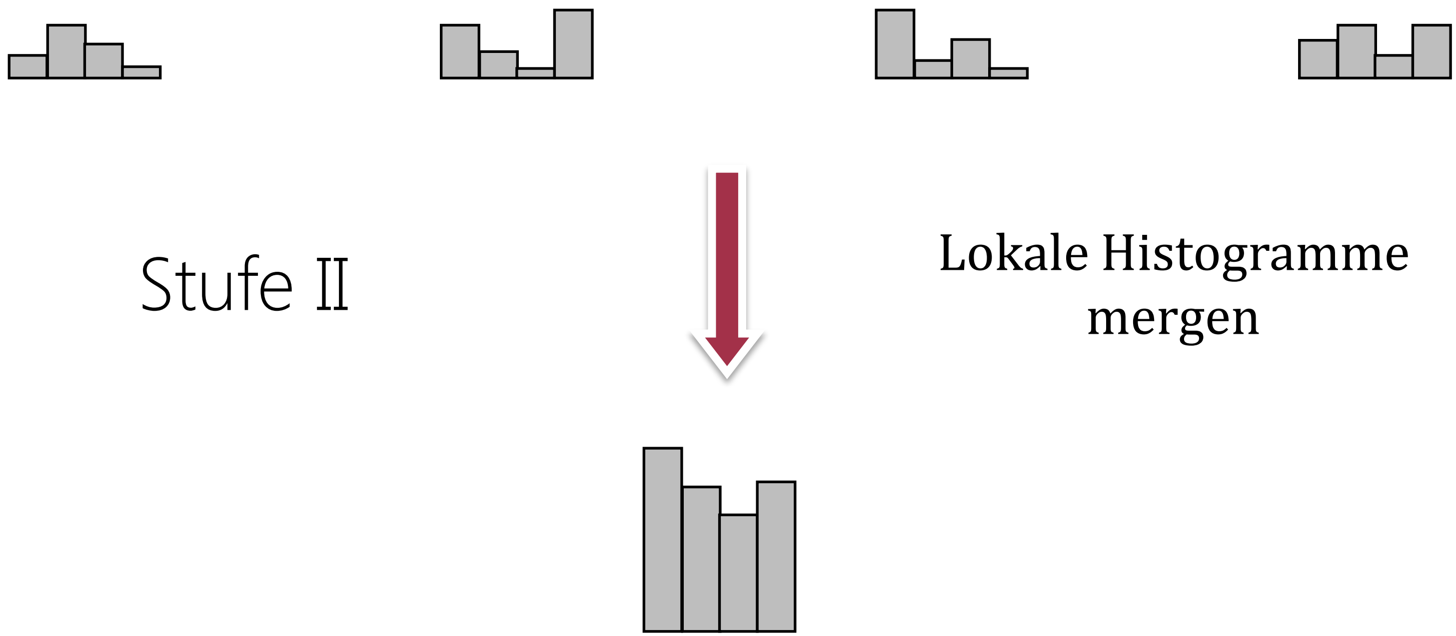
Variante A: Atomic Functions

- Konflikte: ≤ Tile_SIZE per Tile
- Parallelität: ≥ TILE_CNT

Variante B: Sequentiell per Tile

- Konflikte: Keine
- Parallelität: TILE_CNT

Verringerung der Konflikte II



Variante A:
Atomic Functions

- Konflikte: ≤ TILE_CNT per Hist. Elem
- Parallelität = HIST_SIZE

Variante B:
Sequentiell per Hist. Elem.

- Konflikte: Keine
- Parallelität: HIST_SIZE

Variante C:
Reduction per Hist. Elem.

- Konflikte: Keine
- Barrier nötig
- Parallelität >>= HIST_SIZE

  Nicht Datenabhängig

Kapitel

Bewertung & Eigenschaften paralleler Algorithmen

PRAM

- Verbreitetes Modell zur theoretischen Bewertung
- Random Access Maschine (RAM)
  - SISD / v. Neumann
  - Zugriff auf beliebige Speicheradresse: $\boldsymbol{O}(1)$
  - Elementare Berechnung: $\boldsymbol{O}(1)$
- Parallel Random Access Machine (PRAM)
  - p Processing-Elements(PE) des Typs RAM
  - Shared-Memory
  - Kommunikation zwischen PE: $\boldsymbol{O}(1)$
  - R/W-Conflicts: Unklar
  - Achtung: Asymptotisches Verhalten oft je Prozessor angegeben
- Intuitiv & erlaubt Focus auf Parallelität

Weitere Bezeichnungen

- ***Optimal***: Algorithmus unterscheidet sich asymptotisch höchstens um konstanten Faktor von bestmöglicher Lösung
- ***Scalable***: Algorithmus, dessen Parallelität mindestens linear mit der Problemgröße ansteigt
- ***Work-efficient***: Algorithmus, dessen asymptotisches Verhalten sequentieller Variante entspricht
- ***Throughput***: Messung verarbeiteter Datenelemente pro Zeiteinheit
- ***Speedup***: Verhältnis gemessener Laufzeit zu schnellstem sequentiellen Algorithmus
- ***Amdahl‘s Law***: Theoretische Obergrenze für Speedup
  - $f \in [0,1]$: Anteil sequentieller Berechnungen des Algorithmus
  - $T(1)$: Gemessene Ausführungszeit des schnellsten sequentiellen Algorithmus
  - $T(p) = T(1) \cdot \left(f + \frac{1-f}{p}\right)$: Untergrenze für Ausführungszeit auf p Prozessoren
  - $Speedup_{\max}(p) = \frac{T(1)}{T(p)}$

Beispiel: n-D Vektoraddition

Linear

- Laufzeit: O(n)
- Speicher: O(n)
- Optimal: Ja

Parallel

- Laufzeit je PE: O(1)
- Gesamtlaufzeit: O(n)
  - n Prozessing Elements
  - Jedes O(1)
- Optimal: Ja
- Work-efficient: Ja
- Scalable: Ja
- Speedup: ?
- Amdahl / max Speedup: n

Algorithmus

All-Prefix-Sums / Scan

Quelle: GPU Gems 3

Prinzip

- Gegeben: Array aus n Elementen $\{a_0, a_1, \dots, a_{n-1}\}$
- Definiere binären assoziativen Operator $\otimes$ und neutrales Element I
- Exclusive Scan liefert:
  $\{I,\ a_0,\ (a_0 \otimes a_1)\ \dots, (a_0 \otimes a_1 \otimes \cdots \otimes a_{n-2})\}$
- Inclusive Scan liefert:
  $\{a_0,\ (a_0 \otimes a_1)\ \dots, (a_0 \otimes a_1 \otimes \cdots \otimes a_{n-1})\}$
- Beispiel: Inclusive Scan, Operator +, I = 0

In	1	0	1	1	3	5	0	1
Out	1	1	2	3	6	11	11	12

Sequentieller Algorithmus

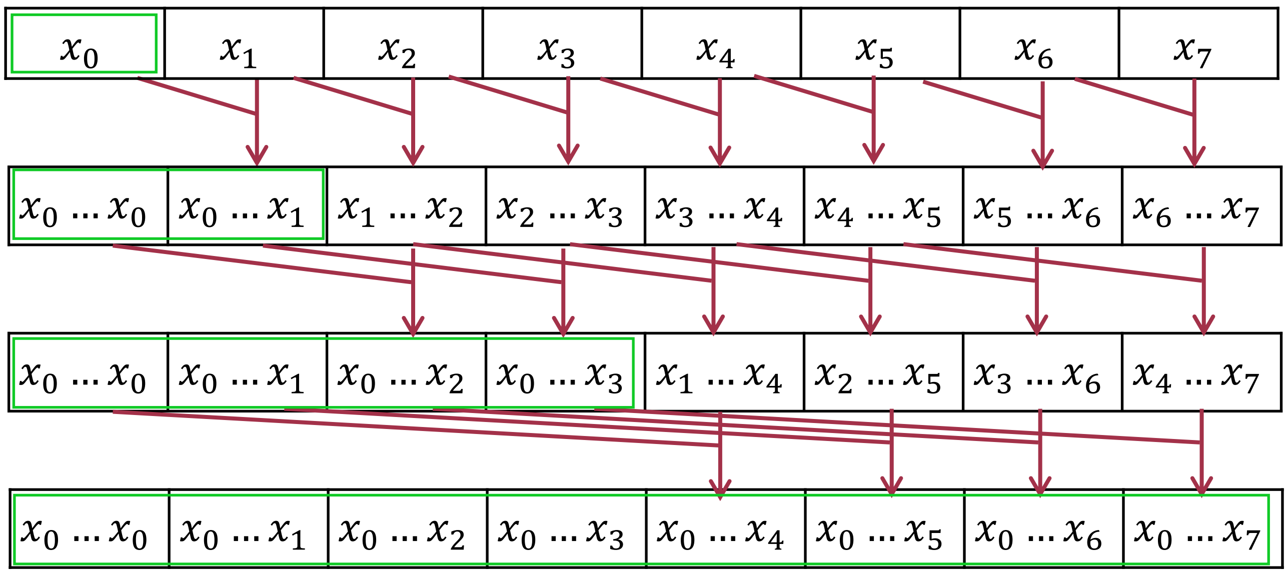
- Exclusive Scan
- Operator +

```
int[] in, initialisiert, Länge: n
int[] out, Länge: n
out[0] ← 0
for(int i = 1; i < n; i++) do
   out[i] ← out[i - 1] + in[i - 1]
end
```

- Beobachtung: Zur Berechnung jedes `out[i]` ist berechneter `out[i-1]` nötig
- Laufzeit: $\boldsymbol{O}(n)$

Paralleler Algorithmus (naiv)

- Inclusive Scan, +
- Bezeichnung (auch folgende Folien): $x_a \dots x_b \coloneqq \sum_{j=a}^{b}(x_j)$

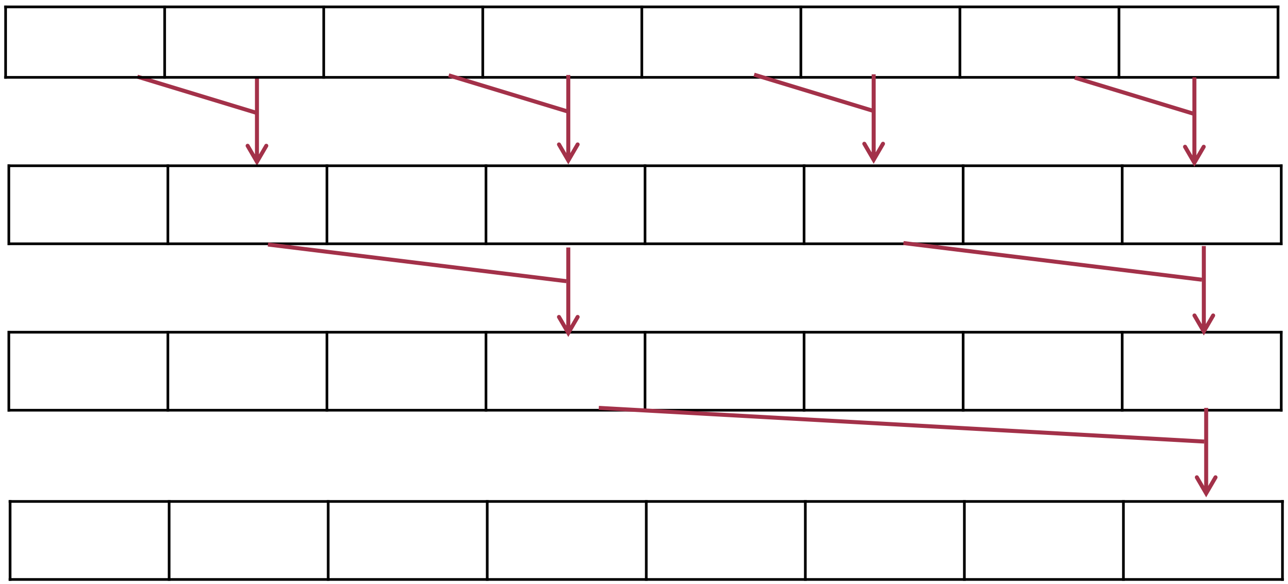


- Laufzeit: $\boldsymbol{O}(n \cdot \log(n))$

  $\Longrightarrow$ Nicht work-efficient
- Scalable

Ansatz

- Verwende binary balanced Tree:

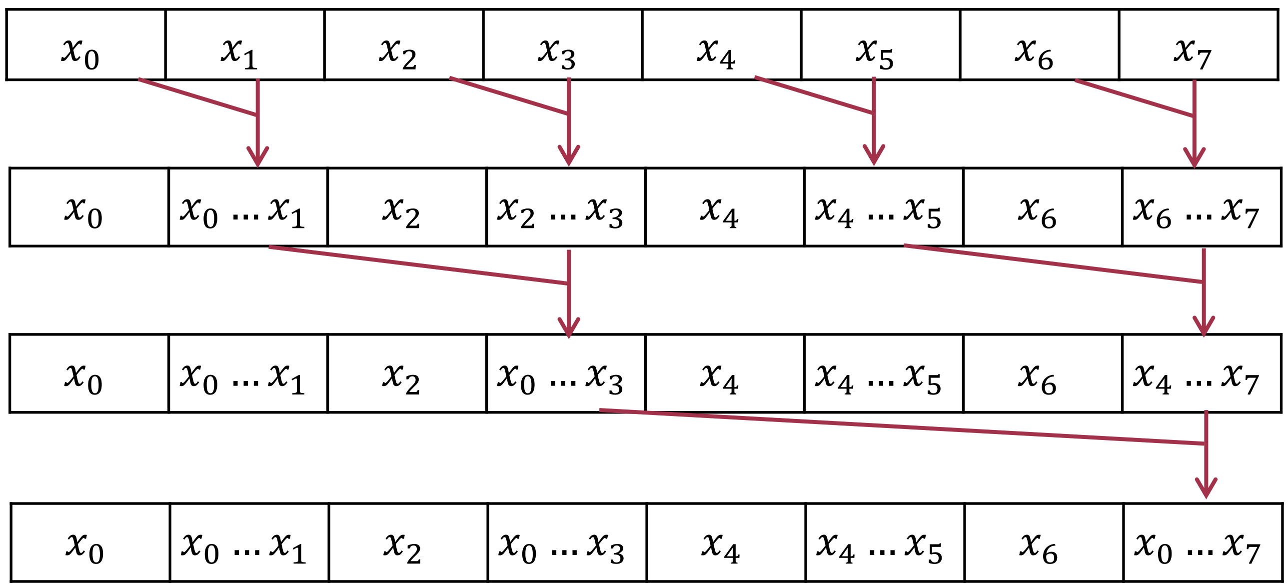


- Schema ergibt Laufzeit $\boldsymbol{O}(n)$
- Ggf. mehrfach verwenden (konstante Anzahl)

  $\Longrightarrow$ Bleibt bei $\boldsymbol{O}(n)$

I. Reduction

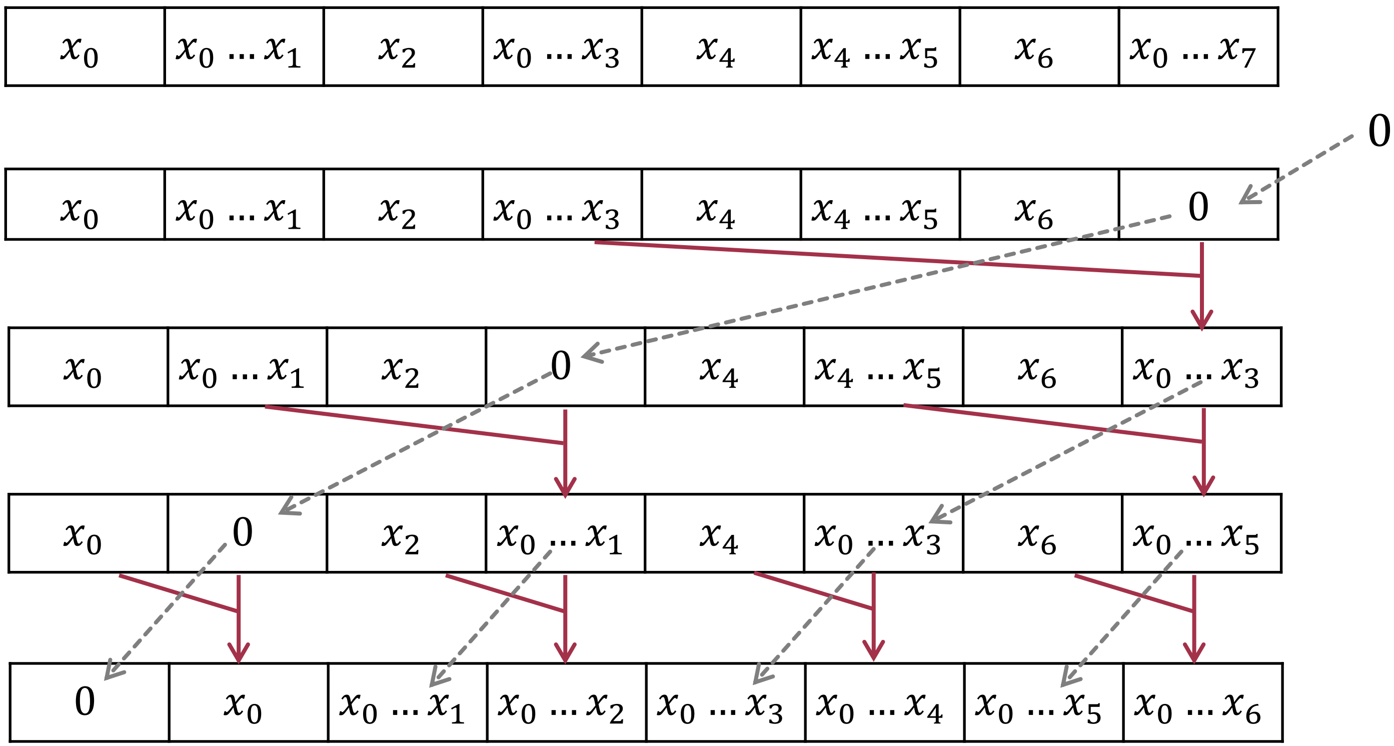
- Erster Schritt: Wende Reduction an



- Nächstes Ziel:
  - Berechne mit zweitem, Reduction-ähnlichem Schritt, Scan
  - Vertausche ggf. Werte

II. Reduction (invers) mit Vertauschungen

- Berechne exclusive Scan, +



- Prinzip: Jeder in der Iteration aktive Knoten speichert…
  - … eigenen Wert in linkem Kindknoten
  - … eigenen Wert + alten Wert des linken childs in rechtem child
- Letzter Wert wird anfänglich auf Null gesetzt

Algorithmus

```
Data: vals[], initialisiert, (length n, mit n zweier Potenz)
stride ← 1, threadCnt ← n / 2
Do ( log₂ n times )
    For Each (Thread t, t ∈ {0, …, threadCnt-1}) in parallel do
        firstId ← 2 · stride · t + stride -1
        secondId ← firstId + stride
        vals[secondId] ← vals[firstId] + vals[secondId]
    End
    stride ← stride · 2
    threadCnt ← threadCnt / 2
End
vals[n-1] ← 0
stride ← n / 2, threadCnt ← 1
Do ( log₂ n times )
    For Each (Thread t, t ∈ {0, …, threadCnt-1}) in parallel do
        firstId ← 2 · stride · t + stride -1
        secondId ← firstId + stride
        tmpVal ← vals[secondId]
        vals[secondId] ← vals[firstId] + tmpVal
        vals[firstId] ← tmpVal
    End
    stride ← stride / 2
    threadCnt ← threadCnt · 2
End
```

Laufzeit: $O(n)$

Work-efficient

Scalable

Weiterer Verlauf

- Geplante Vorlesungsthemen (Stand 2013-04-10)
  - Hello OpenCL
  - Konzepte der parallelen Programmierung
  - Parallele Algorithmen & OpenCL
  - Optimierung
  - Dynamic Parallelism*
  - Distributed Memory*
  - Genauigkeit*
- *Themen nicht mehr im Vorlesungsteil
- Können auch als Projekte vergeben werden
  - V.a. Distributed Memory
- Themenvergabe: Letzter Vorlesungstermin